Probabilistic Inference for Predicate Constraint Satisfaction

Yuki Satake,[1] **Hiroshi Unno,**[1,2] **Hinata Yanagi**[1]
[1]University of Tsukuba, [2]RIKEN AIP
{satake, uhiro, hinata}@logic.cs.tsukuba.ac.jp

Abstract

In this paper, we present a novel constraint solving method for a class of predicate Constraint Satisfaction Problems (pCSP) where each constraint is represented by an *arbitrary* clause of first-order predicate logic over predicate variables. The class of pCSP properly subsumes the well-studied class of Constrained Horn Clauses (CHCs) where each constraint is restricted to a *Horn* clause. The class of CHCs has been widely applied to verification of *linear-time* safety properties of programs in different paradigms. In this paper, we show that pCSP further widens the applicability to verification of *branching-time* safety properties of programs that exhibit finitely-branching non-determinism. Solving pCSP (and CHCs) however is challenging because the search space of solutions is often very large (or unbounded), high-dimensional, and non-smooth. To address these challenges, our method naturally combines techniques studied separately in different literatures: counterexample guided inductive synthesis (CEGIS) and probabilistic inference in graphical models. We have implemented the presented method and obtained promising results on existing benchmarks as well as new ones that are beyond the scope of existing CHC solvers.

1 Introduction

Constrained Horn Clauses (CHCs) (Grebenshchikov et al. 2012; Bjørner et al. 2015) is a class of constraint satisfaction problems where each constraint is represented by a *Horn* clause of first-order predicate logic with free predicate variables over a background theory such as quantifier free linear integer arithmetic (QFLIA). Thanks to its expressiveness, the class has been widely adopted as a verification intermediate language to encode various verification problems of deciding whether the given linear-time safety properties are satisfied by the given programs in different paradigms such as functional (Unno and Kobayashi 2009), multi-threaded (Popeea and Rybalchenko 2012), imperative (Gurfinkel et al. 2015), and object-oriented programming (Kahsai et al. 2016). Accordingly, highly efficient CHC solvers like SPACER (Komuravelli, Gurfinkel, and Chaki 2014), ELDARICA (Hojjat and Rümmer 2018), and HOICE (Champion et al. 2018) have been developed.

This paper studies a generalization of CHCs called predicate Constraint Satisfaction Problems (pCSP) where each constraint is represented by an *arbitrary* (i.e., possibly non-Horn) clause. We show that this generalization further widens the applicability to verification of *branching-time* safety properties of programs that exhibit finitely branching non-determinism. In other words, this paves the way to use pCSP as a common intermediate language for verification of branching-time safety properties in place of CHCs that is limited to a strict subclass (i.e., linear-time safety properties).[1] One of the notable instances of branching-time safety verification is non-termination verification where the goal is to check whether *there is* a non-terminating execution of the given program (Gupta et al. 2008). For example, consider the following program c_{nt}:

```
while (x ≥ 0) do
  if ⋆ then x := x - 1 else x := x + 1
```

The program repeatedly and *non-deterministically* (indicated by $\star$) decrements or increments the integer variable x by 1 while the condition $x \geq 0$ is satisfied. Suppose we would like to verify that *there is* a non-terminating execution of the program under the precondition that the initial value of x is non-negative. The property actually holds because we can always take the else branch to avoid termination. The non-termination verification problem is reduced to the following pCSP $\mathcal{C}_{nt}$:

$$\left(\begin{array}{l} x \geq 0 \Rightarrow I(x), \\ (I(x) \land x \geq 0) \Rightarrow (I(x-1) \lor I(x+1)), \\ (I(x) \land x < 0) \Rightarrow \bot \end{array}\right)$$

Here, the predicate variable I represents a kind of loop invariant (called a *recurrent set* in (Gupta et al. 2008; Chen et al. 2014)) of the while loop and $\bot$ represents the contradiction. The first clause requires that I is satisfied by all the initial valuation conforming to the precondition. The

[1] The generalization however is still not sufficient for capturing programs with infinitely branching non-determinism and the full class of properties (in particular, liveness ones) expressible by branching-time temporal logics such as CTL, CTL*, and modal-mu calculus. This restriction can be easily overcome by further extending pCSP with well-foundedness constraints, though the extended class is out of the scope of this paper for simplicity of exposition.

second clause requires that I is preserved by *either* of the non-deterministic branches. Note here that the use of *non-Horn* clause is essential to capture the *existential or disjunctive* nature of the non-termination property. The third clause ensures that if I is satisfied, the execution never exits the while loop. The problem has a solution (i.e., a satisfying assignment for the free predicate variable I) $\rho_{nt}(I)(x) \triangleq x \geq 0$ and consequently the program is proved to have a diverging execution for any non-negative initial value of x.

Solving pCSP (and CHCs), however, is undecidable in general. Developing an incomplete but practical method is still challenging because the search space of solutions is often (1) very large (or unbounded), (2) high-dimensional, and (3) non-smooth. To address these challenges, there have been proposed data-driven approaches to solving *subclasses of CHCs* based on counterexample guided inductive synthesis (CEGIS) (Solar-Lezama et al. 2006) combined with template-based synthesis via SMT solvers (Sharma et al. 2013a; Garg et al. 2014), greedy set covering with logic minimization (Sharma et al. 2013b; Padhi, Sharma, and Millstein 2016), decision tree learning (Garg et al. 2016; Champion et al. 2018), randomized search (Sharma and Aiken 2016), and reinforcement learning (Si et al. 2018). Here, CEGIS is an iterative method that accumulates and exploits example instances $\mathcal{E}$ of the original constraint set $\mathcal{C}$ in order to address the challenge (1), and has been widely applied to formal verification and synthesis. Each iteration of CEGIS consists of a synthesis phase that attempts to guess candidate solutions (represented as predicate assignments $\rho_1, \ldots, \rho_m$) and a validation phase, where an SMT solver is used to check whether $\mathcal{C}$ is satisfied by some candidate ρ_i. As soon as an iteration is reached where ρ_i satisfies $\mathcal{C}$, we can conclude that $\mathcal{C}$ is satisfiable. The above existing methods vary mainly in the synthesis phase.

