Formal Methods in System Design (2020) 56:154–177
https://doi.org/10.1007/s10703-020-00349-x

Learning inductive invariants by sampling from frequency distributions

Grigory Fedyukovich[1] · Samuel J. Kaufman[2] · Rastislav Bodík[2]

Published online: 16 November 2020

Abstract
Automated verification for program safety is reduced to the discovery safe inductive invariants, i.e., formulas that over-approximate the sets of reachable program states, but precise enough to prove unreachability of the error state. We present a framework, called FreqHorn, that follows the Syntax-Guided Synthesis paradigm to iteratively sample candidate invariants from a formal grammar and check them with an SMT solver. FreqHorn automatically constructs grammars based on either source code or bounded proofs. After each (un-)successful candidate, FreqHorn adjusts the grammars to ensure the candidate is not sampled again. The process continues either until the conjunction of successful candidates (called lemmas) is sufficient, or the search space is exhausted. Additionally, FreqHorn keeps a history of counterexamples-to-induction (CTI) which block learning a lemma. With some periodicity, it checks if there is a CTI which is invalidated by the currently learned lemmas and rechecks the failed lemma if needed. FreqHorn is able to check several candidates at the same time to filter them effectively using the well known Houdini algorithm.

Keywords Software verification · Invariant synthesis · Syntax-guided synthesis

1 Introduction

State-of-the-art techniques for automated program verification certify their outcomes with formal proofs. When verifying a program for safety, such a proof is an inductive invariant [1,3,7,14,25–27,29–31,33,37,40,42–46,48,55] which intuitively describes that no error state is ever reachable. Automated discovery of inductive invariants necessitates exploring a large search space and relies on decision procedures and solvers for Satisfiability Modulo

✉ Grigory Fedyukovich
grigory@cs.fsu.edu

Samuel J. Kaufman
kaufmans@cs.washington.edu

Rastislav Bodík
bodik@cs.washington.edu

[1] Florida State University, Tallahassee, USA

[2] University of Washington, Seattle, USA

Theories (SMT). Existing synthesizers are based on Counterexample-Guided Abstraction Refinement (CEGAR) [10], Counterexample-Guided Inductive Synthesis (CEGIS) [53], Property Directed Reachability (PDR) [6,15], Machine Learning [52], but currently, there is no clear witness that any of these approaches consistently outperforms the others.

This article gives an overview of the approach to synthesize inductive invariants, called FreqHorn, which is based on Syntax-Guided Synthesis (SyGuS) [2]. FreqHorn is an enumeration-based approach that discovers *conjunctive* invariants. Each conjunct considered during the verification process belongs to some formal grammar and can be checked individually for the invariance using an SMT solver (we call it a *lemma*, if it is checked successfully).

Our key insight is that a formal grammar need not necessarily be provided by the user, but instead it could be automatically constructed from the symbolic encoding of the program and interpolation-based proofs of bounded safety of the program. That is, both the syntax and semantics of the program is used to automatically guess and check formal proofs. This method has been successfully applied to proving safety of numerical programs [17,21,22], programs with arrays [23], termination and non-termination [24] of programs, and is compatible with approaches to incremental verification [19,20] and hyperproperties verification [47,50].

The main feature of FreqHorn is its ability to generate *probabilistic production rules* reflecting the statistics of various features of the source code. Such information as occurrences of variables, constants, arithmetic and comparison operators, and their applications is useful for creating candidate invariants. We further extend this approach by on-the-fly adjusting the probabilities based on the successes and failures observed during the verification process. That is, we avoid re-generating candidates that were already considered, and we prioritize the generation of candidates that have not been considered yet. We show that this strategy can be made aggressive, i.e., the completeness of the search-space exploration is traded for speed of the entire process.

We also demonstrate how FreqHorn can be enhanced with elements of PDR, thus improving its scalability. In particular, an ability to check groups of candidates at once (we call them *batches*) decreases the sensitivity of FreqHorn to the order of checks of different candidates. Furthermore, a history of counterexamples-to-induction (CTI), which blocked FreqHorn from learning a lemma, allows re-checking some failed lemmas in the next iterations.

In addition to the revised description of algorithms originally published in [17,21], this article provides a new evaluation of different configurations of FreqHorn and the comparison with the state-of-the-art *learning*-based and PDR-based tools to the inductive invariant synthesis. We considered a wider range of benchmarks and confirmed that our framework is competitive to state-of-the-art.

The article is organized as follows. Section 2 briefly discusses the fundamentals of our verification problem, and Sect. 3 introduces some commonly used methods that address it. Section 4 introduces the framework to learn numerical invariants, Sect. 5 describes its optimizations and drawbacks, and Sect. 6 shows how it can be integrated with well renowned formal methods. Section 7 shows the experimental evaluation, and Sect. 8 outlines several applications of FreqHorn. Finally, the future work, related work, and conclusion complete the paper in Sects. 9–11.

```
int n = 0;
int k = nondet();
assume (k > n);
int i = k;
int j = k;
int b = nondet();

while (nondet()) {
  if (b == 0) {
    i = i + 1;
    b = 1;
  } else {
    j = j + 1;
    b = 0;
  }

  k = k + 1;

  if (i > k ∨ j > k) {
    n = n - 1;
  } else {
    n = n + i + j;
  }
}

assert(n ≥ 0);
```

$$
\left\{
\begin{array}{r}
n = 0 \wedge \\
k > n \wedge \\
i = k \wedge \\
j = k \implies \boldsymbol{Inv}(k, i, j, n, b) \\
\\
\boldsymbol{Inv}(k, i, j, n, b) \wedge \\
i' = ite(b = 0, i + 1, i) \wedge \\
j' = ite(b = 0, j, j + 1) \wedge \\
b' = ite(b = 0, 1, 0) \wedge \\
k' = k + 1 \wedge \\
n' = ite(i' > k' \vee j' > k', \\
n - 1, n + i' + j') \implies \boldsymbol{Inv}(k', i', j', n', b') \\
\\
\boldsymbol{Inv}(k, i, j, n, b) \wedge \\
\neg(n \geq 0) \implies \bot
\end{array}
\right.
$$

Fig. 1 A loop with a safety assertion (left) and its safety verification task (right)

2 Background

In this work, we formulate the safety verification task by using the Satisfiability Modulo Theory (SMT) problem. A first-order theory $\mathcal{T}$ consists of a signature Σ, which gathers variables, function and predicate symbols, and a set *Expr* of Σ-formulas. Formula $\varphi \in Expr$ is called $\mathcal{T}$-satisfiable if there exists an interpretation m of each element (i.e., a variable, a function or a predicate symbol), which simplifies φ to true (denoted $m \models \varphi$); otherwise φ is called $\mathcal{T}$-unsatisfiable (denoted $\varphi \implies \bot$). For an implication between $\varphi, \psi \in Expr$, we write $\varphi \implies \psi$; φ is said to be stronger than ψ, and ψ weaker than φ. The SMT problem for a given theory $\mathcal{T}$ and a formula φ aims at determining whether φ is $\mathcal{T}$-satisfiable.

Definition 1 A program P is a tuple $\langle Var, Init, Tr\rangle$, where $Var \overset{\text{def}}{=} V \cup V'$ is a set of input and output variables; $Init \in Expr$ encodes the initial states over V; and $Tr \in Expr$ encodes the transition relation over Var.

A state is a valuation to all variables in V. The value of every input variable $x \in V$ in the next state is captured by the corresponding output variable $x' \in V'$. The *safety verification task* is a pair $\langle P, Bad\rangle$, where $P = \langle V \cup V', Init, Tr\rangle$ is a program, and *Bad* is a $\mathcal{T}$-encoding of the *error states*. A verification task has a solution if the set of error states is unreachable. A solution to the verification task is represented by a *safe inductive invariant*, a formula that covers every initial state, is closed under the transition relation, and does not cover any of the error states.

Definition 2 Let $P = \langle V \cup V', Init, Tr\rangle$; a formula *Inv* is a *safe inductive invariant* if the following conditions (respectively called an initiation, a consecution, and a safety) hold:

$$Init(V) \implies Inv(V) \tag{1}$$

$$Inv(V) \wedge Tr(V, V') \implies Inv(V') \tag{2}$$

$$Inv(V) \wedge Bad(V) \implies \bot \tag{3}$$

To simplify reading, in the rest of the paper *safe inductive invariants* are referred to as just *invariants*. We assume that an invariant *Inv* has the form of conjunction, i.e., $Inv = \ell_0 \wedge \ldots \wedge \ell_n$, and each ℓ_i is called a *lemma*.

Example 1 The loop in program in Fig. 1 iterates an unknown number of times. Although having multiple statements, the loop body is encoded into a single transition relation (on the right of the figure). In each iteration, it reassigns variables `i`, `j`, `k`, `n`, and/or `b`. We wish to prove that after the loop terminates, `n` is non-negative. An invariant for the program is the conjunction $k \geq i \wedge k \geq j \wedge i \geq 0 \wedge j \geq 0 \wedge n \geq 0$. Intuitively, `k` grows faster than `i` or `j`, thus always falsifying the disjunctive guard before the reassignment of `n`. Finally, since neither `i` nor `j` goes negative, `n` always stays above zero. □

The validity of each implication (1) and (2) is equivalent to the unsatisfiability of the negation of the corresponding formula. Suppose, a formula *Inv* makes (1) valid, but does not make (2) valid. Thus, there exists an interpretation *m* satisfying $Inv(V) \wedge Tr(V, V') \wedge \neg Inv(V')$, to which we refer to as a *counterexample-to-induction* (CTI).

