Automatically inferring loop invariants via algorithmic learning

YUNGBUM JUNG, SOONHO KONG, CRISTINA DAVID, BOW-YAW WANG and KWANGKEUN YI

Mathematical Structures in Computer Science / Volume 25 / Special Issue 04 / May 2015, pp 892 - 915
DOI: 10.1017/S0960129513000078, Published online: 17 December 2014

Math. Struct. in Comp. Science (2015), *vol.* 25, *pp.* 892–915. © Cambridge University Press 2014
doi:10.1017/S0960129513000078 First published online 17 December 2014

Automatically inferring loop invariants via algorithmic learning[†]

YUNGBUM JUNG[‡], SOONHO KONG[§], CRISTINA DAVID[¶], BOW-YAW WANG[‖] and KWANGKEUN YI[††]

[‡]*Program Analysis Department, Fasoo, Seoul, Republic of Korea*
[§]*School of Computer Science, Carnegie Mellon University, Pittsburgh, United States*
Email: soonhok@cs.cmu.edu
[¶]*Department of Computer Science School of Computing, National University of Singapore, Singapore*
[‖]*INRIA and Institute of Information Science, Academia Sinica, Taipei, Taiwan*
[††]*School of Computer Science and Engineering, Seoul National University, Seoul, Republic of Korea*

Received 8 January 2013

By combining algorithmic learning, decision procedures, predicate abstraction and simple templates for quantified formulae, we present an automated technique for finding loop invariants. Theoretically, this technique can find arbitrary first-order invariants (modulo a fixed set of atomic propositions and an underlying satisfiability modulo theories solver) in the form of the given template and exploit the flexibility in invariants by a simple randomized mechanism. In our study, the proposed technique was able to find quantified invariants for loops from the Linux source and other realistic programs. Our contribution is a simpler technique than the previous works yet with a reasonable derivation power.

1. Introduction

Algorithmic learning has been applied to assumption generation in compositional reasoning (Cobleigh *et al.* 2003). In contrast to traditional techniques, the learning approach does not derive assumptions in an off-line manner. It instead finds assumptions by interacting with a model checker progressively. Since assumptions in compositional reasoning are generally not unique, algorithmic learning can exploit the flexibility in assumptions to attain preferable solutions. Applications of algorithmic learning in formal verification and interface synthesis have also been reported (Cobleigh *et al.* 2003; Alur *et al.* 2005a,b; Gupta *et al.* 2007; Chen *et al.* 2009).

Finding loop invariants follows a similar pattern. Invariants are often not unique. Indeed, programmers derive invariants incrementally. They usually have their guesses of invariants in mind, and gradually refine their guesses by further observing program

[†] This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2011-0000971), the National Science Foundation (award no. CNS0926181), the National Science Council of Taiwan projects No. NSC97-2221-E-001-003-MY3, NSC97-2221-E-001-006-MY3, the FORMES Project within LIAMA Consortium, and the French ANR project SIVES ANR-08-BLAN-0326-01. This research was sponsored in part by the National Science Foundation (award no. CNS0926181) and by Republic of Korea Dual Use Program Cooperation Center (DUPC) of Agency for Defense Development (ADD).

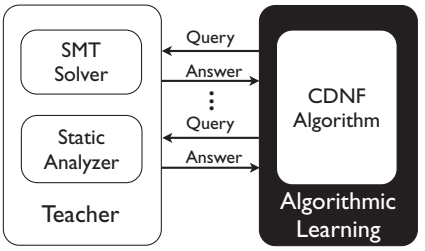


Fig. 1. The learning-based framework.

behaviour. Since in practice there are many invariants for given pre- and post-conditions, programmers have more freedom in deriving invariants. Yet traditional invariant generation techniques do not exploit the flexibility. They have a similar impediment to traditional assumption generation.

This article integrates our experiences applying algorithmic learning to invariant generation (Jung *et al.* 2010; Kong *et al.* 2010). We show that the four technologies (algorithmic learning, decision procedures, predicate abstraction and simple templates) can be arranged in concert to derive loop invariants in first-order (or, quantified) formulae. The new combined technique is able to generate invariants for some Linux device drivers and other realistic programs without any help from static or dynamic analyses.

Figure 1 shows our framework. In the figure, the CDNF algorithm is used to drive the search of invariants. The CDNF algorithm is an exact learning algorithm for Boolean formulae. It computes a representation of an unknown target formula by asking a teacher two types of queries. A membership query asks if a valuation to Boolean variables satisfies the unknown target; an equivalence query asks if a candidate formula is equivalent to the target. Relation between target Boolean formula and first-order formulae is established with predicate abstraction and simple templates. One remaining thing is to automate the query resolution process to infer an invariant.

1.1. *Example for quantifier-free invariants*

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

The while loop assigns a random truth value to the variable b in the beginning of its body. It increases the variable i by 1 if b is true. Observe that the variable b must be true after the while loop. We would like to find an invariant which proves the postcondition $i = 10 \wedge b$. Heuristically, we choose $i = 0$ (precondition) and $(i = 10 \wedge b) \vee i < 10$ (postcondition or loop guard) as under- and over-approximations to invariants respectively. With the help of a decision procedure, these invariant approximations are used to resolve queries made by the learning algorithm. After resolving a number of queries, the learning algorithm asks whether $i \neq 0 \wedge i < 10 \wedge \neg b$ should be included in the invariant. Note that the query

is not stronger than the under-approximation, nor weaker than the over-approximation. Hence decision procedures cannot resolve it due to lack of information. At this point, one could apply static analysis and see that it is possible to have this state at the beginning of the loop. Instead of employing static analysis, we simply give a random answer to the learning algorithm. For this example, this information is crucial: the learning algorithm will ask us to give a counterexample to its best guess $i = 0 \vee (i = 10 \wedge b)$ after it processes the incorrect answer. Since the guess is not an invariant and flipping coins does not generate a counterexample, we restart the learning process. If the query $i \neq 0 \wedge i < 10 \wedge \neg b$ is answered correctly, the learning algorithm infers the invariant $(i = 10 \wedge b) \vee i < 10$ with two more resolvable queries. On average, our algorithm takes about 2 iterations to generate the invariant $i < 10 \vee (i = 10 \wedge b)$ in 0.056 seconds.

1.2. *Example for first-order invariants*

In order to illustrate how our algorithm works for first-order (quantified) invariants, we briefly describe the learning process for the `max` example from Srivastava and Gulwani (2009).

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

The `max` example examines $a[0]$ through $a[n-1]$ and finds the index of the maximal element in the array. This simple loop is annotated with the precondition $m = 0 \wedge i = 0$ and the postcondition $\forall k.0 \leqslant k < n \Rightarrow a[k] \leqslant a[m]$. Under- and over-approximations of an invariant are constructed using the loop guard $\kappa = (i < n)$ and pre- and post-conditions.

