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 templates, we present an automated technique for finding quantified loop invariants. Our technique can find arbitrary first-order invariants 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 sample loops in Linux source code and benchmarks in previous work.

1 Introduction

Recently, algorithmic learning has been successfully applied to invariant generation [28]. 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 [28] 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. By combining algorithmic learning, decision procedures, predicate abstraction, *and* templates, we present an automated technique to infer quantified loop invariants. The proposed technique can find quantified invariants for real-world code such as loops from Linux library, kernel, and device-driver sources.

Our technique is general with respect to the given template. We deploy a learning algorithm to infer propositional formulae (as in [28]) that fill in the template's hole(s) to give quantified formulae. Since the learning algorithm can infer an arbitrary propositional formula (using the given atomic propositions as building blocks), the hole-filling propositional formulae can be in any form. This generality contrasts with existing template-based approach [35] where only

non-disjunctive formulae can fill the holes. In all our experiments (Section 5), we used simple templates like "$\forall k.[]$" or "$\forall k.\exists i.[]$" to find invariants in real-world programs.

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 propositional 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 propositional formulae, we use predicate abstraction and concretization to resolve queries with decision procedures. In reality, because information about loop invariants is incomplete, queries may not be resolvable. When query resolution requires information 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 [28]:

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

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 propositional 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.

Contribution

- We show that algorithmic learning, decision procedures, predicate abstraction, and templates in combination can automatically infer quantified invariants in our simple language.
- The technique can find arbitrary first-order invariants (modulo the underlying SMT solver and a fixed set of atomic propositions) in the form of the given template. The templates are flexible: any template with one hole for propositional formula is supported. The hole-filling propositional formulae can be in any form, not restricted to non-disjunctive formulae, for example.
- We demonstrate that the technique works in realistic settings: we are able to generate quantified invariants for some Linux library, kernel, and device drive sources, as well as for the benchmark code used in the previous work [35].
- The technique can be seen as a simple yet effective framework for invariant generation. The technique is orthogonal to existing techniques. Static analyzers can contribute by providing information to algorithmic learning. And the technique's future improvement is for free. Since our algorithm uses the two key technologies (exact learning algorithm and decision procedures) as black boxes, it can adapt any technique developed in these technologies. Future advances of the two technologies will straightforwardly benefit our technique.

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 propositional and first-order formulae generated from A, respectively.

A *template* $t[] \in \tau$ is a first-order formula over Prop_A with a hole to be filled with a propositional formula in Prop_A.

$$\tau \triangleq [] \mid \neg\tau \mid \mathsf{Prop}_A \wedge \tau \mid \mathsf{Prop}_A \vee \tau \mid \forall I.\tau \mid \exists I.\tau.$$

Let $\theta \in \mathsf{Prop}_A$ be a propositional formula. We write $t[\theta]$ to denote the first-order formula obtained by replacing the hole in $t[]$ with θ.

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* $t[]$ 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 `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 `T` under μ.

Given a first-order formula ρ, a *satisfiability modulo theories (SMT) solver* returns a model of ν if it exists (written $SMT(\rho) \rightarrow \nu$); otherwise, it returns $UNSAT$ (written $SMT(\rho) \rightarrow UNSAT$) [15, 30].

2.1 CDNF Learning Algorithm

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 $|B|$ [9].

3 Framework Overview

We combine algorithmic learning [9], decision procedures [15], predicate abstraction [18], 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 $t[]$, we would like to apply the CDNF algorithm to find an invariant in the form of the given template. Recall that the template $t[]$ is a first-order formula with a hole to be filled by a propositional formula. Finding an invariant in the form of the template amounts to generating a proper propositional formula η such that $t[\eta]$ is a first-order invariant of the annotated loop. Hence, our goal is to infer such a propositional formula η using the CDNF learning algorithm.

