Logical Methods in Computer Science
Vol. 8(3:25)2012, pp. 1–21
www.lmcs-online.org

Submitted Oct. 14, 2011
Published Sep. 29, 2012

PREDICATE GENERATION FOR LEARNING-BASED QUANTIFIER-FREE LOOP INVARIANT INFERENCE *

WONCHAN LEE [a], YUNGBUM JUNG [b], BOW-YAW WANG [c], AND KWANGKUEN YI [d]

[a,d] Seoul National University, Korea
e-mail address: {wclee, kwang}@ropas.snu.ac.kr

[b] Fasoo.com, Korea and Seoul National University, Korea
e-mail address: yb@fasoo.com

[c] Academia Sinica, Taiwan
e-mail address: bywang@iis.sinica.edu.tw

Abstract. We address the predicate generation problem in the context of loop invariant inference. Motivated by the interpolation-based abstraction refinement technique, we apply the interpolation theorem to synthesize predicates implicitly implied by program texts. Our technique is able to improve the effectiveness and efficiency of the learning-based loop invariant inference algorithm of Jung, Kong, Wang and Yi (2010). We report experimental results of examples from Linux, SPEC2000, and the Tar utility.

1. Introduction

One way to prove that an annotated loop satisfies its pre- and post-conditions is by giving loop invariants. In an annotated loop, pre- and post-conditions specify intended effects of the loop. The actual behavior of the annotated loop however does not necessarily conform to its specification. Through loop invariants, verification tools can check whether the annotated loop fulfills its specification automatically [9].

Finding loop invariants is tedious and sometimes requires intelligence. Recently, an automated technique based on algorithmic learning and predicate abstraction is proposed [14]. Given a fixed set of atomic predicates and an annotated loop, the learning-based technique can infer a quantifier-free loop invariant over the given atomic predicates. By employing

1998 ACM Subject Classification: F.3.1.

Key words and phrases: loop invariant, algorithmic learning, predicate generation, interpolation.

* This paper is a revised and extended version of the paper "Predicate Generation for Learning-Based Quantifier-Free Loop Invariant Inference" that has been published in the proceedings of TACAS 2011 [15].
This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grant 2012-0000468), National Science Council of Taiwan Grant Numbers 99-2218-E-001-002-MY3 and 100-2221-E-002-116-, National Science Foundation (award no. CNS0926181), and by Republic of Korea Dual Use Program Cooperation Center(DUPC) of Agency for Defense Development(ADD).

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-8(3:25)2012

a learning algorithm and a mechanical teacher, the new technique is able to generate loop invariants without constructing abstract models nor computing fixed points.

As in other techniques based on predicate abstraction, the selection of atomic predicates is crucial to the effectiveness of the learning-based technique. Oftentimes, users extract atomic predicates from program texts heuristically. If this simple strategy does not yield necessary atomic predicates to express any loop invariants the loop invariant inference algorithm will not be able to infer a loop invariant. Even when the heuristic does give necessary atomic predicates, it may select too many redundant predicates and impede the efficiency of loop invariant inference algorithm.

One way to circumvent this problem is to generate atomic predicates by need. Several techniques have been developed to synthesize atomic predicates by interpolation [8, 12, 19, 20]. Let A and B be logic formulae. An interpolant I of A and B is a formula such that $A \Rightarrow I$ and $I \wedge B$ is inconsistent. Moreover, the non-logical symbols in I must occur in both A and B. By Craig's interpolation theorem, an interpolant I always exists for any first-order formulae A and B when $A \wedge B$ is inconsistent [6]. The interpolant I can be seen as a concise summary of A with respect to B. Indeed, many abstraction refinement techniques for software model checking [8, 11, 12, 19, 20] have used interpolation to synthesize atomic predicates.

Inspired by the refinement technique in software model checking, we develop an interpolation-based technique to synthesize atomic predicates in the context of learning-based loop invariant inference. Our algorithm does not add new atomic predicates by interpolating invalid execution paths in control flow graphs. We instead interpolate the loop body with purported loop invariants from the learning algorithm. We adopt the existing interpolating theorem provers [1, 2, 3, 19] for the interpolation. With our new predicate generation technique, we can improve the effectiveness and efficiency of the existing learning-based loop invariant inference technique [14]. Constructing the set of atomic predicates is fully automatic and on-demand.

1.1. **Example.** Consider the following annotated loop:

$$
\begin{array}{l}
\{\ n \geq 0 \wedge x = n \wedge y = n\ \} \\
\texttt{while}\ x > 0\ \texttt{do} \\
\quad x = x - 1;\ y = y - 1 \\
\texttt{done} \\
\{\ x + y = 0\ \}
\end{array}
$$

Assume that variables x and y both have the value $n \geq 0$ before entering the loop. The loop body decreases each variable by one until the variable x becomes zero. We want to show that $x + y$ is zero after executing the loop. This requires of us to establish the fact that variables x and y have the same value during iterations and eventually become zero after exiting the loop. To express this fact as a loop invariant, we require a predicate $x = y$. The program text however does not reveal this equality explicitly. Moreover, atomic predicates from the program text cannot express any loop invariant that establishes the given specification. Using atomic predicates in the program text is not sufficient in this case. However, we can exploit the fact that any loop invariant ι should be weaker than the pre-condition δ and

stronger than the disjunction of the loop guard κ and the post-condition ϵ ($\delta \Rightarrow \iota \Rightarrow \kappa \vee \epsilon$). Then, we can gen an interpolant from inconsistent formula $\delta \wedge \neg(\kappa \vee \epsilon)$ and extract atomic predicates in it. From the interpolant of $(n \geq 0 \wedge x = n \wedge y = n) \wedge \neg(x > 0 \vee x + y = 0)$, we obtain two atomic predicates $x = y$ and $2y \geq 0$. Observe that the interpolation is able to synthesize the necessary predicate $x = y$. In fact, loop invariant $x = y \wedge x \geq 0$ establishes the specification of the loop.

1.2. **Related Work.** Jung *et al.* [14] introduce the loop invariant inference technique based on algorithmic learning. Kong *et al.* [16] extend this technique to quantified loop invariant inference. Both algorithms require users to provide atomic predicates. The present work addresses this problem for the case of quantifier-free loop invariants.

Recently, Lee *et al.* [18] introduce learning-based technique for termination analysis. The technique infers the transition invariant of a given loop as a proof of termination, by combining algorithmic learning and decision procedures. In the paper, the authors design a heuristic to generate atomic transition predicates. It is an interesting future work to adapt our technique in the present paper for transition invariant inference.

Many interpolation algorithms and their implementations are available [1, 2, 3, 19]. Interpolation-based techniques for predicate refinement in software model checking are proposed in [8, 11, 12, 13, 20]. Abstract models used in these techniques however may require excessive invocations to theorem provers. Another interpolation-based technique for first-order invariants is developed in [21]. The paramodulation-based technique presented in the paper does not construct abstract models as our approach. It however only generates invariants in first-order logic with equality. A template-based predicate generation technique for quantified invariants is proposed [22]. The technique reduces the invariant inference problem to constraint programming and generates predicates in user-provided templates.

1.3. **Paper Organization.** Section 2 gives preliminaries for the presentation. Section 3 reviews the learning-based loop invariant inference framework [14]. Section 4 presents our interpolation-based predicate generation technique. Section 5 presents the loop invariant inference algorithms with automatic predicate generation. Section 6 presents and discusses our experimental results. Section 7 concludes this work.

2. Preliminaries

2.1. **Quantifier-free Formulae.** Let QF denote the quantifier-free logic with equality, linear inequality, and uninterpreted functions. Define the *domain* $\mathbb{D} = \mathbb{Q} \cup \mathbb{B}$ where $\mathbb{Q}$ is the set of rational numbers and $\mathbb{B} = \{F, T\}$ is the Boolean domain. Fix a set X of variables. A *valuation* over X is a function from X to $\mathbb{D}$. The class of valuations over X is denoted by Val_X. For any formula $\theta \in QF$ and valuation ν over free variables in θ, θ is *satisfied* by ν (written $\nu \models \theta$) if θ evaluates to T under ν; θ is *inconsistent* if θ is not satisfied by any valuation. Given a formula $\theta \in QF$, a *satisfiability modulo theories (SMT) solver* returns a satisfying valuation ν of θ if θ is not inconsistent [3, 7].

2.2. **Interpolation Theorem.** For $\theta \in QF$, we denote the set of non-logical symbols occurred in θ by $\sigma(\theta)$. Let $\Theta = [\theta_1, \ldots, \theta_m]$ be a sequence with $\theta_i \in QF$ for $1 \leq i \leq m$. The sequence Θ is *inconsistent* if $\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_m$ is inconsistent. The sequence $\Lambda = [\lambda_0, \lambda_1, \ldots, \lambda_m]$ of quantifier-free formulae is an *inductive interpolant* of Θ if

- $\lambda_0 = T$ and $\lambda_m = F$;
- for all $1 \leq i \leq m$, $\lambda_{i-1} \wedge \theta_i \Rightarrow \lambda_i$; and
- for all $1 \leq i < m$, $\sigma(\lambda_i) \subseteq \sigma(\theta_i) \cap \sigma(\theta_{i+1})$.

The third condition of interpolants makes them attractive to use for predicate generation; since the set of symbols in an interpolant should be an intersection of sets of symbols in two inconsistent formulae, it sometimes consists of predicates which do not appear in the two. The interpolation theorem states that an inductive interpolant exists for any inconsistent sequence [6, 19, 20]. Some of existing theorem provers [1, 2, 3, 19] can generate interpolants from inconsistent sequences.

