Automatically Inferring Quantified Loop Invariants by Algorithmic Learning from Simple Templates*

Soonho Kong[1], Yungbum Jung[1], Cristina David[2], Bow-Yaw Wang[3], and Kwangkeun Yi[1]

[1] Seoul National University
[2] National University of Singapore
[3] INRIA, Tsinghua University, and Academia Sinica

Abstract. By combining algorithmic learning, decision procedures, predicate abstraction, and simple templates, we present an automated technique for finding quantified loop invariants. Our technique can find arbitrary first-order invariants (modulo a fixed set of atomic propositions and an underlying SMT solver) in the form of the given template and exploits the flexibility in invariants by a simple randomized mechanism. The proposed technique is able to find quantified invariants for loops from the Linux source, as well as for the benchmark code used in the previous works. Our contribution is a simpler technique than the previous works yet with a reasonable derivation power.

1 Introduction

Recently, algorithmic learning has been successfully applied to invariant generation. The new approach formalizes the invariant generation problem as an instance of algorithmic learning: to generate an invariant is to learn a concept from a teacher. Using a learning algorithm as a black box, one only needs to design a mechanical teacher that guides the learning algorithm to invariants. The learning-based framework not only simplifies the invariant generation algorithms, the new approach can also automatically generate invariants for realistic C loops at a reasonable cost [15].

Figure 1 shows the new framework proposed in [15]. In the figure, the CDNF algorithm is used to drive the search of quantifier-free invariants. The CDNF algorithm is an exact learning algorithm for Boolean formulae. It computes a

* 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). This work was partly supported by MoE Tier-2 grant R-252-000-411-112 and by the National Science Council of Taiwan projects No. NSC97-2221-E-001-003-MY3, NSC97-2221-E-001-006-MY3, the FORMES Project within LIAMA Consortium, and the French ANR project SIVES ANR-08-BLAN-0326-01

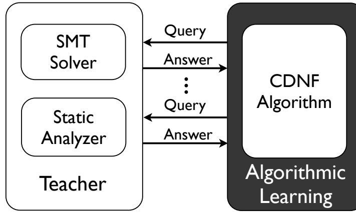

Fig. 1. The learning-based framework

representation of an unknown target formula by asking a teacher two types of queries. A membership query asks if a valuation to Boolean variables satisfies the unknown target; an equivalence query asks if a candidate formula is equivalent to the target. With predicate abstraction, the new approach formulates an unknown quantifier-free invariant as the unknown target Boolean formula. One only needs to automate the query resolution process to infer an invariant.

If an invariant was known, a mechanical teacher to resove queries can be implemented straightforwardly. In the context of invariant generation, no invariant is known. However, a simple randomized automatic teacher is proposed in [15]. With the help of SMT solvers, user-provided annotations, and coin tossing, one can resolve both types of queries by a simple reduction to the satisfiability problem of quantifier-free formulae. An ingenious feature of this design is its random walk. Due to the lack of information, some queries cannot be resolved decisively. In this case, the teacher simply gives a random answer. The learning algorithm will then look for invariants consistent with both decisive and random answers from the teacher. Since there are sufficiently many invariants for an annotated loop in practice, almost certainly the learning algorithm can find one.

The work [15] has, however, one obvious limitation; it can only generate quantifier-free invariants. Yet loops iterating over arrays often require invariants quantified over indices. It will be very useful to extend the new approach to quantified invariants. However, a naïve extension would not work. First of all, it is not clear how to associate an arbitrarily quantified formula with a quantified Boolean formula. There is no counterpart (a Boolean formula) for quantified variables in, say, $\forall i.i > 10$. Second, there is no exact learning algorithm for quantified Boolean formulae to the best of our knowledge. Even if an abstraction for quantified formulae was available, we could not adopt the same learning-based framework. Third, computability issues must be addressed because the satisfiability problem for arbitrarily quantified formulae is undecidable. Developing an effective invariant generation algorithm for quantified invariants is therefore an interesting challenge to the learning-based framework.

This article is about our findings in generating *quantified* invariants with algorithmic learning:

– We show that a simple combination of algorithmic learning, decision procedures, predicate abstraction, and templates can automatically infer quantified

loop invariants. The technique is as powerful as the previous approaches [9,20] yet is much simpler.
- The technique needs a very simple template such as "$\forall k.[]$" or "$\forall k.\exists i.[]$." Our algorithm can generate any quantified invariants expressible by a fixed set of atomic propositions in the form of the given template. Moreover, the correctness of generated invariants is verified by an SMT solver.
- The technique works in realistic settings: The proposed technique can find quantified invariants for some Linux library, kernel, and device driver sources, as well as for the benchmark code used in the previous work [20].
- The technique's future improvement is free. Since our algorithm uses the two key technologies (exact learning algorithm and decision procedures) as black boxes, future advances of these technologies will straightforwardly benefit our approach.

1.1 Motivating Example

In order to illustrate how our algorithm works, we briefly describe the learning process for the `max` example from [20].

$$\{m = 0 \wedge i = 0\}$$
$$\texttt{while}\ i < n\ \texttt{do if}\ a[m] < a[i]\ \texttt{then}\ m = i\ \texttt{fi};\ i = i + 1\ \texttt{end}$$
$$\{\forall k.k < n \Rightarrow a[k] \leq a[m]\}$$

The `max` example examines $a[0]$ through $a[n-1]$ and finds the index of the maximal element in the array. This simple loop is annotated with the precondition $m = 0 \wedge i = 0$ and the postcondition $\forall k.0 \leq k < n \Rightarrow a[k] \leq a[m]$.

Template and Atomic Propositions A template and atomic propositions are provided manually by user. We provide the template $\forall k.[]$. The postcondition is universally quantified with k and gives a hint to the form of an invariant. By extracting from the annotated loop and adding the last two atomic propositions from the user's guidance, we use the following set of atomic propositions:

$$\{i < n,\ m = 0,\ i = 0,\ a[m] < a[i],\ a[k] \leq a[m],\ k < n,\ k < i\}.$$

Query Resolution In this example, 20 membership queries and 6 equivalence queries are made by the learning algorithm on average. For simplicity, let us find an invariant that is weaker than the precondition but stronger than the postcondition. We describe how the teacher resolves some of these queries.