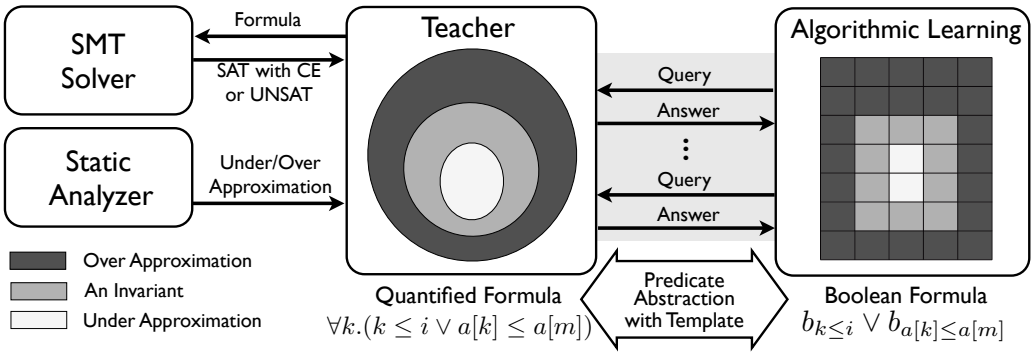

Figure 1: Our framework

To achieve this goal, we need to address two problems. Firstly, the CDNF algorithm is a learning algorithm for Boolean formulae, not propositional formulae. It cannot infer propositional formulae to fill the hole in the given template. Secondly, 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 predicate abstraction to relate Boolean formulae with propositional formulae. In predicate abstraction, an atomic proposition corresponds to a Boolean variable. Instead of inferring a proper propositional formula η for the hole in $t[]$, we will use the CDNF algorithm to deduce a proper Boolean formula λ in the abstract domain. A propositional formula η that corresponds to λ gives a first-order invariant $t[\eta]$ by filling the hole in $t[]$.

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. It is not difficult to concretize queries in the abstract domain. Answering queries however requires information about invariants yet to be computed.

Although an invariant is unknown, its approximations can be derived from the pre- and post-conditions, or computed by static analysis. Our query resolution algorithm (teacher) utilizes the information derived from invariant approximations. If a query cannot be resolved by invariant approximations, our algorithm simply gives a random answer to the CDNF algorithm. 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. Equivalence queries are resolved similarly. 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 counter example. If there are sufficiently many invariants, our simple randomized resolution algorithms will guide the CDNF algorithm to one of them with some luck.

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) develop technical lemmas to relate first-order formulae in the form of the given template (Section 4.2); and (3) design query resolution algorithms for algorithmic learning (Section 4.3).

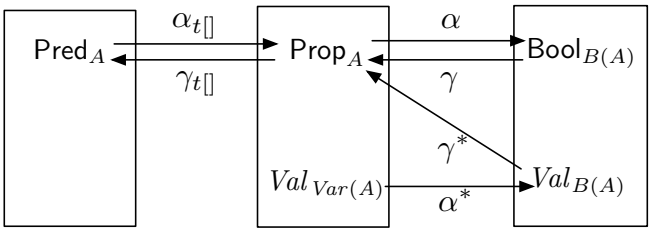

Figure 2: The domains Pred_A, Prop_A, and $\mathsf{Bool}_{B(A)}$

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 propositional formulae generated from A. Since the solution space $\mathcal{S}_{t[]}$ is generated by the fixed template $t[]$, Prop_A is 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 propositional formula $\theta \in \mathsf{Prop}_A$. Note that any relation among atomic propositions is lost after abstraction. For instance, the propositional formula $x = 0 \wedge x > 0$ is not satisfiable whereas its abstraction $\alpha(x = 0 \wedge x > 0) = b_{x=0} \wedge b_{x>0}$ is satisfied by assigning $b_{x=0} = b_{x>0} = \mathtt{T}$.

The pair of $(\gamma_{t[]}, \alpha_{t[]})$ gives the correspondence between Prop_A and Pred_A.

$$\gamma_{t[]} : \mathsf{Prop}_A \to \mathsf{Pred}_A \qquad \alpha_{t[]} : \mathsf{Pred}_A \to \mathsf{Prop}_A$$
$$\gamma_{t[]}(\theta) = t[\theta] \qquad \alpha_{t[]}(\gamma_{t[]}(\theta)) = \theta$$

Given a template $t[]$, the mapping $\gamma_{t[]}(\theta)$ replaces the hole in the template $t[]$ by the propositional formula θ. On the other hand, $\alpha_{t[]}$ returns the propositional subformula θ that corresponds to the hole in the template $t[]$. For instance, assume $t[] = \forall k.(k \not\leq i) \vee []$. Then $\alpha_{t[]}(\forall k.(k \not\leq i) \vee (a[k] \leq a[m])) = a[k] \leq a[m]$ and $\gamma_{t[]}(a[k] = a[m]) = \forall k.(k \not\leq i) \vee (a[k] = a[m])$.