2.3. **Predicate Abstraction.** Let $QF[P]$ denote the set of quantifier-free formulae over the set P of atomic predicates. A cube over P is a conjunction $p_1 \wedge \cdots \wedge p_k \wedge \neg p_{k+1} \wedge \cdots \wedge \neg p_{k+k'}$ where all $p_j \in P$ are distinct. We say that $k + k'$ is the size of the cube. A minterm over P is a cube whose size is $|P|$.

Consider the set $Bool[B_P]$ of Boolean formulae over the set B_P of Boolean variables where $B_P \stackrel{\triangle}{=} \{b_p : p \in P\}$. An *abstract valuation* is a function from B_P to $\mathbb{B}$. We write Val_{B_P} for the set of abstract valuations. A Boolean formula in $Bool[B_P]$ is a *canonical monomial* if it is a conjunction of literals, where each Boolean variable in B_P occurs exactly once. The following functions [14, 15] relate formulae in $QF[P]$ and $Bool[B_P]$ (Figure 1):

$$
\begin{array}{rcl}
\gamma(\beta) & \stackrel{\triangle}{=} & \beta[B_P \mapsto P] \\
\alpha(\theta) & \stackrel{\triangle}{=} & \bigvee\{\beta \in Bool[B_P] : \beta \text{ is a canonical monomial and } \theta \wedge \gamma(\beta) \text{ is satisfiable}\} \\
\gamma^*(\mu) & \stackrel{\triangle}{=} & \bigwedge_{\mu(b_p)=T} \{p\} \wedge \bigwedge_{\mu(b_p)=F} \{\neg p\} \\
\alpha^*(\nu) & \stackrel{\triangle}{=} & \mu \text{ where } \mu(b_p) = \left\{ \begin{array}{ll} T & \text{if } \nu \models p \\ F & \text{if } \nu \not\models p \end{array} \right. \\
\Gamma(\nu) & \stackrel{\triangle}{=} & \bigwedge_{x \in X} x = \nu(x)
\end{array}
$$

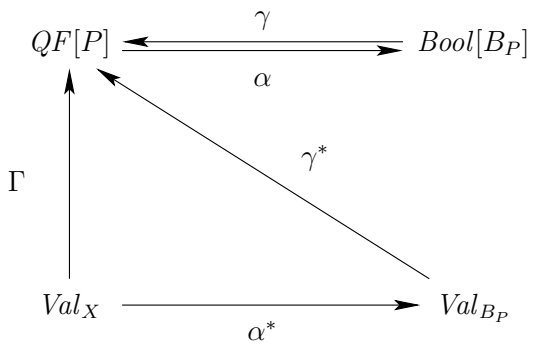

Figure 1. Relating QF and $Bool[B_P]$

The abstraction function α maps any quantifier-free formula to a Boolean formula in $Bool[B_P]$, whereas the concretization function γ maps any Boolean formula in $Bool[B_P]$ to a quantifier-free formula in $QF[P]$. Moreover, the function α^* maps a valuation over X to a valuation over B_P; the function γ^* maps a valuation over B_P to a quantifier-free formula in $QF[P]$. The function $\Gamma(\nu)$ specifies the valuation ν in QF. Observe that quantifier-free formula $\gamma(\beta)$ is a minterm when Boolean formula β is a canonical monomial. Observe also that formula $\gamma(\alpha(\theta))$ is in disjunctive normal form and equivalent to $\theta \in QF[P]$.

Consider, for instance, $P = \{n \geq 0, x = n, y = n\}$ and $B_P = \{b_{n\geq 0}, b_{x=n}, b_{y=n}\}$. We have $\gamma(b_{n\geq 0} \wedge \neg b_{x=n}) = n \geq 0 \wedge \neg(x = n)$ and

$$\begin{array}{rl} & (b_{n\geq 0} \wedge b_{x=n} \wedge \neg b_{y=n}) \vee (b_{n\geq 0} \wedge \neg b_{x=n} \wedge b_{y=n}) \vee \\ \alpha(\neg(x = y)) = & (b_{n\geq 0} \wedge \neg b_{x=n} \wedge \neg b_{y=n}) \vee (\neg b_{n\geq 0} \wedge b_{x=n} \wedge \neg b_{y=n}) \vee \\ & (\neg b_{n\geq 0} \wedge \neg b_{x=n} \wedge b_{y=n}) \vee (\neg b_{n\geq 0} \wedge \neg b_{x=n} \wedge \neg b_{y=n}). \end{array}$$

Moreover, $\alpha^*(\nu)(b_{n\geq 0}) = \alpha^*(\nu)(b_{x=n}) = \alpha^*(\nu)(b_{y=n}) = T$ when $\nu(n) = \nu(x) = \nu(y) = 1$. And $\gamma^*(\mu) = n \geq 0 \wedge x = n \wedge \neg(y = n)$ when $\mu(b_{n\geq 0}) = \mu(b_{x=n}) = T$ but $\mu(b_{y=n}) = F$.

The following lemmas prove useful properties of these abstraction and concretization functions.

Lemma 2.1. *Let P be a set of atomic predicates, $\theta \in QF[P]$, and β a canonical monomial in $Bool[B_P]$. Then $\theta \wedge \gamma(\beta)$ is satisfiable if and only if $\gamma(\beta) \Rightarrow \theta$.*

Proof. Let $\theta' = \bigvee_i \theta_i \in QF[P]$ be a formula in disjunctive normal form such that $\theta' \Leftrightarrow \theta$. Note that each θ_i is a cube over set P. Let $Lit(\theta)$ be a set of literals in formula θ. Then, $Lit(\theta_i) \subseteq P \cup \{\neg p : p \in P\}$.

Assume $\theta \wedge \gamma(\beta)$ is satisfiable. Then $\theta' \wedge \gamma(\beta)$ is satisfiable and $\theta_i \wedge \gamma(\beta)$ is satisfiable for some i. Since β is canonical monomial, $\gamma(\beta)$ is a minterm over set P and $Lit(\theta) \subseteq Lit(\gamma(\beta))$. Hence $\theta_i \wedge \gamma(\beta)$ is satisfiable implies $\gamma(\beta) \Rightarrow \theta_i$. We have $\gamma(\beta) \Rightarrow \theta$.

The other direction is trivial. □

Lemma 2.2. *Let P be a set of atomic predicates, $\theta, \rho \in QF[P]$. Then*

$$\theta \Rightarrow \rho \text{ implies } \alpha(\theta) \Rightarrow \alpha(\rho).$$

Proof. Let $\alpha(\theta) = \bigvee_i \beta_i$ where β_i is a canonical monomial and $\theta \wedge \gamma(\beta_i)$ is satisfiable. By Lemma 2.1, $\gamma(\beta_i) \Rightarrow \theta$. Hence $\gamma(\beta_i) \Rightarrow \rho$ and $\rho \wedge \gamma(\beta_i)$ is satisfiable. □

Lemma 2.3. *Let P be a set of atomic propositions and $\theta \in QF[P]$. Then $\theta \Leftrightarrow \gamma(\alpha(\theta))$.*

Proof. Let $\theta' = \bigwedge_i \theta_i$ be a quantified-free formula in disjunctive normal form such that $\theta' \Leftrightarrow \theta$. Let $\mu \in Bool[B_P]$. Define

$$\chi(\mu) = \bigwedge(\{b_p : \mu(b_p) = T\} \cup \{\neg b_p : \mu(b_p) = F\}).$$

Note that $\chi(\mu)$ is a canonical monomial and $\mu \models \chi(\mu)$.

Assume $\nu \models \theta$. Then $\nu \models \theta_i$ for some i. Consider the canonical monomial $\chi(\alpha^*(\nu))$. Note that $\nu \models \gamma(\chi(\alpha^*(\nu)))$. Thus $\chi(\alpha^*(\nu))$ is a disjunct in $\alpha(\theta)$. We have $\nu \models \gamma(\alpha(\theta))$.

Conversely, assume $\nu \models \gamma(\alpha(\theta))$. Then $\nu \models \gamma(\beta)$ for some canonical monomial β and $\gamma(\beta) \wedge \theta$ is satisfiable. By Lemma 2.1, $\gamma(\beta) \Rightarrow \theta$. Hence $\nu \models \theta$. □

Lemma 2.4. *Let P be a set of atomic propositions, $\theta \in QF[P]$, $\beta \in Bool[B_P]$, and ν a valuation for X. Then*
(1) *$\nu \models \theta$ if and only if $\alpha^*(\nu) \models \alpha(\theta)$; and*
(2) *$\nu \models \gamma(\beta)$ if and only if $\alpha^*(\nu) \models \beta$.*

Proof.
(1) Assume $\nu \models \theta$. $\chi(\alpha^*(\nu))$ is a canonical monomial. Observe that $\nu \models \gamma(\chi(\alpha^*(\nu)))$. Hence $\gamma(\chi(\alpha^*(\nu))) \wedge \theta$ is satisfiable. By the definition of $\alpha(\theta)$ and $\chi(\alpha^*(\nu))$ is canonical, $\chi(\alpha^*(\nu)) \Rightarrow \alpha(\theta)$. $\alpha^*(\nu) \models \alpha(\theta)$ follows from $\alpha^*(\nu) \models \chi(\alpha^*(\nu))$.
Conversely, assume $\alpha^*(\nu) \models \alpha(\theta)$. Then $\alpha^*(\nu) \models \beta$ where β is a canonical monomial and $\gamma(\beta) \wedge \theta$ is satisfiable. By the definition of $\alpha^*(\nu)$, $\nu \models \gamma(\beta)$. Moreover, $\gamma(\beta) \Rightarrow \theta$ by Lemma 2.1. Hence $\nu \models \theta$.
(2) Assume $\nu \models \gamma(\beta)$. By Lemma 2.4 1, $\alpha^*(\nu) \models \alpha(\gamma(\beta))$. Note that $\beta = \alpha(\gamma(\beta))$. Thus $\alpha^*(\nu) \models \beta$.