Lemma 1 *Given program P, if* (1) *and* (2) *hold for* Inv_1 *and* Inv_2*, then* (1) *and* (2) *hold for* $Inv_1 \wedge Inv_2$ *too.*

Note that Lemma 1 does not work in the reverse direction: if a conjunction of formulas is an invariant then for each lemma in isolation (2) may not hold. Lemma 1 enables us to discover invariants incrementally. For example, for the program in Fig. 1, lemmas $k \geq i \wedge k \geq j \wedge i \geq 0 \wedge j \geq 0$ can be found first and thus will enable discovering the remaining lemma $n \geq 0$.

3 Classic approaches

This section outlines two techniques used in symbolic model checking to discover invariants.

3.1 Interpolation-based proofs of bounded safety

Given a program $\langle V \cup V', Init, Tr \rangle$, a set of error states *Bad*, and a non-negative integer number k, the Bounded Model Checking (BMC) task [4] is to (dis-)prove the absence of bugs reachable by k steps from the initial state. The idea is to conjoin k copies of *Tr* with *Init* and *Bad* and to check the satisfiability of the resulting formula (called a BMC formula):

$$Init(V) \wedge \underbrace{Tr(V, V') \wedge Tr(V', V'') \wedge \ldots \wedge Tr(V^{(k-1)}, V^{(k)})}_{k} \wedge Bad(V^{(k)})$$

Here, each $V^{(i)}$ is a fresh copy of V. Each satisfying assignment to the BMC formula represents a counterexample of length k. Otherwise, if the formula is unsatisfiable, then no counterexample of length k exists.

Lemma 2 *If a BMC formula for program P and some k is satisfiable then no invariant exists.*

A proof of *bounded safety* is an over-approximation I of the set of initial states, such that any path of length k, that starts in a state satisfying I, does not end in a state satisfying *Bad*. The extraction of proofs is typically done with the help of Craig interpolation [11].

Algorithm 1: BMCITP: Obtaining bounded proofs, cf. [44,46].

```
Input: $\langle P, Bad \rangle$: verification task, where $P = \langle V \cup V', Init, Tr \rangle$, $k$: bound
Output: $proof \subseteq 2^{Expr}$
1 $unr \leftarrow \top$;
2 for ($i \leftarrow k$; $i > 0$; $i \leftarrow i - 1$) do
3     $unr \leftarrow unr \wedge Tr(V^{(i-1)}, V^{(i)})$;
4     if $unr \wedge Bad(V^{(k)}) \implies \bot$ then
5         $proof \leftarrow$ GETITP($unr, Bad(V^{(k)})$);
6         return;
7 $unr \leftarrow unr \wedge Bad(V^{(k)})$;
8 if $Init(V^{(0)}) \wedge unr \implies \bot$ then $proof \leftarrow$ GETITP($Init(V^{(0)}), unr$);
```

Definition 3 Given two formulas A and B, such that $A \wedge B \implies \bot$, a Craig interpolant I is a formula satisfying three conditions: 1) $A \implies I$, 2) $I \wedge B \implies \bot$, and 3) I is expressed over the common alphabet to A and B.

For an invocation of a procedure of generating an interpolant I for A and B and splitting it to a set of conjunction-free clauses (i.e., $I = \ell_0 \wedge \cdots \wedge \ell_n$), we write $\{\ell_i\} \leftarrow$ GETITP(A, B). Algorithm 1 shows an algorithm to generate interpolation-based proofs of bounded safety for BMC formulas. It iteratively unrolls the transition relation and applies the interpolation to the entire BMC formula. In addition, in spirit of Lazy Annotation [46], while decrementing i, the algorithm checks if an error state is unreachable by $(k - i)$ steps from any state (line 4). It triggers the interpolation to be applied to smaller formulas, and in some cases fastens the proof search (line 5).

Example 2 Let the loop in program in Fig. 1 be unrolled twice, then its BMC formula is constructed as follows:

$$
\begin{aligned}
&Init \; \Big\{ k > 0 \wedge i = k \wedge j = k \wedge n = 0 \wedge \\
&Tr \; \begin{cases} i' = ite(b = 0, i + 1, i) \wedge j' = ite(b = 0, j, j + 1) \wedge b' = ite(b = 0, 1, 0) \wedge \\ \quad k' = k + 1 \wedge n' = ite(i' > k' \vee j' > k', n - 1, n + i' + j') \wedge \end{cases} \\
&Tr' \; \begin{cases} i'' = ite(b' = 0, i' + 1, i') \wedge j'' = ite(b' = 0, j', j' + 1) \wedge b'' = ite(b' = 0, 1, 0) \wedge \\ \quad k'' = k' + 1 \wedge n'' = ite(i'' > k'' \vee j'' > k'', n' - 1, n' + i'' + j'') \wedge \end{cases} \\
&Bad \; \Big\{ \neg(n'' \geq 0)
\end{aligned}
$$

The formula is unsatisfiable, and GETITP($Init, Tr \wedge Tr' \wedge Bad$) returns $proof = \{k \geq i, k \geq j, n + 2 * i + 2 * j \geq 0\}$. Note that the quality of interpolants largely depends on the underlying algorithm. While some approaches, e.g. [5], provide more concise proofs containing e.g., formulas $i \geq 0$ and $j \geq 0$, but in general there is no way to guarantee that the desired proofs will be produced. □

3.2 Inductive subset extraction

Following Definition 2, any given formula over V can be checked for the invariance. It is often the case, however, that stronger formulas have fewer chances to pass the consecution check. If such *candidate* formulas are conjunctive, we could weaken them by

Algorithm 2: HOUDINI: Calculating an inductive subset, cf. [25] and keeping counterexamples-to-induction.

```
Input: $P = \langle V \cup V', Init, Tr\rangle$: program; $Cands \subseteq 2^{Expr}$;
       $CTI \subseteq 2^{V \to \mathbb{Z}}$; $CTImap$: $(V \to \mathbb{Z}) \to 2^{Expr}$
Output: inductive $Cands \subseteq 2^{Expr}$; updated $CTI$ and $CTImap$
1 while $\exists \pi$, s.t. $\pi \models \big( \bigwedge_{cand \in Cands} cand(V) \wedge Tr(V, V') \wedge \bigvee_{cand \in Cands} \neg(cand(V'))\big)$ do
2     for $cand \in Cands$ do
3         if $\pi \models \neg cand(V')$ then
4             $Cands \leftarrow Cands \setminus \{cand\}$;
5             $CTI \leftarrow CTI \cup \{\pi\big|_V\}$;
6             $CTImap(\pi\big|_V) \leftarrow CTImap(\pi\big|_V) \cup \{cand\}$;
```

removing some conjuncts, thus increasing the chance of finding a formula that passes the consecution check. Such a formula is called an *inductive subset*, and it can be found by a sequence of SMT checks driven by CTIs. Algorithm 2 shows a simple implementation of this algorithm, which is known as HOUDINI [25]. Note that HOUDINI is only meaningful for the candidate formulas which are already implied by the initial states.

Note that we extended Algorithm 2 with a routine to extract a CTI π for each element dropped from *Cands* (lines 3–6). We restrict each π to only assignments to variables from V (denoted $\pi\big|_V$) and group all non-inductive formulas from *Cands* by the particular π that *killed* them. This routine is important for optimizing the syntax-guided invariant synthesis algorithm, and it is discussed in more detail in Sect. 6.

Example 3 Consider a set of candidate formulas $\{i \geq k, i \leq k, j \geq k, j \leq k, n \geq 0\}$ and the verification condition in Fig. 1. Algorithm 2 iterates three times before giving the inductive subset $\{i \leq k, j \leq k\}$: formula $i \geq k$ is killed by CTI $\{b \mapsto 1, i \mapsto 0, k \mapsto 0\}$, $j \geq k$ is killed by $\{b \mapsto 0, i \mapsto 0, k \mapsto 0\}$, and $n \geq 0$ is killed by $\{b \mapsto 1, i \mapsto -1, j \mapsto 0, n \mapsto 0\}$. □

4 Syntax-guided invariant synthesis

The Syntax-Guided Synthesis (SyGuS) [2] approach to inductive invariant synthesis generates formulas, restricted by a formal grammar, and searches for a candidate that meets all three conditions from Definition 2. Interestingly, a formal grammar need not necessarily be provided by the user, but instead it could be automatically constructed on the fly from the symbolic encoding of the program. Indeed, the source code often provides useful information, e.g., of occurrences of variables, constants, arithmetic and comparison operators, that could give rise to a range of potentially meaningful candidate formulas.

$$\begin{aligned} c &::= c_1 \mid \ldots \mid c_\ell \mid 0 \\ k &::= k_1 \mid \ldots \mid k_m \mid 1 \mid -1 \\ x &::= x_1 \mid \ldots \mid x_n \mid \varphi(x_1, \cdots, x_n, k_1, \cdots, k_m) \\ \mathit{lincom} &::= k \cdot x \mid k \cdot x + \ldots + k \cdot x \\ \mathit{ineq} &::= \mathit{lincom} > c \mid \mathit{lincom} \geq c \\ \mathit{cand} &::= \mathit{ineq} \mid \mathit{ineq} \vee \mathit{ineq} \mid \ldots \end{aligned}$$

Fig. 2 Parametrizable sampling grammar

4.1 Automatic grammar generation

For the theories of linear (LIA) and nonlinear integer arithmetic (NIA), candidate lemmas could be generated from an instantiation of the grammar shown in Fig. 2 (also referred to as the *sampling grammar*). The grammar is parametrizable by two sets of numeric constants, $C \overset{\text{def}}{=} \{c_1, \ldots, c_\ell\} \cup \{0\}$ and $K \overset{\text{def}}{=} \{k_1, \ldots, k_m\} \cup \{1, -1\}$, by a set of variables $V = \{x_1, \ldots, x_n\}$, and by the number of possible disjuncts – all obtained from the program's source code. Elements of C are used in the right sides of the comparison operators, and elements of K are used in all other places of formulas (mainly, as coefficients in linear combinations). For convenience, any application of a nonlinear operator φ (such as `div` or `mod` that may appear in the code) is treated as a fresh variable.