- Equivalence Query: The learning algorithm starts with an equivalence query $EQ(\texttt{T})$, namely whether $\forall k.\texttt{T}$ can be an invariant. The teacher answers *NO* since $\forall k.\texttt{T}$ is weaker than the postcondition. Additionally, by employing an SMT solver, the teacher returns a counterexample $\{m = 0,\ k = 1,\ n = 2,\ i = 2,\ a[0] = 0,\ a[1] = 1\}$, under which $\forall k.\texttt{T}$ evaluates to true, whereas the postcondition evaluates to false.

– Membership Query: After a few equivalence queries, a membership query asks whether $\bigwedge\{i \geq n,\ m = 0,\ i = 0,\ k \geq n,\ a[k] \leq a[m],\ a[m] \geq a[i]\}$ is a part of an invariant. The teacher replies *YES* since the query is included in the precondition and therefore should also be included in an invariant.
– Membership Query: The membership query $MEM(\bigwedge\{i < n,\ m = 0,\ i \neq 0,\ k < n,\ a[k] > a[m],\ k < i,\ a[m] \geq a[i]\})$ is not resolvable because the template is not *well-formed* (Definition 1) by the given membership query. In this case, the teacher gives a random answer (*YES* or *NO*). Interestingly, each answer leads to a different invariant for this query. If the answer is *YES*, we find an invariant $\forall k.(i < n \wedge k \geq i) \vee (a[k] \leq a[m]) \vee (k \geq n)$; if the answer is *NO*, we find another invariant $\forall k.(i < n \wedge k \geq i) \vee (a[k] \leq a[m]) \vee (k \geq n \wedge k \geq i)$. This shows how our approach exploits a multitude of invariants for the annotated loop.

1.2 Organization

We organize this paper as follows. After preliminaries in Section 2, we present problems and solutions in Section 3. Our abstraction is briefly described in Section 4. The details of our technique are described in Section 5. We report experiments in Section 6, discuss related work in Section 7, then conclude in Section 8.

2 Preliminaries

The abstract syntax of our simple imperative language is given below:

$$
\begin{aligned}
\mathsf{Stmt} &\triangleq \texttt{nop} \mid \mathsf{Stmt};\, \mathsf{Stmt} \mid x := \mathsf{Exp} \mid b := \mathsf{Prop} \mid a[\mathsf{Exp}] := \mathsf{Exp} \mid \\
&\quad\ a[\mathsf{Exp}] := \texttt{nondet} \mid x := \texttt{nondet} \mid b := \texttt{nondet} \mid \\
&\quad\ \texttt{if}\ \mathsf{Prop}\ \texttt{then}\ \mathsf{Stmt}\ \texttt{else}\ \mathsf{Stmt} \mid \{\ \mathsf{Pred}\ \}\ \texttt{while}\ \mathsf{Prop}\ \texttt{do}\ \mathsf{Stmt}\ \{\ \mathsf{Pred}\ \} \\
\mathsf{Exp} &\triangleq n \mid x \mid a[\mathsf{Exp}] \mid \mathsf{Exp} + \mathsf{Exp} \mid \mathsf{Exp} - \mathsf{Exp} \\
\mathsf{Prop} &\triangleq \texttt{F} \mid b \mid \neg\mathsf{Prop} \mid \mathsf{Prop} \wedge \mathsf{Prop} \mid \mathsf{Exp} < \mathsf{Exp} \mid \mathsf{Exp} = \mathsf{Exp} \\
\mathsf{Pred} &\triangleq \mathsf{Prop} \mid \forall x.\mathsf{Pred} \mid \exists x.\mathsf{Pred} \mid \mathsf{Pred} \wedge \mathsf{Pred} \mid \neg\mathsf{Pred}
\end{aligned}
$$

The language has two basic types: Booleans and natural numbers. A term in Exp is a natural number; a term in Prop is a quantifier-free formula and of Boolean type; a term in Pred is a first-order formula. The keyword `nondet` is used for unknown values from user's input or complex structures (e.g, pointer operations, function calls, etc.). In an annotated loop $\{\delta\}$ $\texttt{while}\ \kappa\ \texttt{do}\ S$ $\{\epsilon\}$, $\kappa \in \mathsf{Prop}$ is its *guard*, and $\delta, \epsilon \in \mathsf{Pred}$ are its *precondition* and *postcondition* respectively. Quantifier-free formulae of the forms b, $\pi_0 < \pi_1$, and $\pi_0 = \pi_1$ are called *atomic propositions*. If A is a set of atomic propositions, then Prop_A and Pred_A denote the set of quantifier-free and first-order formulae generated from A, respectively.

A *template* $t[] \in \tau$ is a finite sequence of quantifiers followed by a hole to be filled with a quantifier-free formula in Prop_A.

$$\tau \triangleq [] \mid \forall I.\tau \mid \exists I.\tau.$$

Let $\theta \in \mathsf{Prop}_A$ be a quantifier-free formula. We write $t[\theta]$ to denote the first-order formula obtained by replacing the hole in $t[]$ with θ. Observe that any first-order formula can be transformed into the prenex normal form; it can be expressed in the form of a proper template.

A *precondition* $Pre(\rho, S)$ for $\rho \in \mathsf{Pred}$ with respect to a statement S is a first-order formula that guarantees ρ after the execution of the statement S. Let $\{\delta\}$ `while` κ `do` S $\{\epsilon\}$ be an annotated loop and $t[] \in \tau$ be a template. The *invariant generation problem with template* $t[]$ is to compute a first-order formula $t[\theta]$ such that (1) $\delta \Rightarrow t[\theta]$; (2) $\neg\kappa \wedge t[\theta] \Rightarrow \epsilon$; and (3) $\kappa \wedge t[\theta] \Rightarrow Pre(t[\theta], S)$. Observe that the condition (2) is equivalent to $t[\theta] \Rightarrow \epsilon \vee \kappa$. We have $\delta \Rightarrow t[\theta]$ and $t[\theta] \Rightarrow \epsilon \vee \kappa$ for any invariant $t[\theta]$. δ and $\epsilon \vee \kappa$ are subsequently called the *strongest under-approximation* and *weakest over-approximation* to invariants respectively.