To answer membership queries, we relate a Boolean valuation $\mu \in \mathsf{Bool}_{B(A)}$ with a propositional formula $\gamma^*(\mu)$ or a first order formula $\gamma^*_\tau(\mu, t[])$ in the form of $t[] \in \tau$. 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.

$$\begin{array}{rcl} \gamma^*(\mu) & = & \bigwedge_{p\in A}\{p : \mu(b_p) = \mathtt{T}\} \wedge \bigwedge_{p\in A}\{\neg p : \mu(b_p) = \mathtt{F}\} \\ \gamma^*_\tau(\mu, t[]) & = & \gamma_{t[]}(\gamma^*(\mu)) \\ (\alpha^*(\nu))(b_p) & = & \begin{cases} \mathtt{T} & \text{if } \nu \models p \\ \mathtt{F} & \text{otherwise} \end{cases} \end{array}$$

For conveniences, we introduce $\alpha_\tau(\rho, t[]) = \alpha(\alpha_{t[]}(\rho))$ and $\gamma_\tau(\beta, t[]) = \gamma_{t[]}(\gamma(\beta))$. It is routine to verify the following equalities:

$$\alpha(\gamma(\beta)) = \beta \qquad \alpha_{t[]}(\gamma_\tau(\beta, t[])) = \gamma(\beta) \qquad \alpha_{t[]}(\gamma^*_\tau(\mu, t[])) = \gamma^*(\mu)$$

The following lemmas will be useful in our presentation:

Lemma 4.1 ([28]) *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 ([28]) *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 Template Properties

Let $t[] \in \tau$ be a template and $\theta \in \mathsf{Prop}_A$ a propositional formula. If $t[\theta]$ is too strong to be an invariant, we would like to relax $t[\theta]$ by changing θ. However, the semantics of $t[\theta]$ depends on the context of the hole. For $\theta_1, \theta_2 \in \mathsf{Prop}_A$, $t[\theta_1]$ is not necessarily weaker than $t[\theta_2]$ if θ_1 is weaker (or stronger) than θ_2. We therefore would like to establish a connection between a relation of θ_1 and θ_2, and the corresponding relation of $t[\theta_1]$ and $t[\theta_2]$ (Theorem 4.4). The connection is essential to the design of query resolution algorithms.

We begin with the classification of templates by the polarity of their holes.

Definition Let $t[] \in \tau$ be a template.

1. $t[]$ is *positive* if $\forall\theta_1, \theta_2 \in \mathsf{Prop}_A : (\theta_1 \Rightarrow \theta_2) \Rightarrow (t[\theta_1] \Rightarrow t[\theta_2])$;
2. $t[]$ is *negative* if $\forall\theta_1, \theta_2 \in \mathsf{Prop}_A : (\theta_1 \Rightarrow \theta_2) \Rightarrow (t[\theta_2] \Rightarrow t[\theta_1])$.

The semantic conditions in Definition 4.2 are hard to check. We hence consider the following syntactic classifications:

$$\begin{array}{rcl} \tau_+ & \triangleq & [] \mid \forall I.\tau_+ \mid \exists I.\tau_+ \mid \neg\tau_- \mid \tau_+ \vee \mathsf{Prop}_A \mid \tau_+ \wedge \mathsf{Prop}_A \\ \tau_- & \triangleq & \forall I.\tau_- \mid \exists I.\tau_- \mid \neg\tau_+ \mid \tau_- \vee \mathsf{Prop}_A \mid \tau_- \wedge \mathsf{Prop}_A \end{array}$$

The syntactic classes τ_+ and τ_- characterize the semantic definitions soundly.

Lemma 4.3 *Let $t[] \in \tau$ be a template. If $t[] \in \tau_+$ (or $t[] \in \tau_-$), then $t[]$ is a positive template (or negative template respectively).*

Lemma 4.3 is useful in deducing the relation between $t[\theta_1]$ and $t[\theta_2]$ provided $\theta_1 \Rightarrow \theta_2$. For the other direction, consider the following definition:

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

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

The intuition behind the well-formed template definition is that the relation $t[\theta_1] \Rightarrow t[\theta_2]$ in Pred domain must depend on both propositional formulae θ_1 and θ_2 input into the holes. If $t[\theta_1] \Rightarrow t[\theta_2]$ holds regardless of θ_1 and θ_2, then no relation in Prop domain can be guaranteed. Hence, the well-formedness of t[] allows us to establish the following theorem:

Theorem 4.4 *Let A be a set of atomic propositions, $\theta_1 \in \mathsf{Prop}_A$ and $t[] \in \tau$ a template.*

1. *If $t[]$ is positive and well-formed with respect to θ_1, then $\forall \theta_2 \in \mathsf{Prop}_A.(t[\theta_1] \Rightarrow t[\theta_2]) \Rightarrow (\theta_1 \Rightarrow \theta_2)$;*
2. *If $t[]$ is negative and well-formed with respect to θ_2, then $\forall \theta_1 \in \mathsf{Prop}_A.(t[\theta_1] \Rightarrow t[\theta_2]) \Rightarrow (\theta_2 \Rightarrow \theta_1)$.*

Proof *The proof uses structural induction on t[].*

1. *Let us assume t[] is positive.*

- *Case $t[] = []$. Then, from $t[p_1] \Rightarrow t[p_2]$ we have $p_1 \Rightarrow p_2$.*
- *Case $t[] = t'[] \vee p$. From the assumption we have $t'[p_1] \vee p \Rightarrow t'[p_2] \vee p$. Hence, $t'[p_1] \Rightarrow t'[p_2] \vee p$. Furthermore, $t'[p_1] \Rightarrow t'[p_2] \vee p$ is equivalent with $\neg t'[p_1] \vee t'[p_2] \vee p$. Additionally, from the assumption that t is a well-formed template, $t'[p_1] \wedge \neg p$, we obtain $t'[p_2]$. Therefore, we have $t'[p_1] \Rightarrow t'[p_2]$. $p_1 \Rightarrow p_2$ follows by the induction hypothesis.*
- *Case $t[] = t'[] \wedge p$. From the assumption we have $t'[p_1] \wedge p \Rightarrow t'[p_2] \wedge p$. Hence, $t'[p_1] \wedge p \Rightarrow t'[p_2]$. Furthermore, $t'[p_1] \wedge p \Rightarrow t'[p_2]$ is equivalent with $\neg t'[p_1] \vee \neg p \vee t'[p_2](*)$. Additionally, from the assumption that t is a well-formed template, we have $t'[p_1] \wedge p$ $(**)$. From $(*)$ and $(**)$ we obtain $t'[p_1] \Rightarrow t'[p_2]$. Hence, $p_1 \Rightarrow p_2$ follows by the induction hypothesis.*
- *Case $t[] = \neg t'[]$, where $t'[]$ is a negative template. Then $\neg t'[p_1] \Rightarrow \neg t'[p_2]$ which is equivalent to $t'[p_2] \Rightarrow t'[p_1]$. Conclusion $p_1 \Rightarrow p_2$ follows from the induction hypothesis.*
- *Case $t[] = \forall I.t'[]$. From the assumption we have $\forall I.t'[p_1] \Rightarrow \forall I.t'[p2]$, which is equivalent to $\exists I.(\neg t'[p1]) \vee \forall I.t'[p2]$. After existential quantifier elimination we have $\neg t'[p1] \vee \forall I.t'[p2]$. As $\neg t'[p1] \Rightarrow (t'[p1] \Rightarrow t'[p2])$ and $\forall I.t'[p2] \Rightarrow (t'[p1] \Rightarrow t'[p2])$, through disjunction elimination we have $t'[p1] \Rightarrow t'[p2]$. Conclusion follows from the induction hypothesis.*
- *Case $t[] = \exists I.t'[]$. From the assumption we have $\exists I.t'[p_1] \Rightarrow \exists I.t'[p2]$, which is equivalent to $\forall I.(\neg t'[p1]) \vee \exists I.t'[p2]$. After existential quantifier elimination we have $\forall I.(\neg t'[p1]) \vee t'[p2]$. As $\forall I.\neg t'[p1] \Rightarrow (t'[p1] \Rightarrow t'[p2])$ and $t'[p2] \Rightarrow (t'[p1] \Rightarrow t'[p2])$, through disjunction elimination we have $t'[p1] \Rightarrow t'[p2]$. Conclusion follows from the induction hypothesis.*

2. *Let us assume that t[] is negative. Conclusion follows similar to the case for the positive template.*