Note that there is no production rule with comparison operators other than $>$ and $\geq$. The expressiveness of formulas is achieved by providing large enough sets of K and C for numeric coefficients and constants: if an element is in a set, its additive inverse is also in the set. Thus, the grammar does not contain formula $k_1 \cdot x_1 + \cdots + k_n \cdot x_n < c$, but it contains an equivalent formula $(-k_1) \cdot x_1 + \cdots + (-k_n) \cdot x_n > -c$. Also there is no production rule with a conjunction operator since we aim to discover conjunction-free lemmas and then to conjoin them externally.

For the grammar generation, we first convert the *Init*, *Tr*, and *Bad* formulas to the Conjunctive Normal Form. Then, we populate a multiset *Seeds* by expressions obtained from the parse trees of *Init*, *Tr*, and *Bad*. That is, if the same expression appeared multiple times across the verification condition, it appears in *Seeds* multiple times as well. We need this feature to calculate the frequencies in Sect. 4.2.

To create a grammar from *Seeds*, we drop all expressions that contain variables from both, V and V', and deprime all variables in the remaining expressions. Then, we normalize elements of *Seeds* to have the form of equalities, inequalities, or disjunctions of equalities and inequalities. Finally, formulas are rewritten, such that all terms are moved to the left side, and the $<$, $\leq$, $=$, and, $\neq$ are rewritten as follows:

$$\frac{A < B}{-A > -B} \qquad \frac{A \leq B}{-A \geq -B} \qquad \frac{A = B}{A \geq B \wedge -A \geq -B} \qquad \frac{A \neq B}{A > B \vee -A > -B}$$

Note that in case $a = (A = B)$ the resulting formula is conjunction $a_+ \wedge a_-$, and thus a is replaced by a_+ and a_-, i.e., $\mathit{Seeds} \leftarrow \mathit{Seeds} \backslash \{a\} \cup \{a_+, a_-\}$.

The grammar creation is finalized by the following: (1) the production rule for normalized inequalities consists of choices corresponding to distinct types of inequalities in *Seeds*, (2) the production rule for linear combinations consists of choices corresponding to distinct arities of inequalities in *Seeds*, (3) production rules for variables, coefficients, and constants consist

$$\begin{array}{l} n = 0 \\ k > n \\ i = k \\ j = k \\ b = 0 \\ n \geq 0 \end{array}$$

(a)

$$\begin{array}{l} 1 \cdot n \geq 0 \\ (-1) \cdot n \geq 0 \\ 1 \cdot k + (-1) \cdot n > 0 \\ 1 \cdot n + (-1) \cdot k > 0 \\ 1 \cdot i + (-1) \cdot k \geq 0 \\ 1 \cdot k + (-1) \cdot i \geq 0 \\ 1 \cdot j + (-1) \cdot k \geq 0 \\ 1 \cdot k + (-1) \cdot j \geq 0 \\ 1 \cdot b \geq 0 \\ (-1) \cdot b \geq 0 \end{array}$$

(b)

$$\begin{array}{rl} \alpha & ::= 1 \mid -1 \\ \beta & ::= 0 \\ \gamma & ::= i \mid j \mid k \mid n \mid b \\ \delta & ::= \alpha \cdot \gamma \mid \alpha \cdot \gamma + \alpha \cdot \gamma \\ \mathit{cand} & ::= \delta > \beta \mid \delta \geq \beta \end{array}$$

(c)

Fig. 3 **(a)** Formulas extracted from the encoding shown in Fig. 1; **(b)** normalized formulas; **(c)** the grammar that generalizes the normalized formulas

of choices corresponding respectively to distinct variables, coefficients, and constants that occur in inequalities in *Seeds*.

Definition 4 Each formula contained in set *Seeds*, which is used for constructing grammar G, is called a *seed*. Formula *cand* produced by G is called a *mutant* if $\mathit{cand} \notin \mathit{Seeds}$.

Example 4 For the program in Fig. 1, the verification condition is syntactically split into the *Seeds* set of expressions over V (Fig. 3a), and normalized (Fig. 3b). The grammar containing the normalized expressions from *Seeds* is shown in Fig. 3c. It is easy to see that all lemmas consisting in both invariants mentioned in Example 1 can be generated by applying the grammar's production rules recursively. □

4.2 Probabilistic production rules

The grammar, constructed automatically in the previous subsection, is iteratively used to produce candidate formulas but it does not specify an order in which these formulas should be checked for the invariance. This subsection describes a method for assigning probabilities to different production rules, thus giving priorities for sampling certain candidates. Our idea is based on exploiting the frequencies of certain syntactic features (e.g., frequencies of particular constants or combinations of variables) that belong to the program's symbolic encoding.

Figure 4 shows a schematic representation of the construction of the hierarchy of sets of formulas needed to obtain the frequency distributions.[1] At the top level, we have a multiset A that is populated by formulas of the form *ineq*, obtained through splitting each formula in *Seeds* to disjuncts. That is:

⋆ $A = \{a \mid \exists s \in \mathit{Seeds},\ \text{s.t.}, a \text{ is a disjunct of } s\}$.[2]

Let *max* be the maximal number of disjuncts among formulas in *Seeds*. Then, we split A into *max* disjoint multisets, such that for each $i \in [1, \mathit{max}]$, multiset A_i consists only of formulas originated from some $s \in \mathit{Seeds}$ where s has arity i. More formally:

[1] It is not the only possible way of creating frequencies (e.g., the one described in [21] is slightly different), but in our evaluation it has been shown effective.

[2] Here and later, we assume that the number of occurrences of a in A equals the sum of numbers of occurrences of all s in *Seeds* that have a as a disjunct.

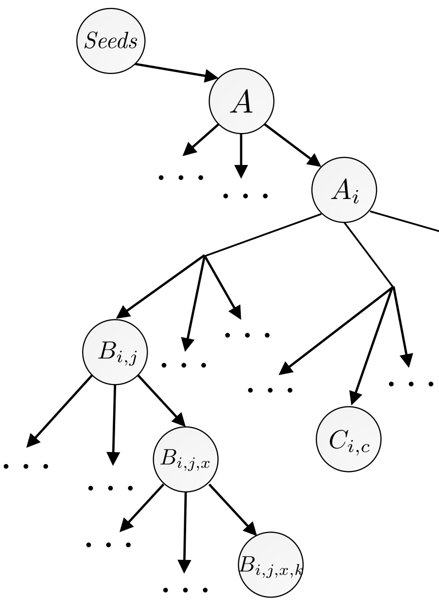


Fig. 4 Hierarchic calculation of frequencies

- ⋆ $A_i = \{a \mid \exists s \in Seeds,$ s.t., s has arity $i,$ and a is a disjunct of $s\}$.

Each non-empty multiset A_i gives rise to three groups of multisets: $B_{i,j}$—based on different linear combinations of the form *lincom*, C_i—based on different constants, and $D_{i,>} \cup D_{i,\geq}$—based on different comparison operators. In particular:

- ⋆ $B_{i,j} = \{b \mid \exists c$ s.t. $(b > c) \in A_i$ or $(b \geq c) \in A_i,$ and b has arity $j\}$;
- ⋆ $C_i = \{c \mid \exists b$ s.t. $(b > c) \in A_i$ or $(b \geq c) \in A_i\}$;
- ⋆ $D_{i,>} = \{c \mid \exists b$ s.t. $(b > c) \in A_i\}$; and
- ⋆ $D_{i,\geq} = \{c \mid \exists b$ s.t. $(b \geq c) \in A_i\}$.

Finally, for each non-empty multiset $B_{i,j}$, we identify its subsets with expressions containing each variable x and each coefficient k:

- ⋆ $B_{i,j,x} = \{b \mid b \in B_{i,j},$ and variable x occurs in $b\}$,
- ⋆ $B_{i,j,x,k} = \{b \mid b \in B_{i,j,x},$ and k is the coefficient for $x\}$.

It remains to calculate the cardinalities of multisets A, $\{A_i\}$, $\{B_{i,j}\}$, $\{B_{i,j,x}\}$, $\{B_{i,j,x,k}\}$, $\{C_i\}$, $\{D_{i,>}\}$, and $\{D_{i,\geq}\}$; and to exploit them to construct frequency distributions as follows[3]:

- ⋆ $p_i \,:\, \{A_i\} \to \mathbb{R}$,
- ⋆ for each i: $p_{i,j} \,:\, \{B_{i,j}\} \to \mathbb{R}$,
- ⋆ for each i, j: $p_{i,j,x} \,:\, \{B_{i,j,x}\} \to \mathbb{R}$,
- ⋆ for each i, j, x: $p_{i,j,x,k} \,:\, \{B_{i,j,x,k}\} \to \mathbb{R}$,
- ⋆ for each i: $p_{i,c} \,:\, \{C_i\} \to \mathbb{R}$, and
- ⋆ for each i: $p_{i,op} \,:\, \{D_{i,>}, D_{i,\geq}\} \to \mathbb{R}$.

[3] All these distributions (except p_i) were referred to as *conditional* in [21]. We do not use this terminology in this presentation.

