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

YUNGBUM JUNG, WONCHAN LEE, BOW-YAW WANG, AND KWANGKUEN YI

Seoul National University, Korea
e-mail address: dreameye@ropas.snu.ac.kr

Seoul National University, Korea
e-mail address: wclee@ropas.snu.ac.kr

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

Seoul National University, Korea
e-mail address: kwang@ropas.snu.ac.kr

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 in [15]. Experiments excerpted from Linux, SPEC2000, and Tar source codes are reported.

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 [5, 10].

Finding loop invariants is tedious and sometimes requires intelligence. Recently, an automated technique based on algorithmic learning and predicate abstraction is proposed [15]. Given a fixed set of atomic predicates and an annotated loop, the learning-based technique

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 [16].

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 2010-0001717), National Science Council of Taiwan Grant Numbers 97-2221-E-001-003-MY3 and 97-2221-E-001-006-MY3, the FORMES Project within LIAMA Consortium, and the French ANR project SIVES ANR-08-BLAN-0326-01.

LOGICAL METHODS
IN COMPUTER SCIENCE

DOI:10.2168/LMCS-???

can infer a quantifier-free loop invariant generated by the given atomic predicates. By employing a learning algorithm and a mechanical teacher, the new technique is able to generate loop invariants without constructing abstract models nor computing fixed points. It gives a new invariant generation framework that can be less sensitive to the number of atomic predicates than traditional techniques.

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 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 [13, 20, 21, 9]. 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, interpolants have been used to synthesize atomic predicates for predicate abstraction refinement in software model checking [9, 12, 13, 20, 21].

Inspired by the refinement technique in software model checking, we develop an interpolation-based technique to synthesize atomic predicates in the context of 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. Our technique can improve the effectiveness and efficiency of the learning-based loop invariant inference algorithm in [15]. Constructing the set of atomic predicates can be fully automatic and on-demand.

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

```
{ $n \geq 0 \wedge x = n \wedge y = n$ }
while $x > 0$ do
  $x = x - 1;\ y = y - 1$
done
{ $x + y = 0$ }
```

Assume that variables x and y both have the value $n \geq 0$ before entering the loop. In the loop body, each variable is decremented by one until the variable x is zero. We want to show that $x+y$ is zero after executing the loop. Note that the predicate $x = y$ is implicitly implied by the loop. The program text however does not reveal this equality explicitly. Moreover, atomic predicates from the program text can not express loop invariants that establish the specification. Using atomic predicates in the program text does not give necessary atomic predicates.

Any loop invariant must be weaker than the pre-condition and stronger than the disjunction of the loop guard and the post-condition. We use the atomic predicates in an interpolant of $n \geq 0 \wedge x = n \wedge y = n$ and $\neg(x + y = 0 \vee x > 0)$ to obtain the initial atomic predicates $\{x = y, 2y \geq 0\}$. Observe that the interpolation theorem is able to synthesize the implicit predicate $x = y$. In fact, $x = y \wedge x \geq 0$ is a loop invariant that establishes the specification of the loop.

1.2. **Related Work.** Loop invariant inference using algorithmic learning is introduced in [15]. In [17], the learning-based technique is extended to quantified loop invariants. Both algorithms require users to provide atomic predicates. The present work addresses this problem for the case of quantifier-free loop invariants.

Many interpolation algorithms and their implementations are available [3, 7, 20]. Interpolation -based techniques for predicate refinement in software model checking are proposed in [9, 12, 13, 21, 14]. 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 [22]. 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 [23]. The technique reduces the invariant inference problem to constraint programming and generates predicates in user-provided templates.

1.3. **Paper Organization.** This paper is organized as follows. After Introduction, preliminaries are given in Section 2. We review the learning-based loop invariant inference framework in Section 3. Our technical results are presented in Section 4. Section 5 gives the loop invariant inference algorithm with automatic predicate generation. We report our experimental results in Section 6. Section 7 concludes this work.

2. Preliminaries

Let *QF* denote the quantifier-free logic with equality, linear inequality, and uninterpreted functions [20, 21]. 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 [8, 18].

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 interpolation theorem states that an inductive interpolant exists for any inconsistent sequence [6, 20, 21].

We consider the following imperative language in this paper:

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

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.

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

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

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

Observe that all universal quantifiers occur positively in $\mathit{Pre}(\theta, S)$ for any S. They can be eliminated by Skolem constants [11, 19].