□

Lemma 2.5. *Let P be a set of atomic propositions, $\theta \in QF[P]$, and μ a Boolean valuation for B_P. Then $\gamma^*(\mu) \Rightarrow \theta$ if and only if $\mu \models \alpha(\theta)$.*

Proof. Assume $\gamma^*(\mu) \Rightarrow \theta$. By Lemma 2.2, $\alpha(\gamma^*(\mu)) \Rightarrow \alpha(\theta)$. Note that $\gamma^*(\mu) = \gamma(\chi(\mu))$. By Lemma 2.3, $\chi(\mu) \Rightarrow \alpha(\theta)$. Since $\mu \models \chi(\mu)$, we have $\mu \models \alpha(\theta)$.

Conversely, assume $\mu \models \alpha(\theta)$. We have $\chi(\mu) \Rightarrow \alpha(\theta)$ by the definition of $\chi(\mu)$. Let $\nu \models \gamma^*(\mu)$, that is, $\nu \models \gamma(\chi(\mu))$. By Lemma 2.4 (2), $\alpha^*(\nu) \models \chi(\mu)$. Since $\chi(\mu) \Rightarrow \alpha(\theta)$, $\alpha^*(\nu) \models \alpha(\theta)$. By Lemma 2.4 (1), $\nu \models \theta$. Therefore, $\gamma^*(\mu) \Rightarrow \theta$. □

2.4. **CDNF Learning Algorithm.** CDNF algorithm [4] is an exact learning algorithm for Boolean formulae based on monotone theory. It infers an unknown target formula by posing queries to a teacher. The teacher is responsible for answering two types of queries. The learning algorithm may ask if a valuation satisfies the target formula by a membership query. Or it may ask if a conjectured formula is equivalent to the target in an equivalence query. Using the answers for the queries, CDNF algorithm infers a Boolean formula equivalent to the unknown target within a polynomial number of queries in the formula size of the target [4].

2.5. **Programs.** We consider the following imperative language in this paper:

$$\begin{aligned} \mathsf{Stmt} &\triangleq \texttt{nop} \mid \mathsf{Stmt}; \mathsf{Stmt} \mid x := \mathsf{Exp} \mid x := \texttt{nondet} \mid \texttt{if}\ \mathsf{BExp}\ \texttt{then}\ \mathsf{Stmt}\ \texttt{else}\ \mathsf{Stmt} \\ \mathsf{Exp} &\triangleq n \mid x \mid \mathsf{Exp} + \mathsf{Exp} \mid \mathsf{Exp} - \mathsf{Exp} \\ \mathsf{BExp} &\triangleq F \mid x \mid \neg\mathsf{BExp} \mid \mathsf{BExp} \wedge \mathsf{BExp} \mid \mathsf{Exp} < \mathsf{Exp} \mid \mathsf{Exp} = \mathsf{Exp} \end{aligned}$$

Two basic types are available: natural numbers and Booleans. A term in Exp is a natural number; a term in BExp is of Boolean type. The keyword `nondet` denotes an arbitrary value in the type of the assigned variable. An *annotated loop* is of the form:

$$\{\delta\}\ \texttt{while}\ \kappa\ \texttt{do}\ S_1; S_2; \cdots; S_m\ \texttt{done}\ \{\epsilon\}$$

The BExp formula κ is the *loop guard*. The BExp formulae δ and ϵ are the *precondition* and *postcondition* of the annotated loop respectively.

Define $X^{\langle k \rangle} = \{x^{\langle k \rangle} : x \in X\}$. For any term e over X, define $e^{\langle k \rangle} = e[X \mapsto X^{\langle k \rangle}]$. A *transition formula* $[\![S]\!]$ for a statement S is a first-order formula over variables $X^{\langle 0 \rangle} \cup X^{\langle 1 \rangle}$ defined as follows.

$$
\begin{aligned}
[\![\mathsf{nop}]\!] &\triangleq \bigwedge_{x \in X} x^{\langle 1 \rangle} = x^{\langle 0 \rangle} \\
[\![x := \mathsf{nondet}]\!] &\triangleq \bigwedge_{y \in X \setminus \{x\}} y^{\langle 1 \rangle} = y^{\langle 0 \rangle} \\
[\![x := e]\!] &\triangleq x^{\langle 1 \rangle} = e^{\langle 0 \rangle} \wedge \bigwedge_{y \in X \setminus \{x\}} y^{\langle 1 \rangle} = y^{\langle 0 \rangle} \\
[\![S_0; S_1]\!] &\triangleq \exists X. [\![S_0]\!][X^{\langle 1 \rangle} \mapsto X] \wedge [\![S_1]\!][X^{\langle 0 \rangle} \mapsto X] \\
[\![\mathsf{if}\ p\ \mathsf{then}\ S_0\ \mathsf{else}\ S_1]\!] &\triangleq (p^{\langle 0 \rangle} \wedge [\![S_0]\!]) \vee (\neg p^{\langle 0 \rangle} \wedge [\![S_1]\!])
\end{aligned}
$$

Let ν and ν' be valuations, and S a statement. We write $\nu \xrightarrow{S} \nu'$ if $[\![S]\!]$ evaluates to true by assigning $\nu(x)$ and $\nu'(x)$ to $x^{\langle 0 \rangle}$ and $x^{\langle 1 \rangle}$ for each $x \in X$ respectively. Given a sequence of statements $S_1; S_2; \cdots; S_m$, a *program execution* $\nu_0 \xrightarrow{S_1} \nu_1 \xrightarrow{S_2} \cdots \xrightarrow{S_m} \nu_m$ is a sequence $[\nu_0, \nu_1, \ldots, \nu_m]$ of valuations such that $\nu_i \xrightarrow{S_i} \nu_{i+1}$ for $0 \leq i < m$.

A *precondition* $Pre(\theta : S)$ for $\theta \in QF$ with respect to the statement S, which is a first-order formula that entails θ after executing the statement S, is defined as follows.

$$
\begin{aligned}
Pre(\theta : \mathsf{nop}) &\triangleq \theta \\
Pre(\theta : x := \mathsf{nondet}) &\triangleq \forall x. \theta \\
Pre(\theta : x := e) &\triangleq \theta[x \mapsto e] \\
Pre(\theta : S_0;\ S_1) &\triangleq Pre(Pre(\theta : S_1) : S_0) \\
Pre(\theta : \mathsf{if}\ p\ \mathsf{then}\ S_0\ \mathsf{else}\ S_1) &\triangleq (p \Rightarrow Pre(\theta : S_0)) \wedge (\neg p \Rightarrow Pre(\theta : S_1))
\end{aligned}
$$

Observe that all universal quantifiers occur positively in $Pre(\theta : S)$ for any S. They can be eliminated by Skolem constants [10, 17].

2.6. **Problem Definition.** Given an annotated loop,

$$\{\delta\}\ \texttt{while}\ \kappa\ \texttt{do}\ S_1; S_2; \cdots; S_m\ \texttt{done}\ \{\epsilon\},$$

the *loop invariant inference problem* is to compute an invariant $\iota \in QF$ that is a formula satisfying

(1) $\delta \Rightarrow \iota$;
(2) $\iota \wedge \neg\kappa \Rightarrow \epsilon$; and
(3) $\iota \wedge \kappa \Rightarrow Pre(\iota : S_1; S_2; \cdots; S_m)$.

Observe that the condition (2) is equivalent to $\iota \Rightarrow \epsilon \vee \kappa$. The first two conditions specify necessary and sufficient conditions of any loop invariants respectively. The formulae δ and $\epsilon \vee \kappa$ are called the *strongest* and *weakest approximations* to loop invariants respectively.

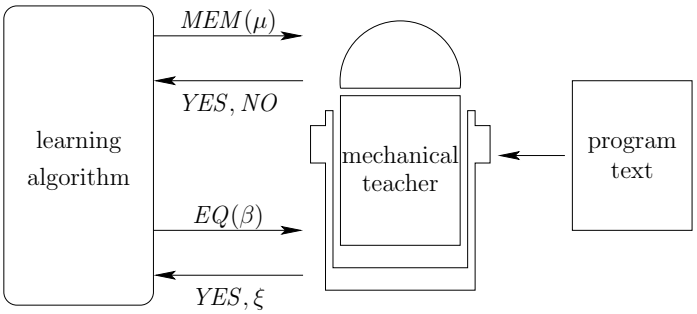

Figure 2. Learning-based Framework

We are particularly interested in the following variant of the loop invariant inference problem:

(a) Given a set P of atomic predicates, finding an invariant $\iota \in QF[P]$; and
(b) Given an annotated loop, finding a suitable set P of atomic predicates that contains enough predicates to express at least one of the invariants.

Jung *et al.* propose a algorithmic-learning-based technique [14] that solves the part (a) of the problem. The technique combines predicate abstraction and decision procedures to make a mechanical teacher that answers the queries from learning algorithm. With predicate abstraction, the learning algorithm becomes an efficient engine for exploring possible combinations of predicates to find an invariant.