A *valuation* ν is an assignment of natural numbers to integer variables and truth values to Boolean variables. If A is a set of atomic propositions and $Var(A)$ is the set of variables occurred in A, $Val_{Var(A)}$ denotes the set of valuations for $Var(A)$. A valuation ν is a *model* of a first-order formula ρ (written $\nu \models \rho$) if ρ evaluates to $\mathtt{T}$ under ν. Let B be a set of Boolean variables. We write Bool_B for the class of Boolean formulae over Boolean variables B. A *Boolean valuation* μ is an assignment of truth values to Boolean variables. The set of Boolean valuations for B is denoted by Val_B. A Boolean valuation μ is a *Boolean model* of the Boolean formula β (written $\mu \models \beta$) if β evaluates to $\mathtt{T}$ under μ.

Given a first-order formula ρ, a *satisfiability modulo theories (SMT) solver* [6,16] returns a model of ν if it exists. In general, SMT solver is incomplete over quantified formulae and may return a potential model (written $SMT(\rho) \xrightarrow{!} \nu$). It returns $UNSAT$ (written $SMT(\rho) \rightarrow UNSAT$) if the solver proves the formula unsatisfiable. Note that an SMT solver can only err when it returns a (potential) model. If $UNSAT$ is returned, the input formula is certainly unsatisfiable.

CDNF Learning Algorithm [3] The CDNF (Conjunctive Disjunctive Normal Form) algorithm is an exact algorithm that computes a representation for any *target* $\lambda \in \mathsf{Bool}_B$ by asking a *teacher* queries. The teacher is required to resolve two types of queries:

- *Membership query* $MEM(\mu)$ where $\mu \in Val_B$. If the valuation μ is a Boolean model of the target Boolean formula λ, the teacher answers YES. Otherwise, the teacher answers NO;
- *Equivalence query* $EQ(\beta)$ where $\beta \in \mathsf{Bool}_B$. If the target Boolean formula λ is equivalent to β, the teacher answers YES. Otherwise, the teacher gives a counterexample. A *counterexample* is a valuation $\mu \in Val_B$ such that β and λ evaluate to different truth values under μ.

For a Boolean formula $\lambda \in \mathsf{Bool}_B$, define $|\lambda|_{CNF}$ and $|\lambda|_{DNF}$ to be the sizes of minimal Boolean formulae equivalent to λ in conjunctive and disjunctive normal forms respectively. The CDNF algorithm infers any target Boolean formula $\lambda \in \mathsf{Bool}_B$ with a polynomial number of queries in $|\lambda|_{CNF}$, $|\lambda|_{DNF}$, and $|B|$ [3].

3 Problems and Solutions

Given an annotated loop and a template, we apply algorithmic learning to find an invariant in the form of the given template. We follow the framework proposed in [15] and deploy the CDNF algorithm to drive the search of invariants. Since the learning algorithm assumes a teacher to answer queries, it remains to mechanize the query resolution process (Figure 1). Let $t[]$ be the given template and $t[\theta]$ an invariant. We will devise a teacher to guide the CDNF algorithm to infer $t[\theta]$.

To achieve this goal, we need to address two problems. First, the CDNF algorithm is a learning algorithm for Boolean formulae, not quantifier-free nor quantified formulae. Second, the CDNF algorithm assumes a teacher who knows the target $t[\theta]$ in its learning model. However, an invariant of the given annotated loop is yet to be computed and hence unknown to us. We need to devise a teacher without assuming any particular invariant $t[\theta]$.

For the first problem, we adopt predicate abstraction to associate Boolean formulae with quantified formulae. Recall that the formula θ in the invariant $t[\theta]$ is quantifier-free. Let α be an abstraction function from quantifier-free to Boolean formulae. Then $\lambda = \alpha(\theta)$ is a Boolean formula and serves as the target function to be inferred by the CDNF algorithm.

For the second problem, we need to design algorithms to resolve queries about the Boolean formula λ without knowing $t[\theta]$. This is achieved by exploiting the information derived from annotations and by making a few random guesses. Recall that any invariant must be weaker than the strongest under-approximation and stronger than the weakest over-approximation. Using an SMT solver, queries can be resolved by comparing with these invariant approximations. For queries unresolvable through approximations, we simply give random answers.

Following a similar framework to [15], we are able to infer quantified invariants of a given template for annotated loops. Our solution to the quantified invariant generation problem for annotated loops is in fact very general. It only requires users to provide a sequence of quantifiers and a fixed set of atomic propositions. With a number of coin tossing, our technique can infer arbitrary quantified invariants representable by the user inputs. This suggests that the algorithmic learning approach to invariant generation has great potential in invariant generation problems.

4 Predicate Abstraction with a Template

We begin with the association between Boolean formulae and first-order formulae in the form of a given template. Let A be a set of atomic propositions and $B(A) \stackrel{\triangle}{=} \{b_p : p \in A\}$ the set of corresponding Boolean variables. Figure 2 shows the abstraction used in our algorithm. The left box represents the class Pred_A of first-order formulae generated from A. The middle box corresponds to the class Prop_A of quantifier-free formulae generated from A. Since we are looking for quantified invariants in the form of the template $t[]$, Prop_A is in fact the essence of generated quantified invariants. The right box contains the class $\mathsf{Bool}_{B(A)}$ of

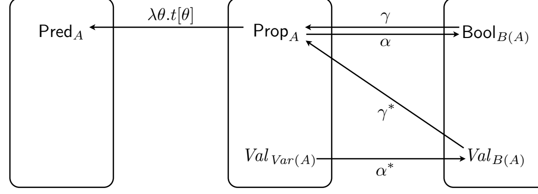

Fig. 2. The domains Pred_A, Prop_A, and $\mathsf{Bool}_{B(A)}$

Boolean formulae over the Boolean variables $B(A)$. The CDNF algorithm infers a target Boolean formula by posing queries in this domain.