□

4.3 Resolving Queries

Fix a set A of atomic propositions. Let an annotated loop and a template $t[]$ be given. Our goal is to apply the CDNF algorithm to infer a Boolean formula $\lambda \in \mathsf{Bool}_{B(A)}$ such that $\gamma_\tau(\lambda, t[])$ is an invariant for the annotated loop. Recall that the CDNF algorithm makes two types of queries. A membership query $MEM(\mu)$ where $\mu \in Val_{B(A)}$ asks if μ is a Boolean model of λ; an equivalence query $EQ(\beta)$ where $\beta \in \mathsf{Bool}_{B(A)}$ asks if β is equivalent to λ. In our setting, the target Boolean function λ is unknown. In order to design query resolution algorithms without knowing λ, we resort to invariant approximations.

Suppose $\iota \in \mathsf{Pred}_A$ is an invariant for the annotated loop $\{\delta\}$ `while` κ `do` S $\{\epsilon\}$. 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 $\overline{\iota} \Rightarrow \epsilon \vee \kappa$. The *strongest* ($\delta$) and *weakest* ($\epsilon \vee \kappa$) approximations are trivial under- and over-approximations to any invariant respectively.

In the following discussion, we assume $\iota = t[\eta]$ is a first-order invariant in the form of the given template $t[]$ for some $\eta \in \mathsf{Prop}_A$. Moreover, $\lambda = \alpha_\tau(\iota, t[]) = \alpha(\eta)$ is the unknown target Boolean formula.

`(* ` $\underline{\iota}$`: an under-approximation; ` $\overline{\iota}$`: an over-approximation *)`
`(* ` $t[]$`: the given template *)`
Input: a valuation μ for $B(A)$
Output: *YES* or *NO*
if $SMT(\gamma^*(\mu)) \rightarrow UNSAT$ **then return** *NO*;
$\rho := \gamma^*_t(\mu, t[])$;
if $t[] \in \tau_+$ **then**
  if $SMT(\rho \wedge \neg\overline{\iota}) \rightarrow \nu$ **then return** *NO*;
  if $isWellFormed(t[], \gamma^*(\mu))$ **and** $SMT(\rho \wedge \neg\underline{\iota}) \rightarrow UNSAT$ **then return** *YES*;
if $t[] \in \tau_-$ **then**
  if $SMT(\underline{\iota} \wedge \neg\rho) \rightarrow \nu$ **then return** *NO*;
  if $isWellFormed(t[], \gamma^*(\mu))$ **and** $SMT(\overline{\iota} \wedge \neg\rho) \rightarrow UNSAT$ **then return** *YES*;
return *YES or NO randomly*

Algorithm 1: Resolving Membership Queries

Membership Queries In a membership query $MEM(\mu)$, our membership query resolution algorithm is required to answer whether $\mu \models \lambda$. Note that any relation between atomic propositions of A is lost in the abstract domain $\mathsf{Bool}_{B(A)}$. A valuation may not correspond to a consistent propositional formula. If the valuation $\mu \in Val_{B(A)}$ corresponds to an inconsistent propositional formula (that is, $\gamma^*(\mu)$ is unsatisfiable), we simply answer *NO* to the membership query. Otherwise, we compare $\gamma^*_\tau(\mu, t[])$ with invariant approximations.

Algorithm 1 gives our membership query resolution algorithm. Given a template $t[]$ and a propositional formula $\theta \in \mathsf{Prop}_A$, $isWellFormed(t[], \theta)$ checks whether $t[]$ is well-formed with respect to θ. It is implemented by a simple structural induction on templates and hence omitted here. In the membership query resolution algorithm, we first check if μ corresponds to a consistent propositional formula. If it does, we split two cases by the syntactic polarity of the hole in the template $t[]$. Assume $t[]$ is syntactically positive. If $\neg(\gamma^*_t(\mu, t[]) \Rightarrow \overline{\iota})$, the algorithm returns *NO*. If $t[]$ is well-formed with respect to $\gamma^*(\mu)$, and $\gamma^*_t(\mu, t[]) \Rightarrow \underline{\iota}$, then the algorithm returns *YES*. When the template $t[]$ is syntactically negative, Algorithm 1 resolves membership queries symmetrically. Note that Algorithm 1 will give a random answer if a membership query cannot be resolved.

Equivalence Queries To answer an equivalence query $EQ(\beta)$, we check if $\gamma_\tau(\beta, t[])$ 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 $\gamma_\tau(\beta, t[])$ with invariant approximations.