In this paper, we present a novel constraint solving method for the full class of pCSP (and obviously CHCs) that combines CEGIS with probabilistic inference in graphical models, which has been studied in statistical machine learning and statistical physics literatures, for guessing candidate solutions in the synthesis phase. In particular, we here adopt survey propagation (SP) (Braunstein, Mézard, and Zecchina 2005; Kroc, Sabharwal, and Selman 2007; Maneva, Mossel, and Wainwright 2007) in factor graphs (Kschischang, Frey, and Loeliger 2001), which has been shown effective for solving boolean Constraint Satisfaction Problems (bCSP) where solutions (i.e., satisfying boolean assignments) are scattered among small clusters in the search space of solutions (Kroc, Sabharwal, and Selman 2007) (i.e., SP addresses the challenges (2) and (3) residing in bCSP as well). To fill the gap between pCSP and bCSP, for each iteration of CEGIS, we apply predicate abstraction to the example instances $\mathcal{E}$ of the original constraint set $\mathcal{C}$ to obtain a boolean abstraction $\mathcal{B}$ of $\mathcal{E}$, which is a bCSP. We then enumerate solutions of $\mathcal{B}$ by SP, concretize them back to solutions $\rho_1, \ldots, \rho_m$ of $\mathcal{E}$, and return them as candidate solutions of $\mathcal{C}$. Here, thanks to the use of SP, our method tends to obtain *simpler* (per Occam's razor) and *different* solutions (belonging to different clusters) for $\mathcal{E}$, which tend to result in a faster convergence of CEGIS in our method. We have implemented the presented method and obtained promising results on existing benchmark sets from SyGuS-Comp 2017 and 2018 (Invariant Synthesis Track), and CHC-COMP 2019 (LIA-nonlin Track) as well as a new benchmark set consisting of pCSP that goes beyond the scope of existing CHC and SyGuS solvers.

The rest of the paper is organized as follows. § 2 formalizes pCSP and § 3 discusses its application to branching-time safety verification of programs with finitely branching nondeterminism. We then present our constraint solving method for pCSP in § 4 and report on the implementation and experimental evaluation in § 5. We discuss related work in § 6 and conclude the paper in § 7.

2 Predicate Constraint Satisfaction Problems

This section defines the class of Predicate Constraint Satisfaction Problems (pCSP). For simplicity, we fix the background first-order theory of our predicate logic to quantifier free linear integer arithmetic (QFLIA). We use ϕ and ψ as meta-variables ranging over formulas of the logic. We assume that ϕ does not contain predicate variables, while ψ may. We also use n as a meta-variable ranging over integer constants. We write $\mathit{ftv}(\phi)$ for the set of free term variables occurring in ϕ. A pCSP $\mathcal{C}$ is a finite set of clauses of the form

$$\phi \vee \left(\bigvee_{i=1}^{\ell} P_i(\widetilde{x}_i) \right) \vee \left(\bigvee_{i=\ell+1}^{m} \neg P_i(\widetilde{x}_i) \right).$$

Here, x and P are meta-variables ranging over term and predicate variables, respectively. We write $\widetilde{x}$ for a sequence of term variables, $|\widetilde{x}|$ for its length, and $\mathrm{ar}(P)$ for the arity of P. We write $\mathit{ftv}(c)$ (resp. $\mathit{ftv}(\mathcal{C})$) for the set of free term variables in the clause c (resp. in the pCSP $\mathcal{C}$). We regard the variables in $\mathit{ftv}(c)$ as universally quantified implicitly. We write $\mathit{fpv}(\mathcal{C})$ for the set of free predicate variables occurring in $\mathcal{C}$ and $\mathit{atms}(\mathcal{C})$ for the set $\{ P_i(\widetilde{x}_i) \mid i = 1, \ldots, m \}$ of predicate variable applications in $\mathcal{C}$. A pCSP $\mathcal{C}$ is called CHCs if $\ell \leq 1$ for all clauses $c \in \mathcal{C}$. A CHCs $\mathcal{C}$ is called linear CHCs if $m - \ell \leq 1$ for all $c \in \mathcal{C}$. A predicate assignment ρ is a finite map from predicate variables P to closed predicates of the form $\lambda x_1, \ldots, x_{\mathrm{ar}(P)}.\phi$ (i.e., a function that takes integer arguments $x_1, \ldots, x_{\mathrm{ar}(P)}$ and returns whether ϕ holds). We write $\rho(\mathcal{C})$ for the application of ρ to $\mathcal{C}$ and $\mathrm{dom}(\rho)$ for the domain of ρ. We call ρ a solution (i.e., a satisfying predicate assignment) for $\mathcal{C}$ if $\mathit{fpv}(\mathcal{C}) \subseteq \mathrm{dom}(\rho)$ and all clauses in $\rho(\mathcal{C})$ are valid (i.e., $\models \bigwedge \rho(\mathcal{C})$). For example, the predicate assignment ρ_{nt} is a solution of the pCSP $\mathcal{C}_{nt}$ in § 1. A boolean Constraint Satisfiability Problem (bCSP) $\mathcal{B}$ is defined as a pCSP that satisfies $\mathrm{ar}(P) = 0$ for all $P \in \mathit{fpv}(\mathcal{B})$. Note that a SAT solver can be used to decide whether the given bCSP has a solution.

3 Application to Verification of Branching-Time Safety Properties

In this section, we show that pCSP is expressible enough to encode verification problems of branching-time safety properties of looping programs with finitely branching non-determinism. The syntax of programs are defined as follows:

$$\begin{aligned}
\text{(programs) } c &::= x := a \mid c_1; c_2 \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\\
&\mid \texttt{while}\ b\ \texttt{do}\ c \mid \texttt{assume}\ b \mid \texttt{assert}\ b\\
&\mid \texttt{if}\ *_q\ \texttt{then}\ c_1\ \texttt{else}\ c_2\\
\text{(bool. exps.) } b &::= \top \mid \bot \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2\\
\text{(arith. exps.) } a &::= n \mid x \mid -a \mid a_1 + a_2 \mid a_1 \times a_2
\end{aligned}$$

The one-step reduction relation $\longrightarrow$ over program configurations (c, σ) is defined as usual (Winskel 1993). Informally, $\texttt{assert}\ b$ aborts if b evaluates to $\bot$ and does nothing otherwise. $\texttt{if}\ *_q\ \texttt{then}\ c_1\ \texttt{else}\ c_2$ non-deterministically branches to c_1 or c_2, where q is either $\forall$ or $\exists$ indicating demonic or angelic non-determinism, respectively. A *branching-time safety verification problem* of the given program c is that of deciding whether $\mathbf{safe}(c, \sigma)$ holds for any valuations σ of the variables in c where $\mathbf{safe}(c, \sigma)$ is co-inductively defined by: $\mathbf{safe}(c, \sigma)$ implies the following

- $c \neq \texttt{assert}\ \bot$,
- if $c = \texttt{if}\ *_\exists\ \texttt{then}\ c_1\ \texttt{else}\ c_2$, then there exists (c', σ') such that $(c, \sigma) \longrightarrow (c', \sigma')$ and $\mathbf{safe}(c', \sigma')$ holds, and
- otherwise, for all (c', σ') such that $(c, \sigma) \longrightarrow (c', \sigma')$, $\mathbf{safe}(c', \sigma')$ holds.

