ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 KOSEF ERC

ROSAEC MEMO
2010-007
January 17, 2010

Inferring Quantified Invariants via Algorithmic Learning, Decision Procedures, and Predicate Abstraction

Cristina David
National University of Singapore
davidcri@comp.nus.edu.sg

Yungbum Jung
Seoul National University
dreameye@ropas.snu.ac.kr

Soonho Kong
Seoul National University
soon@ropas.snu.ac.kr

Bow-Yaw Wang
Academia Sinica, INRIA,
and Tsinghua University
bywang@iis.sinica.edu.tw

Kwangkeun Yi
Seoul National University
kwang@ropas.snu.ac.kr

January 17, 2010

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 the same derivation power.

1 Introduction

Recently, algorithmic learning has been successfully applied to invariant generation [15]. It has been shown that a combination of algorithmic learning, decision procedures, and predicate abstraction can automatically generate invariants for realistic C loops (such as those in Linux device drivers) within a practical cost bound. The work [15] has, however, one obvious limitation; it can only generate propositional, quantifier-free formulae. Yet loops that iterate over aggregate data structures (such as arrays and graphs) often have arbitrarily quantified invariants over the aggregate elements.

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

- We show that a simple-minded 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, 19] yet is much simpler.
- The technique needs very simple templates. It is enough for the templates to specify which variables are existentially or universally quantified, leaving the invariant body part as a single hole (such as "$\forall k.[]$" or "$\forall k.\exists i.[]$").

Our technique finds arbitrary first-order invariants (modulo the underlying SMT solver and a fixed set of atomic propositions) in the form of the given template. We deploy a learning algorithm to infer quantifier-free formulae (as in [15]) that fill in the template's hole to give quantified formulae.

The hole-filling quantifier-free formulae can be in any form. This generality contrasts with existing template-based approach [19] where only non-disjunctive formulae can fill the holes.

- 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 [19].
- 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.

Our algorithm works as follows. For a given loop annotated with its pre- and post-conditions, an exact learning algorithm for Boolean formulae searches for quantifier-free formulae to fill into the given quantified template by asking queries. Because the learning algorithm generates only Boolean formulae but decision procedures, which has to answer the queries, should work in quantifier-free formula, we use predicate abstraction and concretization to resolve queries with decision procedures. In reality, because knowledge (over/under approximations of invariants) about loop invariants is incomplete, queries may not be resolvable. When query resolution requires knowledge unavailable to decision procedures, we simply give a random answer. We surely could use static analysis to compute soundly approximated information other than random answers. Yet, because there are so many invariants for the given annotated loop, providing random answers can make the algorithm deduce different invariants or simply restart in the worst case. In contrast, traditional invariant generation techniques do not have this flexibility; they are rather fixed (by their custom algorithms) to chase one particular invariant.

Let us first illustrate these features with an example for generating propositional invariants.

Example 1. Consider the following code from [15]:

$$\{i = 0\}\ \texttt{while}\ i < 10\ \texttt{do}\ b := \texttt{nondet};\ \texttt{if}\ b\ \texttt{then}\ i := i + 1\ \texttt{end}\ \{i = 10\ \wedge\ b\}$$

Observe that the variable b must be true after the while loop. As under- and over-approximations to invariants $i = 0$ and $(i = 10 \wedge b) \vee i < 10$ are chosen respectively. A decision procedure (an SMT solver) uses these approximations on invariants to resolve queries from the learning algorithm. After a number of queries, the learning algorithm asks whether $i \neq 0 \wedge i < 10 \wedge \neg b$ should be included in the invariant. Because the query is neither stronger than the under-approximation nor weaker than the over-approximation, the decision procedure fails to resolve the query. At this point, we simply give a random answer. In case of an incorrect answer, the learning algorithm will ask us to give a counterexample to its best guess $i = 0 \vee (i = 10 \wedge b)$. Since the guess is not an invariant and coin tossing does not generate a counterexample, we restart the learning process. On the other hand, if the coin tossing answered correctly, the learning algorithm infers the invariant $(i = 10 \wedge b) \vee i < 10$ with after resolving two additional queries.

The next example illustrates a quantified invariant case.