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

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

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

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

— Membership query: after a few equivalence queries, a membership query asks whether $\bigwedge\{i \geqslant n,\ m = 0,\ i = 0,\ k \geqslant n,\ a[k] \leqslant a[m],\ a[m] \geqslant a[i]\}$ is a part of an invariant. The teacher replies *YES* since the query is included in the precondition and therefore should also be included in an invariant.

— Membership query: the membership query $MEM(\bigwedge\{i < n,\ m = 0,\ i \neq 0,\ k < n,\ a[k] > a[m],\ k < i,\ a[m] \geqslant a[i]\})$ is not resolvable because the template is not *well formed* (Definition 6.2) by the given membership query. In this case, the teacher gives a random answer (*YES* or *NO*). Interestingly, each answer leads to a different invariant for this query. If the answer is *YES*, we find an invariant $\forall k.(i < n \wedge k \geqslant i) \vee (a[k] \leqslant a[m]) \vee (k \geqslant n)$; if the answer is *NO*, we find another invariant $\forall k.(i < n \wedge k \geqslant i) \vee (a[k] \leqslant a[m]) \vee (k \geqslant n \wedge k \geqslant i)$. This shows how our approach exploits a multitude of invariants for the annotated loop.

1.3. *Contribution*

— We prove that a simple combination of algorithmic learning, decision procedures, predicate abstraction and templates can automatically infer first-order loop invariants. The technique is as powerful as the previous approaches (Gulwani *et al.* 2008a; Srivastava and Gulwani 2009) yet is much simpler.

— The technique works in realistic settings: the proposed technique can find invariants for some Linux library, kernel, SPEC2000 benchmarks and device driver sources, as well as for the benchmark code used in the previous work (Srivastava and Gulwani 2009).

— The technique can be seen as a framework for invariant generation. Static analysers can contribute by providing information to algorithmic learning. Ours is hence orthogonal to existing techniques.

— The technique needs a very simple template such as '$\forall k.[]$' or '$\forall k.\exists i.[]$'. Our algorithm can generate any quantified invariants expressible by a fixed set of atomic propositions in the form of the given template. Moreover, the correctness of generated invariants is verified by an SMT solver.

— The technique's future improvement is free. Since our algorithm uses the two key technologies (exact learning algorithm and decision procedures) as black boxes, future advances of these technologies will straightforwardly benefit our approach.

1.4. *Organization*

We organize this paper as follows. After preliminaries (Section 2), we present our problem and solutions in Section 3. In Section 4, we review the exact learning algorithm introduced in Bshouty (1995). Section 5 shows how to interact different domains. Section 6 gives the details of our learning approach. We report experiments in Section 7. Section 8 discusses our learning approach, and future work. Section 9 surveys related work. Section 10 concludes our work.

2. The target language and notation

The syntax of statements in our simple imperative language is as follows.

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

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

A *Boolean formula* Bool is a restricted propositional formula constructed from truth values and Boolean variables.

$$\mathsf{Bool} \triangleq \texttt{F} \mid b \mid \neg\mathsf{Bool} \mid \mathsf{Bool} \wedge \mathsf{Bool}$$

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

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

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

A *precondition* $Pre(\rho, S)$ for $\rho \in \mathsf{Pred}$ with respect to a statement S is a first-order formula that guarantees ρ after the execution of the statement S.

$$\begin{aligned}
Pre(\rho, \texttt{nop}) &= \rho \\
Pre(\rho, \texttt{assume}\ \theta) &= \theta \Rightarrow \rho \\
Pre(\rho, S_0; S_1) &= Pre(Pre(\rho, S_1), S_0) \\
Pre(\rho, x := \pi) &= \begin{cases} \forall x.\rho & \text{if } \pi = \texttt{nondet} \\ \rho[x \mapsto \pi] & \text{otherwise} \end{cases} \\
Pre(\rho, b := \pi) &= \begin{cases} \forall b.\rho & \text{if } \pi = \texttt{nondet} \\ \rho[b \mapsto \pi] & \text{otherwise} \end{cases}
\end{aligned}$$

$$Pre(\rho, a[e]:=\pi) = \begin{cases} \forall i.(i \neq e \Rightarrow \rho') \wedge (i = e \Rightarrow \rho'[a[e] \mapsto \pi]) \\ \qquad\qquad \text{if } \rho = \forall i.\rho' \text{ and } \rho' \text{ contains } a[i] \\ \rho[a[e] \mapsto \pi] \quad \text{otherwise} \end{cases}$$

$$Pre(\rho, \texttt{if } \rho \texttt{ then } S_0 \texttt{ else } S_1) = (\rho \Rightarrow Pre(\rho, S_0)) \wedge (\neg\rho \Rightarrow Pre(\rho, S_1))$$

$$Pre(\rho, \texttt{switch } \pi \texttt{ case } \pi_i\texttt{: } S_i) = \bigwedge_i (\pi = \pi_i \Rightarrow Pre(\rho, S_i))$$

$$Pre(\rho, \{\delta\} \texttt{ while } \kappa \texttt{ do } S \{\epsilon\}) = \begin{cases} \delta \text{ if } \epsilon \text{ implies } \rho \\ \texttt{F} \text{ otherwise} \end{cases}$$

A *valuation* v 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 v is a *model* of a first-order formula ρ (written $v \models \rho$) if ρ evaluates to $\texttt{T}$ under v. 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 $\texttt{T}$ under μ.

Given a first-order formula ρ, an *SMT solver* (Dutetre and de Moura 2006; Kroening and Strichman 2008) returns a model of ν if it exists. In general, an SMT solver is incomplete over quantified formulae and may return a potential model (written $SMT(\rho) \xrightarrow{!} \nu$). It returns $UNSAT$ (written $SMT(\rho) \rightarrow UNSAT$) if the solver proves the formula unsatisfiable. Note that an SMT solver can only err when it returns a (potential) model. If $UNSAT$ is returned, the input formula is certainly unsatisfiable.

3. Problems and solutions

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

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

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

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

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

4. The CDNF algorithm

In Bshouty (1995), an exact learning algorithm for Boolean formulae over a finite set B of Boolean variables is introduced. The CDNF algorithm generates a conjunction of formulae in disjunctive normal form equivalent to the unknown Boolean formula λ. It assumes a teacher to answer the following queries:

1. *Membership queries.* Let μ be a Boolean valuation for B. The membership query $MEM(\mu)$ asks if μ is a model of the unknown Boolean formula λ. If $\mu \models \lambda$, the teacher answers *YES* (denoted by $MEM(\mu) \rightarrow YES$). Otherwise, the teacher answers *NO* (denoted by $MEM(\mu) \rightarrow NO$).
2. *Equivalence queries.* Let $\beta \in \mathsf{Bool}_B$. The equivalence query $EQ(\beta)$ asks if β is equivalent to the unknown Boolean formula λ. If so, the teacher answers *YES* (denoted by $EQ(\beta) \rightarrow YES$). Otherwise, the teacher returns a Boolean valuation μ for B such that $\mu \models \beta \oplus \lambda$ as a counterexample (denoted by $EQ(\beta) \rightarrow \mu$).