In this paper, we address the part (b) of the problem using interpolation. As already stated in Section 2.2, interpolation provides a systematic method for predicate generation and widely adopted in software model checking. We explain the application of interpolation in the context of learning-based loop invariant inference.

3. Inferring Loop Invariants with Algorithmic Learning

In this section, we review the learning-based framework for inferring quantifier-free loop invariant due to Jung *et al.* [14]. Given a set P of atomic predicates, the authors show how to apply a learning algorithm for Boolean formulae to infer quantifier-free loop invariants freely generated by P. They first adopt predicate abstraction to relate quantifier-free and Boolean formulae. They then design a mechanical teacher to guide the learning algorithm to a Boolean formula whose concretization is a loop invariant. We first explain the algorithms for resolving queries from the learning algorithm and then the main loop of learning-based loop invariant inference.

3.1. Answering Queries from Algorithmic Learning.

Figure 2 shows a high-level view of learning-based loop invariant inference framework. In the framework, a learning algorithm is used to drive the search of loop invariants. It "learns" an unknown loop invariant by inquiring a mechanical teacher. The mechanical teacher of course does not know any loop invariant. It nevertheless tries to answer these queries by the information derived from program texts. In this case, the teacher uses approximations to loop invariants. By employing a learning algorithm, it suffices to design a mechanical teacher to find loop invariants.

Moreover, the new framework does not construct abstract models nor compute fixed points. It can be more scalable than traditional techniques.

After formulae in QF and valuations in Val_X are abstracted to those in $Bool[B_P]$ and Val_{B_P} respectively, a learning algorithm is used to infer abstractions of loop invariants. Let ξ be an unknown *target* Boolean formula in $Bool[B_P]$. A learning algorithm computes a representation of the target ξ by interacting with a teacher. The *teacher* should answer the following queries [4]:

- *Membership queries.* Let $\mu \in Val_{B_P}$ be an abstract valuation. The membership query $MEM(\mu)$ asks if the unknown target ξ is satisfied by μ. If so, the teacher answers *YES*; otherwise, *NO*.
- *Equivalence queries.* Let $\beta \in Bool[B_P]$ be an *abstract conjecture*. The equivalence query $EQ(\beta)$ asks if β is equivalent to the unknown target ξ. If so, the teacher answers *YES*. Otherwise, the teacher gives an abstract valuation μ such that the exclusive disjunction of β and ξ is satisfied by μ. The abstract valuation μ is called an *abstract counterexample*.

With predicate abstraction and a learning algorithm for Boolean formulae at hand, it remains to design a mechanical teacher to guide the learning algorithm to the abstraction of a loop invariant. The key idea in [14] is to exploit approximations to loop invariants. An *under-approximation* to loop invariants is a quantifier-free formula $\underline{\iota}$ which is stronger than some loop invariants of the given annotated loop; an *over-approximation* is a quantifier-free formula $\overline{\iota}$ which is weaker than some loop invariants.

In the following, we explain exactly how we can answer queries from learning algorithm using under- and over-approximation of loop invariant.

3.1.1. *Answering Membership Queries.* In the membership query $MEM(\mu)$, the teacher is required to answer whether $\mu \models \alpha(\xi)$. We concretize the Boolean valuation μ and check it against the approximations. If the concretization $\gamma^*(\mu)$ is inconsistent (that is, $\gamma^*(\mu)$ is unsatisfiable), we simply answer *NO* for the membership query. Otherwise, there are three cases:

(1) $\gamma^*(\mu) \Rightarrow \underline{\iota}$. Thus $\mu \models \alpha(\underline{\iota})$ (Lemma 2.5). And $\mu \models \alpha(\iota)$ by Lemma 2.2.
(2) $\gamma^*(\mu) \not\Rightarrow \overline{\iota}$. Thus $\mu \not\models \alpha(\overline{\iota})$ (Lemma 2.5). That is, $\mu \models \neg\alpha(\overline{\iota})$. Since $\iota \rightarrow \overline{\iota}$, we have $\mu \not\models \alpha(\iota)$ by Lemma 2.2.
(3) Otherwise, we cannot determine whether $\mu \models \alpha(\iota)$ by the approximations. In this case, we answer YES or NO randomly.

```
/* ι̲, ῑ :  under- and over-approximations to loop invariants          */
```
Input: a membership query $MEM(\mu)$ with $\mu \in Val_{B_P}$
Output: *YES* or *NO*
$\theta := \gamma^*(\mu)$;
if θ *is inconsistent* **then return** *NO*;
if $\theta \Rightarrow \underline{\iota}$ **then return** *YES*;
if $\nu \models \neg(\theta \Rightarrow \overline{\iota})$ **then return** *NO*;
return *YES or NO randomly*;

Algorithm 1: Membership Query Resolution

Algorithm 1 shows our membership query resolution algorithm. Note that instead of giving a random answer when a membership query cannot be resolved by given invariant approximations, one can give more accurate answer by exploiting better approximations from static analyzers. This learning-based framework is orthogonal to existing static analysis techniques [14].

3.1.2. *Answering Equivalence Queries.* To answer the equivalence query $EQ(\beta)$, we concretize the Boolean formula β and check if $\gamma(\beta)$ is indeed an invariant of the `while` statement for the given pre- and post-conditions. If it is, we are done. Otherwise, we use an SMT solver to find a witness to $\alpha(\xi) \oplus \beta$. There are three cases:

(1) There is a ν such that $\nu \models \neg(\underline{\iota} \Rightarrow \gamma(\beta))$. Then $\nu \models \underline{\iota} \wedge \neg\gamma(\beta)$. By Lemma 2.4 and 2.2, we have $\alpha^*(\nu) \models \alpha(\iota)$ and $\alpha^*(\nu) \models \neg\beta$. Thus, $\alpha^*(\nu) \models \alpha(\xi) \wedge \neg\beta$.
(2) There is a ν such that $\nu \models \neg(\gamma(\beta) \Rightarrow \overline{\iota})$. Then $\nu \models \gamma(\beta) \wedge \neg\overline{\iota}$. By Lemma 2.4, $\alpha^*(\nu) \models \beta$. $\alpha^*(\nu) \models \neg\alpha(\iota)$ by Lemma 2.4 and 2.2. Hence $\alpha^*(\nu) \models \beta \wedge \neg\alpha(\xi)$.
(3) Otherwise, we cannot find a witness to $\alpha(\xi) \oplus \beta$ by the approximations. In this case, we give a random abstract counterexample.

`/*` $\{\delta\}$ `while` κ `do` $S_1; S_2; \cdots; S_m$ `done` $\{\epsilon\}$ `: an annotated loop */`
`/*` $\underline{\iota}, \overline{\iota}$ `: under- and over-approximations to loop invariants */`
Input: an equivalence query $EQ(\beta)$ with $\beta \in Bool[B_P]$
Output: *YES* or an abstract counterexample
$\theta := \gamma(\beta)$;
if $\delta \Rightarrow \theta$ *and* $\theta \Rightarrow \epsilon \vee \kappa$ *and* $\theta \wedge \kappa \Rightarrow Pre(\theta : S_1; S_2; \cdots; S_m)$ **then return** *YES*;
if $\nu \models \neg(\underline{\iota} \Rightarrow \theta)$ *or* $\nu \models \neg(\theta \Rightarrow \overline{\iota})$ *or* $\nu \models \neg(\theta \wedge \kappa \Rightarrow Pre(\overline{\iota} : S_1; S_2; \cdots; S_m))$ **then return** $\alpha^*(\nu)$;
return *a random abstract counterexample*;

Algorithm 2: Equivalence Query Resolution

Algorithm 2 shows our equivalence query resolution algorithm. Note that Algorithm 2 returns *YES* only if an invariant is found.

As in the membership query resolution, we give a random answer when an equivalence query is not resolved by given invariant approximations. We can still refine approximations using some static analysis to give more accurate counterexample.

`/*` $\{\delta\}$ `while` κ `do` $S_1; S_2; \cdots; S_m$ `done` $\{\epsilon\}$ `: an annotated loop */`
Output: a loop invariant for the annotated loop
$\underline{\iota} := \delta \vee \epsilon$;
$\overline{\iota} := \epsilon \vee \kappa$;
repeat
  call a learning algorithm for Boolean formulae where membership and equivalence queries are resolved by Algorithms 1 and 2 respectively;
until *a loop invariant is found* ;

Algorithm 3: Main Loop

3.2. **Main Loop of of Inference Framework.** The main loop of loop invariant inference algorithm is given in Algorithm 3. We heuristically choose $\delta \vee \epsilon$ and $\epsilon \vee \kappa$ as the under- and over-approximations respectively. Note that the under-approximation would be stronger if one uses the strongest approximation δ. It is, however, reported that the weaker approximation $\delta \vee \epsilon$ for the under-approximation is more effective in resolving queries [14]. After determining the approximations, a learning algorithm is used to find an invariant. In [14], Jung *et al.* use CDNF algorithm with Algorithms 1 and 2 for resolving queries.

Note that the mechanical teacher may give conflicting answers. Random answers to membership queries may contradict abstract counterexamples from equivalence queries. Moreover, different valuations may correspond to the same abstract valuation. The learning algorithm cannot infer any loop invariant in the presence of conflicting answers. When the mechanical teacher gives conflicting answers, we restart the learning algorithm and search another loop invariant. In practice, there are nevertheless sufficiently many invariants for an annotated loop. The learning-based technique can infer a loop invariant without incurring any conflicts after a small number of restarts. As an empirical evidence, observe the number of restarts in Table 1. Even without the new predicate generation technique, the numbers of restarts in all but three examples are less than three. The number of restarts is dramatically improved with the new technique since the technique generates predicates incrementally on demand so that it can make the abstraction parsimonious.