Algorithm 3: GUESS: Sampling candidates. cf. [21].

```
Input: $G$: grammar, $\{p_\star\}$: frequency distributions
Output: $cand : Expr$
1  $cand \leftarrow \bot$;
2  $n \leftarrow$ SAMPLE$(p_i)$;
3  for $(i \in [1, n])$ do
4      $m \leftarrow$ SAMPLE$(p_{i,j})$;
5      for $(j \in [1, m])$ do
6          $\vec{x}_j \leftarrow$ SAMPLE$(p_{i,j,x})$;
7          $\vec{k}_j \leftarrow$ SAMPLE$(p_{i,j,x,k})$;
8      $c_i \leftarrow$ SAMPLE$(p_{i,c})$;
9      $op_i \leftarrow$ SAMPLE$(p_{i,op})$;
10     $cand \leftarrow cand \vee$ ASSEMBLEINEQ$(\vec{x}_i, \vec{k}_i, c_i, op_i)$;
```

Algorithm 4: FREQHORN: Learning inductive invariants, cf. [21].

```
Input: $\langle P, Bad\rangle$: verification task, where $P = \langle V \cup V', Init, Tr\rangle$
Output: $LearnedLemmas \subseteq 2^{Expr}$, such that their conjunction is an invariant
1  $Seeds \leftarrow$ GETSUBEXPRS$(Init, Tr, Bad)$;
2  $\langle G, \{p_\star\}\rangle \leftarrow$ GETGRAMMARANDDISTRIBUTIONS$(Seeds)$;
3  $LearnedLemmas \leftarrow \varnothing$;
4  while $Bad(V) \wedge \bigwedge_{\ell \in LearnedLemmas} \ell(V) \not\Longrightarrow \bot$ do
5      $cand \leftarrow$ GUESS$(G, \{p_\star\})$;
6      $res \leftarrow$ ISSAT$\big(Init(V) \implies cand(V)\big) \wedge$
           $\big(cand(V) \wedge \bigwedge_{\ell \in LearnedLemmas} \ell(V) \wedge Tr(V, V') \implies cand(V')\big)$;
7      if $res$ then
8          $LearnedLemmas \leftarrow LearnedLemmas \cup \{cand\}$;
9      $\{p_\star\} \leftarrow$ ADJUST$(\{p_\star\}, cand, res)$;
```

For instance, to determine a frequency distribution $p : \{X_1, \ldots, X_n\} \to \mathbb{R}$, we calculate the cardinality of each X_i (say, it equals S_i) and define $p(S_i) = \frac{S_i}{S_1 + \cdots + S_n}$. These distributions guide the sampling process in the synthesis algorithm (to be shown in Sect. 4.3).

4.3 Core algorithm

Algorithm 3 gives a procedure for an iterative "top-to-bottom" generation of candidate formulas from formal grammars. Starting with the probability distribution $p_i : \{A_i\} \to \mathbb{R}$, the algorithm identifies one of the multisets A_n with the probability $p_i(A_n)$, where $n \in [1, max]$. This gives us a number n of slots for operands of $\vee$ (linear inequalities). Then, for each $i \in [1, n]$, the algorithm samples a non-empty subset $\vec{x} \subseteq V$ of variables from given probability distributions $p_{i,j}$ [4] (for the length of $\vec{x}$) and $p_{i,j,x}$ (for each particular element of $\vec{x}$). Then, given probability distributions $p_{i,j,x,k}$ for scalar coefficients for each

[4] We use vector notation to denote sequences (e.g., of variables or constants).

variable $x_j \in \vec{x}$, we sample a coefficient $k_j \in K$. Summing products of each k_j with x_j, we get a linear combination (denoted $\{x_j, k_j\}$). Then, for each inequality, we sample a binary comparison operator (either $>$ or $\geq$) and a constant $c \in C$ from given probability distributions $p_{i,op}$ and $p_{i,c}$, respectively. Lastly, an inequality is assembled from all these pieces (we use a shortcut ASSEMBLEINEQ$(\cdot)$ in pseudocode) and is disjoined with other inequalities.

Each conjunction-free sample *cand* needs to be individually checked for the invariance. Successful samples should be used while checking the invariance of the samples in the future, and unsuccessful samples should be withdrawn. Algorithm 4, called FREQHORN, shows this flow more formally. It relies on an off-the-shelf SMT solver to check formulas (1), (2), and (3) for the satisfiability.[5] A negative result—i.e., whenever the initiation or the consecution check fails—is called an *invariance failure*. Otherwise, the result is positive and is called a *learned lemma*. Each new learned lemma is book-kept (line 8) for the safety check (line 4), and also for the consecution checks (line 6) of candidates coming in the next iterations. The sampler and the checker alternate until a safe inductive invariant is found.

The following theorem expresses the completeness of the search-space exploration. Note, however, that due to the undecidability of the invariant synthesis problem, this is a claim only on a relative completeness.

Theorem 1 *If a safe inductive invariant can be expressed by a conjunction of some seeds and/or mutants, and the* ADJUST *method in Algorithm 4 is instantiated to return a set of probability distributions identical to the input (i.e.,* $\{p_\star\}$*), then the probability that Algorithm 4 eventually discovers it tends to 1.*

The proof of the theorem is straightforward. The only way to skip a candidate from being sampled and checked is to zero its probability in the grammar. Initially, no zero probabilities are introduced, thus any seed or mutant can be sampled and checked. With the progress of the algorithm, the probability of getting a candidate only increases (recall that ADJUST is assumed to not zero it).

Clearly, achieving this theoretical guarantee in practice is often infeasible because of bounded time. Thus, the completeness of the search-space exploration needs to be traded for the speed. The next section discusses how the algorithm nudges the probability distributions after each positive or negative result (line 9). It relies on a non-trivial implementation of method ADJUST, that allows zeroing some probabilities.

5 Prioritizing the search space

The enumeration-based techniques crucially depend on how effectively the search space is pruned after examining a positive or negative candidate. To avoid repeatedly appearing learned lemmas and invariance failures in the future, we keep track of all samples considered before and refer to them whenever a new candidate is sampled. Furthermore, we employ a lightweight syntactic analysis to identify some syntactically related candidates and do not even sample them. Finally, we prioritize some *likely unrelated* candidates for the sampling and checking.

Given a candidate formula $cand = ineq_1 \vee \ldots \vee ineq_n$ consisting of a number of disjuncts, we assume the ith disjunct is over a linear combination $\{\vec{x}_i, \vec{k}_i\}$, a comparison

[5] The algorithm could also be extended (as in [21]) to check if *cand* is a tautology. If so, more expensive checks (1), (2), and (3) should be skipped.

operator op_i, and a constant c_i. We refer to the set of these linear combinations as to $\langle\vec{x}, \vec{k}\rangle \overset{\text{def}}{=} \{\{\vec{x}_i, \vec{k}_i\}\}$.

Definition 5 (Priority distributions) The $priorMap_{\langle\vec{x},\vec{k}\rangle}$ function maps ith linear combination $\{\vec{x}_i, \vec{k}_i\}$ to a joint probability distribution for op_i and c_i.

One goal of $priorMap_{\langle\vec{x},\vec{k}\rangle}$ is to zero the probabilities of sampling candidates which are (1) already checked, (2) stronger than failures (if they can be identified cheaply[6]), and (3) weaker than learned lemmas (if again they can be identified cheaply). Consequently, the probabilities of the remaining candidates should be increased, and we achieve it by exploiting the ordering of constants and comparison operators in the sampling grammar.

Algorithm 4 updates distributions $\{p_\star\}$ by calling the ADJUST($\{p_\star\}$, *cand*, *res*) method (line 9) where *res* is either *true* or *false* depending on the result of checks (1) and (2). At this step, we assume that for $\langle\vec{x}, \vec{k}\rangle$, which corresponds to *cand*, there exists a function $priorMap_{\langle\vec{x},\vec{k}\rangle}$[7]. The act of prioritizing is then defined as follows.

Definition 6 Given $\langle\vec{x}, \vec{k}\rangle$ for some $cand = ineq_1 \vee \ldots \vee ineq_n$, where each $ineq_i$ is over $\{\vec{x}_i, \vec{k}_i\}$, op_i, and c_i, we write:

- PRIORLEARNED($\langle\vec{x}, \vec{k}\rangle$)—to produce a joint probability distribution for each $i \in [1, n]$ to sample op'_i and c'_i and to produce $ineq'_i$ = ASSEMBLEINEQ($\vec{x}_i, \vec{k}_i, c'_i, op'_i$), such that if $ineq_i \implies ineq'_i$ then the probability of sampling op'_i and c'_i is set to zero, and otherwise it *increases* with the growth of c'_i;
- PRIORFAIL($\langle\vec{x}, \vec{k}\rangle$)—to produce a joint probability distribution for each $i \in [1, n]$ to sample op'_i and c'_i and to produce $ineq'_i$ = ASSEMBLEINEQ($\vec{x}_i, \vec{k}_i, c'_i, op'_i$), such that if $ineq'_i \implies ineq_i$ then the probability of sampling op'_i and c'_i is set to zero, and otherwise it *decreases* with the growth of c'_i (symmetrically to PRIORLEARNED($\langle\vec{x}, \vec{k}\rangle$)).

Example 5 Let $C = \{-5, 0, 5\}$, and $ineq_1 \vee ineq_2 = x > -5 \vee x + y \geq 5$ be a learned lemma for some program P. Then, any disjunction $ineq'_1 \vee ineq'_2$ where $ineq'_1 \in \{x \geq -5, x > -5\}$, and $ineq'_2 \in \{x + y \geq -5, x + y > -5, x + y \geq 0, x + y > 0, x + y \geq 5\}$ is weaker or equal to $ineq_1 \vee ineq_2$, and checking its invariance would not affect our verification process. The PRIORLEARNED($\langle\vec{x}, \vec{k}\rangle$) function outputs two probability distributions p_x and p_{x+y} to sample a new comparison operator and a new constant for x and $x + y$, respectively:

[6] We wish to avoid (potentially expensive) SMT checks to test implications.