Let μ and a be Boolean valuations for B. The Boolean valuation $\mu \oplus a$ is defined by $(\mu \oplus a)(b_i) = \mu(b_i) \oplus a(b_i)$ for $b_i \in B$. For any Boolean formula β, $\beta[B \mapsto B \oplus a]$ is the Boolean formula obtained from β by replacing $b_i \in B$ with $\neg b_i$ if $a(b_i) = \mathtt{T}$. For a set S of Boolean valuations for B, define

$$M_{DNF}(\mu) = \bigwedge_{\mu(b_i)=\mathtt{T}} b_i \quad \text{and} \quad M_{DNF}(S) = \bigvee_{\mu \in S} M_{DNF}(\mu).$$

For the degenerate cases, $M_{DNF}(\mu) = \mathtt{T}$ when $\mu \equiv \mathtt{F}$ and $M_{DNF}(\varnothing) = \mathtt{F}$. Algorithm 1 shows the CDNF algorithm (Bshouty 1995). In the algorithm, the step 'walk from μ towards a while keeping $\mu \models \lambda$' takes two Boolean valuations μ and a. It flips the assignments in μ different from those of a and maintains $\mu \models \lambda$. Algorithm 2 implements the walking step by membership queries.

Intuitively, the CDNF algorithm computes the conjunction of approximations to the unknown Boolean formula. In Algorithm 1, H_i records the approximation generated from the set S_i of Boolean valuations with respect to the Boolean valuation a_i. The algorithm checks if the conjunction of approximations H_i's is the unknown Boolean formula (line **5**).

(* $B = \{b_1, b_2, \ldots, b_m\}$: a finite set of Boolean variables *)
Input: A teacher answers membership and equivalence queries for an unknown Boolean formula λ
Output: A Boolean formula equivalent to λ

1 $t := 0$;
2 **if** $EQ(\mathtt{T}) \to YES$ **then return** T;
3 let μ be such that $EQ(\mathtt{T}) \to \mu$;
4 $t := t + 1$; $(H_t, S_t, a_t) := (\mathtt{F}, \varnothing, \mu)$;
5 **if** $EQ(\bigwedge_{i=1}^{t} H_i) \to YES$ **then return** $\bigwedge_{i=1}^{t} H_i$;
6 let μ be such that $EQ(\bigwedge_{i=1}^{t} H_i) \to \mu$;
7 $I := \{i : \mu \not\models H_i\}$;
8 **if** $I = \varnothing$ **then goto 4**;
9 **foreach** $i \in I$ **do**
10 $\quad \mu_i := \mu$;
11 $\quad$ walk from μ_i towards a_i while keeping $\mu_i \models \lambda$;
12 $\quad S_i := S_i \cup \{\mu_i \oplus a_i\}$;
13 **end**
14 $H_i := M_{DNF}(S_i)[B \mapsto B \oplus a_i]$ for $i = 1, \ldots, t$;
15 **goto 5**;

Algorithm 1: The CDNF algorithm.

(* $B = \{b_1, b_2, \ldots, b_m\}$: a finite set of Boolean variables *)
Input: valuations μ and a for B
Output: a model μ of λ by walking towards a

1 $i := 1$;
2 **while** $i \leqslant m$ **do**
3 $\quad$ **if** $\mu(b_i) \neq a(b_i)$ **then**
4 $\quad\quad \mu(b_i) := \neg\mu(b_i)$;
5 $\quad\quad$ **if** $MEM(\mu) \to YES$ **then** $i := 0$ **else** $\mu(b_i) := \neg\mu(b_i)$;
6 $\quad$ **end**
7 $\quad i := i + 1$;
8 **end**
9 **return** μ

Algorithm 2: Walking towards a.

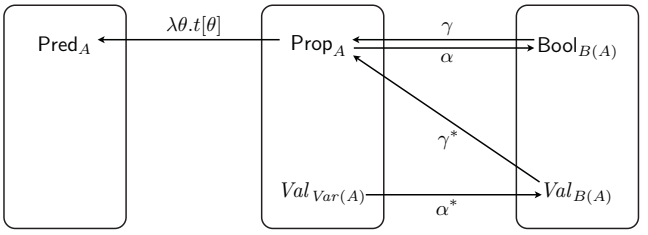


Fig. 2. The domains Pred_A, Prop_A and $\mathsf{Bool}_{B(A)}$.

If it is, we are done. Otherwise, the algorithm tries to refine H_i by expanding S_i. If none of H_i's can be refined (line **8**), another approximation is added (line **4**). The algorithm reiterates after refining the approximations H_i's (line **15**). Let λ be a Boolean formula, $|\lambda|_{DNF}$ and $|\lambda|_{CNF}$ denote the minimum sizes of λ in disjunctive and conjunctive normal forms respectively. The CDNF algorithm learns any Boolean formula λ with a polynomial number of queries in $|\lambda|_{DNF}$, $|\lambda|_{CNF}$, and the number of Boolean variables (Bshouty 1995). Appendix A gives a sample run of the CDNF algorithm.

5. Predicate abstraction with a template

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

The following lemmas are useful in proving our technical results:

Lemma 5.1. Let A be a set of atomic propositions, $\theta, \rho \in \mathsf{Prop}_A$. Then

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

Proof. Let $\alpha(\theta) = \bigvee_i \beta_i$ where β_i is a canonical monomial and $\theta \wedge \gamma(\beta_i)$ is satisfiable. By Lemma 5.2, $\gamma(\beta_i) \Rightarrow \theta$. Hence $\gamma(\beta_i) \Rightarrow \rho$ and $\rho \wedge \gamma(\beta_i)$ is satisfiable. □

Lemma 5.2. Let A be a set of atomic propositions, $\theta \in \mathsf{Prop}_A$, and β a canonical monomial in $\mathsf{Bool}_{B(A)}$. Then $\theta \wedge \gamma(\beta)$ is satisfiable if and only if $\gamma(\beta) \Rightarrow \theta$.

Proof. Let $\theta' = \bigvee_i \theta_i \in \mathsf{Prop}_A$ be a propositional formula in disjunctive normal form such that θ' is equivalent to θ.

Assume $\theta \wedge \gamma(\beta)$ is satisfiable. Then $\theta' \wedge \gamma(\beta)$ is satisfiable and $\theta_i \wedge \gamma(\beta)$ is satisfiable for some i. Since β is canonical, each atomic propositions in A appears in $\gamma(\beta)$. Hence $\theta_i \wedge \gamma(\beta)$ is satisfiable implies $\gamma(\beta) \rightarrow \theta_i$. We have $\gamma(\beta) \Rightarrow \theta$.

The other direction is trivial. □

Recall that a teacher for the CDNF algorithm answers queries in the abstract domain, and an SMT solver computes models in the concrete domain. In order to let an SMT solver play the role of a teacher, more transformations (α^*, β^*) are needed.