`(* ` $\underline{\iota}$ `:  an under-approximation;` $\overline{\iota}$ `:  an over-approximation *)`
`(*` $t[]$`:  the given template *)`
Input: $\beta \in \mathsf{Bool}_{B(A)}$
Output: *YES*, or a counterexample
$\rho := \gamma_\tau(\beta, t[])$;
if ρ *is an invariant weaker than* $\underline{\iota}$ *and stronger than* $\overline{\iota}$ **then return** *YES*;
if $SMT(\underline{\iota} \wedge \neg\rho) \rightarrow \nu$ **then return** $\alpha^*(\nu)$;
if $SMT(\rho \wedge \neg\overline{\iota}) \rightarrow \nu$ **then return** $\alpha^*(\nu)$;
let $SMT(\rho \wedge \neg\underline{\iota}) \rightarrow \nu_0$ and $SMT(\overline{\iota} \wedge \neg\rho) \rightarrow \nu_1$;
return $\alpha^*(\nu_0)$ *or* $\alpha^*(\nu_1)$ *randomly*;

Algorithm 2: Resolving Equivalence Queries

Algorithm 2 gives our equivalence resolution algorithm. It first checks if $\gamma_\tau(\beta, t[])$ is indeed an invariant for the annotated loop by verifying $\underline{\iota} \Rightarrow \rho$, $\rho \Rightarrow \overline{\iota}$, and $\kappa \wedge \rho \Rightarrow Pre(\rho, S)$ with an SMT solver. If not, it computes a valuation to falsify $\underline{\iota} \Rightarrow \gamma_\tau(\beta, t[])$. The valuation in turn gives a counterexample to differentiate β from λ. Otherwise, a valuation invalidating $\gamma_\tau(\beta, t[]) \Rightarrow \overline{\iota}$ gives a counterexample to the equivalence query.

If it fails to find counterexamples, the algorithm returns a valuation distinguishing $\gamma_\tau(\beta, t[])$ from invariant approximations as a random answer. 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.

4.4 Main Loop

Input: $\{\delta\}$ `while` κ `do` S $\{\epsilon\}$: an annotated loop; $t[]$: a template
Output: an invariant in the form of $t[]$
$\underline{\iota}$ `:=` δ;
$\overline{\iota}$ `:=` $\kappa \vee \epsilon$;
repeat
  try λ `:=` **call** CDNF with Algorithm 1 and 2 **when abort** $\rightarrow$ **continue**
until λ *is defined* ;
return $\gamma_\tau(\lambda, t[])$;

Algorithm 3: Main Loop

Our invariant generation algorithm is now straightforward. Given an annotated loop $\{\delta\}$ `while` κ `do` S $\{\epsilon\}$ and a template $t[] \in \tau$, 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 $\gamma_\tau(\lambda, t[])$ is an invariant for the annotated loop in the form of the template $t[]$ (Algorithm 3).

The CDNF algorithm uses the membership query resolution algorithm (Algorithm 1) and the equivalence query resolution algorithm (Algorithm 2) 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

case	Template	AP	MEM	EQ	MEM_R	EQ_R	$RESTART$	Time (sec)
`max`	$\forall k.[]$	7	623	214	214	30	49	1.01
`selection_sort`	$\forall k_1.\exists k_2.[]$	6	6499	4080	3930	209	1577	4.70
`devres`	$\forall k.[]$	7	9017	6415	81	133	1675	2.89
`rm_pkey`	$\forall k.[]$	8	2667	1116	1779	222	126	5.76
`tracepoint1`	$\exists k.[]$	4	108103	115394	53	112	32594	6.10
`tracepoint2`	$\forall k_1.\exists k_2.[]$	7	238448	93654	666	436	22687	196.34

Table 1: Performance Numbers.
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.

are numerous first-order invariants in the form of the given template, a few 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 whole algorithm is restarted.

5 Experiments

We have implemented a prototype[1] in OCaml. In our implementation, we use YICES as the SMT solver to resolve queries (Algorithm 1 and 2). Table 1 shows the performance numbers of our experiments. We took two cases from the benchmark in [35] 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. `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.