We next explain how to encode the given branching-time safety verification problem as a pCSP. To this end, we extend Dijkstra's predicate transformer (Winskel 1993) to our target programs as follows:

$$\begin{aligned}
\mathbf{wp}(x := a, \psi) &\triangleq [a/x]\psi\\
\mathbf{wp}(c_1; c_2, \psi) &\triangleq \mathbf{wp}(c_1, \mathbf{wp}(c_2, \psi))\\
\mathbf{wp}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \psi) &\triangleq (b \Rightarrow \mathbf{wp}(c_1, \psi)) \wedge\\
&\quad\ (\neg b \Rightarrow \mathbf{wp}(c_2, \psi))\\
\mathbf{wp}(\texttt{while}\ b\ \texttt{do}\ c, \psi) &\triangleq I(\widetilde{x}) \wedge\\
&\quad [\widetilde{x}_1/\widetilde{x}](I(\widetilde{x}) \wedge b \Rightarrow \mathbf{wp}(c, I(\widetilde{x})))\\
&\quad \wedge\ [\widetilde{x}_2/\widetilde{x}](I(\widetilde{x}) \wedge \neg b \Rightarrow \psi)\\
\mathbf{wp}(\texttt{assume}\ b, \psi) &\triangleq b \Rightarrow \psi\\
\mathbf{wp}(\texttt{assert}\ b, \psi) &\triangleq b \wedge \psi\\
\mathbf{wp}(\texttt{if}{*_\forall}\texttt{then}\ c_1\ \texttt{else}\ c_2, \psi) &\triangleq \mathbf{wp}(c_1, \psi) \wedge \mathbf{wp}(c_2, \psi)\\
\mathbf{wp}(\texttt{if}{*_\exists}\texttt{then}\ c_1\ \texttt{else}\ c_2, \psi) &\triangleq \mathbf{wp}(c_1, \psi) \vee \mathbf{wp}(c_2, \psi)
\end{aligned}$$

Here, $[a/x]\psi$ is the formula obtained from ψ by replacing the free occurrences of x with a. I represents a fresh predicate variable which is unique to the while loop, $\widetilde{x}$ represents the sequence of the program variables (i.e., $\{\widetilde{x}\} = \mathit{ftv}(b) \cup \mathit{ftv}(c)$), and $\widetilde{x}_1$ and $\widetilde{x}_2$ are fresh program variables such that $|\widetilde{x}| = |\widetilde{x}_1| = |\widetilde{x}_2|$.

Example 3.1. For the running example c_{nt} in § 1, we get[2]

$$\begin{aligned}
\mathbf{wp}(c_{nt}, \psi) = I(x) \wedge&\\
&(I(x_1) \wedge x_1 \geq 0) \Rightarrow (I(x_1 - 1) \vee I(x_1 + 1))\\
&\wedge (I(x_2) \wedge x_2 < 0) \Rightarrow [x_2/x]\psi)
\end{aligned}$$

Note that this can be regarded as a pCSP. □

The branching-time safety verification problem of the given program c is thus reduced to $\mathbf{wp}(c, \top)$. Formally, we can show the correctness of the reduction as follows.

Theorem 3.1. *$\mathbf{safe}(c, \sigma)$ holds for all σ with $\mathit{ftv}(c) \subseteq \mathrm{dom}(\sigma)$ if the pCSP $\mathbf{wp}(c, \top)$ has a solution.*[3]

3.1 Non-Termination Verification

As explained in § 1, the goal of non-termination verification is to check whether *there is* a non-terminating execution of the given program. Non-termination verification can be considered as an instance of branching-time safety verification. More specifically, given a program c, we assume that $q = \exists$ for every occurrence of $\texttt{if}\ *_q\ \texttt{then}\ c_1\ \texttt{else}\ c_2$ in c and $\texttt{assert}\ b$ does not occur in c. Then, the non-termination verification problem with the precondition ϕ can be reduced to the pCSP $\mathbf{wp}(\texttt{assume}\ \phi; c; \texttt{assert}\ \bot, \top)$. Note that for the running example c_{nt}, $\mathbf{wp}(\texttt{assume}\ (x \geq 0); c_{nt}; \texttt{assert}\ \bot, \top)$ coincides with the pCSP $\mathcal{C}_{nt}$ in § 1 (modulo α-conversion).

3.2 Extension to Recursive Functions

We can extend our encoding to support recursive functions. We informally describe the extension using the following example recursive program c_{ntrec} (written in OCaml syntax):

```
let rec loop x =
  if x >= 0 then
    if *∃ then loop(x-1) else loop(x+1)
  else x
let main x =
  assume (x >= 0); loop x; assert ⊥
```

This program behaves similarly to c_{nt} in § 1. The non-termination verification problem of the recursive program is reduced to the following pCSP $\mathcal{C}_{ntrec}$:

$$\left(\begin{array}{l}
(x < 0 \wedge x = y) \Rightarrow P(x, y),\\
(x \geq 0 \wedge P(x-1, y_1) \wedge P(x+1, y_2)) \Rightarrow\\
\qquad (P(x, y_1) \vee P(x, y_2)),\\
(x \geq 0 \wedge P(x, y)) \Rightarrow \bot
\end{array}\right)$$

Here, the predicate variable P represents a postcondition of the `loop` function relating the input x and the output y. The first and the second clauses are respectively generated from the else- and then-branches of the if-expression in the `loop` function. The third clause is generated from the `main` function. The non-deterministic behavior of the program is encoded in the second (non-Horn) clause. The problem has a solution $\rho_{ntrec}(P)(x, y) \triangleq x < 0$ and consequently the `loop` function is proved to have a diverging execution for any non-negative argument x .

4 Our Constraint Solving Method for pCSP

This section describes our CEGIS-based method for finding a solution of the given pCSP $\mathcal{C}$. Our method iteratively accumulates example instances of $\mathcal{C}$, from which we generate a sequence of candidate solutions for $\mathcal{C}$ until a genuine solution is found. We write $\mathcal{E}^{(i)}$ for the set of example instances

[2] We here assume that $*$ in c_{nt} is appropriately annotated with $\exists$ (see § 3.1 for an explanation).