Example 2.

$$\texttt{while}\ i < n\ \texttt{do if}\ a[m] < a[i]\ \texttt{then}\ m = i\ \texttt{fi};\ i = i + 1\ \texttt{end}$$

This simple loop is annotated with the precondition $m = 0 \wedge i = 0$ and postcondition $\forall k.0 \leq k < n \Rightarrow a[k] \leq a[m]$. It examines $a[0]$ through $a[n-1]$ and finds the index of the maximal element in the array. We give template $\forall k.[]$ and the following set of atomic propositions (building blocks):

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

Note that all atomic propositions except $k = i$ and $k < i$ are extracted from the annotated loop. The template introduces a new variable k. It is natural to relate k and the program variables i and n by adding atomic propositions $k = i$, $k < i$ and $k < n$. With these inputs, our technique looks for an invariant ι of the form $\forall k.[]$ such that (1) $(m = 0 \wedge i = 0) \Rightarrow \iota$; and (2) $(\iota \wedge \neg(i < n)) \Rightarrow \forall k.0 \leq k < n \Rightarrow a[k] \leq a[m]$. That is, we would like to find a quantifier-free formula θ such that (1) $(m = 0 \wedge i = 0) \Rightarrow \forall k.\theta$; and (2) $\forall k.\theta \Rightarrow (i < n \vee \forall k.0 \leq k < n \Rightarrow a[k] \leq a[m])$.

Applying algorithmic learning with coin tossing from time to time, our technique successfully generates an invariant : $\forall k.(k \not< i) \vee (a[k] \leq a[m])$. This is of course not the only invariant that our algorithm can generate. Indeed, another separate run of our algorithm generates $\forall k.(i = 0 \wedge k \not< n) \vee (a[k] \leq a[m]) \vee (k \not< i)$ which is another valid loop invariant.

We organize this paper as follows. After preliminaries in Section 2, we present an overview of our framework in Section 3. The details of our technique are described in Section 4. We report experiments in Section 5, discuss related work in Section 6, then conclude in Section 7.

2 Preliminaries

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

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

The language has two basic types: Boolean and natural numbers. A term in Exp is a natural number; a term in Prop is of Boolean type. A variable is assigned to an arbitrary value in its type by the keyword `nondet`. In an annotated loop $\{\delta\}$ `while` κ `do` S $\{\epsilon\}$, κ is its *guard*, δ and ϵ are its *precondition* and *postcondition* respectively. Pre- and post-conditions of annotated loops are terms in Pred, first-order formula. Propositional 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 with 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 prenex normal form. Any first-order formula can be expressed by filling the hole in a proper template.

Let $\{\delta\}$ `while` κ `do` S $\{\epsilon\}$ be an annotated loop and $t[] \in \tau$ be a 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. The *invariant generation problem with template* $t[]$

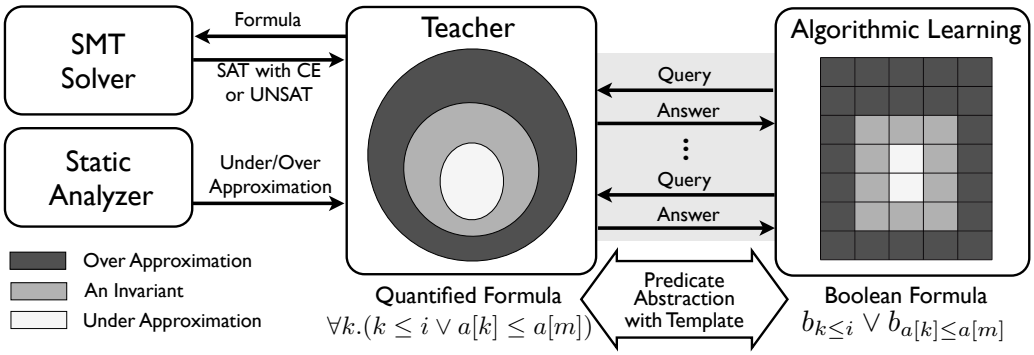

Figure 1: Our framework

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) $\kappa \wedge t[\theta] \Rightarrow Pre(t[\theta], S)$.

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* [5, 16] returns a (potential) model of ν if it exists (written $SMT(\rho) \rightarrow \nu$). In general SMT solver is not complete over quantified formulae and therefore returns a potential model. It returns *UNSAT* (written $SMT(\rho) \rightarrow UNSAT$) if the solver proves the formula unsatisfiable.