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

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

Lemma 5.3. Let A be a set of atomic propositions and $\theta \in \mathsf{Prop}_A$. Then $\theta \leftrightarrow \gamma(\alpha(\theta))$.

Proof. Let $\theta' = \bigwedge_i \theta_i$ be a quantified-free formula in disjunctive normal form such that $\theta' \leftrightarrow \theta$. Let $\mu \in \mathsf{Bool}_{B(A)}$. Define

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

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

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

Conversely, assume $v \models \gamma(\alpha(\theta))$. Then $v \models \gamma(\beta)$ for some canonical monomial β and $\gamma(\beta) \wedge \theta$ is satisfiable. By Lemma 5.2, $\gamma(\beta) \rightarrow \theta$. Hence $v \models \theta$. □

The following lemmas characterize relations among these functions:

Lemma 5.4 (Jung *et al.* 2010). Let A be a set of atomic propositions, $\theta \in \mathsf{Prop}_A$, $\beta \in \mathsf{Bool}_{B(A)}$, and v a valuation for $Var(A)$. Then,

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

Proof.

1. Assume $v \models \theta$. $\chi(\alpha^*(v))$ is a canonical monomial. Observe that $v \models \gamma(\chi(\alpha^*(v)))$. Hence $\gamma(\chi(\alpha^*(v))) \wedge \theta$ is satisfiable. By the definition of $\alpha(\theta)$ and $\chi(\alpha^*(v))$ is canonical, $\chi(\alpha^*(v)) \rightarrow \alpha(\theta)$. $\alpha^*(v) \models \alpha(\theta)$ follows from $\alpha^*(v) \models \chi(\alpha^*(v))$.

Conversely, assume $\alpha^*(v) \models \alpha(\theta)$. Then $\alpha^*(v) \models \beta$ where β is a canonical monomial and $\gamma(\beta) \wedge \theta$ is satisfiable. By the definition of $\alpha^*(v)$, $v \models \gamma(\beta)$. Moreover, $\gamma(\beta) \rightarrow \theta$ by Lemma 5.2. Hence $v \models \theta$.

2. Assume $v \models \gamma(\beta)$. By Lemma 5.4(1), $\alpha^*(v) \models \alpha(\gamma(\beta))$. Note that $\beta = \alpha(\gamma(\beta))$. Thus $\alpha^*(v) \models \beta$.

□

Lemma 5.5 (Jung *et al.* 2010). 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 $\mu \models \alpha(\theta)$.

Proof. Assume $\gamma^*(\mu) \rightarrow \theta$. By Lemma 5.1, $\alpha(\gamma^*(\mu)) \rightarrow \alpha(\theta)$. Note that $\gamma^*(\mu) = \gamma(\chi(\mu))$. By Lemma 5.3, $\chi(\mu) \rightarrow \alpha(\theta)$. Since $\mu \models \chi(\mu)$, we have $\mu \models \alpha(\theta)$.

Conversely, assume $\mu \models \alpha(\theta)$. We have $\chi(\mu) \rightarrow \alpha(\theta)$ by the definition of $\chi(\mu)$. Let $v \models \gamma^*(\mu)$, that is, $v \models \gamma(\chi(\mu))$. By Lemma 5.4(2), $\alpha^*(v) \models \chi(\mu)$. Since $\chi(\mu) \rightarrow \alpha(\theta)$, $\alpha^*(v) \models \alpha(\theta)$. By Lemma 5.4(1), $v \models \theta$. Therefore, $\gamma^*(\mu) \rightarrow \theta$. □

6. Learning invariants

Consider the `while` statement

$$\{\delta\}\ \texttt{while}\ \kappa\ \texttt{do}\ S\ \{\epsilon\}.$$

The propositional formula κ is called the *guard* of the `while` statement; the statement S is called the *body* of the `while` statement. The annotation is intended to denote that if the precondition δ holds, then the postcondition ϵ must hold after the execution of the `while` statement. The *invariant generation problem* is to compute an invariant to justify the pre- and post-conditions.

Definition 6.1. Let $\{\delta\}\ \texttt{while}\ \kappa\ \texttt{do}\ S\ \{\epsilon\}$ be a `while` statement. An *invariant* ι is a propositional formula such that

$$\text{(a)}\delta \Rightarrow \iota \qquad \text{(b)}\kappa \wedge \iota \Rightarrow \mathit{Pre}(\iota, S) \qquad \text{(c)}\neg\kappa \wedge \iota \Rightarrow \epsilon.$$

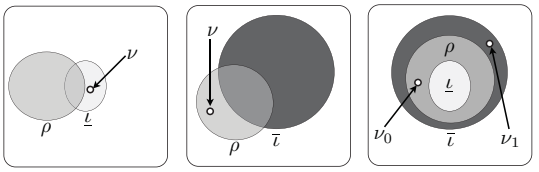
An invariant allows us to prove that the `while` statement fulfills the annotated requirements. Observe that Definition 6.1 (c) is equivalent to $\iota \Rightarrow \epsilon \vee \kappa$. Along with Definition 6.1 (a), we see that any invariant must be weaker than δ but stronger than $\epsilon \vee \kappa$. Hence δ and $\epsilon \vee \kappa$ are called the *strongest* and *weakest* approximations to invariants for $\{\delta\}\ \texttt{while}\ \kappa\ \texttt{do}\ S\ \{\epsilon\}$ respectively.

Our goal is to apply the CDNF algorithm (Algorithm 1) to 'learn' an invariant for an annotated `while` statement. To achieve this goal, we first lift the invariant generation problem to the abstract domain by predicate abstraction and templates. Moreover, we need to devise a mechanism to answer queries from the learning algorithm in the abstract domain. In the following, we show how to answer queries by an SMT solver and invariant approximations.

We present our query resolution algorithms, followed by the invariant generation algorithm. The query resolution algorithms exploit the information derived from the

```
   /* ι : an under-approximation; ῑ : an over-approximation          */
   /* t[]: the given template                                        */
   Input: β ∈ Bool_{B(A)}
   Output: YES, or a counterexample v s.t. α*(v) ⊨ β ⊕ λ
1  ρ := t[γ(β)];
2  if SMT(ι ∧ ¬ρ) → UNSAT and SMT(ρ ∧ ¬ῑ) → UNSAT and
3     SMT(κ ∧ ρ ∧ ¬Pre(ρ, S)) → UNSAT then return YES;
4  if SMT(ι ∧ ¬ρ) -!-> v then return α*(v);
5  if SMT(ρ ∧ ¬ῑ) -!-> v then return α*(v);
6  if SMT(ρ ∧ ¬ι) -!-> v0 or SMT(ῑ ∧ ¬ρ) -!-> v1 then
7     return α*(v0) or α*(v1) randomly;
```

Algorithm 3: Resolving equivalence queries.