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 a formula $\iota \in QF$ 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.

3. Inferring Loop Invariants with Algorithmic Learning

Given an annotated loop $\{\delta\}$ `while` κ `do` $S_1; S_2; \cdots ; S_m$ `done` $\{\epsilon\}$, we would like to infer a loop invariant to establish the pre- and post-conditions. Given a set P of atomic predicates, the work in [15] shows how to apply a learning algorithm for Boolean formulae to infer quantifier-free loop invariants freely generated by P. The authors 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. For the rest of this section, we review the learning-based framework for inferring quantifier-free loop invariant [15].

3.1. Predicate Abstraction.

Let $QF[P]$ denote the set of quantifier-free formulae generated from the set of atomic predicates P. Consider the set of Boolean formulae $Bool[B_P]$ generated by the set of Boolean variables $B_P \stackrel{\triangle}{=} \{b_p : p \in P\}$. 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. Formulae in $QF[P]$ and $Bool[B_P]$ are related by the following functions [15] (Figure 1):

$$\begin{tikzcd}
QF[P] \arrow[r, shift right, "\alpha"'] & Bool[B_P] \arrow[l, shift right, "\gamma"'] \\
Val_X \arrow[u, "\Gamma"] \arrow[r, "\alpha^*"'] & Val_{B_P} \arrow[ul, "\gamma^*"']
\end{tikzcd}$$

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

$$
\begin{array}{rcl}
\gamma(\beta) & \triangleq & \beta[B_P \mapsto P] \\
\alpha(\theta) & \triangleq & \bigvee\{\beta \in Bool[B_P] : \beta \text{ is a canonical monomial and } \theta \wedge \gamma(\beta) \text{ is satisfiable}\} \\
\gamma^*(\mu) & \triangleq & \bigwedge\limits_{\mu(b_p)=T} \{p\} \wedge \bigwedge\limits_{\mu(b_p)=F} \{\neg p\} \\
\alpha^*(\nu) & \triangleq & \mu \text{ where } \mu(b_p) = \begin{cases} T & \text{if } \nu \models p \\ F & \text{if } \nu \not\models p \end{cases}
\end{array}
$$

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, a valuation over X is mapped to a valuation over B_P by α^*; a valuation over B_P is mapped to a quantifier-free formula in $QF[P]$ by γ^*.

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

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

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 $\mu(b_{y=n}) = F$. Observe that the pair (α, γ) forms the Galois correspondence in Cartesian predicate abstraction [2].

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

Lemma 3.1. 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 $\alpha(\theta) = \bigvee\limits_i \beta_i$ where β_i is a canonical monomial and $\theta \wedge \gamma(\beta_i)$ is satisfiable. By Lemma 3.2, $\gamma(\beta_i) \Rightarrow \theta$. Hence $\gamma(\beta_i) \Rightarrow \rho$ and $\rho \wedge \gamma(\beta_i)$ is satisfiable. □

Lemma 3.2. 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 $\theta' = \bigvee\limits_i \theta_i \in QF[P]$ be a propositional formula in disjunctive normal form such that θ' is equivalent to θ.

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, each atomic propositions in A appears in $\gamma(\beta)$. 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 3.3. Let P be a set of atomic propositions and $\theta \in QF[P]$. Then $\theta \Leftrightarrow \gamma(\alpha(\theta))$.

Proof. Let $\theta' = \bigwedge\limits_i \theta_i$ 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 3.2, $\gamma(\beta) \Rightarrow \theta$. Hence $\nu \models \theta$. □

Lemma 3.4. Let P be a set of atomic propositions, $\theta \in \mathit{QF}[P]$, $\beta \in \mathit{Bool}[B_P]$, 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, $\chi(\alpha^*(\nu)) \to \alpha(\theta)$. $\alpha^*(\nu) \models \alpha(\theta)$ 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, $\gamma(\beta) \to \theta$ by Lemma 3.2. Hence $\nu \models \theta$.

(2) Assume $\nu \models \gamma(\beta)$. By Lemma 3.4 1, $\alpha^*(\nu) \models \alpha(\gamma(\beta))$. Note that $\beta = \alpha(\gamma(\beta))$. Thus $\alpha^*(\nu) \models \beta$.