2.0.1 CDNF Learning Algorithm

[3] The CDNF (Conjunctive Disjunctive Normal Form) algorithm is an exact algorithm that computes a representation for any Boolean formula $\lambda \in \mathsf{Bool}_B$ by interacting with a *teacher*. Teacher should 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 Framework Overview

We combine algorithmic learning [3], decision procedures [5], predicate abstraction [8], and templates in our framework. Figure 1 illustrates the relation among these technologies. The

left side (teacher, SMT solver, and static analyzer) represents the concrete domain working with quantified formulae, whereas the right side (algorithmic learning) denotes the abstract domain manipulating Boolean formulae.

Given an annotated loop and a template, our goal is to apply the algorithmic learning to find an invariant in the form of the given template. To achieve this goal, we need to address two problems. First, the CDNF algorithm is a learning algorithm for Boolean formula, not quantified formula. Second, the CDNF algorithm assumes a teacher who knows the target in its learning model. In order to automatically compute invariants, we have to design a mechanical procedure to play the role of a teacher.

For the first problem, we use the predicate abstraction and a template to relate Boolean formulae with quantified formulae. In predicate abstraction, an atomic proposition corresponds to a Boolean variable. Instead of inferring a proper quantifier-free formula for the hole in the template, we will use the CDNF algorithm to deduce a proper Boolean formula λ in the abstract domain. The corresponding first-order invariant $t[\gamma(\lambda)]$ is obtained by concretizing the found formula λ and filling the hole in the template $t[]$.

For the second problem, we need to design algorithms to resolve queries about the Boolean formula λ in the previous paragraph. There are two types of queries: membership queries ask whether a Boolean valuation is a model of an invariant; equivalence queries ask whether a Boolean formula is an invariant and demand a counterexample if it is not. Without knowing an invariant, the teacher should answer queries. With under/over approximations provided by static analyzers or derived from the pre- and post-conditions of the annotated loop, the teacher resolves queries by resorting to these under/over approximations. An SMT solver is deployed to prove satisfiability of formulae.

If a query cannot be resolved by invariant approximations, our algorithm simply gives a random answer to the CDNF algorithm. For equivalence query, we check if the guess is an invariant. If the concretization is not weaker than the under-approximation or not stronger than the over-approximation, a counterexample can be generated by an SMT solver. Otherwise, the learning process gives a random counterexample. For a membership query, we check if its concretization is in the under-approximation or outside the over-approximation by an SMT solver. If it is in the under-approximation, the answer is affirmative; if it is out of the over-approximation, the answer is negative. Otherwise, we simply give a random answer. If there are sufficiently many invariants, our simple randomized resolution algorithms will guide the CDNF algorithm to one of them.

4 Learning Quantified Invariants

Our goal is to infer quantified invariants through algorithmic learning with templates. For this goal, we will (1) identify correspondences between the three domains of interest (Section 4.1); (2) present the main loop of our framework (Section 4.2); (3) design query resolution algorithms for algorithmic learning (Section 4.3 and Section 4.4). (4) develop the technical lemmas for the sound membership query resolution (Section 4.5).

4.1 Predicate Abstraction with Templates

Let A be a set of atomic propositions and $B(A) \triangleq \{b_p : p \in A\}$ the set of corresponding Boolean variables. Figure 2 shows the domains used in our algorithm. The left box represents the class Pred_A of first-order formulae generated from A. Here, we are interested in the class of first-order formulae in the form of a given template $t[] \in \tau$. Thus, the subclass $\mathcal{S}_{t[]} \triangleq \{t[\theta] : \theta \in \mathsf{Prop}_A\} \subseteq \mathsf{Pred}_A$ forms the *solution space* of the invariant generation problem with the template $t[]$. The middle box corresponds to the class Prop_A of quantifier-free formulae generated from A. Since the solution space $\mathcal{S}_{t[]}$ is generated by the fixed template $t[]$, Prop_A is

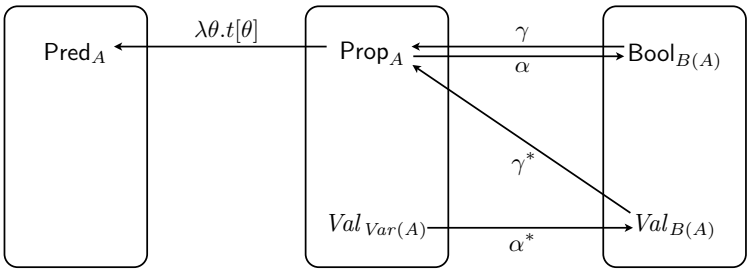

Figure 2: The domains Pred_A, Prop_A, and $\mathsf{Bool}_{B(A)}$

in fact the essence of $\mathcal{S}_{t[]}$. The right box contains the class $\mathsf{Bool}_{B(A)}$ of 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 $\theta \in \mathsf{Prop}_A$.