(a) (b) (c)

Fig. 3. Counterexamples in equivalence query resolution (c.f. Algorithm 3): (a) a counterexample inside the under-approximation $\underline{\iota}$ but outside the candidate ρ (line **4**); (b) a counterexample inside the candidate ρ but outside the over-approximation $\bar{\iota}$ (line **5**); (c) a random counterexample v_0 (or v_1) inside the candidate ρ (or over-approximation $\bar{\iota}$) but out of the under-approximation $\underline{\iota}$ (or candidate ρ), respectively (line **6** and **7**).

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

6.1. *Equivalence queries*

An equivalence query $EQ(\beta)$ with $\beta \in \mathsf{Bool}_{B(A)}$ asks if β is equivalent to the unknown target λ. Algorithm 3 gives our equivalence resolution algorithm. It first checks if $\rho = t[\gamma(\beta)]$ is indeed an invariant for the annotated loop by verifying $\underline{\iota} \Rightarrow \rho$, $\rho \Rightarrow \bar{\iota}$ and $\kappa \wedge \rho \Rightarrow Pre(\rho, S)$ with an SMT solver (line **2** and **3**). If so, the CDNF algorithm has generated an invariant and our teacher acknowledges that the target has been found. If the candidate ρ is not an invariant, we need to provide a counterexample. Figure 3 describes the process of counterexample discovery. The algorithm first tries to generate a counterexample inside of under-approximation (a), or outside of over-approximation

(b). If it fails to find such counterexamples, the algorithm tries to return a valuation distinguishing ρ from invariant approximations as a random answer (c).

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

6.2. *Membership queries*

```
   /* ι : an under-approximation; ῑ : an over-approximation          */
   /* t[]: the given template                                        */
   Input: a valuation μ for B(A)
   Output: YES or NO
1  if SMT(γ*(μ)) → UNSAT then return NO;
2  ρ := t[γ*(μ)];
3  if SMT(ρ ∧ ¬ῑ) --!--> v then return NO;
4  if SMT(ρ ∧ ¬ι) → UNSAT and isWellFormed(t[], γ*(μ)) then return YES;
5  return YES or NO randomly
```

Algorithm 4: Resolving membership queries.

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

Lemma 6.1. Let $t[] \in \tau$ be a template. For any $\theta_1, \theta_2 \in \mathsf{Prop}_A, \theta_1 \Rightarrow \theta_2$ implies $t[\theta_1] \Rightarrow t[\theta_2]$.

By Lemma 6.1 and $t[\gamma^*(\mu)] \not\Rightarrow \bar{\iota}$ (line **3**), we have $\gamma^*(\mu) \not\Rightarrow \gamma(\lambda)$. Hence $\mu \not\models \lambda$ (Lemma 5.5).

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

Definition 6.2. 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.

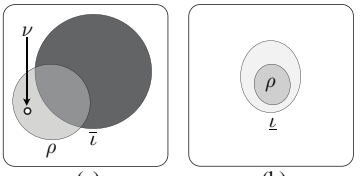


Fig. 4. Resolving a membership query with invariant approximations (c.f. Algorithm 4): (a) the guess ρ is not included in the over-approximation $\bar{\iota}$ (line **3**); (b) the guess ρ is included in the under-approximation $\underline{\iota}$ (line **4**).

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

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

Lemma 6.2. Let A be a set of atomic propositions, $\theta_1 \in \mathsf{Prop}_A$, and $t[] \in \tau$ a well-formed template with respect to θ_1. For any $\theta_2 \in \mathsf{Prop}_A$, $t[\theta_1] \Rightarrow t[\theta_2]$ implies $\theta_1 \Rightarrow \theta_2$.

By Lemma 6.2 and 5.5, we have $\mu \models \lambda$ from $\rho \Rightarrow \underline{\iota}$ (line **4**) and the well formedness of $t[]$ with respect to $\gamma^*(\mu)$. As in the case of the equivalence query resolution algorithm, incorrect models from SMT solvers (line **3**) simply guide the CDNF algorithm to other quantified invariants. Note that Algorithm 4 also gives a random answer if a membership query cannot be resolved through invariant approximations. The correctness of generated invariants is ensured by SMT solvers in the equivalence query resolution algorithm (Algorithm 3).

6.3. *Main loop*

Algorithm 5 shows our invariant generation algorithm. It invokes the CDNF algorithm in the main loop. Whenever a query is made, our algorithm uses one of the query resolution algorithms (Algorithm 3 or 4) to give an answer. In both query resolution algorithms, we use the strongest under-approximation δ and the weakest over-approximation $\kappa \vee \epsilon$ to resolve queries from the learning algorithm. Observe that the equivalence and membership query resolution algorithms give random answers independently. They may send inconsistent answers to the CDNF algorithm. When inconsistencies arise, the main loop forces the learning algorithm to restart (line **6**). If the CDNF algorithm infers a Boolean formula $\lambda \in \mathsf{Bool}_{B(A)}$, the first-order formula $t[\gamma(\lambda)]$ is an invariant for the annotated loop in the form of the template $t[]$.

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

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

Algorithm 5: Main loop.

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

Similar to other invariant generation techniques based on predicate abstraction, our algorithm is not guaranteed to generate invariants. If no invariant can be expressed by the template with a given set of atomic propositions, our algorithm will not terminate. Moreover, if no invariant in the form of the given template can be verified by SMT solvers, our algorithm does not terminate either. On the other hand, if there is one verifiable invariant in the form of the given template, there is a sequence of random answers that leads to the verifiable invariant. If sufficiently many verifiable invariants are expressible in the form of the template, random answers almost surely guide the learning algorithm to one of them. Since our algorithmic learning approach with random answers does not commit to any particular invariant, it can be more flexible and hence effective than traditional deterministic techniques in practice.

7. Experiments

We have implemented a prototype in OCaml[†]. In our implementation, we use YICES as the SMT solver to resolve queries (Algorithm 4 and 3).

Table 1 shows experimental results. For quantifier-free invariant generation, we chose five `while` statements from SPEC2000 benchmarks and Linux device drivers. Among

[†] Available at http://ropas.snu.ac.kr/aplas10/qinv-learn-released.tar.gz

Table 1. *Experimental results.*

case	Template	*AP*	*MEM*	*EQ*	MEM_R	EQ_R	*ITER*	Time (s)	σ_{Time}(s)
`vpr`	[]	5	18.8	7.8	71%	22%	1.6	0.05	0.03
`ide_ide_tape`	[]	5	22.2	12.8	43%	38%	2.5	0.07	0.03
`ide_wait_ireason`	[]	5	3,103	1,804	38%	26%	187	1.10	1.07
`usb_message`	[]	10	1,260	198	29%	25%	20	2.29	1.90
`parser`	[]	15	809	139	25%	5%	4	2.96	2.05
`max`	$\forall k.[]$	7	5,968	1,742	65%	26%	269	5.71	7.01
`selection_sort`	$\forall k_1.\exists k_2.[]$	6	9,630	5,832	100%	4%	1,672	9.59	11.03
`devres`	$\forall k.[]$	7	2,084	1,214	91%	21%	310	0.92	0.66
`rm_pkey`	$\forall k.[]$	8	2,204	919	67%	20%	107	2.52	1.62
`tracepoint1`	$\exists k.[]$	4	246	195	61%	25%	31	0.26	0.15
`tracepoint2`	$\forall k_1.\exists k_2.[]$	7	33,963	13,063	69%	5%	2,088	157.55	230.40

