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

ROSAEC MEMO
2009-004
August 31, 2009

Deriving Invariants by Algorithmic Learning, Decision Procedures, and Predicate Abstraction

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
bywang@iis.sinica.edu.tw

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

August 28, 2009

Abstract

By combining algorithmic learning, decision procedures, and predicate abstraction, we present an automated technique for finding loop invariants in propositional formulae. Given invariant approximations derived from pre- and post-conditions, our new technique exploits the flexibility in invariants by a simple randomized mechanism. The proposed technique is able to generate invariants for some Linux device drivers and SPEC2000 benchmarks in our experiments.

1 Introduction

Algorithmic learning has been applied to assumption generation in compositional reasoning [1]. 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 in verifying concurrent systems have been reported [1, 2, 3, 4].

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 observing program behavior more. 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 reports our first findings in applying algorithmic learning to invariant generation. We show that the three technologies (algorithmic learning, decision procedures, and predicate abstraction) can be arranged in concert to derive loop invariants in propositional (or, quantifier-free) formulae. The new technique is able to generate invariants for some Linux device drivers and SPEC2000 benchmarks without any help from static or dynamic analyses.

For a while loop, an exact learning algorithm for Boolean formulae searches for invariants by asking queries. Queries can be resolved (not always, see below) by decision procedures automatically. Recall that the learning algorithm generates only Boolean formulae but decision procedures work in propositional formulae. We thus perform predicate abstraction and concretization to integrate the two components.

In reality, information about loop invariant is incomplete. Queries may not be resolvable due to insufficient information. One striking feature of our learning approach is to exploit the flexibility in invariants. 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 there are so many invariants for the given pre- and post-conditions. A little bit of incorrect information does not prevent algorithmic learning from inferring correct invariants. Indeed, the learning algorithm is able to derive invariants in our experiments by coin tossing.

Example

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

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 $i = 0$ and $(i = 10 \wedge b) \vee i < 10$ 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.

Contribution

- We prove that algorithmic learning, decision procedures, and predicate abstraction in combination can automatically infer invariants in propositional formulae for programs in our simple language.
- We demonstrate that the technique works in realistic settings: we are able to generate invariants for some Linux device drivers and SPEC2000 benchmarks in our experiments.
- The technique can be seen as a framework for invariant generation. Static analyzers can contribute by providing information to algorithmic learning. Ours is hence orthogonal to existing techniques.

We organize this paper as follows. After preliminaries (Section 2), we present an overview of the framework in Section 3. In Section 4, we review the exact learning algorithm introduced in [5]. Section 5 gives the details of our learning approach. We report experiments in Section 6. Section 7 briefly discusses our learning approach, future work, and related work. Section 8 concludes our work.

2 The Target Language and Notation

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

$$\begin{array}{rcl} \mathsf{Stmt} & \triangleq & \texttt{nop} \mid \texttt{assume}\ \mathsf{Prop} \mid \mathsf{Stmt};\ \mathsf{Stmt} \mid \\ & & x := \mathsf{Exp} \mid x := \texttt{nondet} \mid b := \mathsf{Bool} \mid b := \texttt{nondet} \mid \\ & & \texttt{if}\ \mathsf{Prop}\ \texttt{then}\ \mathsf{Stmt}\ \texttt{else}\ \mathsf{Stmt} \mid \texttt{switch}\ \mathsf{Exp}\ \texttt{do case}\ \mathsf{Exp} :\ \mathsf{Stmt}\ \cdots \mid \\ & & \{\ \mathsf{Prop}\ \}\ \texttt{while}\ \mathsf{Prop}\ \texttt{do}\ \mathsf{Stmt}\ \{\ \mathsf{Prop}\ \} \end{array}$$

Natural number variables and Boolean variables are allowed. They assign to arbitrary values in their respective domains by the keyword `nondet`. Note that `while` statements are annotated. Programmers are asked to specify a *precondition* before a `while` statement, and a *postcondition* after the statement.