The pair (γ, α) gives the correspondence between the domains $\mathsf{Bool}_{B(A)}$ and Prop_A. Let us call a Boolean formula $\beta \in \mathsf{Bool}_{B(A)}$ a *canonical monomial* if it is a conjunction of literals, where each variable appears exactly once. Define

$$\gamma : \mathsf{Bool}_{B(A)} \to \mathsf{Prop}_A \qquad \alpha : \mathsf{Prop}_A \to \mathsf{Bool}_{B(A)}$$

$$\gamma(\beta) = \beta[\overline{b}_p \mapsto \overline{p}]$$
$$\alpha(\theta) = \bigvee\{\beta \in \mathsf{Bool}_{B(A)} : \beta \text{ is a canonical monomial and } \theta \wedge \gamma(\beta) \text{ is satisfiable}\}.$$

Concretization function $\gamma(\beta) \in \mathsf{Prop}_A$ simply replaces Boolean variables in $B(A)$ by corresponding atomic propositions in A. On the other hand, $\alpha(\theta) \in \mathsf{Bool}_{B(A)}$ is the abstraction for any quantifier-free formula $\theta \in \mathsf{Prop}_A$.

A Boolean valuation $\mu \in Val_{B(A)}$ is associated with a quantifier-free formula $\gamma^*(\mu)$ and a first-order formula $t[\gamma^*(\mu)]$. A valuation $\nu \in Var(A)$ moreover induces a natural Boolean valuation $\alpha^*(\nu) \in Val_{B(A)}$.

$$\gamma^*(\mu) = \bigwedge_{p \in A}\{p : \mu(b_p) = \mathtt{T}\} \wedge \bigwedge_{p \in A}\{\neg p : \mu(b_p) = \mathtt{F}\}$$
$$\alpha^*(\nu)(b_p) = \nu \models p$$

The following lemmas characterize relations among these functions:

Lemma 1 ([15]). *Let A be a set of atomic propositions, $\theta \in \mathsf{Prop}_A$, $\beta \in \mathsf{Bool}_{B(A)}$, and ν a valuation for $Var(A)$. Then*

1. *$\nu \models \theta$ if and only if $\alpha^*(\nu) \models \alpha(\theta)$; and*
2. *$\nu \models \gamma(\beta)$ if and only if $\alpha^*(\nu) \models \beta$.*

Lemma 2 ([15]). *Let A be a set of atomic propositions, $\theta \in \mathsf{Prop}_A$, and μ a Boolean valuation for $B(A)$. Then $\gamma^*(\mu) \Rightarrow \theta$ if and only if $\mu \models \alpha(\theta)$.*

5 Learning Quantified Invariants

We present our query resolution algorithms, followed by the invariant generation algorithm. The query resolution algorithms exploit the information derived from the given annotated loop $\{\delta\}$ `while` κ `do` S $\{\epsilon\}$. Let $\underline{\iota}, \overline{\iota} \in \mathsf{Pred}$. We say $\underline{\iota}$ is an

Algorithm 1: Resolving Equivalence Queries

```
/* ι̲ : an under-approximation; ῑ : an over-approximation */
/* t[]: the given template */
```

Input: $\beta \in \mathsf{Bool}_{B(A)}$

Output: *YES*, or a counterexample ν s.t. $\alpha^*(\nu) \models \beta \oplus \lambda$

1 $\rho := t[\gamma(\beta)]$;

2 **if** $SMT(\underline{\iota} \wedge \neg\rho) \rightarrow UNSAT$ **and** $SMT(\rho \wedge \neg\bar{\iota}) \rightarrow UNSAT$ **and**

3 $\quad SMT(\kappa \wedge \rho \wedge \neg Pre(\rho, S)) \rightarrow UNSAT$ **then return** *YES*;

4 **if** $SMT(\underline{\iota} \wedge \neg\rho) \overset{!}{\rightarrow} \nu$ **then return** $\alpha^*(\nu)$;

5 **if** $SMT(\rho \wedge \neg\bar{\iota}) \overset{!}{\rightarrow} \nu$ **then return** $\alpha^*(\nu)$;

6 **if** $SMT(\rho \wedge \neg\underline{\iota}) \overset{!}{\rightarrow} \nu_0$ **or** $SMT(\bar{\iota} \wedge \neg\rho) \overset{!}{\rightarrow} \nu_1$ **then**

7 $\quad$ **return** $\alpha^*(\nu_0)$ *or* $\alpha^*(\nu_1)$ *randomly*;

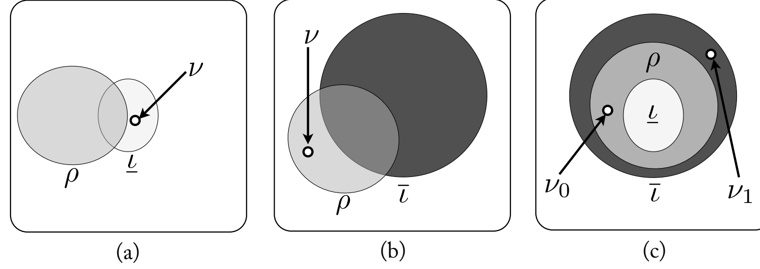

Fig. 3. Counterexamples in equivalence query resolution (c.f. Algorithm 1): (a) a counterexample inside the under-approximation $\underline{\iota}$ but outside the candidate ρ (line 4); (b) a counterexample inside the candidate ρ but outside the over-approximation $\bar{\iota}$ (line 5); (c) a random counterexample ν_0 (or ν_1) inside the candidate ρ (or over-approximation $\bar{\iota}$) but out of the under-approximation $\underline{\iota}$ (or candidate ρ), respectively (line 6 and 7).

under-approximation to invariants if $\delta \Rightarrow \underline{\iota}$ and $\underline{\iota} \Rightarrow \iota$ for some invariant ι of the annotated loop. Similarly, $\bar{\iota}$ is an *over-approximation* to invariants if $\bar{\iota} \Rightarrow \epsilon \vee \kappa$ and $\iota \Rightarrow \bar{\iota}$ for some invariant ι. The strongest under-approximation δ is an under-approximation; the weakest over-approximation $\epsilon \vee \kappa$ is an over-approximation. Better invariant approximations can be obtained by other techniques; they can be used in our query resolution algorithms.