5.1 `devres` from Linux Library

$\{\ i = 0 \wedge \neg ret\ \}$
1 `while` $i < n \wedge \neg ret$ `do`
2 `if` $tbl[i] = addr$ `then`
3 $tbl[i]$:=0; ret:=`true`
4 `else`
5 i:=$i + 1$
6 `end`
$\{\ (\neg ret \Rightarrow \forall k.\ k < n \Rightarrow tbl[k] \neq addr) \wedge (ret \Rightarrow tbl[i] = 0)\ \}$

Figure 3: `devres` from Linux library

Figure 3 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 $addr$ 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:

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

[1] Available at `http://ropas.snu.ac.kr/cav10/qinv-learn-released.tar.gz`
[2] The source code can be found in function `devres` of `lib/devres.c` in Linux 2.6.28

Observe that our algorithm is able to infer an arbitrary propositional 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.

5.2 `selection_sort` from [35]

$\{\ i = 0\ \}$
1 `while` $i < n - 1$ `do`
2 $\quad min{:=}i;$
3 $\quad j{:=}i + 1;$
4 $\quad$ `while` $j < n$ `do`
5 $\quad\quad$ `if` $a[j] < a[min]$ `then` $min{:=}j;$
6 $\quad\quad j{:=}j + 1;$
7 $\quad$ `if` $i{\neq}min$ `then`
8 $\quad\quad tmp{:=}a[i];\ a[i]{:=}a[min];\ a[min]{:=}tmp;$
9 $\quad i{:=}i + 1;$
$\{\ i \geq (n-1) \wedge \forall k_1.k_1 < n \Rightarrow \exists k_2.k_2 < n \wedge a[k_1] = {'}a[k_2]\ \}$

Figure 4: `selection_sort` from [35]

Consider the selection sort algorithm in Figure 4. 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:

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

Templates allow us to infer not only universally quantified invariants but also first-order invariants with alternating quantifications. Inferring arbitrary propositional formulae over a fixed set of atomic propositions again greatly simplify the form of templates used in this example.

5.3 `tracepoint1` from Linux Kernel

$\{\ n = 0 \wedge i = 0\ \}$
1 `while` $A[i] \neq 0$ `do`
2 $\quad$ `if` $(p = 0 \vee A[i] = p)$ `then`
3 $\quad\quad n{:=}n{+}1;$
4 $\quad i{:=}i{+}1;$
$\{\ n > 0 \wedge p \neq 0 \Rightarrow \exists k.k < i \wedge A[k] = p\ \}$

Figure 5: `tracepoint1` from Linux Kernel

Figure 5 is an annotated loop extracted from a Linux Kernel.[3]. From the loop body, we infer the postcondition. If the true branch of `if` statement (line 3) is taken at least once and

[3]The source code can be found in function `tracepoint_entry_remove_probe` of `kernel/tracepoint.c` in Linux 2.6.28

also p is not zero, then it implies that the second disjunct of the branch condition $A[i] = p$ (line 2) should hold. This means that there should be an appropriate value k such that the condition $A[k] = p$ holds. From this postcondition, we find a hint that the invariant could be existentially quantified with k. Thus, we use the template $\exists k.[]$. With the template and the set of atomic propositions $\{p = 0,\ n < 0,\ k < i,\ A[k] = p\}$, our algorithm finds the following invariant

$$\exists k.(p = 0) \vee (k < n \wedge A[k] = p) \vee (n <= 0).$$

5.4 `tracepoint2` from Linux Kernel

$\{\ i = 0 \wedge j = 0\ \}$
1 `while` $old[i] \neq 0$ `do`
2 `if` $(probe \wedge old[i] \neq probe)$ `then`
3 $new[j]{:=}old[i]$;
4 $j{:=}j{+}1$;
5 $i{:=}i{+}1$;
$\{\ \forall k_2.k_2 < j \Rightarrow \exists k_1.k_1 < i \wedge old[k_1] = new[k_2] \wedge new[k_2] \neq probe\ \}$

Figure 6: `tracepoint2` from Linux Kernel

Figure 6 is another annotated loop extracted from a Linux Kernel.[4]. From the loop body, we infer the postcondition. The value of j is incremented only if the program executes the true branch of `if` statement (line 3 and 4). Thus, at the loop head, for all $new[0] \ldots new[j-1]$ each element of new has the same value with one of $old[0] \ldots old[i-1]$ (line 3). From this postcondition, we find a hint that an invariant can be in the form of $\forall k_1.\exists k_2.[]$. With this template and the set of atomic propositions $\{old[i] = 0,\ old[i] = probe,\ k < i,\ k < j,\ old[k] = p,\ new[k] = p,\ old[k] = new[k]\}$, our algorithm finds the following invariant

$$\forall k_2.\exists k_1(k_2 < j) \Rightarrow (k_1 < i) \wedge (old[k_1] = new[k_2]) \wedge (old[k_1] \neq probe).$$