[7] Whenever $\langle\vec{x}, \vec{k}\rangle$ is sampled for the first time, we initialize each $priorMap_{\langle\vec{x},\vec{k}\rangle}(i)$ as a *uniform* joint distribution.

$x > 5 \mapsto {}^4\!/_{10}$	$x + y > 5 \mapsto 1$
$x \geq 5 \mapsto {}^3\!/_{10}$	$x + y \geq 5 \mapsto 0$
$x > 0 \mapsto {}^2\!/_{10}$	$x + y > 0 \mapsto 0$
$x \geq 0 \mapsto {}^1\!/_{10}$	$x + y \geq 0 \mapsto 0$
$x > -5 \mapsto 0$	$x + y > -5 \mapsto 0$
$x \geq -5 \mapsto 0$	$x + y \geq -5 \mapsto 0$

□

Distributions $priorMap_{\langle\vec{x},\vec{k}\rangle}$ might be too aggressive, i.e., they might zero the probabilities of candidates that are strictly stronger (resp. weaker) that the learned lemmas (resp. invariance failures). Thus some critical lemmas might be missed, and the algorithm might diverge. However, the intention of our synthesis procedure is to encourage exploring a wide range of candidates. The increase probabilities for constants that are "further" from the already explored ones diversifies the search and forces the algorithm to try to make progress.

Example 6 Let $C = \{-5, 0, 5\}$, and $ineq_3 \vee ineq_4 = x \geq 5 \vee x + y > 5$ be an invariance failure for P. The PRIORFAIL($\langle\vec{x}, \vec{k}\rangle$) function outputs two probability distributions p'_x and p'_{x+y}:

$x > 5 \mapsto 0$	$x + y > 5 \mapsto 0$
$x \geq 5 \mapsto 0$	$x + y \geq 5 \mapsto {}^1\!/_{15}$
$x > 0 \mapsto {}^1\!/_{10}$	$x + y > 0 \mapsto {}^2\!/_{15}$
$x \geq 0 \mapsto {}^2\!/_{10}$	$x + y \geq 0 \mapsto {}^3\!/_{15}$
$x > -5 \mapsto {}^3\!/_{10}$	$x + y > -5 \mapsto {}^4\!/_{15}$
$x \geq -5 \mapsto {}^4\!/_{10}$	$x + y \geq -5 \mapsto {}^5\!/_{15}$

□

Definition 7 Given two probability distributions p, p' on set A, a probability distribution $p_m(p, p')$ on A is defined as follows. Let $s \overset{\text{def}}{=} \sum_{a \in A} min(p(a), p'(a))$, then $\forall a \in A.\ p_m(p, p')(a) \overset{\text{def}}{=} \frac{min(p(a), p'(a))}{s}$.

Method ADJUST($\{p_\star\}$, *cand*, *res*) (line 9 of Algorithm 4) produces distributions p_i, such that:

$$\{p_i\} = \begin{cases} \text{PRIORLEARNED}(\langle\vec{x}, \vec{k}\rangle), \text{ if } res = true \\ \text{PRIORFAIL}(\langle\vec{x}, \vec{k}\rangle), \text{ otherwise} \end{cases}$$

Then, for each i, it replaces the distribution $priorMap_{\langle\vec{x},\vec{k}\rangle}(i)$ by a distribution $p_m(priorMap_{\langle\vec{x},\vec{k}\rangle}(i), p_i)$. These new distributions are ultimately used to generate candidates in the next iterations. Intuitively, this schema resembles a widely used binary search, where the next element to be checked is not fixed, but identified with respect to the probability distribution.

Example 7 Given p_x and p_{x+y}, p'_x and p'_{x+y} obtained in Examples 5–6 respectively, two distributions $p_m(p_x, p'_x)$ and $p_m(p_{x+y}, p'_{x+y})$ are as follows. Note that the latter is *undefined* since all formulas are mapped to 0, thus violating the definition of probability distribution.

$$\begin{aligned} x > 5 &\mapsto 0 & x + y > 5 &\mapsto \textit{undefined} \\ x \geq 5 &\mapsto 0 & x + y \geq 5 &\mapsto \textit{undefined} \\ x > 0 &\mapsto {}^1\!/_2 & x + y > 0 &\mapsto \textit{undefined} \\ x \geq 0 &\mapsto {}^1\!/_2 & x + y \geq 0 &\mapsto \textit{undefined} \\ x > -5 &\mapsto 0 & x + y > -5 &\mapsto \textit{undefined} \\ x \geq -5 &\mapsto 0 & x + y \geq -5 &\mapsto \textit{undefined} \end{aligned}$$

This way, since one of the two distributions at $\mathit{priorMap}_{\langle \vec{x}, \vec{k} \rangle}$ is *undefined*, the entire $\langle \vec{x}, \vec{k} \rangle$ is withdrawn by the algorithm and is not going to be considered in the next iterations. □

6 Accelerating syntax-guided invariant synthesis

To gain efficiency, the enumeration-based invariant synthesis algorithm from the previous section can be generalized to consider multiple candidates at a time and to recheck failed candidates after new lemmas are learned. In this section, we present its extension that exploits the inductive-subset extraction algorithm to identify good and bad candidates while checking *batches* of candidates. Additionally, it stores the history of counter-examples-to-induction (CTI), the analysis of which lets the algorithm give some failed candidates a *second chance*. Finally, the algorithm creates the grammar not only from the syntax of the problem but also from the set of interpolation-based bounded proofs.

6.1 Overview

The pseudocode of the accelerated invariant-synthesis algorithm is shown in Algorithm 5. It takes as input a verification task, and it is parametrized by the following values

- the length of unrolling N used for interpolation,
- the size of a batch of candidates M, and
- the periodicity of CTI-checks K.

Similarly to Algorithm 4, Algorithm 5 obtains a set of expressions *Seeds* from *Init*, *Tr*, and *Bad* (line 1). Then, *Seeds* gets additional formulas obtained by Craig interpolation from bounded unrollings of the transition relation (lines 2–6). This enlarges the search space for the further mutants. After their creation in Algorithm 1, these proofs are normalized to match the grammar in Fig. 2.[8] If there is a counterexample of length $k < N$ discoverable by the BMC engine then an invariant does not exist (recall Lemma 2), and Algorithm 5 terminates (line 5).

[8] However in the examples below we present them in a simplified way.

Algorithm 5: FreqHorn-2: Sampling inductive invariants with Houdini, bmcItp, and the second-chance candidates.

Input: $\langle P, Bad\rangle$: verification task, where $P = \langle V \cup V', Init, Tr\rangle$; N, M, K: knobs
Output: $LearnedLemmas \subseteq 2^{Expr}$, such that their conjunction is an invariant or $\bot$ if no invariant exists

```
1  $Seeds \leftarrow$ getSubExprs$(Init, Tr, Bad)$;
2  for $(k \leftarrow 0;\ k < N;\ k \leftarrow k+1)$ do
3      $proof \leftarrow$ bmcItp$(\langle P, Bad\rangle, k)$;
4      if $proof = \varnothing$ then
5          return $\bot$;
6      else
7          $Seeds \leftarrow Seeds \cup proof$;
8  $\langle G, \{p_\star\}\rangle \leftarrow$ getGrammarAndDistributions$(Seeds)$;
9  $CTI, CTImap, LearnedLemmas \leftarrow \varnothing$;
10 $\#learned \leftarrow 0$;
11 $Cands \leftarrow Seeds$;
12 while $Bad(V) \wedge \bigwedge_{\ell \in LearnedLemmas} \ell(V) \not\Longrightarrow \bot$ do
13     while $|Cands| < M$ do
14         $Cands \leftarrow Cands \cup \{$guess$(G, \{p_\star\})\}$;
15     for $cand \in Cands$ do
16         if $Init(V) \not\Longrightarrow cand(V)$ then
17             $\{p_\star\} \leftarrow$ adjust$(\{p_\star\}, cand, false)$;
18             $Cands \leftarrow Cands \setminus \{cand\}$;
19     $\langle LearnedLemmas', CTI, CTImap\rangle \leftarrow$
        Houdini$(P, Cands \cup LearnedLemmas, CTI, CTImap)$;
20     for $\ell' \in LearnedLemmas' \setminus LearnedLemmas$ do
21         $\{p_\star\} \leftarrow$ adjust$(\{p_\star\}, \ell', true)$;
22     $NewLemmas = \{\ell' \mid \ell' \in LearnedLemmas', s.t. \bigwedge_{\ell \in LearnedLemmas} \ell \not\Longrightarrow \ell'\}$;
23     $\#learned \leftarrow \#learned + |NewLemmas|$;
24     $LearnedLemmas = LearnedLemmas \cup NewLemmas$;
25     $Cands \leftarrow \varnothing$;
26     if $\#learned > K$ then
27         $\#learned \leftarrow 0$;
28         for $m \in CTI$ do
29             if $m \not\models \bigwedge_{\ell \in LearnedLemmas} \ell(V)$ then
30                 $CTI \leftarrow CTI \setminus \{m\}$;
31                 $Cands \leftarrow Cands \cup CTImap(m)$;
```

The main algorithmic advantage against Algorithm 4 lies in checking the candidate formulas in batches. Each batch *Cands* is created either directly from *Seeds* (in the first checking round, line 11) or from additional mutants (lines 13–14). The algorithm checks the initiation condition for all formulas in a batch individually, and the set is filtered accordingly (lines 15–

18). Then, for the remaining formulas, the algorithm extracts an inductive subset (line 19) using Algorithm 2, which finally becomes a set of new lemmas (line 22), i.e., ones that are not implied by the lemmas learned previously.