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

five `while` statements, the cases `parser` and `vpr` are extracted from PARSER and VPR in SPEC2000 benchmarks respectively. The other three cases are extracted from Linux 2.6.28 device drivers: both `ide-ide-tape` and `ide-wait-ireason` are from IDE driver; `usb-message` is from USB driver. For quantified invariant generation, we took two cases from the ten benchmarks in Srivastava and Gulwani (2009) with the same annotation (`max` and `selection_sort`). We also chose four `for` statements from Linux 2.6.28. Benchmark `devres` is from library, `tracepoint1` and `tracepoint2` are from kernel, and `rm_pkey` is from InfiniBand device driver. We translated them into our language and annotated pre and post conditions manually. Sets of atomic proposition are manually chosen from the program texts.

For each case, we report the number of atomic propositions (*AP*), the number of membership queries (*MEM*), the number of equivalence queries (*EQ*), the percentages of randomly resolved membership queries (MEM_R) and equivalence queries (EQ_R), the number of the CDNF algorithm invocations (*ITER*), the execution time and standard deviation of the running time (σ_{Time}). The data are the average of 500 runs and collected on a 2.6GHz Intel E5300 Duo Core with 3GB memory running Linux 2.6.28.

7.1. *Quantifier-free invariants*

For inferring quantifier-free invariants specific templates are not required. We use identity template [].

7.1.1. *`ide-ide-tape` from Linux IDE driver.* Figure 5 is a `while` statement extracted from Linux IDE driver.It copies data of size *n* from tape records. The variable *count* contains the size of the data to be copied from the current record (*bh_b_size* and

```
{ ret = 0 ∧ bh_b_count ⩽ bh_b_size }
1 while n > 0 do
2    if (bh_b_size − bh_b_count) < n then count := bh_b_size − bh_b_count
3    else count := n;
4    b := nondet;
5    if b then ret := 1;
6    n := n − count; bh_b_count := bh_b_count + count;
7    if bh_b_count = bh_b_size then
8      bh_b_size := nondet; bh_b_count := nondet; bh_b_count := 0;
9 end
{ n = 0 ∧ bh_b_count ⩽ bh_b_size }
```

Fig. 5. A sample loop in Linux IDE driver.

bh_b_count). If the current tape record runs out of data, more data are copied from the next record. The flexibility in invariants can be witnessed in the following run. After successfully resolving 3 equivalence and 7 membership queries, the CDNF algorithm makes the following membership query unresolvable by the invariant approximations:

$$\overbrace{n > 0 \wedge (bh\_b\_size - bh\_b\_count) < n \wedge ret \neq 0}^{\rho} \wedge bh\_b\_count = bh\_b\_size.$$

Answering *NO* to this query leads to the following unresolvable membership query after successfully resolving two more membership query:

$$\rho \wedge bh\_b\_count \neq bh\_b\_size \wedge bh\_b\_count \leqslant bh\_b\_size.$$

We proceed with a random answer *YES*. After successfully resolving one more membership queries, we reach the following unresolvable membership query:

$$\rho \wedge bh\_b\_count \neq bh\_b\_size \wedge bh\_b\_count > bh\_b\_size.$$

For this query, both answers lead to invariants. Answering *YES* yields the following invariant:

$$n \neq 0 \vee (bh\_b\_size - bh\_b\_count) \geqslant n.$$

Answering *NO* yields the following invariant:

$$(bh\_b\_count \leqslant bh\_b\_size \wedge n \neq 0) \vee (bh\_b\_size - bh\_b\_count) \geqslant n.$$

Note that they are two different invariants. The equivalence query resolution algorithm (Algorithm 3) ensures that both fulfill the conditions in Definition 6.1.

7.1.2. *`parser` from VPR in SPEC2000 benchmarks.* Figure 6 shows a sample `while` statement from the `parser` program in SPEC2000 benchmark.In the `while` body, there are three locations where *give_up* or *success* is set to `T`. Thus one of these conditions in the `if` statements must hold (the first conjunct of postcondition). Variable *valid* may get an arbitrary value if *linkages* is not zero. But it cannot be greater than *linkages* by the `assume` statement (the second conjunct of postcondition). The variable *linkages* gets

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

Fig. 6. A sample loop in SPEC2000 benchmark PARSER.

an arbitrary value near the end of the `while` body. But it cannot be greater than 5000 (the fourth conjunct), and always equal to the variable *canonical* (the third conjunct of postcondition). Despite the complexity of the postcondition and the `while` body, our approach is able to compute an invariant in 4 iterations on average.

One of the found invariants is the following:

$$
\begin{aligned}
&success \Rightarrow (valid \leqslant linkages \wedge linkages \leqslant 5000 \wedge canonical = linkages) \bigwedge \\
&success \Rightarrow (\neg search \vee count > words \vee valid \neq 0) \bigwedge \\
&success \Rightarrow (count > words \vee cutoff = maxcost \vee \\
&\qquad (canonical \neq 0 \wedge valid \neq 0 \wedge linkages \neq 0)) \bigwedge \\
&give\_up \Rightarrow ((valid = 0 \wedge linkages = 0 \wedge canonical = linkages) \vee \\
&\qquad (canonical \neq 0 \wedge valid \leqslant linkages \wedge linkages \leqslant 5000 \wedge canonical = linkages)) \bigwedge \\
&give\_up \Rightarrow (cutoff = maxcost \vee count > words \vee \\
&\qquad\qquad (canonical \neq 0 \wedge valid \neq 0 \wedge linkages \neq 0)) \bigwedge \\
&give\_up \Rightarrow (\neg search \vee count > words \vee valid \neq 0)
\end{aligned}
$$

This invariant describes the conditions when *success* or *give_up* are true. For instance, it specifies that $valid \leqslant linkages \wedge linkages \leqslant 5000 \wedge canonical = linkages$ should hold if *success* is true. In Figure 6, we see that *success* is assigned to `T` at line 18 when *valid* is positive. Yet *valid* is set to 0 at line 14. Hence line 16 and 17 must be executed. Thus,

the first ($valid \leqslant linkages$) and the third ($canonical = linkages$) conjuncts hold. Moreover, line 13 ensures that the second conjunct ($linkages \leqslant 5000$) holds as well.

7.2. *Quantified invariants*

For generating quantified invariants, user should carefully provide templates to our framework. In practice, the quantified variables in postconditions can give hints to users. In our experiments all templates are able to be extracted from postconditions. We find that simple templates, such as $\forall k_1.\exists k_2.[]$, which only specify the quantified variables and type of quantification are useful enough to derive quantified invariants in our experiments.