An *expression* Exp is a natural number ($n \in \mathbb{N}$), a variable (x), or a summation or the difference of two expressions.

$$\mathsf{Exp} \ \triangleq\ n \mid x \mid \mathsf{Exp} + \mathsf{Exp} \mid \mathsf{Exp} - \mathsf{Exp}$$

A *propositional formula* Prop is either: the falsehood symbol ($\texttt{F}$), a Boolean variable (b), the negation of a propositional formula, the conjunction of two propositional formulae, or comparisons ($E_0 < E_1$ or $E_0 = E_1$).

$$\mathsf{Prop} \ \triangleq\ \texttt{F} \mid b \mid \neg\mathsf{Prop} \mid \mathsf{Prop} \wedge \mathsf{Prop} \mid \mathsf{Exp} < \mathsf{Exp} \mid \mathsf{Exp} = \mathsf{Exp}$$

Let ρ_0 and ρ_1 be propositional formulae, π_0 and π_1 be expressions. We write $\texttt{T}$ for $\neg\texttt{F}$, $\rho_0 \vee \rho_1$ for $\neg(\neg\rho_0 \wedge \neg\rho_1)$, $\rho_0 \Rightarrow \rho_1$ for $\neg\rho_0 \vee \rho_1$, $\rho_0 \Leftrightarrow \rho_1$ for $(\rho_0 \Rightarrow \rho_1) \wedge (\rho_1 \Rightarrow \rho_0)$, $\rho_0 \oplus \rho_1$ for $\neg(\rho_0 \Leftrightarrow \rho_1)$, $\pi_0 \leq \pi_1$ for $\pi_0 < \pi_1 \vee \pi_0 = \pi_1$, and $\pi_0 \neq \pi_1$ for $\neg(\pi_0 = \pi_1)$. 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, Prop_A denotes the set of propositional formulae generated from A.

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 *valuation* ν is an assignment of natural numbers to variables and truth values to Boolean variables. A *Boolean valuation* μ is an assignment of 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 $Var(A)$. Let ρ be a propositional formula. The valuation ν is a *model* of ρ (written $\nu \models \rho$) if ρ evaluates to $\texttt{T}$ under the valuation ν. Similarly, the Boolean valuation μ is a *Boolean model* of the Boolean formula β (written $\mu \models \beta$) if β evaluates to $\texttt{T}$ under μ. If B is a set of Boolean variables, the set of Boolean valuations for B is denoted by Val_B. Given a propositional formula ρ, a *satisfiability modulo theories (SMT) solver* returns a model of ρ if it exists (written $SMT(\rho) \rightarrow \nu$); otherwise, it returns $UNSAT$ (written $SMT(\rho) \rightarrow UNSAT$) [6, 7].

A *precondition* $Pre(\phi, S)$ for $\phi \in \mathsf{Prop}$ with respect to a statement S is a universally

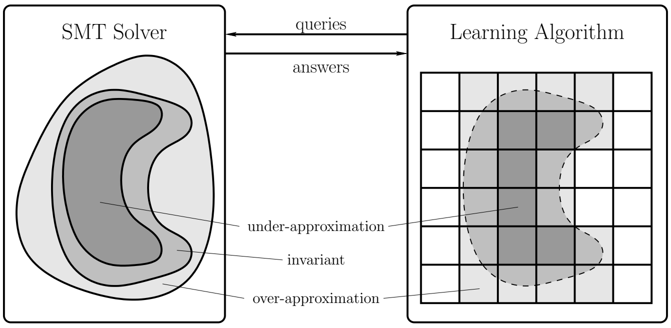

Figure 1: Overview

quantified formula that guarantees ϕ after the execution of the statement S.

$$\begin{array}{rcl}
\mathit{Pre}(\phi, \mathtt{nop}) & = & \phi \\
\mathit{Pre}(\phi, \mathtt{assume}\ \theta) & = & \theta \Rightarrow \phi \\
\mathit{Pre}(\phi, S_0;\ S_1) & = & \mathit{Pre}(\mathit{Pre}(\phi, S_1), S_0) \\
\mathit{Pre}(\phi, x := \pi) & = & \begin{cases} \forall x.\phi & \text{if } \pi = \mathtt{nondet} \\ \phi[x \mapsto \pi] & \text{otherwise} \end{cases} \\
\mathit{Pre}(\phi, b := \rho) & = & \begin{cases} \forall b.\phi & \text{if } \rho = \mathtt{nondet} \\ \phi[b \mapsto \rho] & \text{otherwise} \end{cases} \\
\mathit{Pre}(\phi, \mathtt{if}\ \rho\ \mathtt{then}\ S_0\ \mathtt{else}\ S_1) & = & (\rho \Rightarrow \mathit{Pre}(\phi, S_0)) \wedge (\neg\rho \Rightarrow \mathit{Pre}(\phi, S_1)) \\
\mathit{Pre}(\phi, \mathtt{switch}\ \pi\ \mathtt{case}\ \pi_i \colon S_i) & = & \bigwedge_i (\pi = \pi_i \Rightarrow \mathit{Pre}(\phi, S_i)) \\
\mathit{Pre}(\phi, \{\delta\}\ \mathtt{while}\ \rho\ \mathtt{do}\ S\ \{\epsilon\}) & = & \begin{cases} \delta & \text{if } \epsilon \text{ implies } \phi \\ \mathtt{F} & \text{otherwise} \end{cases}
\end{array}$$