□

Lemma 3.5. Let P be a set of atomic propositions, $\theta \in \mathit{QF}[P]$, 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 3.1, $\alpha(\gamma^*(\mu)) \Rightarrow \alpha(\theta)$. Note that $\gamma^*(\mu) = \gamma(\chi(\mu))$. By Lemma 3.3, $\chi(\mu) \Rightarrow \alpha(\theta)$. 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 3.4 (2), $\alpha^*(\nu) \models \chi(\mu)$. Since $\chi(\mu) \Rightarrow \alpha(\theta)$, $\alpha^*(\nu) \models \alpha(\theta)$. By Lemma 3.4 (1), $\nu \models \theta$. Therefore, $\gamma^*(\mu) \Rightarrow \theta$. □

3.2. **Answering Queries from Algorithmic Learning.**

After formulae in QF and valuations in Val_X are abstracted to $\mathit{Bool}[B_P]$ 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 $\mathit{Bool}[B_P]$. A *learning algorithm* computes a representation of the target ξ by interacting with a teacher. The *teacher* should answer the following queries [1, 4]:

- *Membership queries.* Let $\mu \in \mathit{Val}_{B_P}$ be an abstract valuation. The membership query $\mathit{MEM}(\mu)$ asks if the unknown target ξ is satisfied by μ. If so, the teacher answers *YES*; otherwise, *NO*.
- *Equivalence queries.* Let $\beta \in \mathit{Bool}[B_P]$ be an *abstract conjecture*. The equivalence query $\mathit{EQ}(\beta)$ 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 [15] is to exploit approximations to loop invariants. An

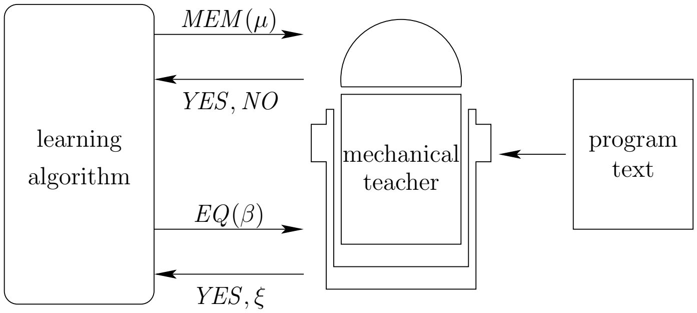

Figure 2. Learning-based Framework

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 $\bar{\iota}$ which is weaker than some loop invariants.

Figure 2 shows the 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 can try to answer these queries by the information derived from program texts. In this case, approximations to loop invariants are used. Observe the simplicity of the learning-based framework. 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.

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

3.2.1. *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 3.5). And $\mu \models \alpha(\iota)$ by Lemma 3.1.
(2) $\gamma^*(\mu) \not\Rightarrow \bar{\iota}$. Thus $\mu \not\models \alpha(\bar{\iota})$ (Lemma 3.5). That is, $\mu \models \neg\alpha(\bar{\iota})$. Since $\iota \to \bar{\iota}$, we have $\mu \not\models \alpha(\iota)$ by Lemma 3.1.
(3) Otherwise, we cannot determine whether $\mu \models \alpha(\iota)$ by the approximations. In this case, we answer YES or NO randomly.

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 [15].

```
/* $\underline{\iota}, \bar{\iota}$ :  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 $\theta$ is inconsistent then return NO;
if $\theta \Rightarrow \underline{\iota}$ then return YES;
if $\nu \models \neg(\theta \Rightarrow \bar{\iota})$ then return NO;
return YES or NO randomly;
```

Algorithm 1: Membership Query Resolution

```
/* $\{\delta\}$ while $\kappa$ do $S_1; S_2; \cdots ; S_m$ done $\{\epsilon\}$ :  an annotated loop        */
/* $\underline{\iota}, \bar{\iota}$ :  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 $\nu \models \neg(\theta \Rightarrow \bar{\iota})$ or $\nu \models \neg(\theta \wedge \kappa \Rightarrow Pre(\bar{\iota} : S_1; S_2; \cdots ; S_m))$ then
    return $\alpha^*(\nu)$;
return a random abstract counterexample;
```

Algorithm 2: Equivalence Query Resolution

3.2.2. *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 3.4 and 3.1, 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 $\nu \models \neg(\gamma(\beta) \Rightarrow \bar{\iota})$. Then $\nu \models \gamma(\beta) \wedge \neg\bar{\iota}$. By Lemma 3.4, $\alpha^*(\nu) \models \beta$. $\alpha^*(\nu) \models \neg\alpha(\iota)$ by Lemma 3.4 and 3.1. 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.