To answer membership queries (Section 4.4), we relate a Boolean valuation $\mu \in Val_{B(A)}$ 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 $\alpha^*(\nu) \in Val_{B(A)}$. It is useful in finding counterexamples for equivalence queries (Section 4.3).

$$\begin{aligned} \gamma^*(\mu) &= \bigwedge_{p \in A} \{p : \mu(b_p) = \mathtt{T}\} \wedge \bigwedge_{p \in A} \{\neg p : \mu(b_p) = \mathtt{F}\} \\ (\alpha^*(\nu))(b_p) &= \begin{cases} \mathtt{T} & \text{if } \nu \models p \\ \mathtt{F} & \text{otherwise} \end{cases} \end{aligned}$$

The following lemmas characterize relations between these functions:

Lemma 4.1 ([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 4.2 ([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 $\mu \models \alpha(\theta)$.*

4.2 Main Loop

Given an annotated loop $\{\delta\}$ `while` κ `do` S $\{\epsilon\}$ and a template $t[] \in \tau$, we want to find an invariant $\iota \in \mathsf{Pred}_A$. We say $\underline{\iota} \in \mathsf{Pred}_A$ is an *under-approximation* to the invariant ι if $\delta \Rightarrow \underline{\iota}$ and $\underline{\iota} \Rightarrow \iota$. Similarly, $\overline{\iota} \in \mathsf{Pred}_A$ is an *over-approximation* to the invariant ι if $\iota \Rightarrow \overline{\iota}$ and

Algorithm 1: Main Loop

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

```
1 ι̲ := δ;
2 ῑ := κ ∨ ϵ;
3 repeat
4     try
5         λ := call CDNF with query resolution algorithms (Algorithm 2 and 3)
6     when abort → continue
7 until λ is defined ;
8 return t[γ(λ)];
```

$\overline{\iota} \Rightarrow \epsilon \vee \kappa$. The *strongest* ($\delta$) and *weakest* ($\epsilon \vee \kappa$) approximations are trivial under- and over-approximations to any invariant respectively. In the following discussion, λ is the unknown target Boolean formula. We assume $\iota = t[\gamma(\lambda)]$ is a first-order invariant.

Algorithm 1 shows our invariant generation algorithm. The target Boolean function λ is unknown. In order to design query resolution algorithms without knowing λ, we resort to invariant approximations. We use the under-approximation δ and over-approximation $\kappa \vee \epsilon$ to resolve queries from the CDNF algorithm. 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 $t[]$ (Algorithm 1).

The CDNF algorithm uses the equivalence query resolution algorithm (Algorithm 2) and the membership query resolution algorithm (Algorithm 3) to resolve queries. When a query cannot be resolved decisively, our query resolution algorithm may give a random answer. Since the equivalence query resolution algorithm uses an SMT solver to *verify* the found first-order formula is indeed an invariant. Random answers do not yield incorrect results. On the other hand, random answers allow our algorithm to explore the multitude of invariants. If there are numerous first-order invariants in the form of the given template, random answers will not prevent our algorithm from hitting one of them. Indeed, our experiments suggest that our simple randomized algorithm can find different first-order invariants in different runs. However our algorithm may fail when conflicts occur as too many wrong random answers are accumulated. Then CDNF algorithm is restarted.

Our algorithm does not guarantee its termination. If invariants cannot be expressed in the form of provided template, the algorithm goes into an infinite loop. The algorithm fails to find an invariant if a given set of atomic propositions is not sufficient to compose an invariant. Due to the incompleteness of SMT solvers over quantified formulae, our algorithm could fail to recognize an invariant and leads to non-termination as a result.

4.3 Equivalence Queries

An equivalence query $EQ(\beta)$ with $\beta \in \mathsf{Bool}_{B(A)}$ asks if β is equivalent to λ. To answer an equivalence query $EQ(\beta)$, we check if $t[\gamma(\beta)]$ is indeed an invariant of the annotated loop. If it is, we are done. Otherwise, our equivalence query resolution algorithm finds a counterexample to distinguish λ from β by comparing $t[\gamma(\beta)]$ with invariant approximations.