Observe that all universal quantifiers occur positively in $\mathit{Pre}(\phi, S)$ for any S. They can be eliminated by Skolem constants [8, 9].

3 Framework Overview

We combine algorithmic learning, decision procedures [6], and predicate abstraction [10] in our framework. Figure 1 illustrates the relation among these technologies. In the figure, the left side represents the concrete domain; the right side represents the abstract domain. Assume there is an invariant for a `while` statement with respect to the given pre- and post-conditions in the concrete domain. We would like to apply algorithmic learning to find such an invariant.

To this purpose, we use the CDNF algorithm [5]. The CDNF algorithm is an exact learning algorithm for Boolean formulae. It is an active learning algorithm that makes queries about an unknown Boolean formula and outputs a Boolean formula that is equivalent to the unknown one [11, 5]. We perform predicate abstraction to represent propositional formulae as Boolean formulae in the abstract domain. Since the CDNF algorithm is able to learn arbitrary Boolean formulae, our technique can infer arbitrary invariants in propositional formulae by answering queries.

To realize this idea, we devise a mechanism (a teacher) to resolve queries in the abstract domain. 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. Hence, we estimate invariant approximations heuristically and adopt decision procedures for query resolution. For a membership query, we check if its concretization is in the under-approximation or out of the over-approximation by an SMT solver. If it is in the under-approximation, the answer is affirmative; if it is outside the over-approximation, the answer is negative. Otherwise, we simply give a random answer. Equivalence queries are resolved similarly.

4 The CDNF Algorithm

In [5], 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 $EQ(\beta) \rightarrow \mu$).

```
   (* $B = \{b_1, b_2, \ldots, b_m\}$: a finite set of Boolean variables *)
   **Input**: A teacher answers membership and equivalence queries for an unknown Boolean
         formula $\lambda$
   **Output**: A Boolean formula equivalent to $\lambda$
   $t := 0$;
   **if** $EQ(\mathtt{T}) \rightarrow YES$ **then return** $\mathtt{T}$;
   let $\mu$ be such that $EQ(\mathtt{T}) \rightarrow \mu$;
0  $t := t + 1$; $(H_t, S_t, a_t) := (\mathtt{F}, \emptyset, \mu)$;
1  **if** $EQ(\bigwedge_{i=1}^{t} H_i) \rightarrow YES$ **then return** $\bigwedge_{i=1}^{t} H_i$;
   let $\mu$ be such that $EQ(\bigwedge_{i=1}^{t} H_i) \rightarrow \mu$;
   $I := \{i : \mu \not\models H_i\}$;
2  **if** $I = \emptyset$ **then goto 0**;
   **foreach** $i \in I$ **do**
       $\mu_i := \mu$;
       walk from $\mu_i$ towards $a_i$ while keeping $\mu_i \models \lambda$;
       $S_i := S_i \cup \{\mu_i \oplus a_i\}$;
   **end**
   $H_i := M_{DNF}(S_i)[B \mapsto B \oplus a_i]$ for $i = 1, \ldots, t$;
3  **goto 1**;
```

Algorithm 1: The CDNF Algorithm [5]