If a batch of candidates still misses some lemmas necessary for an invariant, then Algorithm 5 proceeds to a new iteration. It enlarges the set of learned lemmas with the new ones (line 24), and the generation of a new batch of candidates starts from scratch (line 25). Importantly, the new batch can have candidates that did not pass the inductiveness check if the corresponding CTI has been invalidated (lines 26–31). The algorithm periodically checks whether some CTI m is inconsistent with the set of learned lemmas (line 29) and gives all candidates killed by m a second chance.

Example 8 Recall the program in Fig. 1. After the HOUDINI run in Example 3, the candidate formula $n \geq 0$ is killed by CTI $\{b \mapsto 1, i \mapsto -1, j \mapsto 0, n \mapsto 0\}$. However, after a lemma $i \geq 0$ is learned, the CTI is invalidated (because $i \geq 0$ is inconsistent with $i = -1$), and formula $n \geq 0$ is given the second chance. □

Theorem 2 *If Algorithm 5 terminates, then either an actual bug is found (line 5), or an invariant is synthesized (after the while-loop).*

Finally, Algorithm 5 exploits a new *learning strategy* (to be discussed in the next subsection) that reduces the dependency of the algorithm on heuristics for prioritizing the search-space traversal.

6.2 Learning strategy

A more elaborated process of sampling and checking candidates in Algorithm 5, compare to Algorithm 4, enables a more careful treatment of the search space with respect to the learned lemmas and failures. In both algorithms, the grammar adjustments are performed by changing the probabilities assigned to the production rules. In addition to zeroing the probability of sampling a candidate *cand* itself, after a positive check, algorithms zero the probabilities of sampling some formulas which are weaker than *cand*, and after a negative check—the probabilities of sampling some formulas which are stronger than *cand* (recall Sect. 5).

In Algorithm 4, the search space gets adjusted after each individual check. Given that the whole process boils down to a one-by-one enumeration, each candidate can be checked at most once. But since the progress is sensitive to the order of checks of different candidates, in the worst-case scenario, some critically-important lemmas (and thus, the whole invariant) cannot be found (recall the discussion in Sect. 5).

In contrast, Algorithm 5 takes into account only the failed candidates after the initiation check (line 17) and the successful candidates after the consecution check (line 21). Otherwise, if the consecution check failed for a candidate *cand* (inside Algorithm 2), Algorithm 5 does not ban *cand* from being checked again in the future, and instead keeps *cand* (and its CTI) temporarily and seeks an opportunity to give it a second chance. Thus, in this case, the probabilities assigned to the production rules of the grammar remain the same, and there is a non-zero probability to sample and check a candidate that is stronger than *cand*.

7 Evaluation

Implementations of the algorithms of FreqHorn and FreqHorn-2 are available in a self-titled tool[9] that takes as input a verification task in a form of Constrained Horn Clauses (CHC), automatically performs its unrolling, searches for counterexamples, generates proofs of bounded safety, and performs the Houdini-style extraction of inductive subsets. All the symbolic reasoning is performed by the Z3 SMT solver [12].

Benchmarks We evaluated FreqHorn and FreqHorn-2 on various safe and buggy programs taken from SVCOMP[10], literature (e.g., [14,27]), and crafted by ourselves. All the programs were encoded using the theories of linear (LIA) and nonlinear integer arithmetic (NIA). We did run FreqHorn-2 on unsafe instances for the testing purposes only. It was able to detect a counterexample, but since no invariant exists in these cases, we do not discuss this experience here.

Experimental setup We used m4.xlarge instances on Amazon Web Services, which have Intel Xeon E5-2676 v3 processors (“base clock rate of 2.4 GHz and can go as high as 3.0 GHz with Intel Turbo Boost”) and 16GiB of RAM. Due to the stochastic nature of our learning, the FreqHorn and FreqHorn-2 timings are the means of 3 independent runs. We used a timeout of 1 min for all runs.

7.1 Comparing configurations of FreqHorn

In total, we considered **114** safe programs. For **68** of them, the seeds, generated by breaking the symbolic encoding to pieces, did already contain all lemmas needed for invariants. For our BMC implementation, we considered bounds 1, 2, and 3. For **74** benchmarks, the interpolants and the seeds together did already contain all lemmas needed for invariants. [11]

Within a timeout of 60 s, the basic FreqHorn configuration (i.e., without prioritizing the search space) was able to solve 86 instances. The average run took 3.52 s[12]. In contrast, the optimized FreqHorn’s configuration (i.e., with prioritizing the search space) solved 96 instances and took 2.68 s in average. Figure 5a shows a scatterplot comparing these runs.[13] This lets us conclude that the strategy of prioritizing the search space results in an improvement of the overall synthesis procedure.

Figure 5b shows a scatterplot comparing the runs of the basic FreqHorn-2 configuration (i.e., without interpolation, but with Houdini and counterexamples-to-induction) and the configuration with interpolation. The basic configuration solved 99 instances (0.94 s in average), while its counterpart—100 instances (1.02 s in average). Interestingly, there were several benchmarks for which the interpolation greatly improved the runtime, and there were several other benchmarks for which the interpolation exceeded timeout, but the basic configuration succeeded quickly. With this in mind, we cannot yet conclude that interpolation results in a meaningful improvement for FreqHorn-2.

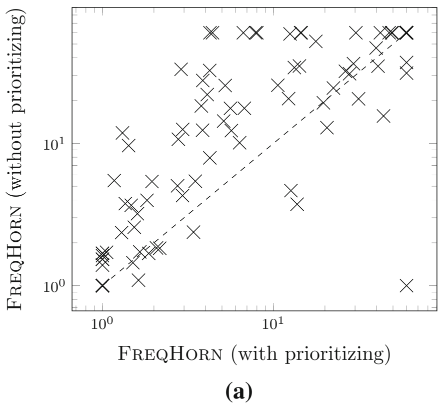
[9] The source code and benchmarks can be found at https://github.com/grigoryfedyukovich/aeval/tree/rnd.

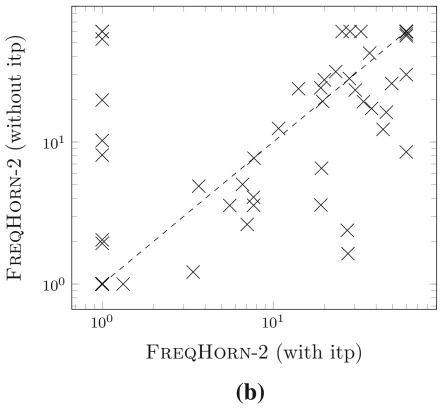
[10] Software Verification Competition, http://sv-comp.sosy-lab.org/, `loop-*` categories.

[11] Currently interpolation in FreqHorn-2 is limited to LIA, so we had to skip interpolation for 17 benchmarks over NIA.

[12] Here and later, average runtimes are calculated as a geometric mean among all successful runs (i.e., timeouts are not taken into account).

[13] Here and later, each point in a plot represents a pair of runs for the same benchmark.





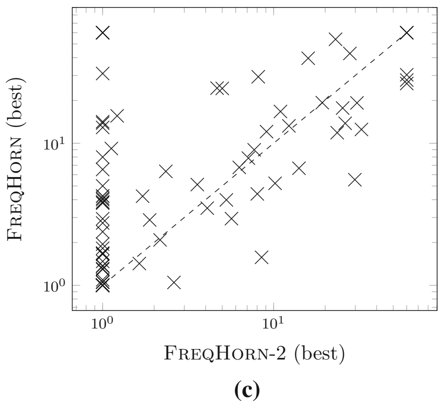


Fig. 5 FreqHorn versus FreqHorn-2 and configurations

Finally, we compare the best running times (among all related configurations) of FreqHorn and FreqHorn-2 in Fig. 5c: in average the former is 2.5 times slower that the latter, and it solved fewer instances.

7.2 Comparing FreqHorn with state-of-the-art

We compared the best running times (among all configurations) of FreqHorn-2 (as the winner in the "local competition") against the μZ [32,41], Spacer [41], ICE- DT [27], and MCMC [52] invariant synthesizers (versions of 2017). Figure 6 shows four scatterplots for these experiments.

The PDR-based tools (μZ and Spacer) are built on top of Z3, and they are in general highly optimized. On most of our benchmarks, they either terminated immediately, or exceeded the timeout. For the successful runs, μZ was in average 7.5 times faster than FreqHorn-2, and Spacer was 10 times faster. However, they solved fewer instances than FreqHorn-2: 68 by μZ, 74 by Spacer (compared to 100 by FreqHorn-2). The biggest bottleneck for the PDR-based tools is currently benchmarks over NIA: μZ and Spacer rely on quantifier elimination which is supported only partially in Z3.