Algorithm 2 gives our equivalence resolution algorithm. It first checks if $\rho = t[\gamma(\beta)]$ is indeed an invariant for the annotated loop by verifying $\underline{\iota} \Rightarrow \rho$, $\rho \Rightarrow \overline{\iota}$, and $\kappa \wedge \rho \Rightarrow \mathit{Pre}(\rho, S)$ with an SMT solver (line 2 and 3 in Algorithm 2). 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

Algorithm 2: Resolving Equivalence Queries

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** $SMT(\underline{\iota} \wedge \neg\rho) \to UNSAT$ **and** $SMT(\rho \wedge \neg\bar{\iota}) \to UNSAT$ **and**
3 $SMT(\theta \wedge \rho \wedge \neg Pre(\rho, S)) \to UNSAT$ **then return** *YES*;
4 **if** $SMT(\underline{\iota} \wedge \neg\rho) \to \nu$ **then return** $\alpha^*(\nu)$;
5 **if** $SMT(\rho \wedge \neg\bar{\iota}) \to \nu$ **then return** $\alpha^*(\nu)$;
6 **if** $SMT(\rho \wedge \neg\underline{\iota}) \to \nu_0$ **or** $SMT(\bar{\iota} \wedge \neg\rho) \to \nu_1$ **then**
7 **return** $\alpha^*(\nu_0)$ *or* $\alpha^*(\nu_1)$ *randomly*;
8 **abort**;

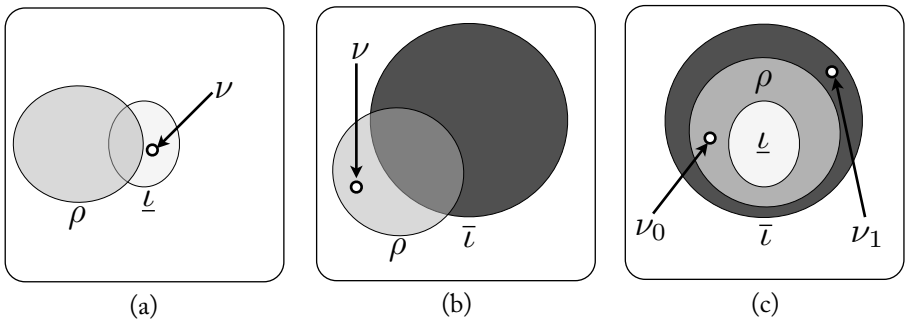

Figure 3: Finding a counterexample in equivalence query resolution: (a) SMT solver finds a counterexample which is a model for the under-approximation $\underline{\iota}$ but not for an invariant candidate ρ (line 4 in Algorithm 2); (b) SMT solver finds a counterexample which is a model for an invariant candidate ρ but not for the over-approximation $\bar{\iota}$ (line 5 in Algorithm 2); (c) Otherwise, the algorithm guesses a counterexample ν_0 (or ν_1) which is a model for the candidate ρ (or over-approximation $\bar{\iota}$) but not for under-approximation $\underline{\iota}$ (or candidate ρ), respectively (line 6 and 7 in Algorithm 2).

to return a valuation distinguishing ρ from invariant approximations as a random answer (c). However, it is possible to have *UNSAT* results for all the SMT queries through lines 4 - 6. Then we abort the equivalence query resolution (line 8 in Algorithm 2). This failure will make the main function restart the CDNF algorithm.

4.4 Membership Queries

In a membership query $MEM(\mu)$, our membership query resolution algorithm (Algorithm 3) 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 (e.g., $x = 0 \wedge x < 0$). 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 in Algorithm 3). Otherwise, we compare $\rho = t[\gamma^*(\mu)]$ with invariant approximations.

Figure 4 shows the two cases when the queries can be answered by comparing ρ with approximations. In case (a), if $\rho \Rightarrow \bar{\iota}$ does not hold, the algorithm returns *NO*. The soundness of this answer is guaranteed by Lemma 4.3. In case (b), we check if $\rho \Rightarrow \underline{\iota}$ holds. If it holds then we want to answer *Yes*. However soundness of this answer requires another condition.