[3] The converse also holds if we adopt the set-theoretic notion of solutions instead of the formula-definable ones.

accumulated before the iteration i. Starting from $\mathcal{E}^{(1)} = \emptyset$, for each iteration $i \geq 1$, we perform the following:

1. **Synthesis Phase**: Find solutions $\rho_1^{(i)}, \dots, \rho_m^{(i)}$ (with $\mathrm{dom}(\rho_1^{(i)}) = \cdots = \mathrm{dom}(\rho_m^{(i)}) = \mathit{fpv}(\mathcal{C})$) for $\mathcal{E}^{(i)}$, which will be used as candidate solutions for $\mathcal{C}$.
2. **Validation Phase**: Check whether $\rho_j^{(i)}$ is a genuine solution of $\mathcal{C}$ for some j by using an off-the-shelf SMT solver. If so, we return $\rho_j^{(i)}$ as a solution. Otherwise, for each clause $c \in \mathcal{C}$ not satisfied by some $\rho_j^{(i)}$, we obtain a counterexample, i.e., an integer assignment σ_c such that $\mathrm{dom}(\sigma_c) = \mathit{ftv}(c)$ and $\not\models \sigma_c(\rho_j^{(i)}(c))$. We then update $\mathcal{E}^{(i+1)} = \mathcal{E}^{(i)} \cup \{\, \sigma_c(c) \mid c \in \mathcal{C} \wedge \exists j.\ \not\models \rho_j^{(i)}(c) \,\}$ and check whether $\mathcal{E}^{(i+1)}$ has no solution by using a SAT solver.[4] If it is the case, we report that $\mathcal{C}$ has no solution. Otherwise, we proceed to the next iteration with $\mathcal{E}^{(i+1)}$.

In the synthesis phase, we apply predicate abstraction and survey inspired decimation for enumerating solutions of $\mathcal{E}^{(i)}$, which are respectively explained in § 4.1 and § 4.2.

The above CEGIS method is not guaranteed to terminate due to the undecidability of pCSP but satisfies the so called *progress property*: any counterexample σ_c found in an iteration i is never generated again in the succeeding iterations.

Example 4.1. Recall the running example $\mathcal{C}_{nt}$ in § 1. For the first iteration, our implementation reported in § 5 obtains a candidate solution $\rho_1^{(1)}(I)(x) \triangleq \top$ from the initial set of example instances $\mathcal{E}^{(1)} \triangleq \emptyset$ in the synthesis phase. In the validation phase, we obtain a counterexample $\{x \mapsto -1\}$ for the third clause, from which we obtain the updated example instances $\mathcal{E}^{(2)} \triangleq \{\neg I(-1)\}$. In the second iteration, we get a candidate solution $\rho_1^{(1)}(I)(x) \triangleq x \geq 0$ from $\mathcal{E}^{(2)}$, which satisfies all the clauses of $\mathcal{C}_{nt}$ in the validation phase. We thus return $\rho_1^{(1)} = \rho_{nt}$ as a solution of $\mathcal{C}_{nt}$. □

4.1 Predicate Abstraction of pCSP into bCSP

We now describe our predicate abstraction technique which uses predicates for abstracting the example instances $\mathcal{E}^{(i)}$ into a bCSP $\mathcal{B}$. For linear CHCs, predicate abstraction has been studied in (Srivastava and Gulwani 2009). We here present predicate abstraction for pCSP $\mathcal{E}$ that satisfies the condition $\mathit{ftv}(\mathcal{E}) = \emptyset$ (recall $\mathit{ftv}(\mathcal{E}^{(i)}) = \emptyset$ for any $i \geq 1$), which helps greatly reduce the computational cost.

We first prepare, for each $P \in \mathit{fpv}(\mathcal{E})$, the following set $\mathcal{Q}_P$ of predicates from the octahedron abstract domain:

$$\begin{aligned}&\{\lambda x_1, \dots, x_{\mathrm{ar}(P)}.\, \Sigma_{i=1}^{\mathrm{ar}(P)} c_i \cdot x_i \geq c_0 \mid \\ &\quad c_0 \in \{0\} \cup \{\Sigma_{i=1}^{\mathrm{ar}(P)} c_i \cdot n_i \mid P(\widetilde{n}) \in \mathit{atms}(\mathcal{E})\} \wedge \\ &\quad c_1, \dots, c_{\mathrm{ar}(P)} \in \{-1, 0, 1\}\}.\end{aligned}$$

We then abstract each clause $c \in \mathcal{E}$ of the form[5]

$$(\bigvee_{i=1}^{\ell} P_i(\widetilde{n}_i)) \vee (\bigvee_{i=\ell+1}^{m} \neg P_i(\widetilde{n}_i))$$

into the following boolean clause $\alpha(c)$:

$$(\bigvee_{i=1}^{\ell} \alpha(P_i(\widetilde{n}_i))) \vee (\bigvee_{i=\ell+1}^{m} \neg\alpha(P_i(\widetilde{n}_i)))$$

where

$$\alpha(P(\widetilde{n})) \triangleq \bigvee_{j=1}^{nd} \bigwedge_{k=1}^{|\mathcal{Q}_P|} \psi_{j,k}$$

$$\psi_{j,k} \triangleq \begin{cases} \top & (\models p_k(\widetilde{n})) \\ \neg b_{j,k} & (\not\models p_k(\widetilde{n})). \end{cases}$$

Here, p_k represents the k-th predicate of $\mathcal{Q}_P$, nd specifies the maximum number of disjuncts allowed in disjunctive normal form (DNF) solutions for P to be searched, and $b_{j,k}$ is a boolean variable indicating whether the predicate p_k is used in the j-th disjunct of the DNF solutions for P. We thus obtain the bCSP $\mathcal{B}$ as $\alpha(\mathcal{E}) \triangleq \{\, \alpha(c) \mid c \in \mathcal{E} \,\}$. We can show the following correctness theorem of predicate abstraction.