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, $M_{DNF}(\mu) = \texttt{T}$ when $\mu \equiv \texttt{F}$ and $M_{DNF}(\emptyset) = \texttt{F}$. Algorithm 1 shows the CDNF algorithm [5]. 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.

```
(* $B = \{b_1, b_2, \ldots, b_m\}$: a finite set of Boolean variables *)
Input: valuations $\mu$ and $a$ for $B$
Output: a model $\mu$ of $\lambda$ by walking towards $a$
$i := 1$;
while $i \leq m$ do
    if $\mu(b_i) \neq a(b_i)$ then
        $\mu(b_i) := \neg\mu(b_i)$;
        if $MEM(\mu) \rightarrow YES$ then $i := 0$ else $\mu(b_i) := \neg\mu(b_i)$;
    end
    $i := i + 1$;
end
return $\mu$
```

Algorithm 2: Walking towards a

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 **1**). 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 **2**), another approximation is added (line **0**). The algorithm reiterates after refining the approximations H_i's (line **3**). 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 [5]. Appendix A gives a sample run of the CDNF algorithm.

5 Learning Invariants

Consider the `while` statement

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

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

(a) $\delta \wedge \rho \Rightarrow \iota$ (b) $\rho \wedge \iota \Rightarrow Pre(\iota, S)$ (c) $\neg\rho \wedge \iota \Rightarrow \epsilon.$

An invariant allows us to prove that the `while` statement fulfills the annotated requirements. Observe that Definition 5 (c) is equivalent to $\iota \Rightarrow \epsilon \vee \rho$. Along with Definition 5 (a), we see that any invariant must be weaker than $\delta \wedge \rho$ but stronger than $\epsilon \vee \rho$. Hence $\delta \wedge \rho$ and $\epsilon \vee \rho$ are called the *strongest* and *weakest* approximations to invariants for $\{\delta\}$ `while` ρ `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. 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.

5.1 Predicate Abstraction to Connect Algorithmic Learning and SMT Solvers

Domains for an SMT solver and algorithmic learning are adjoined via the predicate abstraction [10]. The α, α^*, γ, and γ^* are the abstraction (α, α^*) and concretization (γ, γ^*) maps between the two domains. SMT solvers work in propositional formulae. Algorithmic learning works in Boolean formulae.

$$\begin{tikzcd}
\mathsf{Prop}_A \arrow[r, shift right, "\alpha"'] & \mathsf{Bool}_{B(A)} \arrow[l, shift right, "\gamma"'] \\
\mathit{Val}_{\mathit{Var}(A)} \arrow[r, "\alpha^*"'] & \mathit{Val}_{B(A)} \arrow[ul, "\gamma^*"]
\end{tikzcd}$$

Let A be a fixed set of atomic propositions. For each atomic proposition $p \in A$, we use a Boolean variable b_p to represent p. Let $B(A) = \{b_p : p \in A\}$ be the set of Boolean variables corresponding to the atomic propositions in A. Consider the *concrete domain* Prop_A and the *abstract domain* $\mathsf{Bool}_{B(A)}$. A Boolean formula $\beta \in \mathsf{Bool}_{B(A)}$ is called a *canonical monomial* if it is a conjunction of literals such that each Boolean variable in $B(A)$ appears exactly once. Define the mappings $\gamma : \mathsf{Bool}_{B(A)} \to \mathsf{Prop}_A$ and $\alpha : \mathsf{Prop}_A \to \mathsf{Bool}_{B(A)}$:

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

where $\overline{b}_p$ and $\overline{p}$ are the Boolean variables in $B(A)$ and their corresponding atomic propositions respectively.

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*

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

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 $\gamma(\beta) \to \theta_i$. 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 are needed. A valuation induces a natural Boolean valuation. Precisely, define the Boolean valuation $\alpha^*(\nu)$ for the valuation ν as follows.

$$(\alpha^*(\nu))(b_p) = \begin{cases} \mathtt{T} & \text{if } \nu \models p \\ \mathtt{F} & \text{otherwise} \end{cases}$$

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

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

Lemma 5.4 *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 $\alpha^*(\nu) \models \beta$.*

Proof

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

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

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

A Boolean valuation on the other hand induces a propositional formula. Define the propositional formula $\gamma^*(\mu)$ for the Boolean valuation μ as follows.

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

Lemma 5.5 *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 $\gamma^*(\mu) \to \theta$. By Lemma 5.1, $\alpha(\gamma^*(\mu)) \to \alpha(\theta)$. Note that $\gamma^*(\mu) = \gamma(\chi(\mu))$. By Lemma 5.3, $\chi(\mu) \to \alpha(\theta)$. Since $\mu \models \chi(\mu)$, we have $\mu \models \alpha(\theta)$.