Algorithm 3: Resolving Membership Queries

Input: a valuation μ for $B(A)$
Output: *YES* or *NO*
1 **if** $SMT(\gamma^*(\mu)) \to UNSAT$ **then return** *NO*;
2 $\rho := t[\gamma^*(\mu)]$;
3 **if** $SMT(\rho \wedge \neg\bar{\iota}) \to \nu$ **then return** *NO*;
4 **if** $isWellFormed(t[], \gamma^*(\mu))$ **and** $SMT(\rho \wedge \neg\underline{\iota}) \to UNSAT$ **then return** *YES*;
5 **return** *YES or NO randomly*

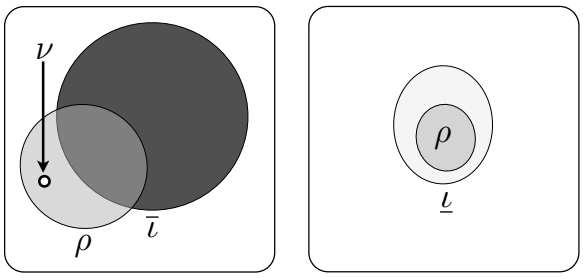

Figure 4: The membership query resolution can be resolved by invariant approximations: (a) the guess ρ is not included in the over-approximation $\bar{\iota}$ (line 3 in Algorithm 3); (b) the guess ρ is included in the under-approximation $\underline{\iota}$ (line 4 in Algorithm 3).

The inclusion relation in concrete domain does not always guarantee the inclusion relation in abstract domain. For example, consider the template $\forall i.[]$, $\theta_1 = i < 10$, and $\theta_2 = i < 1$. We have $\forall i.i < 10 \Rightarrow \forall i.i < 1$ but $i < 10 \Rightarrow i < 1$ does not hold. Indeed the soundness of this answer is established by Lemma 4.4 and well-formedness check is defined in Section 4.5. Note that Algorithm 3 will give a random answer if a membership query cannot be resolved.

Since answers to equivalence queries and to membership queries are generated independently, inconsistencies between these two types of answers can occur. Our invariant generation algorithm simply restarts when an inconsistent answer is observed.

4.5 Template Properties

Let $t[] \in \tau$ be a template and $\theta \in \mathsf{Prop}_A$ a quantifier-free formula. If $t[\theta]$ is stronger than under-approximation, we would like to relax $t[\theta]$ by changing θ. This motivates our investigations on properties about templates. First, templates are monotonic. More precisely, they have the following property.

Lemma 4.3. *Let $t[] \in \tau$ be a template. For any $\theta_1, \theta_2 \in \mathsf{Prop}_A, \theta_1 \Rightarrow \theta_2$ implies $t[\theta_1] \Rightarrow t[\theta_2]$.*

The contraposition of this Lemma 4.3 shows the soundness of our membership query resolution (line 3 in Algorithm 3). We can answer *NO* in the membership query resolution because $t[\theta_1] \not\Rightarrow t[\theta_2]$ implies $\theta_1 \not\Rightarrow \theta_2$. For the other resolution (line 4 in Algorithm 3), we need to check well-formedness of template $t[]$ and quantifier-free formula θ:

Definition 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;

case	Template	AP	MEM	EQ	MEM_R	EQ_R	$RESTART$	Time (sec)
`max`	$\forall k.[]$	7	15	9	0	1	2	0.05
`selection_sort`	$\forall k_1.\exists k_2.[]$	6	12021	7396	12021	223	2158	11.11
`devres`	$\forall k.[]$	7	1854	1078	1693	232	275	0.84
`rm_pkey`	$\forall k.[]$	8	2477	1034	1661	202	121	2.72
`tracepoint1`	$\exists k.[]$	4	260	209	158	53	33	0.266
`tracepoint2`	$\forall k_1.\exists k_2.[]$	7	35239	13538	24359	642	2168	167.59

Table 1: Experimental Results.
AP : # of atomic propositions, MEM : # of membership queries, EQ : # of equivalence queries, MEM_R : # of randomly resolved membership queries, EQ_R # of randomly resolved equivalence queries, and $RESTART$: # of the CDNF algorithm invocations.

- $\forall I.t'[]$ 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 $\neg t'[\theta]$.

The intuition behind the well-formed templates is to give a sufficient condition to infer properties about a quantifier-free formula from the corresponding first-order formula in the form of well-formed templates.

Lemma 4.4. *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 $\theta_1 \Rightarrow \theta_2$.*

Proof. By induction on $t[]$. For the basis ($t[] \equiv []$), this is trivial.