5.1 Equivalence Queries

An equivalence query $EQ(\beta)$ with $\beta \in \mathsf{Bool}_{B(A)}$ asks if β is equivalent to the unknown target λ. Algorithm 1 gives our equivalence resolution algorithm. It first checks if $\rho = t[\gamma(\beta)]$ is indeed an invariant for the annotated loop by verifying $\underline{\iota} \Rightarrow \rho$, $\rho \Rightarrow \bar{\iota}$, and $\kappa \wedge \rho \Rightarrow Pre(\rho, S)$ with an SMT solver (line 2 and 3). If so,

the CDNF algorithm has generated an invariant and our teacher acknowledges that the target has been found. If the candidate ρ is not an invariant, we need to provide a counterexample. Figure 3 describes the process of counterexample discovery. The algorithm first tries to generate a counterexample inside of under-approximation (a), or outside of over-approximation (b). If it fails to find such counterexamples, the algorithm tries to return a valuation distinguishing ρ from invariant approximations as a random answer (c).

Recall that SMT solvers may err when a potential model is returned (line 4 – 6). If it returns an incorrect model, our equivalence resolution algorithm will give an incorrect answer to the learning algorithm. Incorrect answers effectively guide the CDNF algorithm to different quantified invariants. Note also that random answers do not yield incorrect results because the equivalence query resolution algorithm uses an SMT solver to *verify* that the found first-order formula is indeed an invariant.

5.2 Membership Queries

Algorithm 2: Resolving Membership Queries

/* $\underline{\iota}$: an under-approximation; $\bar{\iota}$: an over-approximation */
/* $t[]$: the given template */
Input: a valuation μ for $B(A)$
Output: *YES* or *NO*

1 **if** $SMT(\gamma^*(\mu)) \rightarrow UNSAT$ **then return** *NO*;
2 $\rho := t[\gamma^*(\mu)]$;
3 **if** $SMT(\rho \wedge \neg\bar{\iota}) \xrightarrow{!} \nu$ **then return** *NO*;
4 **if** $SMT(\rho \wedge \neg\underline{\iota}) \rightarrow UNSAT$ **and** $isWellFormed(t[], \gamma^*(\mu))$ **then return** *YES*;
5 **return** *YES or NO randomly*

In a membership query $MEM(\mu)$, our membership query resolution algorithm (Algorithm 2) should answer whether $\mu \models \lambda$. Note that any relation between atomic propositions A is lost in the abstract domain $\mathsf{Bool}_{B(A)}$. A valuation may not correspond to a consistent quantifier-free formula (for example, $b_{x=0} = b_{x>0} = \mathtt{T}$). If the valuation $\mu \in Val_{B(A)}$ corresponds to an inconsistent quantifier-free formula (that is, $\gamma^*(\mu)$ is unsatisfiable), we simply answer *NO* to the membership query (line 1). Otherwise, we compare $\rho = t[\gamma^*(\mu)]$ with invariant approximations. Figure 4 shows the scenarios when queries can be answered by comparing ρ with invariant approximations. In case 4(a), $\rho \Rightarrow \bar{\iota}$ does not hold and we would like to show $\mu \not\models \lambda$. This requires the following lemma:

Lemma 3. *Let* $t[] \in \tau$ *be a template. For any* $\theta_1, \theta_2 \in \mathsf{Prop}_A, \theta_1 \Rightarrow \theta_2$ *implies* $t[\theta_1] \Rightarrow t[\theta_2]$.[1]

[1] Complete proofs are in [5]

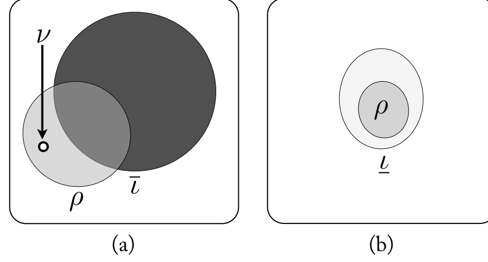

Fig. 4. Resolving a membership query with invariant approximations (c.f. Algorithm 2): (a) the guess ρ is not included in the over-approximation $\bar{\iota}$ (line 3); (b) the guess ρ is included in the under-approximation $\underline{\iota}$ (line 4).

By Lemma 3 and $t[\gamma^*(\mu)] \not\Rightarrow \bar{\iota}$ (line 3), we have $\gamma^*(\mu) \not\Rightarrow \gamma(\lambda)$. Hence $\mu \not\models \lambda$ (Lemma 2).

For case 4(b), we have $\rho \Rightarrow \underline{\iota}$ and would like to show $\mu \models \lambda$. However, the implication $t[\theta_1] \Rightarrow t[\theta_2]$ carries little information about the relation between θ_1 and θ_2. Consider $t[] \equiv \forall i.[]$, $\theta_1 \equiv i < 10$, and $\theta_2 \equiv i < 1$. We have $\forall i.i < 10 \Rightarrow \forall i.i < 1$ but $i < 10 \not\Rightarrow i < 1$. In order to infer more information from $\rho \Rightarrow \underline{\iota}$, we introduce a subclass of templates.

Definition 1. *Let $\theta \in \mathsf{Prop}_A$ be a quantifier-free formula over A. A* well-formed *template $t[]$ with respect to θ is defined as follows.*

- *$[]$ is well-formed with respect to θ;*
- *$\forall I.t'[]$ is well-formed with respect to θ if $t'[]$ is well-formed with respect to θ and $t'[\theta] \Rightarrow \forall I.t'[\theta]$;*
- *$\exists I.t'[]$ is well-formed with respect to θ if $t'[]$ is well-formed with respect to θ and $\neg t'[\theta]$.*

Using an SMT solver, it is straightforward to check if a template $t[]$ is well-formed with respect to a quantifier-free formula θ by a simple recursion. For instance, when the template is $\forall I.t'[]$, it suffices to check $SMT(t'[\theta] \wedge \exists I.\neg t'[\theta]) \to UNSAT$ and $t'[]$ is well-formed with respect to θ. More importantly, well-formed templates allow us to infer the relation between hole-filling quantifier-free formulae.