Theorem 4.1. *Let $\mathcal{B} \triangleq \alpha(\mathcal{E})$ be the bCSP abstracted from the given pCSP $\mathcal{E}$ with $\mathit{ftv}(\mathcal{E}) = \emptyset$. Let σ be a satisfying boolean assignment for $\mathcal{B}$ (i.e., $\models \sigma(\mathcal{B})$) and $\gamma(\sigma)$ be the concretization of σ defined as the predicate assignment:*

$$\gamma(\sigma)(P) \triangleq \lambda x_1, \dots, x_{\mathrm{ar}(P)}.\, \bigvee_{j=1}^{nd} \bigwedge_{k=1}^{|\mathcal{Q}_P|} \phi_{j,k}$$

$$\phi_{j,k} \triangleq \begin{cases} p_k(x_1, \dots, x_{\mathrm{ar}(P)}) & (\models \sigma(b_{j,k})) \\ \top & (\not\models \sigma(b_{j,k})). \end{cases}$$

Then, $\gamma(\sigma)$ is a solution for $\mathcal{E}$.

Thus, predicate abstraction reduces the problem of finding a solution for example instances $\mathcal{E}$ to that of finding a solution for the bCSP $\mathcal{B} \triangleq \alpha(\mathcal{E})$. In our implementation reported in § 5, we iteratively increment nd starting from 1 until we obtain a solution for $\mathcal{B}$. The convergence of the iterations follows from the facts that $\mathcal{E}$ has a solution and for any satisfying *boolean* assignment for $\mathit{atms}(\mathcal{E})$, there is a satisfying *predicate* assignment for $\mathit{fpv}(\mathcal{E})$ expressible as disjunctions of conjunctions of $\mathcal{Q}_P$ (in other words, $\mathcal{Q}_P$ is *adequate* in the sense defined in Section 4.1 of (Sharma et al. 2013b)).

Example 4.2. Recall the running example $\mathcal{C}_{nt}$ in § 1. In the second iteration of CEGIS, our implementation obtains the following set $\mathcal{Q}_I$ of predicates for abstraction of I:

$$\{\lambda x.\top, \lambda x.x \geq 0, \lambda x.x \geq -1\}\,.$$

Predicate abstraction of $\mathcal{E}^{(2)} = \{\neg I(-1)\}$ by $\mathcal{Q}_I$ yields the bCSP $\mathcal{B} = \{b_{1,2}\}$, which has a satisfying boolean assignment $\sigma \triangleq \{b_{1,2} \mapsto \top\}$, indicating that the second predicate $\lambda x.x \geq 0$ of $\mathcal{Q}_I$ must be used. We thus obtain the candidate solution $\rho_1^{(1)} = \gamma(\sigma) = \{I \mapsto \lambda x.x \geq 0\}$. □

[4] Note that $\mathit{ftv}(\mathcal{E}^{(i+1)}) = \emptyset$ holds since $\mathcal{E}^{(i+1)}$ is obtained from $\mathcal{C}$ by substituting all the term variables with concrete values. Thus, by regarding each $P(\widetilde{n}) \in \mathit{atms}(\mathcal{E}^{(i+1)})$ as a distinct boolean variable, $\mathcal{E}^{(i+1)}$ can be considered as a bCSP.

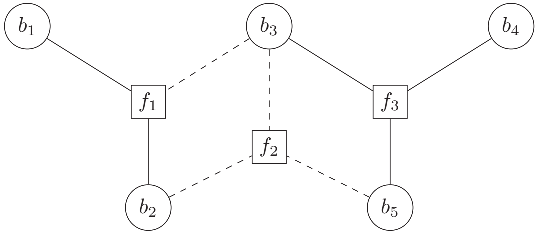
[5] We here omit ϕ because it is equivalent to either $\top$ or $\bot$.

4.2 Survey Inspired Decimation for bCSP

Our motivation to use SP is to enumerate *simpler* and essentially *different* solutions for $\mathcal{E}$ that accelerate and stabilize the convergence of CEGIS in our method. To this end, we enumerate solutions of the bCSP $\alpha(\mathcal{E})$ that (1) belong to different clusters in the space of boolean assignments and (2) assign as much variables as possible the boolean value $\perp$ (indicating that the corresponding predicate is *not* used).

We achieve the requirement (1) by survey inspired decimation (SID) (Braunstein, Mézard, and Zecchina 2005; Kroc, Sabharwal, and Selman 2007; Maneva, Mossel, and Wainwright 2007). SID iteratively assigns a boolean value to a boolean variable with the highest bias towards the value, which is computed from the result of SP as briefly reviewed later in this section, in order not to kill too many clusters. SID thus returns a partial assignment σ_0 to some boolean variables of $\alpha(\mathcal{E})$. We then use an ordinary SAT solver to enumerate solutions of $\alpha(\mathcal{E})$ belonging to different clusters by enumerating solutions of $\sigma_0(\alpha(\mathcal{E}))$. We try to satisfy the requirement (2) by heuristically detecting and assigning don't-care variables $\perp$.

Factor Graphs Any bCSP can be modeled as a factor graph which is a bipartite graph representing the factorization of a probability distribution function. For example, the following is the factor graph of the bCSP $\{b_1 \vee b_2 \vee \neg b_3, \neg b_2 \vee \neg b_3 \vee \neg b_5, b_3 \vee b_4 \vee b_5\}$:



Here, the square nodes labeled with f_1, f_2, and f_3 represent the clauses of the bCSP and the dashed (resp. solid) edges connect each variable node b_i to the square nodes that represent a clause containing $\neg b_i$ (resp. b_i). The graph represents the following probability density distribution over boolean *random* variables $b_1, \dots, b_5$.

$$\mu(b_1, \dots, b_5) = \frac{1}{Z} f_1(b_1, b_2, b_3) f_2(b_2, b_3, b_5) f_3(b_3, b_4, b_5)$$

$$f_1(b_1, b_2, b_3) = e^{-\delta(b_1 \vee b_2 \vee \neg b_3)}$$

$$f_2(b_2, b_3, b_5) = e^{-\delta(\neg b_2 \vee \neg b_3 \vee \neg b_5)}$$

$$f_3(b_3, b_4, b_5) = e^{-\delta(b_3 \vee b_4 \vee b_5)}.$$

Here, Z is the normalization constant and the function δ is defined by $\delta(\perp) = 1, \delta(\top) = 0$. Note that $\mu(b_1, \dots, b_5)$ has the maximum value (i.e., $1/Z$) if and only if the valuation of $b_1, \dots, b_5$ is a satisfying assignment for the bCSP. We can thus solve the bCSP by performing probabilistic inference in the factor graph to compute $\underset{b_1,\dots,b_5}{\arg\max}\, \mu(b_1, \dots, b_5)$.