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 resolve by given invariant approximations. We can still refine approximations using some static analysis to give more accurate counterexample.

3.3. **Main Loop of of Inference Framework.** The main loop of loop invariant inference algorithm is given in Algorithm 3. For an annotated loop $\{\delta\}$ `while` κ `do` $S_1; S_2; \cdots ; S_m$ `done` $\{\epsilon\}$, we heuristically choose $\delta \vee \epsilon$ and $\epsilon \vee \kappa$ as the under- and over-approximations respectively. Note that the under-approximation is different from the strongest approximation δ. It is reported that the approximations $\delta \vee \epsilon$ and $\epsilon \vee \kappa$ are more effective in resolving queries [15]. After determining the approximations, a learning algorithm is used to find an invariant. In [15], we used CDNF algorithm [4] with Algorithms 1 and 2 for resolving resolve queries.

Note that the mechanical teacher may give conflicting answers. Random answers to membership queries may contradict to abstract counterexamples from equivalence queries.

```
/* {$\delta$} while $\kappa$ 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: Loop Invariant Inference

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.

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 theorem is essential to our predicate generation technique [6, 22, 21, 13]. 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 $\lambda \wedge \kappa \Rightarrow Pre(\lambda : S_1; S_2; \cdots ; S_m)$. 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 $\beta \in Bool[B_P]$ 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:

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

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, $\phi \Rightarrow Pre(\psi : S_1; S_2; \cdots; S_m)$ 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 $\phi \Rightarrow Pre(\psi : S_1; S_2; \cdots; S_m)$. The following proposition will be handy.

Proposition 4.1. Let $S_1; S_2; \cdots; S_m$ be a sequence of statements. For any ϕ with $\phi \Rightarrow Pre(\psi : S_1; S_2; \cdots; S_m)$, $\Xi(\phi, S_1, \ldots, S_m, \psi)$ 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 $\phi \Rightarrow Pre(\psi : S_1; S_2; \cdots; S_m)$. By definition of *Pre*, $Pre(\psi : S_1; S_2; \cdots; S_m) = Pre(Pre(\psi : S_2; \cdots; S_m) : S_1)$ 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 4.1 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 4.2. 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 4.2 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 4.1 is applicable with the pair $\theta \wedge \kappa$ and $\bar{\iota}$.

Corollary 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 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 4.1. □

4.2.2. *Interpolating Under-Approximation.* For under-approximations, there is no necessary condition. Nevertheless, Proposition 4.1 is applicable with the pair $\underline{\iota} \wedge \kappa$ and θ.

Corollary 4.4. 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 4.1. □

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 4.2, 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 4.3). 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 4.4).

```
/* $\{\delta\}$ while $\kappa$ do $S_1;\cdots;S_m$ done $\{\epsilon\}$ :  an annotated loop            */
/* $\underline{\iota},\bar{\iota}$ :  under- and over-approximations to loop invariants             */
Input: a formula $\theta \in QF[P]$ such that $\theta \wedge \kappa \Rightarrow Pre(\bar{\iota} : S_1; S_2; \cdots ; S_m)$
Output: a set of atomic predicates
$I$ := an inductive interpolant of $\Xi(\theta \wedge \kappa, S_1, S_2, \ldots, S_m, \bar{\iota})$;
$Q$ := desuperscripted atomic predicates in $I$;
if $\underline{\iota} \wedge \kappa \Rightarrow Pre(\theta : S_1; S_2; \cdots ; S_m)$ then
    $J$ := an inductive interpolants of $\Xi(\underline{\iota} \wedge \kappa, S_1, S_2, \ldots, S_m, \theta)$;
    $R$ := desuperscripted atomic predicates in $J$;
    $Q := Q \cup R$;
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 $\lambda \in Bool[b_{y=0}]$. 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. Define

$$\Gamma(\nu) \stackrel{\triangle}{=} \bigwedge_{x \in X} x = \nu(x).$$

The function $\Gamma(\nu)$ specifies the valuation ν in QF (Figure 1). 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.