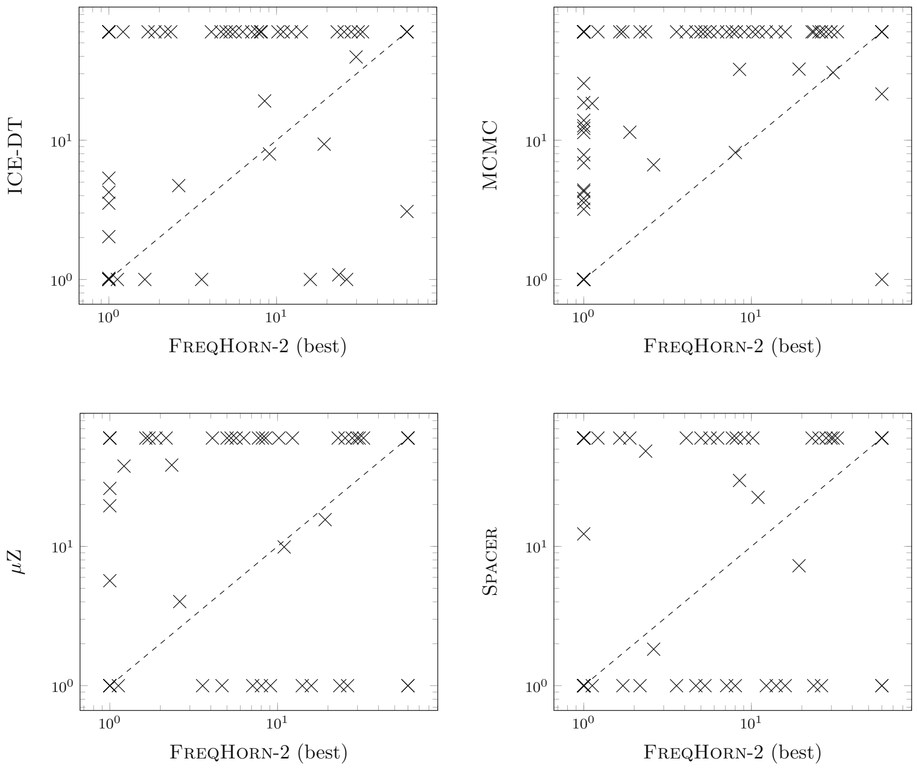


Fig. 6 FREQHORN-2 versus external tools

Compared to the data-driven tools, MCMC and ICE- DT, our FREQHORN-2 exhibited more consistent runtime and solved significantly larger number of instances. In particular, MCMC solved 39 benchmarks only and took in average 1.53 s, and ICE- DT solved 35 benchmarks and took 0.52 s in average.

Note that FREQHORN-2 still has some performance anomalies, which we believe are connected to the limited use of constants, inability to generate large disjunctions, and possible inefficiencies of the black-box interpolation engine. We are going to address these issues in our future work.

8 Applications

In this section, we outline several applications of FREQHORN that demonstrate the success of SyGuS in a range of verification tools.

Constrained horn clauses The task of program verification is an instance of a more general problem of determining the satisfiability of Constrained Horn Clauses (CHC) (e.g., [9,13, 28,32,33,38,41,57]). A CHC is a logical implication over a set of unknown predicates, and a solution to a system of CHCs is an interpretation for all predicates that makes all implications valid. Thus, the task considered in this article is a CHC instance with one predicate.

```
int x = 0;
while (x < 256) {
  x = x + 2;
}
assert (x == 256);
```

Fig. 7 `count_by_2`: a bottleneck

In [22], we presented an algorithm for solving CHC tasks of arbitrary structure: each unknown predicate of the CHC system gets its own formal grammar that encodes the search space for a solution. The general principle of enumerative invariant discovery can be applied for arbitrary (even, non-linear) CHCs, but it does not scale in practice. Instead, we developed an algorithm to generate a candidate for one unknown predicate at a time and to propagate it to candidates of the remaining unknowns via quantifier elimination [19]. This algorithm still relies on the automated grammar construction algorithm and frequency distributions.

Quantified invariants We extended the approach to handle CHCs with arrays [23] and LIA. The invariants are universally quantified formulas describing ranges of array elements, repeatedly accessed in loops. In particular, FreqHorn guesses candidates of form $\forall q . q \in \mathit{Range} \implies \mathit{ArrGuess}(q, \mathit{vars})$, where q is a fresh quantified variable, $q \notin \mathit{vars}$, a *Range* is a loop invariant over the loop counter, and *ArrGuess* is a formula obtained from the grammar as in Sect. 4 where the loop counter is replaced by q. With the support for quantified reasoning in SMT solvers, FreqHorn exhibits a competitive performance to state-of-the-art.

Proving termination We developed the FreqTerm [24] algorithm that adds a decrementing counter to a loop, iteratively guesses lower bounds on its initial value, which lead to the safety verification tasks to be solved by FreqHorn. Existence of an inductive invariant guarantees termination, and the algorithm converges. Otherwise FreqTerm proceeds to strengthening the lower bounds by adding another guess. FreqTerm has also been extended with the support for generation of lexicographic termination arguments.

Proving non-termination The FreqTerm algorithm can iteratively search for a restriction on the loop guard, that *might lead* to infinite traces. It exploits a solver for the validity of ∀∃-formulas [19]. Importantly, FreqTerm relies on the generator of constraints adapted from Sect. 4 that is applied to sample termination and, independently, non-termination constraints.

9 Future work

Consider a simple program in Fig. 7: it increments a counter, initially assigned 0, by 2. When its value is no longer less than 256, we wish to prove that it actually equals 256. An invariant $x = 0 \vee x = 2 \vee \ldots \vee x = 256$ could be discovered by μZ in about 40 s (and this time grows nonlinearly when 256 gets replaced by larger constants: e.g., for 512 μZ agonizes for 11 min). That said, there exists a much more compact invariant $x \mod 2 = 0 \wedge x \leq 256$, and it could be easily discovered by a human and not by a machine.

Unfortunately, FreqHorn is currently able to generate neither of them: its sampling grammar does not have neither the modulo operator nor the whole range of even numbers. In the future work, it would be useful to have a set of heuristics to automatically enhance the sampling grammar by (preferably minimal and property-driven) ingredients. With this additional information, we believe, SyGuS-based algorithms would be able to generate small invariants and would not be crucially dependent on particular constants in the source code.

Alternatively one could transform the program by adding a ghost variable c which counts the number of iterations of the loop (and make sure it is not sliced during the encoding). Then, FreqHorn would automatically take c into account while constructing the sampling grammar, and this would lead to the discovery of a invariant $x = 2 \cdot c \wedge x \leq 256$. In the future, it would make sense to consider various types of program transformations (also, to tackle the tasks with several loops [47]) in the preprocessing steps of FreqHorn.

10 Related work

Our enumerative approach can be considered as *data-driven* since it treats frequencies of various features in the source code as *data*. Among other enumerative-learning techniques, MCMC [52] employs Metropolis Hastings MCMC sampler to produce candidates for the whole invariant. Similarly to our approach, it obtains some statistics from the code (namely, constants), but as can be seen from our evaluation, it is not enough to guarantee consistent results. In [26,27,29], the *data* is obtained by executing programs. Then, the learning of invariants proceeds by analyzing the program traces and does not take into account the source code. This idea, in fact, could also be used in our syntax-guided approach: at least we could add more constants to the grammar. However we are currently unaware of a strategy to find meaningful constants and to avoid over-population of the grammar by too many constants. Our preliminary experiments resulted so far in a performance decrease.

There is a large body of inductive and non-enumerative SMT-based techniques for invariant synthesis, and due to lack of space we list only a few here. IC3/PDR [7,8,15,32,41] and abductive inference [14] depend crucially on the background theory of verification conditions which should admit interpolation and/or quantifier elimination. We use some insights from these approaches to accelerate our synthesis algorithm, but we do not necessarily depend on them. This, for instance, allows us to discover invariants over non-linear arithmetic, which at the moment is an unattainable goal for most of the state-of-the-art tools.

The idea of keeping CTIs and analyzing them to push previously considered lemmas is exploited in the implementations of IC3 [54]. The idea of applying Houdini to extract invariants from proofs of bounded safety was fundamental for the first version of Spacer [42]. They, however, keep obtaining proofs along the entire verification process. In contrast, we obtain proofs only at the very beginning, and the remaining progress of the algorithm is entirely dictated by the success of sampling.

Some prior work considered non-enumerative invariant inference from the source code. In Formula Slicing [39], a variant of Houdini, candidate invariants are chosen from the *Init* formulas (not from *Tr* or *Bad*, as in our case). In Niagara [18,20] candidate invariants are obtained from the previous versions of the program. Despite those techniques proceed in the “*guess-and-check*” manner, for each new guess they just weaken the formula from the previous guess. In contrast, we permit much wider search space.

Techniques for automatic construction of grammars were applied outside of formal verification, but in the domains of security analysis and dynamic test generation [34,35]. Indeed, mutations of the input data for some program can in fact be used as new input data and therefore can increase the testing coverage.

Finally, SyGuS [2] keep being used in program synthesis more frequently than in the inductive invariant synthesis. For instance, in applications [16,36,49,51,56] a formal grammar is additionally provided, and it is considered a part of specification. In contrast, in our application, the verification condition contains the encoding of the entire program and the safety specification, which together are enough for the completely automatic formal grammar construction. This is in fact the main driving idea behind FreqHorn, and it leaves us a spacious room for its further adaptations, e.g., in proving and disproving program termination, automated repair of software regressions, and security analysis.

11 Conclusion

This article has given an overview of the SyGuS-based approach to inductive invariant synthesis. Our algorithm discovers conjunctive invariants by iteratively sampling candidate formulas from a formal grammar, constructed automatically from the syntax and bounded semantics of the given program. The learning process is guided by frequency distributions, collected after a simple statistical analysis over the seed formulas, and it further prunes and prioritizes the search space after each positive or negative sample. Our implementation FreqHorn is the first tool that constructs the grammars and distributions completely automatically, and enables fast verification of numeric programs. Similarly to most of the state-of-the-art verification techniques, our approach enjoys a tight integration with well renowned formal methods and should be treated as an example of successful interchange of ideas across application domains.