Lemma 4. *Let A be a set of atomic propositions, $\theta_1 \in \mathsf{Prop}_A$, and $t[] \in \tau$ a well-formed template with respect to θ_1. For any $\theta_2 \in \mathsf{Prop}_A$, $t[\theta_1] \Rightarrow t[\theta_2]$ implies $\theta_1 \Rightarrow \theta_2$.*

By Lemma 4 and 2, we have $\mu \models \lambda$ from $\rho \Rightarrow \underline{\iota}$ (line 4) and the well-formedness of $t[]$ with respect to $\gamma^*(\mu)$. As in the case of the equivalence query resolution algorithm, incorrect models from SMT solvers (line 3) simply guide the CDNF algorithm to other quantified invariants. Note that Algorithm 2 also gives a

random answer if a membership query cannot be resolved through invariant approximations. The correctness of generated invariants is ensured by SMT solvers in the equivalence query resolution algorithm (Algorithm 1).

5.3 Main Loop

Algorithm 3: Main Loop

Input: $\{\delta\}$ `while` κ `do` S $\{\epsilon\}$: an annotated loop; $t[]$: a template
Output: an invariant in the form of $t[]$

1 $\underline{\iota}$:= δ;
2 $\overline{\iota}$:= $\kappa \vee \epsilon$;
3 **repeat**
4   **try**
5   λ := **call** CDNF with query resolution algorithms (Algorithm 1 and 2)
6   **when inconsistent** $\rightarrow$ **continue**
7 **until** λ *is defined* ;
8 **return** $t[\gamma(\lambda)]$;

Algorithm 3 shows our invariant generation algorithm. It invokes the CDNF algorithm in the main loop. Whenever a query is made, our algorithm uses one of the query resolution algorithms (Algorithm 1 or 2) to give an answer. In both query resolution algorithms, we use the strongest under-approximation δ and the weakest over-approximation $\kappa \vee \epsilon$ to resolve queries from the learning algorithm. Observe that the equivalence and membership query resolution algorithms give random answers independently. They may send inconsistent answers to the CDNF algorithm. When inconsistencies arise, the main loop forces the learning algorithm to restart (line 6). If the CDNF algorithm infers a Boolean formula $\lambda \in \mathsf{Bool}_{B(A)}$, the first-order formula $t[\gamma(\lambda)]$ is an invariant for the annotated loop in the form of the template $t[]$.

In contrast to traditional deterministic algorithms, our algorithm gives random answers in both query resolution algorithms. Due to the undecidability of first-order theories in SMT solvers, verifying quantified invariants and comparing invariant approximations are not solvable in general. If we committed to a particular solution deterministically, we would be forced to address computability issues. Random answers simply divert the learning algorithm to search for other quantified invariants and try the limit of SMT solvers. They could not be effective if there were very few solutions. Our thesis is that there are sufficiently many invariants for any given annotated loop in practice. As long as our random answers are consistent with one verifiable invariant, the CDNF algorithm is guaranteed to generate an invariant for us.

Similar to other invariant generation techniques based on predicate abstraction, our algorithm is not guaranteed to generate invariants. If no invariant can be expressed by the template with a given set of atomic propositions, our algorithm will not terminate. Moreover, if no invariant in the form of the given template can be verified by SMT solvers, our algorithm does not terminate either. On the

case	Template	AP	MEM	EQ	MEM_R	EQ_R	$ITER$	Time (s)	σ_{Time}(s)
`max`	$\forall k.[]$	7	5,968	1,742	65%	26%	269	5.71	7.01
`selection_sort`	$\forall k_1.\exists k_2.[]$	6	9,630	5,832	100%	4%	1,672	9.59	11.03
`devres`	$\forall k.[]$	7	2,084	1,214	91%	21%	310	0.92	0.66
`rm_pkey`	$\forall k.[]$	8	2,204	919	67%	20%	107	2.52	1.62
`tracepoint1`	$\exists k.[]$	4	246	195	61%	25%	31	0.26	0.15
`tracepoint2`	$\forall k_1.\exists k_2.[]$	7	33,963	13,063	69%	5%	2,088	157.55	230.40

Table 1. Experimental Results.
AP : # of atomic propositions, MEM : # of membership queries, EQ : # of equivalence queries, MEM_R : fraction of randomly resolved membership queries to MEM, EQ_R fraction of randomly resolved equivalence queries to EQ, $ITER$: # of the CDNF algorithm invocations, and σ_{Time} : standard deviation of the running time.

other hand, if there is one verifiable invariant in the form of the given template, there is a sequence of random answers that leads to the verifiable invariant. If sufficiently many verifiable invariants are expressible in the form of the template, random answers almost surely guide the learning algorithm to one of them. Since our algorithmic learning approach with random answers does not commit to any particular invariant, it can be more flexible and hence effective than traditional deterministic techniques in practice.

6 Experiments

We have implemented a prototype[2] in OCaml. In our implementation, we use Yices as the SMT solver to resolve queries (Algorithm 1 and 2). Table 1 shows experimental results. We took two cases from the ten benchmarks in [20] with the same annotation (`max` and `selection_sort`). We also chose four `for` statements from Linux 2.6.28. We translated them into our language and annotated pre- and post-conditions manually. Sets of atomic proposition are manually chosen from the program texts. Benchmark `devres` is from library, `tracepoint1` and `tracepoint2` are from kernel, and `rm_pkey` is from InfiniBand device driver. The data are the average of 500 runs and collected on a 2.66GHz Intel Core2 Quad CPU with 8GB memory running Linux 2.6.28.