5.5 `rm_pkey` from Linux InfiniBand Driver

$\{\ i = 0 \wedge key \neq 0 \wedge \neg ret \wedge \neg break\}$
1 `while`$(i < n\ \wedge\ \neg break)$ `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`
$\{\ (\neg ret \wedge \neg break \Rightarrow (\forall k.(k < n) \Rightarrow pkeys[k] \neq key))$
$\wedge(\neg ret \wedge break \Rightarrow pkeys[i] = key \wedge pkeyrefs[i] \neq 0)$
$\wedge(ret \Rightarrow pkeyrefs[i] = 0 \wedge pkeys[i] = 0)\ \}$

Figure 7: `rm_pkey` from Linux InfiniBand driver

Figure 7 is a `while` statement extracted from Linux InfiniBand driver.[5] The conjuncts in the postcondition P represent (1) if the loop terminates without break, all elements of $pkeys$

[4]The source code can be found in function `tracepoint_entry_remove_probe` of `kernel/tracepoint.c` in Linux 2.6.28

[5]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.

6 Related Work

In contrast to previous template based approaches [35, 21], our template is more general as it supports arbitrary hole-filling propositional formulae. The technique introduced in [35] 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. [21] 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 [15, 17, 36] and theorem provers [33, 7, 34]. Moreover, our approach can be sped up when using more accurate approximations provided by existing invariant generation techniques. Gupta et al. [24] devised a tool InvGen. This tool collects reached states satisfying the program invariants, and also computes a collection of invariants for efficient invarinat 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. [22] proposed an approach based on constraint analysis. Invariants in the combined theory of linear arithmetic and uninterpreted functions are synthesized in [8], while InvGen [24] presents an efficient approach to generation of linear arithmetic invariants. In the area of quantified loop invariants generation, Flanagan et al. [16] use Skolemization for generating universally quantified invariants. In [33] 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 [16, 33] only generate universally quantified invariants.

With respect to the analysis of properties of array contents, Halbwachs et al. [25] 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 [26]. For inferring range predicates, Jhala and Mcmillan [27] 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 [27] 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.

References

[1] Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL, ACM (2005) 98–109

[2] Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: CAV. Volume 3576 of LNCS., Springer (2005) 548–562

[3] Alur, R., Černý, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL, New York, NY, USA, ACM (2005) 98–109

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

[5] Balaban, I., Pnueli, A., Zuck, L.: Shape analysis by predicate abstraction. In: VMCAI. Volume 3385 of LNCS., Springer (2005)

[6] Ball, T., Cook, B., Das, S., Rajamani, S.K.: Refining approximations in software predicate abstraction. In: TACAS. Volume 2988 of LNCS., Springer (2004) 388–403

[7] 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)

[8] Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: VMCAI. (2007) 378–394

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

[10] Chen, Y.F., Farzan, A., Clarke, E.M., Tsay, Y.K., Wang, B.Y.: Learning minimal separating DFA's for compositional verification. In: TACAS. Volume 5505 of LNCS., Springer (2009) 31–45

[11] 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

[12] Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: TACAS. Volume 2619 of LNCS., Springer (2003) 331–346

[13] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, ACM (1978) 84–96

[14] David, C., Jung, Y., Kong, S., Bow-Yaw, W., 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)

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

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

[17] 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

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

[19] Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI, ACM (2009) 375–385

[20] Gulwani, S., Jojic, N.: Program verification as probabilistic inference. In: POPL, ACM (2007) 277–289

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

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

[23] Gupta, A., McMillan, K.L., Fu, Z.: Automated assumption generation for compositional verification. In: CAV. Volume 4590 of LNCS., Springer (2007) 420–432

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

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

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

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

[28] 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)

[29] Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: FASE. LNCS, Springer (2009) 470–485

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

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

[32] Lahiri, S.K., Bryant, R.E., Bryant, A.E., Cook, B.: A symbolic approach to predicate abstraction. In: CAV. Volume 2715 of LNCS., Springer (2003) 141–153

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

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

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

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