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

5.2 Answering Queries from Algorithmic Learning

Suppose $\iota \in \mathsf{Prop}_A$ is an invariant for the statement $\{\delta\}$ `while` ρ `do` S $\{\epsilon\}$. Let $\underline{\iota}, \overline{\iota} \in \mathsf{Prop}_A$. We say $\underline{\iota}$ is an *under-approximation* to an invariant ι if $\delta \wedge \rho \Rightarrow \underline{\iota}$ and $\underline{\iota} \Rightarrow \iota$. Similarly, $\overline{\iota}$ is an *over-approximation* to an invariant ι if $\iota \Rightarrow \overline{\iota}$ and $\overline{\iota} \Rightarrow \epsilon \vee \rho$. The strongest and weakest approximations are trivial under- and over-approximations to any invariant respectively.

Recall that the CDNF algorithm makes the following queries: (1) Membership queries $\mathit{MEM}(\mu)$ where $\mu \in \mathit{Val}_{B(A)}$, and (2) equivalence queries $\mathit{EQ}(\beta)$ where $\beta \in \mathsf{Bool}_{B(A)}$. In the following, we show how to resolve these queries by means of an SMT solver and the invariant approximations ($\underline{\iota}$ and $\overline{\iota}$).

5.2.1 Membership Queries

In the membership query $\mathit{MEM}(\mu)$, the teacher is required to answer whether $\mu \models \alpha(\iota)$. We concretize the Boolean valuation μ and check it against the approximations. If the concretizationl $\gamma^*(\mu)$ is inconsistent (that is, $\gamma^*(\mu)$ is unsatisfiable), we simply answer *NO* for the membership query. Otherwise, there are three cases:

1. $\gamma^*(\mu) \Rightarrow \underline{\iota}$. Thus $\mu \models \alpha(\underline{\iota})$ (Lemma 5.5). And $\mu \models \alpha(\iota)$ by Lemma 5.1.
2. $\gamma^*(\mu) \not\Rightarrow \overline{\iota}$. Thus $\mu \not\models \alpha(\overline{\iota})$ (Lemma 5.5). That is, $\mu \models \neg\alpha(\overline{\iota})$. Since $\iota \rightarrow \overline{\iota}$, we have $\mu \not\models \alpha(\iota)$ by Lemma 5.1.
3. Otherwise, we cannot determine whether $\mu \models \alpha(\iota)$ by the approximations.

```
(* $\underline{\iota}$:  an under-approximation; $\overline{\iota}$:  an over-approximation *)
**Input**: a valuation $\mu$ for $B(A)$
$\theta := \gamma^*(\mu)$;
**if** $\mathit{SMT}(\theta) \rightarrow \mathit{UNSAT}$ **then return** $\mathit{NO}$;
**if** $\mathit{SMT}(\theta \wedge \neg\underline{\iota}) \rightarrow \mathit{UNSAT}$ **then return** $\mathit{YES}$;
**if** $\mathit{SMT}(\theta \wedge \neg\overline{\iota}) \rightarrow \nu$ **then return** $\mathit{NO}$;
**abort** with $\theta$;
```

Algorithm 3: Resolving Membership Queries

Algorithm 3 shows our membership query resolution algorithm. Note that when a membership query cannot be resolved by an SMT solver given invariant approximations, one can use better approximations from static analyzers. Our framework is therefore orthogonal to existing static analysis techniques.

5.2.2 Equivalence Queries

To answer the equivalence query $\mathit{EQ}(\beta)$, we concretize the Boolean formula β and check if $\gamma(\beta)$ is indeed an invariant of the `while` statement for the given pre- and post-conditions. If it is, we are done. Otherwise, we use an SMT solver to find a witness to $\alpha(\iota) \oplus \beta$. There are three cases:

1. There is a ν such that $\nu \models \neg(\underline{\iota} \Rightarrow \gamma(\beta))$. Then $\nu \models \underline{\iota} \wedge \neg\gamma(\beta)$. By Lemma 5.4 and 5.1, we have $\alpha^*(\nu) \models \alpha(\iota)$ and $\alpha^*(\nu) \models \neg\beta$. Thus, $\alpha^*(\nu) \models \alpha(\iota) \wedge \neg\beta$.
2. There is a ν such that $\nu \models \neg(\gamma(\beta) \Rightarrow \overline{\iota})$. Then $\nu \models \gamma(\beta) \wedge \neg\overline{\iota}$. By Lemma 5.4, $\alpha^*(\nu) \models \beta$. $\alpha^*(\nu) \models \neg\alpha(\iota)$ by Lemma 5.4 and 5.1. Hence $\alpha^*(\nu) \models \beta \wedge \neg\alpha(\iota)$.
3. Otherwise, we cannot find a witness to $\alpha(\iota) \oplus \beta$ by the approximations.

```
(* $\{\delta\}$ while $\rho$ do $S$ $\{\epsilon\}$ *)
(* $\underline{\iota}$: an under-approximation; $\overline{\iota}$: an over-approximation *)
Input: $\beta \in \mathsf{Bool}_{B(A)}$
$\theta := \gamma(\beta)$;
if $SMT(\underline{\iota} \wedge \neg\theta) \rightarrow UNSAT$ and $SMT(\theta \wedge \neg\overline{\iota}) \rightarrow UNSAT$ and
$SMT(\rho \wedge \theta \wedge \neg Pre(\theta, S)) \rightarrow UNSAT$ then
    return $YES$;
if $SMT(\underline{\iota} \wedge \neg\theta) \rightarrow \nu$ then return $\alpha^*(\nu)$;
if $SMT(\theta \wedge \neg\overline{\iota}) \rightarrow \nu$ then return $\alpha^*(\nu)$;
abort with $\theta$;
```

Algorithm 4: Resolving Equivalence Queries

Algorithm 4 shows our equivalence query resolution algorithm. Note that Algorithm 4 returns *YES* only if an invariant is found.

Similar to membership query resolution, one can refine approximations by static analysis when an equivalence query is not resolvable by an SMT solver given invariant approximations. For simplicity, Algorithm 4 aborts the learning algorithm with the unresolved equivalence query.

5.3 Main Loop of Our Approach

Algorithm 5 gives the top-level loop of our framework. Initially, we use the disjunction of strongest approximation and the postcondition as the under-approximation; the weakest approximation is the over-approximation. The under-approximation aims to find an invariant that establishes the postcondition. This heuristic is proved very useful in practice.

```
(* $\{\delta\}$ while $\rho$ do $S$ $\{\epsilon\}$ *)
function randomized_membership $\mu$ =
    try Algorithm 3 with input $\mu$ when abort $\rightarrow$ return YES or NO randomly;

$\underline{\iota}$ := $(\delta \wedge \rho) \vee \epsilon$; $\overline{\iota}$ := $\epsilon \vee \rho$;
repeat
    try $\iota$ := Algorithm 1 with randomized_membership and Algorithm 4
    when abort $\rightarrow$ continue
until an invariant $\iota$ is found ;
```

Algorithm 5: Main Loop

After determining the approximations, Algorithm 1 is used to find an invariant. We use Algorithms 3 and 4 to resolve queries with an SMT solver given the invariant approximations. If Algorithm 3 aborts with an unresolved membership query, a random answer is returned by *randomized_membership*. If Algorithm 4 aborts with an unresolved equivalence query, the learning algorithm is restarted.

Since algorithmic learning does not commit to any specific target, it always finds an invariant consistent with answers to previous queries. In other words, the learning algorithm will always generate an invariant if there is one consistent with our random answers. Although our random answers may not reflect the real program behavior, an invariant can still be inferred. Verifying whether a formula is an invariant is done by checking the sufficient conditions of Definition 5 in our equivalence query resolution algorithm (Algorithm 4).

case	$SIZE$	AP	MEM	EQ	coin tossing	iterations	time (sec)
ide-ide-tape	16	6	16.1	5.3	4.3	1.2	0.048
vpr	8	7	21.3	9.8	18.3	3.3	0.064
ide-wait-ireason	9	6	41.7	24.1	9.4	2.2	0.130
usb-message	18	10	31.1	10.6	7.0	1.0	0.200
parser	37	20	7157.0	876.3	1058.5	14.5	31.196