We remark that the learning-based loop invariant inference is semi-algorithm; Algorithm 3 terminates with a loop invariant only when there exists one for the loop that can be expressed with the given set of predicates. If there are not enough atomic predicates to express any invariant, the algorithm will iterate indefinitely. For example, `tar` example in Section 6 timed out because it turned out to have no invariant with only atomic predicates from the program text.

4. Predicate Generation by Interpolation

One drawback in the learning-based approach to loop invariant inference is to require a set of atomic predicates. It is essential that at least one quantifier-free loop invariant is representable by the given set P of atomic predicates. Otherwise, concretization of formulae in $Bool[B_P]$ cannot be loop invariants. The mechanical teacher never answers *YES* to equivalence queries. To address this problem, we will synthesize new atomic predicates for the learning-based loop invariant inference framework progressively.

The interpolation is essential to our predicate generation technique. Let $\Theta = [\theta_1, \theta_2, \ldots, \theta_m]$ be an inconsistent sequence of quantifier-free formula and $\Lambda = [\lambda_0, \lambda_1, \lambda_2, \ldots, \lambda_m]$ its inductive interpolant. By definition, $\theta_1 \Rightarrow \lambda_1$. Assume $\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_i \Rightarrow \lambda_i$. We have $\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_{i+1} \Rightarrow \lambda_{i+1}$ since $\lambda_i \wedge \theta_{i+1} \Rightarrow \lambda_{i+1}$. Thus, λ_i is an over-approximation to $\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_i$ for $0 \leq i \leq m$. Moreover, $\sigma(\lambda_i) \subseteq \sigma(\theta_i) \cap \sigma(\theta_{i+1})$. Hence λ_i can be seen as a concise summary of $\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_i$ with restricted symbols. Since each λ_i is written in a less expressive vocabulary, new atomic predicates among variables can be synthesized. We therefore apply the interpolation theorem to synthesize new atomic predicates and refine the abstraction.

Our predicate generation technique consists of three components. Before the learning algorithm is invoked, an initial set of atomic predicates is computed (Section 4.1). When the learning algorithm is failing to infer loop invariants, new atomic predicates are generated to refine the abstraction (Section 4.2). Lastly, conflicting answers to queries may incur from

predicate abstraction. We further refine the abstraction with these conflicting answers (Section 4.3). Throughout this section, we consider the annotated loop $\{\delta\}$ `while` κ `do` $S_1; S_2; \cdots ; S_m$ `done` $\{\epsilon\}$ with the under-approximation $\underline{\iota}$ and over-approximation $\overline{\iota}$.

4.1. **Initial Atomic Predicates.** The under- and over-approximations to loop invariants must satisfy $\underline{\iota} \Rightarrow \overline{\iota}$. Otherwise, there cannot be any loop invariant ι such that $\underline{\iota} \Rightarrow \iota$ and $\iota \Rightarrow \overline{\iota}$. Thus, the sequence $[\underline{\iota}, \neg\overline{\iota}]$ is inconsistent. For any interpolant $[T, \lambda, F]$ of $[\underline{\iota}, \neg\overline{\iota}]$, we have $\underline{\iota} \Rightarrow \lambda$ and $\lambda \Rightarrow \overline{\iota}$. The quantifier-free formula λ can be a loop invariant if it satisfies $\lambda \wedge \kappa \Rightarrow \mathit{Pre}(\lambda : S_1; S_2; \cdots ; S_m)$. It is however unlikely that λ happens to be a loop invariant. Yet our loop invariant inference algorithm can generalize λ by taking the atomic predicates in λ as the initial atomic predicates. The learning algorithm will try to infer a loop invariant freely generated by these atomic predicates.

4.2. **Atomic Predicates from Incorrect Conjectures.** Consider an equivalence query $EQ(\beta)$ where $\beta \in \mathit{Bool}[B_P]$ is an abstract conjecture. If the concretization $\theta = \gamma(\beta)$ is not a loop invariant, we interpolate the loop body with the incorrect conjecture θ. For any quantifier-free formula θ over variables $X^{\langle 0\rangle} \cup X^{\langle 1\rangle}$, define $\theta^{\langle k\rangle} = \theta[X^{\langle 0\rangle} \mapsto X^{\langle k\rangle}, X^{\langle 1\rangle} \mapsto X^{\langle k+1\rangle}]$. The *desuperscripted* form of a quantifier-free formula λ over variables $X^{\langle k\rangle}$ is $\lambda[X^{\langle k\rangle} \mapsto X]$. Moreover, if ν is a valuation over $X^{\langle 0\rangle} \cup \cdots \cup X^{\langle m\rangle}$, $\nu{\downarrow_{X^{\langle k\rangle}}}$ represents a valuation over X such that $\nu{\downarrow_{X^{\langle k\rangle}}}(x) = \nu(x^{\langle k\rangle})$ for $x \in X$. Let ϕ and ψ be quantifier-free formulae over X. Define the following sequence:

$$\Xi(\phi, S_1, \ldots, S_m, \psi) \stackrel{\triangle}{=} [\phi^{\langle 0\rangle}, [\![S_1]\!]^{\langle 0\rangle}, [\![S_2]\!]^{\langle 1\rangle}, \ldots, [\![S_m]\!]^{\langle m-1\rangle}, \neg\psi^{\langle m\rangle}].$$

Observe that

- $\phi^{\langle 0\rangle}$ and $[\![S_1]\!]^{\langle 0\rangle}$ share the variables $X^{\langle 0\rangle}$;
- $[\![S_m]\!]^{\langle m-1\rangle}$ and $\neg\psi^{\langle m\rangle}$ share the variables $X^{\langle m\rangle}$; and
- $[\![S_i]\!]^{\langle i-1\rangle}$ and $[\![S_{i+1}]\!]^{\langle i\rangle}$ share the variables $X^{\langle i\rangle}$ for $1 \leq i < m$.

Starting from the program states satisfying $\phi^{\langle 0\rangle}$, the formula

$$\phi^{\langle 0\rangle} \wedge [\![S_1]\!]^{\langle 0\rangle} \wedge [\![S_2]\!]^{\langle 1\rangle} \wedge \cdots \wedge [\![S_i]\!]^{\langle i-1\rangle}$$

characterizes the images of $\phi^{\langle 0\rangle}$ during the execution of $S_1; S_2; \cdots ; S_i$.

Lemma 4.1. *Let X denote the set of variables in the statement $S_1; S_2; \cdots ; S_i$, and ϕ a quantifier-free formula over X. For any valuation ν over $X^{\langle 0\rangle} \cup X^{\langle 1\rangle} \cup \cdots \cup X^{\langle i\rangle}$, the formula $\phi^{\langle 0\rangle} \wedge [\![S_1]\!]^{\langle 0\rangle} \wedge [\![S_2]\!]^{\langle 1\rangle} \wedge \cdots \wedge [\![S_i]\!]^{\langle i-1\rangle}$ is satisfied by ν if and only if $\nu{\downarrow_{X^{\langle 0\rangle}}} \xrightarrow{S_1} \nu{\downarrow_{X^{\langle 1\rangle}}} \xrightarrow{S_2} \cdots \xrightarrow{S_i} \nu{\downarrow_{X^{\langle i\rangle}}}$ is a program execution and $\nu{\downarrow_{X^{\langle 0\rangle}}} \models \phi$.*

Proof. By induction on the length of statement $S_1; S_2; \cdots ; S_i$. Suppose that the lemma is true for statement $S_1; S_2; \cdots ; S_i$. By definition of program execution, if $\nu{\downarrow_{X^{\langle i\rangle}}} \xrightarrow{S_{i+1}} \nu{\downarrow_{X^{\langle i+1\rangle}}}$, then ν satisfies $[\![S_{i+1}]\!]^{\langle i\rangle}$ and vice versa. By induction hypothesis, the formula $\phi^{\langle 0\rangle} \wedge [\![S_1]\!]^{\langle 0\rangle} \wedge [\![S_2]\!]^{\langle 1\rangle} \wedge \cdots \wedge [\![S_{i+1}]\!]^{\langle i\rangle}$ is satisfied by ν and the statement follows by it. □

By definition, $\phi \Rightarrow \mathit{Pre}(\psi : S_1; S_2; \cdots ; S_m)$ implies that the image of ϕ must satisfy ψ after the execution of $S_1; S_2; \cdots ; S_m$. The sequence $\Xi(\phi, S_1, \ldots, S_m, \psi)$ is inconsistent if $\phi \Rightarrow \mathit{Pre}(\psi : S_1; S_2; \cdots ; S_m)$. The following proposition will be handy.

Proposition 4.2. *Let $S_1; S_2; \cdots ; S_m$ be a sequence of statements. For any ϕ with $\phi \Rightarrow \mathit{Pre}(\psi : S_1; S_2; \cdots ; S_m)$, $\Xi(\phi, S_1, \ldots, S_m, \psi)$ has an inductive interpolant.*

Proof. By induction on the length of statement $S_1; S_2; \cdots ; S_m$. Suppose the proposition holds for statement $S_2; \cdots ; S_m$ and an arbitrary formula ϕ with $\phi \Rightarrow \mathit{Pre}(\psi : S_1; S_2; \cdots ; S_m)$. By definition of *Pre*, $\mathit{Pre}(\psi : S_1; S_2; \cdots ; S_m) = \mathit{Pre}(\mathit{Pre}(\psi : S_2; \cdots ; S_m) : S_1)$ Let ϕ' be a formula such that ϕ satisfies ϕ' after execution of S_1. By induction hypothesis, $\Xi(\phi', S_2, \ldots, S_m, \psi)$ has an inductive interpolant. Thus, $\Xi(\phi, S_1, \ldots, S_m, \psi)$ also has inductive interpolant.