Assume $t[]\forall I.t'[]$ and $\forall I.t[\theta_1] \Rightarrow \forall I.t'[\theta_2]$. Let $\nu \models t'[\theta_1]$. By the definition of well-formedness, $\nu \models \forall I.t'[\theta_1]$. By assumption, $\nu \models \forall I.t'[\theta_2]$. Hence $\nu \models t'[\theta_2]$. That is, $t'[\theta_1] \Rightarrow t'[\theta_2]$. We have $\theta_1 \Rightarrow \theta_2$ by inductive hypothesis.

Assume $t[] \equiv \exists I.t'[]$ and $\exists I.t'[\theta_1] \Rightarrow \exists I.t'[\theta_2]$. By the definition of well-formedness, $t'[\theta_1]$ is not satisfiable and thus $t'[\theta_1] \Rightarrow t'[\theta_2]$. Hence $\theta_1 \Rightarrow \theta_2$ by inductive hypothesis. □

5 Experiments

We have implemented a prototype[1] in OCaml. In our implementation, we use Yices as the SMT solver to resolve queries (Algorithm 2 and 3). Table 1 shows experimental results. We took two cases from the benchmark in [19] 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.

5.0.1 `devres` from Linux Library

Figure 5.(A) shows an annotated loop extracted from a Linux library.[2] In the postcondition, we assert that *ret* implies $tbl[i] = 0$, and every element in the array $tbl[]$ is not equal to *addr*

[1] Available at `http://ropas.snu.ac.kr/sas10/qinv-learn-released.tar.gz`
[2] The source code can be found in function `devres` of `lib/devres.c` in Linux 2.6.28

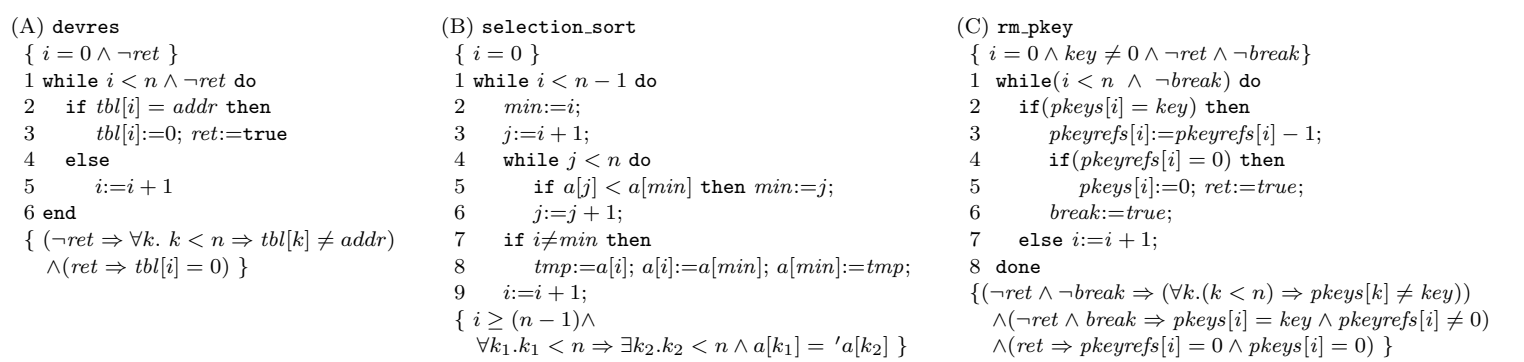

Figure 5: Benchmark Examples: (A) `devres` from Linux library, (B) `selection_sort` from [19], and (C) `rm_pkey` from Linux InfiniBand driver

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 the following quantified invariant:

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

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.

5.0.2 `selection_sort` from [19]

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 array $a[]$ is a permutation of its old content. In this example, 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 the following invariant:

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

Note that all membership queries are resolved randomly. In spite of more than 12,000 coin tosses, our algorithm is still able to derive invariants in each of 500 runs. This suggests that invariants are abundant. 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 simplify the form of templates used in this example.

5.0.3 `rm_pkey` from Linux InfiniBand Driver

Figure 5.(C) is a `while` statement extracted from Linux InfiniBand driver.[3] The conjuncts in the postcondition P represent (1) if the loop terminates without break, all elements of *pkeys*

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

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 the following quantified invariant:

$$(\forall k.(k < i) \Rightarrow pkeys[k] \neq key) \wedge (\neg ret \wedge break \Rightarrow pkeys[i] = key \wedge pkeyrefs[i] \neq 0) \\ \wedge (ret \Rightarrow pkeyrefs[i] = 0 \wedge pkeys[i] = 0)$$

The generality of our learning-based approach is again observed in this example. Our algorithm is able to infer a quantified invariant with very little help from the user. Our solution can be more applicable than other template based approaches in practice.