Algorithm 5 generates atomic predicates from conflicting abstract counterexamples. Let ν and ν' be distinct valuations in Val_X. We compute formulae $\chi = \Gamma(\nu)$ and $\chi' = \Gamma(\nu')$. Since ν and ν' are conflicting, they correspond to the same abstract valuation $\alpha^*(\nu) = \alpha^*(\nu')$. Let $\rho = \gamma^*(\alpha^*(\nu))$. We have $\chi \Rightarrow \rho$ and $\chi' \Rightarrow \rho$ [15]. Recall that $\chi \wedge \chi'$ is inconsistent. $[\chi, \chi' \vee \neg\rho]$ is also inconsistent for $\chi \Rightarrow \rho$. Algorithm 5 returns atomic predicates in an inductive interpolant of $[\chi, \chi' \vee \neg\rho]$.

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

Algorithm 5: PredicatesFromConflict (ν, ν')

5. Loop Invariant Inference Algorithms with Automatic Predicate Generation

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

```
/* $\{\delta\}$ while $\kappa$ do $S_1; S_2; \cdots ; S_m$ done $\{\epsilon\}$ :  an annotated loop        */
Output: a loop invariant for the annotated loop
$\underline{\iota} := \delta \vee \epsilon$;
$\overline{\iota} := \epsilon \vee \kappa$;
$P :=$ atomic predicates in an inductive interpolant of $[\underline{\iota}, \neg\overline{\iota}]$;
repeat
    try
        call a learning algorithm for Boolean formulae where membership and
        equivalence queries are resolved by Algorithms 1 and 7 respectively;
    catch conflict abstract counterexamples →
        find distinct valuations $\nu$ and $\nu'$ such that $\alpha^*(\nu) = \alpha^*(\nu')$;
        $P := P \cup$ PredicatesFromConflict$(\nu, \nu')$;
until a loop invariant is found ;
```

Algorithm 6: Loop Invariant Inference

Before invoking a learning algorithm, we compute the initial atomic predicates by interpolating $\underline{\iota}$ and $\neg\overline{\iota}$ (Section 4.1). With initial set of predicates, we start learning process until it fins a loop invariant or conflict which means the current predicate abstraction is too coarse to find an invariant. If learning algorithm finds a conflict during its process, the loop invariant inference algorithm adds more atomic predicates by Algorithm 5. Then the main loop reiterates with the new set of atomic predicates. Note that we use the same algorithm for membership query resolution (Algorithm 1), but different one for equivalence query resolution, which is detailed next.

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* 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

```
/* $\tau$ :  a threshold to generate new atomic predicates                */
/* $\{\delta\}$ while $\kappa$ do $S_1; S_2; \cdots ; S_m$ done $\{\epsilon\}$ :  an annotated loop        */
/* $\underline{\iota}, \bar{\iota}$ :  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 $\nu \models \neg(\theta \Rightarrow \bar{\iota})$ or $\nu \models \neg(\theta \wedge \kappa \Rightarrow Pre(\bar{\iota} : S_1; S_2; \cdots ; S_m))$ then
    record $\nu$; return $\alpha^*(\nu)$;
if the number of random abstract counterexamples $\leq \tau$ then
    return a random abstract counterexample;
else
    $P := P \cup$ PredicatesFromConjecture($\theta$);
    $\tau := \lceil 1.3^{|P|} \rceil$;
    reiterate the main loop;
end
```

Algorithm 7: Equivalence Query Resolution

over-approximation, or it does not satisfy the necessary condition given in Proposition 4.2, an abstract counterexample is returned after recording the witness valuation [15, 17]. The witnessing valuations are needed to synthesize atomic predicates 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 invoke Algorithm 5 to synthesize more atomic predicates from the incorrect conjecture, update the threshold to $\lceil 1.3^{|P|} \rceil$, and then restart the main loop.

6. Experimental Results

We have implemented the proposed technique in OCaml. In our implementation, the SMT solver Yices and the interpolating theorem prover CSIsat [3] are used for query resolution and interpolation respectively. In addition to the examples in [15], 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

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

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

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 [15]					Current					BLAST [21]	
		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)

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).

In the table, the column Previous represents the work in [15] 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 [21]. 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:

```
{ $size = M \wedge copy = N$ }
while $size > 0$ do
    $available$ := nondet;
    if $available > size$ then
        $copy := copy + available$;
        $size := size - available$;
done
{ $size = 0 \implies copy = M + N$ }
```