□

Let $\Lambda = [T, \lambda_1, \lambda_2, \ldots, \lambda_{m+1}, F]$ be an inductive interpolant of $\Xi(\phi, S_1, \ldots, S_m, \psi)$. Recall that λ_i is a quantifier-free formula over $X^{\langle i-1 \rangle}$ for $1 \leq i \leq m+1$. It is also an over-approximation to the image of ϕ after executing $S_1; S_2; \cdots ; S_{i-1}$. Proposition 4.2 can be used to generate new atomic predicates. One simply finds a pair of quantifier-free formulae ϕ and ψ with $\phi \Rightarrow \mathit{Pre}(\psi : S_1; S_2; \cdots ; S_m)$, applies the interpolation theorem, and collects desuperscripted atomic predicates in an inductive interpolant of $\Xi(\phi, S_1, \ldots, S_m, \psi)$. In the following, we show how to obtain such pairs with under- and over-approximations to loop invariants.

4.2.1. *Interpolating Over-Approximation.* It is not hard to see that an over-approximation to loop invariants characterizes loop invariants after the execution of the loop body. Recall that $\iota \Rightarrow \bar{\iota}$ for some loop invariant ι. Moreover, $\iota \wedge \kappa \Rightarrow \mathit{Pre}(\iota : S_1; S_2; \cdots ; S_m)$. By the monotonicity of $\mathit{Pre}(\bullet : S_1; S_2; \cdots ; S_m)$, we have $\iota \wedge \kappa \Rightarrow \mathit{Pre}(\bar{\iota} : S_1; S_2; \cdots ; S_m)$.

Proposition 4.3. *Let $\bar{\iota}$ be an over-approximation to loop invariants of the annotated loop $\{\delta\}$ `while` κ `do` $S_1; S_2; \cdots ; S_m$ `done` $\{\epsilon\}$. For any loop invariant ι with $\iota \Rightarrow \bar{\iota}$, $\iota \wedge \kappa \Rightarrow \mathit{Pre}(\bar{\iota} : S_1; S_2; \cdots ; S_m)$.*

Proof. Since ι is a loop invariant, $\iota \wedge \kappa \Rightarrow \mathit{Pre}(\iota : S)$. The statement follows by the monotonicity of $\mathit{Pre}(\bullet : S)$. □

Proposition 4.3 gives a necessary condition to loop invariants of interest. Recall that $\theta = \gamma(\beta)$ is an incorrect conjecture of loop invariants. If $\nu \models \neg(\theta \wedge \kappa \Rightarrow \mathit{Pre}(\bar{\iota} : S_1; S_2; \cdots ; S_m))$, the mechanical teacher returns the abstract counterexample $\alpha^*(\nu)$. Otherwise, Proposition 4.2 is applicable with the pair $\theta \wedge \kappa$ and $\bar{\iota}$.

Corollary 4.4. *Let $\bar{\iota}$ be an over-approximation to loop invariants of the annotated loop $\{\delta\}$ `while` κ `do` $S_1; S_2; \cdots ; S_m$ `done` $\{\epsilon\}$. For any θ with $\theta \wedge \kappa \Rightarrow \mathit{Pre}(\bar{\iota} : S_1; S_2; \cdots ; S_m)$, the sequence $\Xi(\theta \wedge \kappa, S_1, S_2, \ldots, S_m, \bar{\iota})$ has an inductive interpolant.*

Proof. By Proposition 4.2. □

4.2.2. *Interpolating Under-Approximation.* For under-approximations, there is no necessary condition. Nevertheless, Proposition 4.2 is applicable with the pair $\underline{\iota} \wedge \kappa$ and θ.

Corollary 4.5. *Let $\underline{\iota}$ be an under-approximation to loop invariants of the annotated loop $\{\delta\}$ `while` κ `do` $S_1; S_2; \cdots; S_m$ `done` $\{\epsilon\}$. For any θ with $\underline{\iota} \wedge \kappa \Rightarrow \mathit{Pre}(\theta : S_1; S_2; \cdots; S_m)$, the sequence $\Xi(\underline{\iota} \wedge \kappa, S_1, S_2, \ldots, S_m, \theta)$ has an inductive interpolant.*

Proof. By Proposition 4.2. □

Generating atomic predicates from an incorrect conjecture θ should now be clear (Algorithm 4). Assuming that the incorrect conjecture satisfies the necessary condition in Proposition 4.3, we simply collect all desuperscripted atomic predicates in an inductive interpolant of $\Xi(\theta \wedge \kappa, S_1, S_2, \ldots, S_m, \bar{\iota})$ (Corollary 4.4). More atomic predicates can be obtained from an inductive interpolant of $\Xi(\underline{\iota} \wedge \kappa, S_1, S_2, \ldots, S_m, \theta)$ if additionally $\underline{\iota} \wedge \kappa \Rightarrow \mathit{Pre}(\theta : S_1; S_2; \cdots; S_m)$ (Corollary 4.5).

```
/* {δ} while κ do S1;⋯;Sm done {ϵ} :  an annotated loop          */
/* ι̲, ῑ :  under- and over-approximations to loop invariants     */
Input: a formula θ ∈ QF[P] such that θ ∧ κ ⇒ Pre(ῑ : S1; S2; ⋯; Sm)
Output: a set of atomic predicates
I := an inductive interpolant of Ξ(θ ∧ κ, S1, S2, ..., Sm, ῑ);
Q := desuperscripted atomic predicates in I;
if ι̲ ∧ κ ⇒ Pre(θ : S1; S2; ⋯; Sm) then
    J := an inductive interpolants of Ξ(ι̲ ∧ κ, S1, S2, ..., Sm, θ);
    R := desuperscripted atomic predicates in J;
    Q := Q ∪ R;
end
return Q
```

Algorithm 4: `PredicatesFromConjecture`(θ)

4.3. **Atomic Predicates from Conflicting Abstract Counterexamples.** Because of the abstraction, conflicting abstract counterexamples may be given to the learning algorithm. Consider the example in Section 1. Recall that $n \geq 0 \wedge x = n \wedge y = n$ and $x + y = 0 \vee x > 0$ are the under- and over-approximations respectively. Suppose there is only one atomic predicate $y = 0$. The learning algorithm tries to infer a Boolean formula $\lambda \in \mathit{Bool}[b_{y=0}]$. Let us resolve the equivalence queries $EQ(T)$ and $EQ(F)$. On the equivalence query $EQ(F)$, we check if F is weaker than the under-approximation by an SMT solver. It is not, and the SMT solver gives the valuation $\nu_0(n) = \nu_0(x) = \nu_0(y) = 1$ as a witness. Applying the abstraction function α^* to ν_0, the mechanical teacher returns the abstract counterexample $b_{y=0} \mapsto F$. The abstract counterexample is intended to notify that the target formula λ and F have different truth values when $b_{y=0}$ is F. That is, λ is satisfied by the valuation $b_{y=0} \mapsto F$.

On the equivalence query $EQ(T)$, the mechanical teacher checks if T is stronger than the over-approximation. It is not, and the SMT solver now returns the valuation $\nu_1(x) = 0, \nu_1(y) = 1$ as a witness. The mechanical teacher in turn computes $b_{y=0} \mapsto F$ as the corresponding abstract counterexample. The abstract counterexample notifies that the target formula λ and T have different truth values when $b_{y=0}$ is F. That is, λ is not

satisfied by the valuation $b_{y=0} \mapsto F$. Yet the target formula λ cannot be satisfied and unsatisfied by the valuation $b_{y=0} \mapsto F$. We have conflicting abstract counterexamples.

Such conflicting abstract counterexamples arise because the abstraction is too coarse. This gives us another chance to refine the abstraction. For distinct valuations ν and ν', $\Gamma(\nu) \wedge \Gamma(\nu')$ is inconsistent. For instance, $\Gamma(\nu_0) = (n = 1) \wedge (x = 1) \wedge (y = 1)$, $\Gamma(\nu_1) = (x = 0) \wedge (y = 1)$, and $\Gamma(\nu_1) \wedge \Gamma(\nu_0)$ is inconsistent.

`/*` $\{\delta\}$ `while` κ `do` $S_1; S_2; \cdots; S_m$ `done` $\{\epsilon\}$ `: an annotated loop */`
Input: distinct valuations ν and ν' such that $\alpha^*(\nu) = \alpha^*(\nu')$
Output: a set of atomic predicates
$X := \Gamma(\nu)$;
$X' := \Gamma(\nu')$;
`/*` $X \wedge X'$ `is inconsistent */`
$\rho := \gamma^*(\alpha^*(\nu))$;
$Q :=$ atomic predicates in an inductive interpolant of $[X, X' \vee \neg\rho]$;
return Q;

Algorithm 5: `PredicatesFromConflict`(ν, ν')

Algorithm 5 generates atomic predicates from conflicting abstract counterexamples. Let ν and ν' be distinct valuations in Val_X. We compute formulae $X = \Gamma(\nu)$ and $X' = \Gamma(\nu')$. Since ν and ν' are conflicting, they correspond to the same abstract valuation $\alpha^*(\nu) = \alpha^*(\nu')$. Let $\rho = \gamma^*(\alpha^*(\nu))$. We have $X \Rightarrow \rho$ and $X' \Rightarrow \rho$ [14]. Recall that $X \wedge X'$ is inconsistent. $[X, X' \vee \neg\rho]$ is also inconsistent for $X \Rightarrow \rho$. Algorithm 5 returns atomic predicates in an inductive interpolant of $[X, X' \vee \neg\rho]$.