Survey Propagation (SP) SID relies on SP to find a boolean variable with the highest bias towards a boolean value; The bias for each boolean variable b_i is computed by using SP to approximate the marginal probability $\mu(b_i) = \sum_{b_1,\dots,b_{i-1},b_{i+1},\dots,b_m} \mu(b_1, \dots, b_m)$ after extending the domain of variables b_i to the three values $\top$, $\perp$ and $*$ respectively meaning that the variable b_i is forced to be $\top$ by a clause, forced to be $\perp$ by a clause, and not forced at all (see (Braunstein, Mézard, and Zecchina 2005; Kroc, Sabharwal, and Selman 2007; Maneva, Mossel, and Wainwright 2007) for details of message passing algorithms for SP). If SP converges, we obtain $W_i^{\top} \triangleq \mu(b_i = \top)$, $W_i^{\perp} \triangleq \mu(b_i = \perp)$, and $W_i^{*} \triangleq \mu(b_i = *)$ for each variable b_i. Then the variable b_j with the highest bias $|W_j^{\top} - W_j^{\perp}|$ is assigned to $\top$ if $W_j^{\top} - W_j^{\perp} > 0$ and $\perp$ if $W_j^{\perp} - W_j^{\top} > 0$.

5 Implementation and Evaluation

We have implemented a constraint solver PCSAT for the full class of pCSP based on the presented method. We adopted MINISAT (Eén and Sörensson 2004) and Z3 (de Moura and Bjørner 2008) as the backend SAT and SMT solvers, respectively. We conducted three sets of experiments. In the first experiment (§ 5.1), we used the benchmark set from SyGuS-Comp 2018 (Invariant Synthesis Track) to evaluate the effectiveness of our SID-based technique to enumerate multiple candidate solutions for a faster convergence of CEGIS. In the second experiment (§ 5.2), we used the benchmark sets from SyGuS-Comp 2017 and 2018 (Invariant Synthesis Track)[6], and CHC-COMP 2019 (LIA-nonlin Track)[7] to compare PCSAT with existing SyGuS and CHC solvers for solving linear CHCs and (non-linear) CHCs that are proper subclasses of pCSP. Finally (§ 5.3), we tested PCSAT on a new pCSP benchmark set consisting of non-Horn clauses that go beyond the scope of existing SyGuS and CHC solvers. All the experiments were conducted on 3.1GHz Intel Xeon Platinum 8000 CPU and 32 GiB RAM.

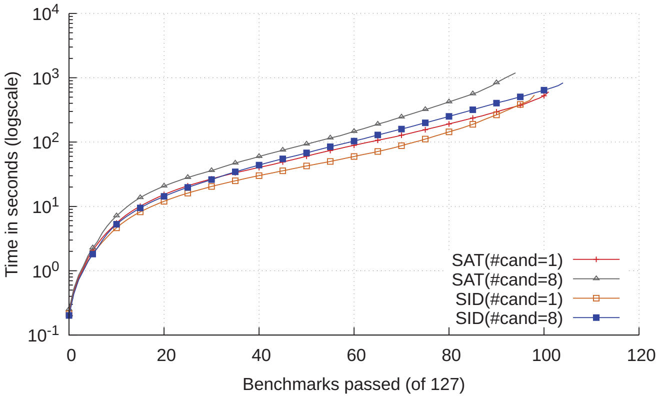
5.1 Evaluation of Multiple Solution Enumeration

We evaluated the impact of our SID-based multiple solution enumeration technique on the convergence speed of CEGIS using the benchmark set consisting of 127 linear CHCs from the InvTrack category of SyGuS-Comp 2018 with 300s timeout. Figure 1 summarizes the results of running PCSAT with different configurations obtained by varying the number "#cand" of candidate solutions generated in the synthesis phase, and enabling or disabling SID, which are respectively represented by "SAT(...)" and "SID(...)". There, "#SAT" and "#UNSAT" respectively represent the number of solved SAT and UNSAT instances (out of 127). Note that PCSAT obtained the best result with SID(#cand=8). The number of solved instances however slowly got worse with larger #cand. This is because the overhead of synthesizing and checking candidate solutions becomes non-negligible for larger #cand. We believe the overhead can be further reduced by tweaking the implementation and expect that better results can be obtained with larger #cand.

We also observed that: (1) the average number of iterations taken for the commonly solved instances by SID with

[6] http://sygus.org/comp/

[7] https://chc-comp.github.io/



	#SAT	#UNSAT
SID(#cand=1)	91	8
SID(#cand=2)	92	8
SID(#cand=4)	96	8
SID(#cand=8)	100	8
SID(#cand=16)	96	8
SID(#cand=32)	95	7
SID(#cand=64)	94	6
SAT(#cand=1)	94	8
SAT(#cand=2)	92	8
SAT(#cand=4)	91	8
SAT(#cand=8)	89	7

Figure 1: Comparisons of cumulative runtime (left) and number of solved instances (right) among different configurations

#cand=1, 2, 4, 8 were 15.980, 12.684, 11.510, and 10.663, respectively, and (2) the average elapsed time taken for the commonly solved instances by SID with #cand=1, 2, 4, 8 were 6.246, 4.257, 5.558, and 6.201 (sec.), respectively. These results show that SP accelerates and stabilizes the convergence of CEGIS, though the improvement of average time reached the ceiling at #cand=2. The rest of the experiments was conducted using the configuration SID(#cand=8).

5.2 Comparison with CHC and SyGuS Solvers

To compare with state-of-the-art SyGuS solvers for invariant synthesis, we tested PCSAT on the benchmark sets from the InvTrack category of SyGuS-Comp 2017 and 2018 with 300s timeout. The benchmark sets consist of linear CHCs. PCSAT solved 61 SAT and 7 UNSAT (out of 74), and 100 SAT and 8 UNSAT (out of 127) instances respectively from the SyGuS-Comp 2017 and 2018 benchmark sets. The champion solver LOOPINVGEN (Padhi, Sharma, and Millstein 2016) of SyGuS-Comp 2017 was reported to have solved 65 SAT instances within 3600s.[8] The top three solvers LOOPINVGEN, CVC4, and DRYADSYNTH of SyGuS-Comp 2018 were reported to have solved 115, 109, and 103 SAT instances respectively within 3600s.[9] The results show that PCSAT is comparable to the state of the art invariant generation tools for the task of solving linear CHCs, despite the facts that PCSAT is designed for the strictly larger class of constraints, not tuned for the class of linear CHCs, and less mature than these tools.