Our algorithm does not guarantee its termination. If invariants cannot be expressed in the form of provided template, the algorithm goes infinite. The algorithm fails to find an invariant if a given set of atomic propositions is not sufficient to compose an invariant. Due to the incompleteness of SMT solvers over quantified formulae, our algorithm could fail to recognize an invariant and leads to non-termination as a result.

6 Related Work

In contrast to previous template based approaches [19, 9], our template is more general as it supports arbitrary hole-filling quantifier-free formulae. The technique introduced in [19] handles templates whose holes are restricted to formulae over conjunctions of predicates from a given set, while disjunctions must be explicitly specified by the templates. Gulwani et al. [9] consider invariants restricted to $E \wedge \bigwedge_{j=1}^{n} \forall U_j(F_j \Rightarrow e_j)$, where E, F_j and e_j are quantifier free facts.

Existing technologies can strengthen our framework. Firstly, its completeness can be increased by powerful decision procedures [5, 7, 20] and theorem provers [17, 1, 18]. Moreover, our approach can be sped up when using more accurate approximations provided by existing invariant generation techniques. Gupta et al. [11] devised a tool InvGen. This tool collects reached states satisfying the program invariants, and also computes a collection of invariants for efficient invariant generation. These two results can be used by our framework as under and over approximations, respectively.

Regarding the generation of unquantified invariants, Gulwani et al. [10] proposed an approach based on constraint analysis. Invariants in the combined theory of linear arithmetic and uninterpreted functions are synthesized in [2], while InvGen [11] presents an efficient approach to generation of linear arithmetic invariants. In the area of quantified loop invariants generation, Flanagan et al. [6] use Skolemization for generating universally quantified invariants. In [17] a paramodulation-based saturation prover is extended to an interpolating prover that is complete for universally quantified interpolants. As opposed to our proposal, other approaches [6, 17] only generate universally quantified invariants.

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.

7 Conclusions

By combining algorithmic learning, decision procedures, predicate abstraction, and templates we presented a technique for generating quantified invariants. The new technique searches for invariants in the given template form guided by query resolution algorithms. Algorithmic learning gives a platform to integrate various techniques for invariant generation; with simple templates for quantified invariants it suffices to design new query resolution algorithms based on existing techniques.

Our technique shows that the flexibility of algorithmic learning over plentiful invariants works in finding real-world quantified invariants of a given template form. We exploit the flexibility by deploying a randomized query resolution algorithm. When a query cannot be resolved, a random answer is given to the learning algorithm. Since the learning algorithm does not commit to any specific invariant beforehand, it always finds a solution consistent with query results. Our experiments show that algorithmic learning is able to infer non-trivial quantified invariants with this naïve randomized resolution. In experiments, templates just need to specify which variables are universally or existentially quantified.

Acknowledgment We are grateful to Wontae Choi, Suwon Jang, Will Klieber, Wonchan Lee, Bruno Oliveira, 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] Dutertre, B., Moura, L.D.: The Yices SMT solver. Technical report, SRI International (2006)

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

[6] Ge, Y., Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification, Berlin, Heidelberg, Springer-Verlag (2009) 306–320

[7] Graf, S., Saïdi, H.: Construction of abstract state graphs with pvs. In: CAV. Volume 1254 of LNCS., Springer (1997) 72–83

[8] Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, ACM (2008) 235–246

[9] Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: VMCAI. Volume 5403 of LNCS., Springer (2009) 120–135

[10] Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: CAV. Volume 5643 of LNCS., Springer (2009) 634–640

[11] Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI. (2008) 339–348

[12] Henzinger, T.A., Hottelier, T., Kovács, L., Voronkov, A.: Invariant and type inference for matrices. In: VMCAI. (2010) 163–179

[13] Jhala, R., Mcmillan, K.L.: Array abstractions from proofs. In: CAV, volume 4590 of LNCS, Springer (2007)

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

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

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

[17] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)

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

[19] Srivastava, S., Gulwani, S., Foster, J.S.: Vs3: Smt solvers for program verification. In: CAV '09: Proceedings of Computer Aided Verification 2009. (2009)