5. Loop Invariant Inference Algorithms with Predicate Generation

Algorithm 6 is the main loop of inference framework with predicate generation. The algorithm is the same as Algorithm 3 except the gray-boxed parts.

We first compute the initial set of atomic predicates by interpolating $\underline{\iota}$ and $\neg\bar{\iota}$ (Section 4.1). With the initial set, we start the learning process until the algorithm finds a loop invariant or there is an exception raised. Exceptions basically mean that the current set of predicates might not be enough to find a loop invariant. We need in this case to find more predicates using one of the algorithms explained in Section 4.

The learning algorithm finds conflicting abstract counterexamples when the equivalence query resolution algorithm gives a random counterexample that contradicts the previous ones or the current predicate abstraction is too coarse. Since we cannot distinguish the two, we always generate more predicates using Algorithm 5, hoping that we can find a loop invariant in the next iteration.

The `ExcessiveRandomAnswers` exception is raised when our new equivalence query resolution algorithm, which is detailed later, suspects that it generates too many random counterexamples because of the coarse predicate abstraction. In this case, we generate more predicates using Algorithm 4.

Note that we start the learning algorithm from the scratch every time we generate more predicates. The reason is because we use CDNF algorithm for learning that handles only a fixed number of Boolean variables. Recently, Chen *et al.* [5] propose a variant of CDNF

/* CEX : a set of counterexamples */
/* τ : a threshold to generate new atomic predicates */
/* $\{\delta\}$ while κ do $S_1; S_2; \cdots; S_m$ done $\{\epsilon\}$: an annotated loop */
Output: a loop invariant for the annotated loop
$\underline{\iota} := \delta \vee \epsilon$;
$\overline{\iota} := \epsilon \vee \kappa$;
$P :=$ InitialAtomicPredicates()
repeat
  try
    call a learning algorithm for Boolean formulae where membership and equivalence queries are resolved by Algorithms 1 and 7 respectively;
  catch ConflictAbstractCEX $\rightarrow$
    find distinct valuations ν and ν' in CEX such that $\alpha^*(\nu) = \alpha^*(\nu')$;
    $P := P \cup$ PredicatesFromConflict(ν, ν');
  catch ExcessiveRandomAnswers$(\theta) \rightarrow$
    $P := P \cup$ PredicatesFromConjecture(θ);
  $\tau := \lceil 1.3^{|P|} \rceil$;
until *a loop invariant is found* ;

Algorithm 6: Main Loop with Predicate Generation

algorithm that supports incremental learning. We can also adopt this algorithm to improve the efficiency of the overall technique.

/* CEX : a set of counterexamples */
/* τ : a threshold to generate new atomic predicates */
/* $\{\delta\}$ while κ do $S_1; S_2; \cdots; S_m$ done $\{\epsilon\}$: an annotated loop */
/* $\underline{\iota}, \overline{\iota}$: under- and over-approximations to loop invariants */
Input: an equivalence query $EQ(\beta)$ with $\beta \in Bool[B_P]$
Output: *YES* or an abstract counterexample
$\theta := \gamma(\beta)$;
if $\delta \Rightarrow \theta$ *and* $\theta \Rightarrow \epsilon \vee \kappa$ *and* $\theta \wedge \kappa \Rightarrow Pre(\theta : S_1; S_2; \cdots; S_m)$ **then return** *YES*;
if $\nu \models \neg(\underline{\iota} \Rightarrow \theta)$ *or* $\nu \models \neg(\theta \Rightarrow \overline{\iota})$ *or* $\nu \models \neg(\theta \wedge \kappa \Rightarrow Pre(\overline{\iota} : S_1; S_2; \cdots; S_m))$ **then**
  $CEX := CEX \cup \{\nu\}$; **return** $\alpha^*(\nu)$;
if the number of random abstract counterexamples $\leq \tau$ **then**
  return *a random abstract counterexample*;
else
  throw ExcessiveRandomAnswers(θ);

Algorithm 7: Equivalence Query Resolution with Predicate Generation

The equivalence query resolution algorithm is given in Algorithm 7. Again, we put gray-boxes to denote the modified parts. As Algorithm 2, the mechanical teacher first checks if the concretization of the abstract conjecture is a loop invariant. If so, it returns *YES*

Table 1. Experimental Results.
P : # of atomic predicates, MEM : # of membership queries, EQ : # of equivalence queries, RE : # of the learning algorithm restarts, T : total elapsed time (s).

case	$SIZE$	Previous [14]					Current					BLAST [20]	
		P	MEM	EQ	RE	T	P	MEM	EQ	RE	T	P	T
`ide-ide-tape`	16	6	13	7	1	0.05	4	6	5	1	0.05	21	1.31(1.07)
`ide-wait-ireason`	9	5	790	445	33	1.51	5	122	91	7	1.09	9	0.19(0.14)
`parser`	37	17	4,223	616	13	13.45	9	86	32	1	0.46	8	0.74(0.49)
`riva`	82	20	59	11	2	0.51	7	14	5	1	0.37	12	1.50(1.17)
`tar`	7	6	∞	∞	∞	∞	2	2	5	1	0.02	10	0.20(0.17)
`usb-message`	18	10	21	7	1	0.10	3	7	6	1	0.04	4	0.18(0.14)
`vpr`	8	5	16	9	2	0.05	1	1	3	1	0.01	4	0.13(0.10)

and concludes the loop invariant inference algorithm. Otherwise, the mechanical teacher compares the concretization of the abstract conjecture with approximations to loop invariants. If the concretization is stronger than the under-approximation, weaker than the over-approximation, or it does not satisfy the necessary condition given in Proposition 4.3, an abstract counterexample is returned after recording the witness valuation [14, 16]. The witnessing valuations are needed to synthesize atomic predicates in Algorithm 6 when conflicts occur.

If the concretization is not a loop invariant and falls between both approximations to loop invariants, there are two possibilities. The current set of atomic predicates is sufficient to express a loop invariant; the learning algorithm just needs a few more iterations to infer a solution. Or, the current atomic predicates are insufficient to express any loop invariant; the learning algorithm cannot derive a solution with these predicates. Since we cannot tell which scenario arises, a threshold is deployed heuristically. If the number of random abstract counterexamples is less than the threshold, we give the learning algorithm more time to find a loop invariant. Only when the number of random abstract counterexamples exceeds the threshold, can we synthesize more atomic predicates for abstraction refinement. Intuitively, the current atomic predicates are likely to be insufficient if lots of random abstract counterexamples have been generated. In this case, we raise `ExcessiveRandomAnswers` exception to synthesize more atomic predicates from the incorrect conjecture in Algorithm 6. Observe that in Algorithm 6, threshold τ is set to $\lceil 1.3^{|P|} \rceil$, the approximate size of the search space, which we found empirically.

6. Experimental Results

We have implemented the proposed technique in OCaml. In our implementation, the SMT solver Yices and the interpolating theorem prover CSIsat [1] are used for query resolution and interpolation respectively. In addition to the examples in [14], we add two more examples: `riva` is the largest loop expressible in our simple language from Linux[1], and `tar` is extracted from Tar[2]. All examples are translated into annotated loops manually. Data are the average of 100 runs and collected on a 2.4GHz Intel Core2 Quad CPU with 8GB memory running Linux 2.6.31 (Table 1).

[1]In Linux 2.6.30 `drivers/video/riva/riva_hw.c:nv10CalcArbitration()`

[2]In Tar 1.13 `src/mangle.c:extract_mangle()`

```
{ size = M ∧ copy = N }
1 while size > 0 do
2     available := nondet;
3     if available > size then
4         copy := copy + available;
5         size := size − available;
6 done
{ size = 0 ⟹ copy = M + N }
```

Figure 3. A Sample Loop in Tar

In the table, the column PREVIOUS represents the work in [14] where atomic predicates are chosen heuristically. Specifically, all atomic predicates in pre- and post-conditions, loop guards, and conditions of `if` statements are selected. The column CURRENT gives the results for our automatic predicate generation technique. Interestingly, heuristically chosen atomic predicates suffice to infer loop invariants for all examples except `tar`. For the `tar` example, the learning-based loop invariant inference algorithm fails to find a loop invariant due to ill-chosen atomic predicates. In contrast, our new algorithm is able to infer a loop invariant for the `tar` example in 0.02s. The number of atomic predicates can be significantly reduced as well. Thanks to a smaller number of atomic predicates, loop invariant inference becomes more economical in these examples. Without predicate generation, four of the six examples take more than one second. Only one of these examples takes more than one second using the new technique. Particularly, the `parser` example is improved in orders of magnitude.

The column BLAST gives the results of lazy abstraction technique with interpolants implemented in BLAST [20]. In addition to the total elapsed time, we also show the preprocessing time in parentheses. Since the learning-based framework does not construct abstract models, our new technique outperforms BLAST in all cases but one (`ide-wait-ireason`). If we disregard the time for preprocessing in BLAST, the learning-based technique still wins three cases (`ide-ide-tape`, `tar`, `vpr`) and ties one (`usb-message`). Also note that the number of atomic predicates generated by the new technique is always smaller except `parser`. Given the simplicity of the learning-based framework, our preliminary experimental results suggest a promising outlook for further optimizations.