We also tested PCSAT on the CHC benchmark set from CHC-COMP 2019 (LIA-nonlin Track) with 300s timeout, and compared with state-of-the-art CHC solvers. PCSAT solved 58 SAT and 33 UNSAT instances out of 283 instances from the CHC-COMP 2019 benchmark set. According to the report of CHC-COMP 2019[10], the state-of-the-art CHC solver SPACER (Komuravelli, Gurfinkel, and Chaki 2014) has solved 153 SAT and 117 UNSAT instances, and one of the best CHC solvers HOICE (Champion et al. 2018) has solved 110 SAT and 66 UNSAT instances. The results show that we need engineering efforts of tuning PCSAT for CHC solving to compete with the mature and highly tuned CHC solvers. In particular, the current version of PCSAT cannot exploit the restricted (i.e. Horn) constraint form when applied to CHC solving. In other words, the generality of PCSAT resulted in a negative impact on the speed. In particular, the current implementation is not well-tuned for proving the unsatisfiability of CHCs, though it is rather out of the scope of this paper. We could exploit the Horn form of constraints for finding a resolution derivation of the contradiction via an efficient SLD-resolution. We could also implement constraint simplification and specialization for pre-processing, which are supported by the other solvers, to obtain better results on SAT instances.

5.3 Evaluation on the New pCSP Benchmark Set

The new benchmark set consists of pCSP benchmarks that encode branching-time safety verification problems, which cannot be handled by existing SyGuS and CHC solvers. Table 1 summarizes the experiment results on the benchmark set. The problems there are as follows:

- "nt-intro" is the pCSP $\mathcal{C}_{nt}$ generated from the non-termination verification problem of c_{nt} in § 1.
- "nt-rec" is the pCSP $\mathcal{C}_{ntrec}$ generated from the non-termination verification problem of c_{ntrec} in § 3.2.
- "nt-sum" is the pCSP

  $$\left(\begin{array}{l}(x>0\land y=0)\Rightarrow I(x,y),\\ (I(x,y)\land x>0)\Rightarrow(I(x,y)\lor I(x-1,y+x)),\\ (I(x,y)\land x\le 0)\Rightarrow\bot\end{array}\right)$$

  generated from the non-termination verification problem:

  ```
  assume (x > 0 ∧ y = 0);
  (while(x > 0) do
    if *∃ then y := y + x; x := x - 1)
  assert ⊥
  ```
- "nt-sum-mod" is the pCSP obtained from "nt-sum" by modifying the precondition to $x \geq 0 \land y = 0$, which makes the pCSP unsatisfiable.

[8] https://sygus.org/comp/2017/report.pdf

[9] https://sygus.org/comp/2018/report.pdf

[10] https://chc-comp.github.io/2019/chc-comp19.pdf

Table 1: Results on the new pCSP benchmark set

problems	results	time (sec.)	#iterations	problems	results	time (sec.)	#iterations
nt-intro	SAT	0.425	1	eq	SAT	1.332	5
nt-rec	SAT	1.557	2	eq-mod	UNSAT	1.261	5
nt-sum	SAT	0.454	1	eq-rec	SAT	0.932	2
nt-sum-mod	UNSAT	0.313	0	eq-mod-rec	UNSAT	1.142	3
nt-sum-rec	SAT	0.686	2	nt-sum-mod-rec	UNSAT	0.284	0

- "nt-sum-rec" is the pCSP

$$\left(\begin{array}{l} (x > 0 \land P(x-1, y_1) \land P(x, y_2)) \Rightarrow \\ \qquad (P(x, x+y_1) \lor P(x, y_2)), \\ (x \leq 0 \land y = 0) \Rightarrow P(x, y), \\ (x > 0 \land P(x, y)) \Rightarrow \bot \end{array}\right)$$

  generated from the recursive version of "nt-sum":

```
let rec sum x =
  if x > 0 then
    if ⋆∃ then x + sum(x-1) else sum x
  else 0
let main x =
  assume (x > 0); sum x; assert ⊥
```

- "nt-sum-mod-rec" is the pCSP generated from the recursive version of "nt-sum-mod".
- "eq" is the pCSP

$$\left(\begin{array}{l} (x = x_0 \land y = 0) \Rightarrow I(x_0, x, y), \\ (I(x_0, x, y) \land x \neq 0) \Rightarrow \\ \qquad (I(x_0, x-1, y+1) \lor I(x_0, x-1, y)), \\ (I(x_0, x, y) \land x = 0) \Rightarrow y = x_0 \end{array}\right)$$

  from the branching-time safety verification problem:

```
assume (x = x0 ∧ y = 0);
(while(x ≠ 0) do
  x := x - 1; if ⋆∃ then y := y + 1);
assert (y = x0)
```

- "eq-mod" is the pCSP obtained from "eq" by modifying the conditional expression of the while loop to $x > 0$, which makes the pCSP unsatisfiable.
- "eq-rec" is the pCSP

$$\left(\begin{array}{l} (x \neq 0 \land P(x-1, y, z_1) \land P(x-1, y+1, z_2)) \Rightarrow \\ \qquad (P(x, y, z_1) \lor P(x, y, z_2)), \\ (x = 0 \land z = y) \Rightarrow P(x, y, z), \\ P(x, 0, z) \Rightarrow z = x \end{array}\right)$$

  generated from the recursive version of "eq":

```
let rec loop x y =
  if x <> 0 then
    if ⋆∃ then loop (x - 1) y
    else loop (x - 1) (y + 1)
  else y
let main x =
  let z = loop x 0 in assert (z = x)
```

- "eq-mod-rec" is the pCSP generated from the recursive version of "eq-mod".

Note that PCSAT successfully solved these benchmarks, which are impossible by existing SyGuS and CHC solvers, in a reasonable amount of time.

6 Related Work

To our knowledge, our method is the first to apply survey propagation in factor graphs to constraint solving for verification. Except the data-driven approaches for solving subclasses of CHCs mentioned in § 1, various methods for solving CHCs have been proposed. To name a few: methods based on counterexample guided abstraction refinement and Craig interpolation (Unno and Kobayashi 2009; Hojjat and Rümmer 2018), generalized property directed reachability (Hoder and Bjørner 2012; Komuravelli, Gurfinkel, and Chaki 2014), constraint specialization (Angelis et al. 2014; Kafle, Gallagher, and Morales 2016), and inductive theorem proving (Unno, Torii, and Sakamoto 2017). For linear CHCs, a number of existing invariant synthesis techniques can be applied straightforwardly. The most related to ours among them is the method in (Gulwani and Jojic 2007), which models constraints as factor graphs and performs a local search based on Gibbs sampling to find a peak of the joint probability distribution represented by the factor graph. In contrast to the above methods, our constraint solving method supports the full class of pCSP.