`devres` from Linux Library Figure 5(c) shows an annotated loop extracted from a Linux library.[3] In the postcondition, we assert that ret implies $tbl[i] = 0$, and every element in the array $tbl[]$ is not equal to $addr$ otherwise. Using the set of atomic propositions $\{tbl[k] = addr, i < n, i = n, k < i, tbl[i] = 0, ret\}$ and the simple template $\forall k.[]$, our algorithm finds following quantified invariants in different runs:

$$\forall k.(k < i \Rightarrow tbl[k] \neq addr) \wedge (ret \Rightarrow tbl[i] = 0) \text{ and } \forall k.(k < i) \Rightarrow tbl[k] \neq addr.$$

[2] Available at `http://ropas.snu.ac.kr/aplas10/qinv-learn-released.tar.gz`
[3] The source code can be found in function `devres` of `lib/devres.c` in Linux 2.6.28

(a) rm_pkey
```
{ i = 0 ∧ key ≠ 0 ∧ ¬ret ∧ ¬break}
1 while(i < n ∧ ¬break) do
2    if(pkeys[i] = key) then
3       pkeyrefs[i]:=pkeyrefs[i] − 1;
4       if(pkeyrefs[i] = 0) then
5          pkeys[i]:=0; ret:=true;
6       break:=true;
7    else i:=i + 1;
8 done
{(¬ret ∧ ¬break) ⇒ (∀k.k < n ⇒ pkeys[k] ≠ key)
  ∧(¬ret ∧ break) ⇒ (pkeys[i] = key ∧ pkeyrefs[i] ≠ 0)
  ∧ ret ⇒ (pkeyrefs[i] = 0 ∧ pkeys[i] = 0) }
```

(c) devres
```
{ i = 0 ∧ ¬ret }
1 while i < n ∧ ¬ret do
2   if tbl[i] = addr then
3      tbl[i]:=0; ret:=true
4   else
5      i:=i + 1
6 end
{(¬ret ⇒ ∀k. k < n ⇒ tbl[k] ≠ addr)
  ∧(ret ⇒ tbl[i] = 0) }
```

(b) selection_sort
```
  { i = 0 }
 1 while i < n − 1 do
 2    min:=i;
 3    j:=i + 1;
 4    while j < n do
 5       if a[j] < a[min] then
 6          min:=j;
 7       j:=j + 1;
 8    done
 9    if i≠min then
10       tmp:=a[i];
11       a[i]:=a[min];
12       a[min]:=tmp;
13    i:=i + 1;
14 done
  {(i ≥ n − 1)
   ∧ (∀k1.k1 < n ⇒
      (∃k2.k2 < n ∧ a[k1] = ′a[k2]))}
```

Fig. 5. Benchmark Examples: (a) `rm_pkey` from Linux InfiniBand driver, (b) `selection_sort` from [20], and (c) `devres` from Linux library.

Observe that our algorithm is able to infer an arbitrary quantifier-free formula (over a fixed set of atomic propositions) to fill the hole in the given template. A simple template such as $\forall k.[]$ suffices to serve as a hint in our approach.

`selection_sort` from [20] Consider the selection sort algorithm in Figure 5(b). Let $'a[]$ denote the content of the array $a[]$ before the algorithm is executed. The postcondition states that the contents of array $a[]$ come from its old contents. In this test case, we apply our invariant generation algorithm to compute an invariant to establish the postcondition of the outer loop. For computing the invariant of the outer loop, we make use of the inner loop's specification.

We use the following set of atomic propositions: $\{k_1 \geq 0,\ k_1 < i,\ k_1 = i,\ k_2 < n,\ k_2 = n,\ a[k_1] = {'a}[k_2],\ i < n-1,\ i = min\}$. Using the template $\forall k_1.\exists k_2.[]$, our algorithm infers following invariants in different runs:

$$\forall k_1.(\exists k_2.[(k_2 < n \wedge a[k_1] = {'a}[k_2]) \vee k_1 \geq i]); \text{and}$$
$$\forall k_1.(\exists k_2.[(k_1 \geq i \vee min = i \vee k_2 < n) \wedge (k_1 \geq i \vee (min \neq i \wedge a[k_1] ={'} a[k_2]))]).$$

Note that all membership queries are resolved randomly due to the alternation of quantifiers in array theory. Still a simple random walk suffices to find invariants in this example. Moreover, templates allow us to infer not only universally quantified invariants but also first-order invariants with alternating quantifications. Inferring arbitrary quantifier-free formulae over a fixed set of atomic propositions again greatly simplifies the form of templates used in this example.

rm_pkey from Linux InfiniBand Driver Figure 5(a) is a `while` statement extracted from Linux InfiniBand driver.[4] The conjuncts in the postcondition represent (1) if the loop terminates without break, all elements of *pkeys* are not equal to *key* (line 2); (2) if the loop terminates with break but *ret* is false, then $pkeys[i]$ is equal to *key* (line 2) but $pkeyrefs[i]$ is not equal to zero (line 4); (3) if *ret* is true after the loop, then both $pkeyrefs[i]$ (line 4) and $pkeys[i]$ (line 5) are equal to zero. From the postcondition, we guess that an invariant can be universally quantified with k. Using the simple template $\forall k.[]$ and the set of atomic propositions $\{ret,\ break,\ i < n,\ k < i,\ pkeys[i] = 0,\ pkeys[i] = key,\ pkeyrefs[i] = 0,\ pkeyrefs[k] = key\}$, our algorithm finds following quantified invariants in different runs:

$$
\begin{gathered}
(\forall k.(k < i) \Rightarrow pkeys[k] \neq key) \wedge (ret \Rightarrow pkeyrefs[i] = 0 \wedge pkeys[i] = 0) \\
\wedge\ (\neg ret \wedge break \Rightarrow pkeys[i] = key \wedge pkeyrefs[i] \neq 0);\ \text{and} \\
(\forall k.(\neg ret \vee \neg break \vee (pkeyrefs[i] = 0 \wedge pkeys[i] = 0)) \wedge (pkeys[k] \neq key \vee k \geq i) \\
\wedge(\neg ret \vee (pkeyrefs[i] = 0 \wedge pkeys[i] = 0 \wedge i < n \wedge break)) \\
\wedge\ (\neg break \vee pkeyrefs[i] \neq 0 \vee ret) \wedge (\neg break \vee pkeys[i] = key \vee ret)).
\end{gathered}
$$

In spite of undecidability of first-order theories in YICES and random answers, each of the 3000 ($= 6 \times 500$) runs in our experiments infers an invariant successfully. Moreover, several quantified invariants are found in each case among 500 runs. This suggests that invariants are abundant. Note that the templates in the test cases `selection_sort` and `tracepoint2` have alternating quantification. Satisfiability of alternating quantified formulae is in general undecidable. That is why both cases have substantially more restarts than the others. Interestingly, our algorithm is able to generate a verifiable invariant in each run. Our simple randomized mechanism proves to be effective even for most difficult cases.