7.2.1. `devres` *from Linux library*. Figure 7(c) shows an annotated loop extracted from a Linux library.In the postcondition, we assert that *ret* implies $tbl[i] = 0$, and every element in the array $tbl[]$ is not equal to *addr* otherwise. Using the set of atomic propositions $\{tbl[k] = addr,\ i < n,\ i = n,\ k < i,\ tbl[i] = 0,\ ret\}$ and the simple template $\forall k.[]$, our algorithm finds following quantified invariants in different runs:

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

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.

7.2.2. `selection_sort`. Consider the selection sort algorithm (Srivastava and Gulwani 2009) in Figure 7(b). Let $'a[]$ denote the content of the array $a[]$ before the algorithm is executed. The postcondition states that the contents of array $a[]$ come from its old contents. In this test case, we apply our invariant generation algorithm to compute an invariant to establish the postcondition of the outer loop. For computing the invariant of the outer loop, we make use of the inner loop's specification.

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

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

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

7.2.3. `rm_pkey` *from Linux InfiniBand driver*. Figure 7(a) is a `while` statement extracted from Linux InfiniBand driver.The conjuncts in the postcondition represent (1) if the loop terminates without break, all elements of *pkeys* are not equal to *key* (line 2); (2) if the loop

(a) `rm_pkey`

$\{\ i = 0 \wedge key \neq 0 \wedge \neg ret \wedge \neg break\}$

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

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

(b) `selection_sort`

$\{\ i = 0\ \}$

```
1  while i < n − 1 do
2    min:=i;
3    j:=i + 1;
4    while j < n do
5      if a[j] < a[min] then
6        min:=j;
7      j:=j + 1;
8    done
9    if i≠min then
10     tmp:=a[i];
11     a[i]:=a[min];
12     a[min]:=tmp;
13   i:=i + 1;
14 done
```

$\{(i \geqslant n - 1)$
$\wedge (\forall k_1.k_1 < n \Rightarrow$
$(\exists k_2.k_2 < n \wedge a[k_1] = {}'a[k_2]))\}$

(c) `devres`

$\{\ i = 0 \wedge \neg ret\ \}$

```
1 while i < n ∧ ¬ret do
2   if tbl[i] = addr then
3     tbl[i]:=0; ret:=true
4   else
5     i:=i + 1
6 end
```

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

Fig. 7. Benchmark examples: (a) `rm_pkey` from Linux InfiniBand driver, (b) `selection_sort` and (c) `devres` from Linux library.

terminates with break but *ret* is false, then $pkeys[i]$ is equal to *key* (line 2) but $pkeyrefs[i]$ is not equal to zero (line 4); (3) if *ret* is true after the loop, then both $pkeyrefs[i]$ (line 4) and $pkeys[i]$ (line 5) are equal to zero. From the postcondition, we guess that an invariant can be universally quantified with k. Using the simple template $\forall k.[]$ and the set of atomic propositions $\{ret,\ break,\ i < n,\ k < i,\ pkeys[i] = 0,\ pkeys[i] = key,\ pkeyrefs[i] = 0,\ pkeyrefs[k] = key\}$, our algorithm finds following quantified invariants in different runs:

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

In spite of undecidability of first-order theories in Yices and random answers, each of the 500 runs in our experiments infers an invariant successfully. Moreover, several quantified invariants are found in each case among 500 runs. This suggests that invariants are abundant. Note that the templates in the test cases `selection_sort` and `tracepoint2` have alternating quantification. Satisfiability of alternating quantified formulae is in general undecidable. That is why both cases have substantially more restarts than the

others. Interestingly, our algorithm is able to generate a verifiable invariant in each run. Our simple randomized mechanism proves to be effective even for most difficult cases.

8. Discussion and future work

The complexity of our technique depends on the distribution of invariants. It works most effectively if invariants are abundant. The number of iterations depends on the outcomes of coin tossing. The main loop may reiterate several times or not even terminate. Our experiments suggest that there are sufficiently many invariants in practice. Our technique always generates an invariant, even though it takes 2,088 iterations for the most complicated case `tracepoint2` on average.

Since plentiful of invariants are available, sometimes one of them seems to be generated by merely coin tossing. In `selection_sort`, simple over- and under-approximations cannot help resolving membership queries. All membership queries are resolved randomly. However, if we also randomly resolve equivalence queries then our technique does not terminate. Invariant approximations are essential to our framework.

Atomic predicates are collected from program texts, pre and post conditions in our experiments. Recently, automatic predicate generation technique is proposed (Jung *et al.* 2011). However, this technique supports atomic predicates only for quantifier-free invariants. Generating atomic predicates for templates is challenging because quantified variables do not appear on the program text. Better invariant approximations ($\underline{\iota}$ and $\overline{\iota}$) computed by static analysis can be used in our framework. More precise approximations of $\underline{\iota}$ and $\overline{\iota}$ will improve the performance by reducing the number of iterations via increasing the number of resolvable queries. Also, a variety of techniques from static analysis or loop invariant generation (Flanagan and Qadeer 2002; Gulwani *et al.* 2009; Srivastava and Gulwani 2009; Gulwani *et al.* 2008b; Gupta and Rybalchenko 2009; Kovács and Voronkov 2009; Lahiri *et al.* 2004; McMillan 2008) in particular can be integrated to resolve queries in addition to one SMT solver with coin tossing. Such a set of multiple teachers will increase the number of resolvable queries because it suffices to have just one teacher to answer the query to proceed.

9. Related work

Existing impressive techniques for invariant generation can be adopted as the query resolution components (teachers) in our algorithmic learning-based framework. Srivastava and Gulwani (2009) devise three algorithms, two of them use fixed point computation and the other uses a constraint based approach (Gulwani *et al.* 2009, 2008b) to derive quantified invariants. Gupta and Rybalchenko (2009) present an efficient invariant generator. They apply dynamic analysis to make invariant generation more efficient. Flanagan and Qadeer use predicate abstraction to infer universally quantified loop invariants (Flanagan and Qadeer 2002). Predicates over Skolem constants are used to handle unbounded arrays. McMillan (2008) extends a paramodulation-based saturation prover to an interpolating prover that is complete for universally quantified interpolants. He also solves the problem of divergence in interpolated-based invariant generation. Completeness of our framework

can be increased by more powerful decision procedures (Dutetre and de Moura 2006; Ge and Moura 2009; Srivastava *et al.* 2009) and theorem provers (McMillan 2008; Bertot and Castéran 2004; Nipkow *et al.* 2002).

In contrast to previous template-based approaches (Srivastava and Gulwani 2009; Gulwani *et al.* 2008a), our template is more general as it allows arbitrary hole-filling quantifier-free formulae. The templates in Srivastava and Gulwani (2009) can only be filled with formulae over conjunctions of predicates from a given set. Any disjunction must be explicitly specified as part of a template. In Gulwani *et al.* (2008a), the authors consider invariants of the form $E \wedge \bigwedge_{j=1}^{n} \forall U_j(F_j \Rightarrow e_j)$, where E, F_j and e_j must be quantifier free finite conjunctions of atomic facts.