Figure 3. A Sample Loop in Tar

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

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

```
{ $phase = \mathtt{F} \wedge success = \mathtt{F} \wedge give\_up = \mathtt{F} \wedge cutoff = 0 \wedge count = 0$ }
while $\neg(success \vee give\_up)$ do
    $entered\_phase := \mathtt{F}$;
    if $\neg phase$ then
        if $cutoff = 0$ then $cutoff := 1$;
        else if $cutoff = 1 \wedge maxcost > 1$ then $cutoff := maxcost$;
              else $phase := \mathtt{T}$; $entered\_phase := \mathtt{T}$; $cutoff := 1000$;
        if $cutoff = maxcost \wedge \neg search$ then $give\_up := \mathtt{T}$;
    else
        $count := count + 1$;
        if $count > words$ then $give\_up := \mathtt{T}$;
    if $entered\_phase$ then $count := 1$;
    $linkages$ := nondet;
    if $linkages > 5000$ then $linkages := 5000$;
    $canonical := 0$; $valid := 0$;
    if $linkages \neq 0$ then
        $valid$ := nondet;
        assume $0 \leq valid \wedge valid \leq linkages$;
        $canonical := linkages$;
    if $valid > 0$ then $success := \mathtt{T}$;
done
{ $(valid > 0 \vee count > words \vee (cutoff = maxcost \wedge \neg search)) \wedge$
  $valid \leq linkages \wedge canonical = linkages \wedge linkages \leq 5000$ }
```

Figure 4. A Sample Loop in SPEC2000 Benchmark PARSER

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:

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

The invariant is simpler and thus easier to understand than the one presented in [15]. 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 equal to 0.06. In contrast, the execution time for [15] 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.

```
{ $retries = 100 \land (\neg ireason\_has\_ATAPI\_COD \lor ireason\_has\_ATAPI\_IO)$ }
while $retries \neq 0 \land (\neg ireason\_has\_ATAPI\_COD \lor ireason\_has\_ATAPI\_IO)$ do
    $retries := retries - 1$;
    $ireason\_has\_ATAPI\_COD :=$ nondet;
    $ireason\_has\_ATAPI\_IO :=$ nondet;
    if $retries = 0$ then
        $ireason\_has\_ATAPI\_COD :=$ T;
        $ireason\_has\_ATAPI\_IO :=$ F;
done
{ $retries < 100 \land ireason\_has\_ATAPI\_COD \land \neg ireason\_has\_ATAPI\_IO$ }
```

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] Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2) (1987) 87–106

[2] Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking c programs. In: TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, London, UK, Springer-Verlag (2001) 268–283

[3] Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: CAV. (2008) 304–308

[4] Bshouty, N.H.: Exact learning boolean functions via the monotone theory. Information and Computation **123** (1995) 146–153

[5] Canet, G., Cuoq, P., Monate, B.: A value analysis for c programs. In: Source Code Analysis and Manipulation, IEEE (2009) 123–124

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

[7] D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: VMCAI. (2010) 129–145

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

[9] Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with craig interpolation and symbolic pushdown systems. In: TACAS. (2006) 489–503

[10] Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In Davies, J., Schulte, W., Barnett, M., eds.: Formal Methods and Software Engineering. Volume 3308 of LNCS., Springer (2004) 15–29

[11] Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, ACM (2002) 191–202

[12] Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL '04, New York, NY, USA, ACM (2004) 232–244

[13] Jhala, R., Mcmillan, K.L.: A practical and complete approach to predicate refinement. In: TACAS. Volume 3920 of LNCS., Springer (2006) 459–473

[14] Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: CAV, volume 4590 of LNCS, Springer (2007) 193–206

[15] Jung, Y., Kong, S., Wang, B.Y., Yi, K.: Deriving invariants in propositional logic by algorithmic learning, decision procedure, and predicate abstraction. In: VMCAI. Volume 5944 of LNCS., Springer (2010) 180–196

[16] 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 LNCS., Springer (2011) 205–219

[17] 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. (2010) to appear.

[18] Kroening, D., Strichman, O.: Decision Procedures an algorithmic point of view. EATCS. Springer (2008)

[19] Lahiri, S.K., Bryant, R.E., Bryant, A.E.: Constructing quantified invariants via predicate abstraction. In: VMCAI. Volume 2937 of LNCS., Springer (2004) 267–281

[20] McMillan, K.L.: An interpolating theorem prover. Theoretical Computer Science **345**(1) (2005) 101–121

[21] McMillan, K.L.: Lazy abstraction with interpolants. In Ball, T., Jones, R.B., eds.: CAV. Volume 4144 of LNCS., Springer (2006) 123–136

[22] McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: TACAS. Volume 4693 of LNCS., Springer (2008) 413–427

[23] Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI, ACM (2009) 223–234