Table 1: Performance Numbers

```
{ $ret = 0 \wedge bh\_b\_count \leq bh\_b\_size$ }
while $n > 0$ do
    if $(bh\_b\_size - bh\_b\_count) < n$ then $count := bh\_b\_size - bh\_b\_count$
    else $count := n$;
    $b$ :=nondet;
    if $b$ then $ret := 1$;
    $n := n - count$; $bh\_b\_count := bh\_b\_count + count$;
    if $bh\_b\_count = bh\_b\_size$ then
      $bh\_b\_size$ := nondet; $bh\_b\_count$ := nondet; $bh\_b\_count := 0$;
end
{ $n = 0 \wedge bh\_b\_count \leq bh\_b\_size$ }
```

Figure 2: A Sample Loop in Linux IDE Driver

6 Experiments

We have implemented a prototype [1] in OCaml. In our implementation, we use Yices as the SMT solver to resolve queries (Algorithm 3 and 4). From SPEC2000 benchmarks and Linux device drivers we chose five while statements. We translated them into our language and added postcondition manually. Table 1 shows the performance numbers of our experiments. Among 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 each case, we report the number of language constructs in the loop ($SIZE$), the number of atomic propositions (AP), the number of membership queries (MEM), the number of equivalence queries (EQ), the number of randomly resolved membership queries (coin tossing), the number of the CDNF algorithm invocations (iterations), and the execution time. The data are the average of 500 runs and collected on a 2.8GHz Intel E7400 Duo Core with 3GB memory running Linux 2.6.28.

Our technique is able to find invariants for four cases within 1 second. Most interestingly, the learning algorithm is able to find an invariant for usb-message regardless of the outcomes of coin tossing. Although about 7 membership queries are resolved randomly, an invariant can always be found in that case. For the most complicated case parser, our technique is able to generate an invariant with 1059 random membership resolutions in about 31 seconds.

6.1 ide-ide-tape from Linux IDE Driver

Figure 2 is a while statement extracted from Linux IDE driver.[2] The flexibility in invariants can be witnessed in the following run. After successfully resolving 3 equivalence and 5 membership queries, the CDNF algorithm makes the following membership query unresolvable by

[1] Available at http://ropas.snu.ac.kr/vmcai10/inv-learn-released.tar.gz

[2] The source code can be found in function idetape_copy_stage_from_user() of drivers/ide/ide-tape.c in Linux 2.6.28

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

Figure 3: A Sample Loop in SPEC2000 Benchmark PARSER

the invariant approximations:

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

Answering *YES* to this query leads to the following unresolvable membership query after successfully resolving one more membership query:

$$\rho \land bh\_b\_count \neq bh\_b\_size \land bh\_b\_count \leq bh\_b\_size$$

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

$$\rho \land bh\_b\_count \neq bh\_b\_size \land bh\_b\_count > bh\_b\_size$$

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

$$n \neq 0 \lor (bh\_b\_size - bh\_b\_count) \geq n$$

Answering *NO* yields the following invariant:

$$(bh\_b\_count \leq bh\_b\_size \land n \neq 0) \lor (bh\_b\_size - bh\_b\_count) \geq n$$

Note that they are two different invariants. The equivalence query resolution algorithm (Algorithm 4) ensures that both fulfill the conditions in Definition 5.

6.2 `parser` from VPR in SPEC2000 Benchmarks

Figure 3 shows a sample `while` statement from the `parser` program in SPEC2000 benchmark.[3] In the `while` body, there are three locations where *give_up* or *success* is set to `T`. Thus one of

[3] The source code can be found in function `loop()` of `CINT2000/197.parser/main.c` in SPEC2000.

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 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 15 iterations. The execution time and number of iterations vary significantly. They range from 2.25s to 163.46s and 1 to 78 with standard deviations 29.42 and 14.0 respectively. By Chebyshev's inequality [12], our technique infers an invariant within two minutes with probability 0.891.