With respect to the analysis of properties of array contents, Halbwachs *et al.* (Halbwachs and Péron 2008) 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 Henzinger *et al.* (2010). For inferring range predicates, Jhala and McMillan (2007) 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 (Jhala and McMillan 2007) fails to prove that an implementation of insertion sort correctly sorts an array.

10. Conclusions

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

Under- and over-approximations are presently derived from annotations provided by users. They can in fact be obtained by other techniques such as static analysis. The integration of various refinement techniques for predicate abstraction will certainly be an important future work.

Appendix A. An Example of the CDNF Algorithm

Let us apply Algorithm 1 to learn the Boolean formula $b_0 \oplus b_1$. The algorithm first makes the query $EQ(\mathtt{T})$ (Figure 8). The teacher responds by giving the valuation $\mu_1(b_0) = \mu_1(b_1) = 0$ (denoted by $\mu_1(b_0 b_1) = 00$). Hence Algorithm 1 assigns $\varnothing$ to S_1, $\mathtt{F}$ to H_1, and μ_1 to a_1. Next, the query $EQ(H_1)$ is made and the teacher responds with the valuation $\mu_2(b_0 b_1) = 01$. Since $\mu_2 \not\models \mathtt{F}$, we have $I = \{1\}$. Algorithm 1 now walks from μ_2 towards a_1. Since flipping $\mu_2(b_1)$ would not give us a model of $b_0 \oplus b_1$, we have $S_1 = \{\mu_2\}$ and

Equivalence query	Answer	I	S_i	H_i	a_i
T	$\mu_1(b_0b_1) = 00$		$S_1 = \varnothing$	$H_1 = \mathtt{F}$	$a_1 = \mu_1$
F	$\mu_2(b_0b_1) = 01$	$\{1\}$	$S_1 = \{\mu_2\}$	$H_1 = b_1$	
b_1	$\mu_3(b_0b_1) = 11$	$\varnothing$	$S_2 = \varnothing$	$H_2 = \mathtt{F}$	$a_2 = \mu_3$
$b_1 \wedge \mathtt{F}$	$\mu_4(b_0b_1) = 01$	$\{2\}$	$S_2 = \{\mu_5\}^\dagger$	$H_2 = \neg b_0$	
$b_1 \wedge \neg b_0$	$\mu_6(b_0b_1) = 10$	$\{1,2\}$	$S_1 = \{\mu_2, \mu_6\}$ $S_2 = \{\mu_5, \mu_7\}^\dagger$	$H_1 = b_1 \vee b_0$ $H_2 = \neg b_0 \vee \neg b_1$	
$(b_1 \vee b_0) \wedge (\neg b_0 \vee \neg b_1)$	*YES*				

† $\mu_5(b_0b_1) = 10$ and $\mu_7(b_0b_1) = 01$

Fig. 8. Learning $b_0 \oplus b_1$.

$H_1 = b_1$. In this example, Algorithm 1 generates $(b_1 \vee b_0) \wedge (\neg b_0 \vee \neg b_1)$ as a representation for the unknown Boolean formula $b_0 \oplus b_1$. Observe that the generated Boolean formula is a conjunction of two Boolean formulae in disjunctive normal form.

References

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

Alur, R., Madhusudan, P. and Nam, W. (2005b) Symbolic compositional verification by learning assumptions. In: CAV. *Springer Lecture Notes in Computer Science* **3576** 548–562.

Bertot, Y. and Castéran, P. (2004) *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science, Springer Verlag.

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

Chen, Y.-F., Farzan, A., Clarke, E. M., Tsay, Y.-K. and Wang, B.-Y. (2009) Learning minimal separating DFA's for compositional verification. In: TACAS. *Springer Lecture Notes in Computer Science* **5505** 31–45.

Cobleigh, J. M., Giannakopoulou, D. and Păsăreanu, C. S. (2003) Learning assumptions for compositional verification. In: TACAS. *Springer Lecture Notes in Computer Science* **2619** 331–346.

Dutetre, B. and de Moura, L. D. (2006) The Yices SMT solver, *Technical report*, SRI International. Available at: `http://yices.csl.sri.com/tool-paper.pdf`

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

Ge, Y. and Moura, L. (2009) Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: CAV. *Springer-Verlag Lecture Notes in Computer Science* **5643** 306–320.

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

Gulwani, S., Srivastava, S. and Venkatesan, R. (2008b) Program analysis as constraint solving. In: *PLDI*, ACM 281–292.

Gulwani, S., Srivastava, S. and Venkatesan, R. (2009) Constraint-based invariant inference over predicate abstraction. In: VMCAI. *Springer Lecture Notes in Computer Science* **5403** 120–135.

Gupta, A., McMillan, K. L. and Fu, Z. (2007) Automated assumption generation for compositional verification. In: CAV. *Springer Lecture Notes in Computer Science* **4590** 420–432.

Gupta, A. and Rybalchenko, A. (2009) Invgen: An efficient invariant generator. In: CAV. *Springer Lecture Notes in Computer Science* **5643** 634–640.

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

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

Jhala, R. and McMillan, K. L. (2007) Array abstractions from proofs. In: CAV. *Springer Lecture Notes in Computer Science* **4590** 193–206.

Jung, Y., Kong, S., Wang, B.-Y. and Yi, K. (2010) Deriving invariants in propositional logic by algorithmic learning, decision procedure and predicate abstraction. In: VMCAI. *Springer Lecture Notes in Computer Science* **5944** 180–196.

Jung, Y., Lee, W., Wang, B.-Y. and Yi, K. (2011) Predicate generation for learning-based quantifier-free loop invariant inference. In: *TACAS* 205–219.

Kong, S., Jung, Y., David, C., Wang, B.-Y. and Yi, K. (2010) Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In: *The 8th ASIAN Symposium on Programming Languages and Systems (APLAS'10)* 328–343.

Kovács, L. and Voronkov, A. (2009) Finding loop invariants for programs over arrays using a theorem prover. In: FASE. *Springer Lecture Notes in Computer Science* **5503** 470–485.

Kroening, D. and Strichman, O. (2008) *Decision Procedures An Algorithmic Point of View*, EATCS, Springer.

Lahiri, S. K., Bryant, R. E. and Bryant, A. E. (2004) Constructing quantified invariants via predicate abstraction. In: VMCAI. *Springer Lecture Notes in Computer Science* **2937** 267–281.

McMillan, K. L. (2008) Quantified invariant generation using an interpolating saturation prover. In: TACAS. *Springer Lecture Notes in Computer Science* **4693** 413–427.

Nipkow, T., Paulson, L. C. and Wenzel, M. (2002) *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science, volume 2283.

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

Srivastava, S., Gulwani, S. and Foster, J. S. (2009) Vs3: Smt solvers for program verification. In: *CAV '09: Proceedings of Computer Aided Verification* 702–708.