An extension of CHCs called existentially quantified CHCs has been proposed in (Beyene, Popeea, and Rybalchenko 2013). We observe that any pCSP can be reduced to a problem of solving existentially quantified CHCs. Their constraint solving method relies on a dedicated technique for Skolemization of existential quantifiers, while our method supports non-Horn clauses without any additional twist.

7 Conclusion

We have presented a novel constraint solving method for the full class of pCSP based on CEGIS and probabilistic inference in graphical models, in particular, SP in factor graphs. We have experimentally confirmed that our technique to enumerate multiple candidate solutions accelerates and stabilizes the convergence of CEGIS. We expect our predicate abstraction technique from pCSP to bCSP paves the way to applying other techniques from the constraint satisfaction literature. Another interesting research direction is to directly model pCSP as factor graphs representing joint probability distributions over random predicate variables and apply other probabilistic inference such as variational inference and approximate model counting.

Acknowledgments We would like to thank anonymous referees for their useful comments. This work was partly supported by JSPS KAKENHI Grant Numbers 15H05706, 16H05856, 17H01720, 17H01723, and 19H04084.

References

Angelis, E. D.; Fioravanti, F.; Pettorossi, A.; and Proietti, M. 2014. VeriMAP: A tool for verifying programs through transformations. In *TACAS '14*, 568–574. Springer.

Beyene, T. A.; Popeea, C.; and Rybalchenko, A. 2013. Solving existentially quantified Horn clauses. In *CAV '13*, 869–882.

Bjørner, N.; Gurfinkel, A.; McMillan, K. L.; and Rybalchenko, A. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, 24–51.

Braunstein, A.; Mézard, M.; and Zecchina, R. 2005. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms* 27(2):201–226.

Champion, A.; Chiba, T.; Kobayashi, N.; and Sato, R. 2018. ICE-based refinement type discovery for higher-order functional programs. In *TACAS '18*, 365–384. Springer.

Chen, H. Y.; Cook, B.; Fuhs, C.; Nimkar, K.; and O'Hearn, P. W. 2014. Proving nontermination via safety. In *TACAS '14*, volume 8413 of *LNCS*, 156–171. Springer.

de Moura, L., and Bjørner, N. 2008. Z3: An efficient SMT solver. In *TACAS '08*, 337–340. Springer.

Eén, N., and Sörensson, N. 2004. An extensible SAT-solver. In *SAT '04*, 502–518. Springer.

Garg, P.; Löding, C.; Madhusudan, P.; and Neider, D. 2014. ICE: A robust framework for learning invariants. In *CAV '14*, 69–87. Springer.

Garg, P.; Neider, D.; Madhusudan, P.; and Roth, D. 2016. Learning invariants using decision trees and implication counterexamples. In *POPL '16*, 499–512. ACM.

Grebenshchikov, S.; Lopes, N. P.; Popeea, C.; and Rybalchenko, A. 2012. Synthesizing software verifiers from proof rules. In *PLDI '12*, 405–416. ACM.

Gulwani, S., and Jojic, N. 2007. Program verification as probabilistic inference. In *POPL '07*, 277–289. ACM.

Gupta, A.; Henzinger, T. A.; Majumdar, R.; Rybalchenko, A.; and Xu, R.-G. 2008. Proving non-termination. In *POPL '08*, 147–158. ACM.

Gurfinkel, A.; Kahsai, T.; Komuravelli, A.; and Navas, J. A. 2015. The SeaHorn verification framework. In *CAV '15*, 343–361. Springer.

Hoder, K., and Bjørner, N. 2012. Generalized property directed reachability. In *SAT '12*, 157–171. Springer.

Hojjat, H., and Rümmer, P. 2018. The Eldarica horn solver. In *FMCAD '18*. IEEE.

Kafle, B.; Gallagher, J. P.; and Morales, J. F. 2016. RAHFT: A tool for verifying horn clauses using abstract interpretation and finite tree automata. In *CAV '16*, 261–268. Springer.

Kahsai, T.; Rümmer, P.; Sanchez, H.; and Schäf, M. 2016. JayHorn: A framework for verifying Java programs. In *CAV '16*, volume 9779, 352–358. Springer.

Komuravelli, A.; Gurfinkel, A.; and Chaki, S. 2014. SMT-based model checking for recursive programs. In *CAV '14*, volume 8559 of *LNCS*, 17–34. Springer.

Kroc, L.; Sabharwal, A.; and Selman, B. 2007. Survey propagation revisited. In *UAI '07*, 217–226. AUAI Press.

Kschischang, F. R.; Frey, B. J.; and Loeliger, H. . 2001. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory* 47(2):498–519.

Maneva, E.; Mossel, E.; and Wainwright, M. J. 2007. A new look at survey propagation and its generalizations. *Journal of the ACM* 54(4).

Padhi, S.; Sharma, R.; and Millstein, T. D. 2016. Data-driven precondition inference with learned features. In *PLDI '16*, 42–56.

Popeea, C., and Rybalchenko, A. 2012. Compositional termination proofs for multi-threaded programs. In *TACAS '12*, volume 7214 of *LNCS*. Springer. 237–251.

Sharma, R., and Aiken, A. 2016. From invariant checking to invariant inference using randomized search. *Form. Methods Syst. Des.* 48(3):235–256.

Sharma, R.; Gupta, S.; Hariharan, B.; Aiken, A.; Liang, P.; and Nori, A. V. 2013a. A data driven approach for algebraic loop invariants. In *ESOP '13*, 574–592. Springer.

Sharma, R.; Gupta, S.; Hariharan, B.; Aiken, A.; and Nori, A. V. 2013b. Verification as learning geometric concepts. In *SAS '13*, 388–411. Springer.

Si, X.; Dai, H.; Raghothaman, M.; Naik, M.; and Song, L. 2018. Learning loop invariants for program verification. In *NeurIPS '18*, 7762–7773. Curran Associates, Inc.

Solar-Lezama, A.; Tancau, L.; Bodik, R.; Seshia, S.; and Saraswat, V. 2006. Combinatorial sketching for finite programs. In *ASPLOS XII*, 404–415. ACM.

Srivastava, S., and Gulwani, S. 2009. Program verification using templates over predicate abstraction. In *PLDI '09*, 223–234. ACM.

Unno, H., and Kobayashi, N. 2009. Dependent type inference with interpolants. In *PPDP '09*, 277–288. ACM.

Unno, H.; Torii, S.; and Sakamoto, H. 2017. Automating induction for solving horn clauses. In *CAV '17*, 571–591. Springer.

Winskel, G. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.