7 Related Work

Comparing with the work [15] of generating quantifier-free invariants, we develop the following technical extensions. First, we integrate potential counterexamples in resolving equivalence query algorithm (line 6 - 7 in Algorithm 1, and line 3 in Algorithm 2) instead of restarting. Due to the undecidability of satisfiability of quantified formulae, SMT solvers often give potential counterexamples. We exploit potential counterexamples to enhance our algorithm. Second, a new condition (Definition 1) to answer positively in resolving membership queries is proposed. Without this condition, we can answer negatively to membership queries.

In contrast to previous template-based approaches [20,9], our template is more general as it allows arbitrary hole-filling quantifier-free formulae. The templates in [20] can only be filled with formulae over conjunctions of predicates from a given set. Any disjunction must be explicitly specified as part of a template. In [9], the authors consider invariants of the form $E \wedge \bigwedge_{j=1}^{n} \forall U_j(F_j \Rightarrow e_j)$, where E, F_j and e_j must be quantifier free finite conjuctions of atomic facts.

[4] The source code can be found in function `rm_pkey` of `drivers/infiniband/hw/ipath/ipath_mad.c` in Linux 2.6.28

Existing technologies can strengthen our framework. Firstly, its completeness can be increased by more powerful decision procedures [6,8,21] and theorem provers [18,1,19]. Moreover, our approach can be improved by using more accurate approximations from existing invariant generation techniques. The tool InvGen collects reached states satisfying the program invariants, and also computes a collection of invariants for efficient invariant generation [11]. They can be used as under- and over-approximations, respectively.

Regarding the generation of unquantified invariants, a constraint analysis approach is proposed in [10]. Invariants in the combined theory of linear arithmetic and uninterpreted functions are synthesized in [2], while InvGen [11] presents an efficient approach for linear arithmetic invariants. For quantified loop invariants, Skolemization is used for generating universally quantified invariants [7]. In [18], a paramodulation-based saturation prover is extended to generate universally quantified invariants by interpolation.

With respect to the analysis of properties of array contents, Halbwachs et al. [12] handle programs which manipulate arrays by sequential traversal, incrementing (or decrementing) their index at each iteration, and which access arrays by simple expressions of the loop index. A loop property generation method for loops iterating over multi-dimensional arrays is introduced in [13]. For inferring range predicates, Jhala and McMillan [14] described a framework that uses infeasible counterexample paths. As a deficiency, the prover may find proofs refuting short paths, but which do not generalize to longer paths. Due to this problem, this approach [14] fails to prove that an implementation of insertion sort correctly sorts an array.

8 Conclusions

By combining algorithmic learning, decision procedures, predicate abstraction, and templates, we present a technique for generating quantified invariants. The new technique searches for invariants in the given template form guided by query resolution algorithms. We exploit the flexibility of algorithmic learning by deploying a randomized query resolution algorithm. When there are sufficiently many invariants, random answers will not prevent algorithmic learning from inferring verifiable invariants. Our experiments show that our learning-based approach is able to infer non-trivial quantified invariants with this naïve randomized resolution for some loops extracted from Linux drivers.

Under- and over-approximations are presently derived from annotations provided by users. They can in fact be obtained by other techniques such as static analysis. For deciding the set of atomic propositions, it will be interesting whether existing techniques [4,17] are applicable. The integration of various refinement techniques for predicate abstraction will certainly be an important future work.

Acknowledgment We are grateful to Wontae Choi, Suwon Jang, Will Klieber, Wonchan Lee, Ben Lickly, Bruno Oliveira, and Sungwoo Park for their detailed comments and helpful suggestions. We also thank Heejae Shin for implementing OCaml binding for Yices.

References

1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)
2. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: VMCAI. (2007) 378–394
3. Bshouty, N.H.: Exact learning boolean functions via the monotone theory. Information and Computation **123** (1995) 146–153
4. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. Volume 1855 of LNCS., Springer (2000) 154–169
5. David, C., Jung, Y., Kong, S., Wang, B.Y., Yi, K.: Inferring quantified invariants via algorithmic learning, decision procedure, and predicate abstraction. Technical Memorandum ROSAEC-2010-007, Research On Software Analysis for Error-Free Computing (2010)
6. Dutertre, B., Moura, L.D.: The Yices SMT solver. Technical report, SRI International (2006)
7. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, ACM (2002) 191–202
8. Ge, Y., Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: CAV. Volume 5643 of LNCS., Springer-Verlag (2009) 306–320
9. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, ACM (2008) 235–246
10. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: VMCAI. Volume 5403 of LNCS., Springer (2009) 120–135
11. Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: CAV. Volume 5643 of LNCS., Springer (2009) 634–640
12. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI. (2008) 339–348
13. Henzinger, T.A., Hottelier, T., Kovács, L., Voronkov, A.: Invariant and type inference for matrices. In: VMCAI. (2010) 163–179
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. Kroening, D., Strichman, O.: Decision Procedures an algorithmic point of view. EATCS. Springer (2008)
17. McMillan, K.L.: Lazy abstraction with interpolants. In Ball, T., Jones, R.B., eds.: CAV. Volume 4144 of LNCS., Springer (2006) 123–136
18. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: TACAS. Volume 4693 of LNCS., Springer (2008) 413–427
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
20. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI, ACM (2009) 223–234
21. Srivastava, S., Gulwani, S., Foster, J.S.: VS3: SMT solvers for program verification. In: CAV. Volume 5643 of LNCS. (2009) 702–708