6.1. **`tar` from Tar.** This simple fragment is excerpted from the code for copying two buffers. M items in the source buffer are copied to the target buffer that already has N items. The variable *size* keeps the number of remaining items in the source buffer and *copy* denotes the number of items in the target buffer after the last copy. In each iteration, an arbitrary number of items are copied and the values of *size* and *copy* are updated accordingly.

Observe that the atomic predicates in the program text cannot express any loop invariant that proves the specification. However, our new algorithm successfully finds the following loop invariant in this example:

$$M + N \leq copy + size \wedge copy + size \leq M + N$$

The loop invariant asserts that the number of items in both buffers is equal to $M + N$. It requires atomic predicates unavailable from the program text. Predicate generation is essential to find loop invariants for such tricky loops.

6.2. `parser` **from SPEC2000 Benchmarks.** For the `parser` example (Figure 4), 9 atomic predicates are generated. These atomic predicates are a subset of the 17 atomic predicates from the program text. Every loop invariant found by the loop invariant inference algorithm contains all 9 atomic predicates. This suggests that there are no redundant predicates. Few atomic predicates make loop invariants easier to comprehend. For instance, the following loop invariant summarizes the condition when *success* or *give_up* is true:

$$
\begin{array}{l}
(\mathit{success} \vee \mathit{give\_up}) \Rightarrow \\
\quad (\mathit{valid} \neq 0 \vee \mathit{cutoff} = \mathit{maxcost} \vee \mathit{words} < \mathit{count}) \wedge \\
\quad (\neg \mathit{search} \vee \mathit{valid} \neq 0 \vee \mathit{words} < \mathit{count}) \wedge \\
\quad (\mathit{linkages} = \mathit{canonical} \wedge \mathit{linkages} \geq \mathit{valid} \wedge \mathit{linkages} \leq 5000)
\end{array}
$$

The invariant is simpler and thus easier to understand than the one presented in [14]. The right side of the implication summarizes the condition when *success* or *give_up* becomes true.

Fewer atomic predicates also lead to a smaller standard deviation of the execution time. The execution time now ranges from 0.36s to 0.58s with the standard deviation

```
{ phase = F ∧ success = F ∧ give_up = F ∧ cutoff = 0 ∧ count = 0 }
 1 while ¬(success ∨ give_up) do
 2     entered_phase := F;
 3     if ¬phase then
 4         if cutoff = 0 then cutoff := 1;
 5         else if cutoff = 1 ∧ maxcost > 1 then cutoff := maxcost;
 6                 else phase := T; entered_phase := T; cutoff := 1000;
 7         if cutoff = maxcost ∧ ¬search then give_up := T;
 8     else
 9         count := count + 1;
10         if count > words then give_up := T;
11     if entered_phase then count := 1;
12     linkages := nondet;
13     if linkages > 5000 then linkages := 5000;
14     canonical := 0; valid := 0;
15     if linkages ≠ 0 then
16         valid := nondet;
17         assume 0 ≤ valid ∧ valid ≤ linkages;
18         canonical := linkages;
19     if valid > 0 then success := T;
20 done
{ (valid > 0 ∨ count > words ∨ (cutoff = maxcost ∧ ¬search))∧
  valid ≤ linkages ∧ canonical = linkages ∧ linkages ≤ 5000 }
```

Figure 4. A Sample Loop in SPEC2000 Benchmark PARSER

equal to 0.06. In contrast, the execution time for [14] ranges from 1.20s to 80.20s with the standard deviation equal to 14.09. By Chebyshev's inequality, the new algorithm infers a loop invariant in one second with probability greater than 0.988. With a compact set of atomic predicates, loop invariant inference algorithm performs rather predictably.

```
{ retries = 100 ∧ (¬ireason_has_ATAPI_COD ∨ ireason_has_ATAPI_IO) }
1 while retries ≠ 0 ∧ (¬ireason_has_ATAPI_COD ∨ ireason_has_ATAPI_IO) do
2     retries := retries − 1;
3     ireason_has_ATAPI_COD := nondet;
4     ireason_has_ATAPI_IO := nondet;
5     if retries = 0 then
6         ireason_has_ATAPI_COD := T;
7         ireason_has_ATAPI_IO := F;
8 done
{ retries < 100 ∧ ireason_has_ATAPI_COD ∧ ¬ireason_has_ATAPI_IO }
```

Figure 5. A Sample Loop in Linux IDE Driver

6.3. `ide-wait-ireason` **from Linux Device Driver.** In the `ide-wait-ireason` example (Figure 5), predicate generation performs better even though it generates the same number of atomic predicates. This is because the technique can synthesize the atomic predicate $retries \leq 100$ which does not appear in the program text but is essential to loop invariants. Surely this atomic predicate is expressible by the two atomic predicates $retries = 100$ and $retries < 100$ from the program text. However the search space is significantly reduced with the more succinct atomic predicate $retries \leq 100$. Subsequently, the learning algorithm only needs a quarter of queries to infer a loop invariant.

7. Conclusions

A predicate generation technique for learning-based loop invariant inference was presented. The technique applies the interpolation theorem to synthesize atomic predicates implicitly implied by program texts. To compare the efficiency of the new technique, examples excerpted from Linux, SPEC2000, and Tar source codes were reported. The learning-based loop invariant inference algorithm is more effective and performs much better in these realistic examples.

More experiments are always needed. Especially, we would like to have more realistic examples which require implicit predicates unavailable in program texts. Additionally, loops manipulating arrays often require quantified loop invariants with linear inequalities. Extension to quantified loop invariants is also important.

References

[1] Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In Gupta, A., Malik, S., eds.: CAV. Volume 5123 of Lecture Notes in Computer Science., Springer (2008) 304–308

[2] Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: Beyond quantifier-free interpolation in extensions of presburger arithmetic. In Jhala, R., Schmidt, D.A., eds.: VMCAI. Volume 6538 of Lecture Notes in Computer Science., Springer (2011) 88–102

[3] Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The mathsat 4smt solver. In Gupta, A., Malik, S., eds.: CAV. Volume 5123 of Lecture Notes in Computer Science., Springer (2008) 299–303

[4] Bshouty, N.H.: Exact learning boolean function via the monotone theory. Inf. Comput. **123**(1) (1995) 146–153

[5] Chen, Y.F., Wang, B.Y.: Learning boolean functions incrementally. In Madhusudan, P., Seshia, S.A., eds.: CAV. Volume 7358 of Lecture Notes in Computer Science., Springer (2012) 55–70

[6] Craig, W.: Linear reasoning. a new form of the herbrand-gentzen theorem. J. Symb. Log. **22**(3) (1957) 250–268

[7] Dutertre, B., de Moura, L.: The Yices SMT solver. Technical report, SRI International (2006)

[8] Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with craig interpolation and symbolic pushdown systems. In Hermanns, H., Palsberg, J., eds.: TACAS. Volume 3920 of Lecture Notes in Computer Science., Springer (2006) 489–503

[9] Filliâtre, J.C., Marché, C.: Multi-prover verification of c programs. In Davies, J., Schulte, W., Barnett, M., eds.: ICFEM. Volume 3308 of Lecture Notes in Computer Science., Springer (2004) 15–29

[10] Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In Launchbury, J., Mitchell, J.C., eds.: POPL, ACM (2002) 191–202

[11] Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In Jones, N.D., Leroy, X., eds.: POPL, ACM (2004) 232–244

[12] Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In Hermanns, H., Palsberg, J., eds.: TACAS. Volume 3920 of Lecture Notes in Computer Science., Springer (2006) 459–473

[13] Jhala, R., McMillan, K.L.: Array abstractions from proofs. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of Lecture Notes in Computer Science., Springer (2007) 193–206

[14] Jung, Y., Kong, S., Wang, B.Y., Yi, K.: Deriving invariants by algorithmic learning, decision procedures, and predicate abstraction. In Barthe, G., Hermenegildo, M.V., eds.: VMCAI. Volume 5944 of Lecture Notes in Computer Science., Springer (2010) 180–196

[15] Jung, Y., Lee, W., Wang, B.Y., Yi, K.: Predicate generation for learning-based quantifier-free loop invariant inference. In Abdulla, P.A., Leino, K.R.M., eds.: TACAS. Volume 6605 of Lecture Notes in Computer Science., Springer (2011) 205–219

[16] Kong, S., Jung, Y., David, C., Wang, B.Y., Yi, K.: Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In Ueda, K., ed.: APLAS. Volume 6461 of Lecture Notes in Computer Science., Springer (2010) 328–343

[17] Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In Steffen, B., Levi, G., eds.: VMCAI. Volume 2937 of Lecture Notes in Computer Science., Springer (2004) 267–281

[18] Lee, W., Wang, B.Y., Yi, K.: Termination analysis with algorithmic learning. In Madhusudan, P., Seshia, S.A., eds.: CAV. Volume 7358 of Lecture Notes in Computer Science., Springer (2012) 88–104

[19] McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. **345**(1) (2005) 101–121

[20] McMillan, K.L.: Lazy abstraction with interpolants. In Ball, T., Jones, R.B., eds.: CAV. Volume 4144 of Lecture Notes in Computer Science., Springer (2006) 123–136

[21] McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In Ramakrishnan, C.R., Rehof, J., eds.: TACAS. Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 413–427

[22] Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In Hind, M., Diwan, A., eds.: PLDI, ACM (2009) 223–234