One of the found invariants is the following:

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

This invariant describes the conditions when *success* or *give_up* are true. For instance, it specifies that $\mathit{valid} \neq 0 \land \mathit{canonical} \neq 0 \land \mathit{valid} \leq \mathit{linkages} \land \mathit{linkages} \leq 5000 \land \mathit{canonical} = \mathit{linkages}$ should hold if *success* is true. In Figure 3, we see that *success* is assigned to `T` at line 18 with condition $\mathit{valid} > 0$. The first conjunct ($\mathit{valid} \neq 0$) is valid. Since *valid* is set to 0 at line 14, we know that the condition $\mathit{linkages} \neq 0$ (line 15) is valid. From line 16 and 17, we see that the third ($\mathit{valid} \leq \mathit{linkages}$), fourth ($\mathit{linkages} \leq 5000$), and fifth conjunct ($\mathit{canonical} = \mathit{linkages}$) are valid. Finally, we have $\mathit{linkages} \neq 0$ and $\mathit{canonical} = \mathit{linkages}$. The second conjunct ($\mathit{canonical} \neq 0$) is also valid.

7 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. For each of the 2500 ($= 5 \times 500$) runs, our technique always generates an invariant. On average, it takes 14.5 iterations for the most complicated case `parser`, and less than 5 iterations for the other cases.

Since plentiful of invariants are available, it may appear that one of them can be generated by merely coin tossing. But this is not the case. In `parser`, our technique does not terminate if the under- and over-approximations are the strongest and weakest approximations respectively. Indeed, 7157 membership and 876 equivalence queries are resolved by invariant approximations in this case. Invariant approximations are essential to our framework.

Better invariant approximations ($\underline{\iota}$ and $\overline{\iota}$) can be computed by static analysis and 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 [8, 13, 14, 15, 16, 17, 9, 18] in particular can be used together 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.

In comparison with previous invariant generation techniques [8, 13, 14, 15, 16, 17, 9, 18], we have the following distinguishing features. (1) We do not use fixed point computation nor any static or dynamic analyses. Instead, we use algorithmic learning [5] to search for loop invariants. (2) Templates for invariants are not needed. Our approach does not restrict to specific forms of invariants imposed by templates. (3) We employ SMT solvers instead of theorem provers in our technique. This allows us to take advantages of recent development in efficient SMT algorithms. (4) Our method can be extended and combined with the existing loop invariant techniques.

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 [14] devise three algorithms, two of them use fixed point computation and the other uses a constraint based approach [13, 15] to derive quantified invariants. Gupta and Rybalchenko [16] 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 [8]. Predicates over Skolem constants are used to handle unbounded arrays. McMillan [18] 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.

8 Conclusions

By combining algorithmic learning, decision procedures, and predicate abstraction, we introduced a technique for invariant generation. The new technique finds invariants guided by query resolution algorithms. Algorithmic learning gives a platform to integrate various techniques for invariant generation; it suffices to design new query resolution algorithms based on existing techniques. The learning algorithm will utilize the information provided by these techniques.

To illustrate the flexibility of algorithmic learning, we deploy a randomized query resolution algorithm. When a membership query cannot be resolved, a random answer is returned 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 indeed show that algorithmic learning is able to infer non-trivial invariants with this naïve membership resolution. It is important to exploit the power of coin tossing in our technique.

Acknowledgment We would like to thank Wontae Choi, Deokhwan Kim, Will Klieber, Sasa Misailovic, Bruno Oliveira, Corneliu Popeea, Hongseok Yang, and Karen Zee for their detailed comments and helpful suggestions.We also would like to thank Heejae Shin for implementing OCaml binding of Yices.

References

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

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

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

[4] 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

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

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

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

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

[9] 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

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

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

[12] Rosen, K.H.: Discrete Mathematics and Its Applications. McGraw-Hill Higher Education (2006)

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

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

[15] Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI, ACM (2008) 281–292

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

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

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

[19] 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

[20] 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

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

[22] Zee, K., Kuncak, V., Rinard, M.C.: An integrated proof language for imperative programs. In: PLDI, ACM (2009) 338–351

[23] Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: PLDI, ACM (2008) 349–361

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

[25] Podelski, A., Wies, T.: Boolean heaps. In: SAS. Volume 3672 of LNCS., Springer (2005) 268–283

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

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

[28] 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

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

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

Figure 4: Learning $b_0 \oplus b_1$

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 4). The teacher responds by giving the valuation $\mu_1(b_0) = \mu_1(b_1) = 0$ (denoted by $\mu_1(b_0b_1) = 00$). Hence Algorithm 1 assigns $\emptyset$ 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 $\mu_2(b_0b_1) = 01$. 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 $H_1 = b_1$. 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.