References

1. Albarghouthi A, Gurfinkel A, Chechik M (2012) From under-approximations to over-approximations and back. In: TACAS, volume 7214 of LNCS. Springer, Berlin, pp 157–172
2. Alur R, Bodík R, Juniwal G, Martin MMK, Raghothaman M, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A (2013) Syntax-guided synthesis. In: FMCAD. IEEE, pp 1–17
3. Beyer D, Dangl M, Wendler P (2015) Boosting k-Induction with Continuously-Refined Invariants. In: CAV, Part I, volume 9206 of LNCS, pp 622–640
4. Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic Model Checking without BDDs. In: TACAS, volume 1579 of LNCS. Springer, Berlin, pp 193–207
5. Blicha M, Hyvärinen AEJ, Kofron J, Sharygina N (2019) Decomposing farkas interpolants. In: TACAS, Part I, volume 11427 of LNCS. Springer, Berlin, pp 3–20
6. Bradley AR (2011) SAT-based model checking without unrolling. In: VMCAI, volume 6538 of LNCS. Springer, Berlin, pp 70–87
7. Bradley AR (2012) Understanding IC3. In: SAT, volume 7317 of LNCS. Springer, Berlin, pp 1–14
8. Bradley AR, Manna Z (2008) Property-directed incremental invariant generation. Formal Asp Comput 20(4–5):379–405
9. Champion A, Kobayashi N, Sato R (2018) HoIce: an ICE-based non-linear horn clause solver. In: APLAS, volume 11275 of LNCS. Springer, Berlin, pp 146–156
10. Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: CAV, volume 1855 of LNCS. Springer, Berlin, pp 154–169
11. Craig W (1957) Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J Symb Log 22:269–285
12. de Moura LM, Bjørner N (2008) Z3: an efficient SMT Solver. In: TACAS, volume 4963 of LNCS. Springer, Berlin, pp 337–340
13. Dietsch D, Heizmann M, Hoenicke J, Nutz A, Podelski A (2019) Ultimate TreeAutomizer. In: HCVS/PERR, volume 296 of EPTCS, pp 42–47
14. Dillig I, Dillig T, Li B, McMillan KL (2013) Inductive invariant generation via abductive inference. In: OOPSLA. ACM, London, pp 443–456
15. Eén N, Mishchenko A, Brayton RK (2011) Efficient implementation of property directed reachability. In: FMCAD. IEEE, pp 125–134

16. Fedyukovich G, Ahmad MBS, Bodík R (2017) Gradual synthesis for static parallelization of single-pass array-processing programs. In: PLDI. ACM, London, pp 572–585
17. Fedyukovich G, Bodík R (2018) Accelerating syntax-guided invariant synthesis. In: TACAS, Part I, volume 10805 of LNCS. Springer, Berlin, pp 251–269
18. Fedyukovich G, Gurfinkel A, Sharygina N (2014) Incremental verification of compiler optimizations. In: NFM, volume 8430 of LNCS. Springer, Berlin, pp 300–306
19. Fedyukovich G, Gurfinkel A, Sharygina N (2015) Automated discovery of simulation between programs. In: LPAR, volume 9450 of LNCS. Springer, Berlin, pp 606–621
20. Fedyukovich G, Gurfinkel A, Sharygina N (2016) Property directed equivalence via abstract simulation. In: CAV, vol 9780. Part II of LNCS. Springer, Berlin, pp 433–453
21. Fedyukovich G, Kaufman S, Bodík R (2017) Sampling Invariants from Frequency Distributions. In: FMCAD. IEEE, pp 100–107
22. Fedyukovich G, Prabhu S, Madhukar K, Gupta A (2018) Solving constrained horn clauses using syntax and data. In: FMCAD. IEEE, pp 170–178
23. Fedyukovich G, Prabhu S, Madhukar K, Gupta A (2019) Quantified invariants via syntax-guided synthesis. In: CAV, Part I, volume 11561 of LNCS. Springer, Berlin, pp 259–277
24. Fedyukovich G, Zhang Y, Gupta A (2018) Syntax-guided termination analysis. In: CAV, Part I, volume 10981 of LNCS. Springer, Berlin, pp 124–143
25. Flanagan C, Leino KRM (2001) Houdini: an Annotation Assistant for ESC/Java. In: FME, volume 2021 of LNCS. Springer, Berlin, pp 500–517
26. Garg P, Löding C, Madhusudan P, Neider D (2014) ICE: a robust framework for learning invariants. In: CAV, volume 8559 of LNCS. Springer, Berlin, pp 69–87
27. Garg P, Neider D, Madhusudan P, Roth D (2016) Learning invariants using decision trees and implication counterexamples. In: POPL. ACM, London, pp 499–512
28. Grebenshchikov S, Lopes NP, Popeea C, Rybalchenko A (2012) Synthesizing software verifiers from proof rules. In: PLDI. ACM, London, pp 405–416
29. Gulwani S, Jojic N (2007) Program verification as probabilistic inference. In: POPL. ACM, London, pp 277–289
30. Heizmann M, Hoenicke J, Podelski A (2010) Nested interpolants. In: POPL. ACM, London, pp 471–482
31. Henzinger TA, Jhala R, Majumdar R, McMillan KL (2004) Abstractions from proofs. In: POPL. ACM, London, pp 232–244
32. Hoder K, Bjørner N (2012) Generalized property directed reachability. In: SAT, volume 7317 of LNCS. Springer, Berlin, pp 157–171
33. Hojjat H, Konecný F, Garnier F, Iosif R, Kuncak V, Rümmer P (2012) A verification toolkit for numerical transition systems—tool paper. In: FM, volume 7436 of LNCS. Springer, Berlin, pp 247–251
34. Höschele M, Zeller A (2016) Mining input grammars from dynamic taints. In: ASE. ACM, London, pp 720–725
35. Höschele M, Zeller A (2017) Mining input grammars with AUTOGRAM. In: ICSE—companion volume. IEEE Computer Society, pp 31–34
36. Inala JP, Polikarpova N, Qiu X, Lerner BS, Solar-Lezama A (2017) Synthesis of recursive ADT transformations from reusable templates. In: TACAS, Part I, volume 10205 of LNCS, pp 247–263
37. Jovanovic D, Dutertre B (2016) Property-directed k-induction. In: FMCAD. IEEE, pp 85–92
38. Kafle B, Gallagher JP, Morales JF (2016) Rahft: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In: CAV, Part I, volume 9779 of LNCS. Springer, Berlin, pp 261–268
39. Karpenkov EG, Monniaux D (2016) Formula slicing: inductive invariants from preconditions. In: HVC, volume 10028 of LNCS. Springer, Berlin, pp 169–185
40. Kincaid Z, Cyphert J, Breck J, Reps TW (2018) Non-linear reasoning for invariant synthesis. PACMPL 2(POPL):54:1–54:33
41. Komuravelli A, Gurfinkel A, Chaki S (2014) SMT-based model checking for recursive programs. In: CAV, volume 8559 of LNCS, pp 17–34
42. Komuravelli A, Gurfinkel A, Chaki S, Clarke EM (2013) Automatic abstraction in SMT-based unbounded software model checking. In: CAV, volume 8044 of LNCS. Springer, Berlin, pp 846–862
43. Le TC, Zheng G, Nguyen T (2019) SLING: using dynamic analysis to infer program invariants in separation logic. In: PLDI. ACM, London, pp 788–801
44. McMillan KL (2003) Interpolation and SAT-based model checking. In: CAV, volume 2725 of LNCS. Springer, Berlin, pp 1–13
45. McMillan KL (2006) Lazy abstraction with interpolants. In: CAV, volume 4144 of LNCS. Springer, Berlin, pp 123–136
46. McMillan KL (2014) Lazy annotation revisited. In: CAV, volume 8559 of LNCS. Springer, Berlin, pp 243–259

47. Mordvinov D, Fedyukovich G (2017) Synchronizing Constrained Horn Clauses. In: LPAR, volume 46 of EPiC Series in Computing. EasyChair, pp 338–355
48. Padon O, McMillan KL, Panda A, Sagiv M, Shoham S (2016) Ivy: safety verification by interactive generalization. In: PLDI. ACM, London, pp 614–630
49. Phothilimthana PM, Jelvis T, Shah R, Totla N, Chasins S, Bodík R (2014) Chlorophyll: synthesis-aided compiler for low-power spatial architectures. In: PLDI. ACM, London, pp 396–407
50. Pick L, Fedyukovich G, Gupta A (2018) Exploiting synchrony and symmetry in relational verification. In: CAV, Part I, volume 10981 of LNCS. Springer, Berlin, pp 164–182
51. Pu Y, Bodík R, Srivastava S (2011) Synthesis of first-order dynamic programming algorithms. In: OOPSLA. ACM, London, pp 83–98
52. Sharma R, Aiken A (2014) From invariant checking to invariant inference using randomized search. In: CAV, volume 8559 of LNCS. Springer, Berlin, pp 88–105
53. Solar-Lezama A, Tancau L, Bodík R, Seshia SA, Saraswat VA (2006) Combinatorial sketching for finite programs. In: ASPLOS. ACM, London, pp 404–415
54. Suda M (2013) Triggered clause pushing for IC3. CoRR, arXiv:1307.4966
55. Vazou N, Seidel EL, Jhala R, Vytiniotis D, Jones SLP (2014) Refinement types for Haskell. In: ICFP. ACM, London, pp 269–282
56. Yang W, Fedyukovich G, Gupta A (2019) lemma synthesis for automating induction over algebraic data types. In: CP, volume 11802 of LNCS. Springer, Berlin, pp 600–617
57. Zhu H, Magill S, Jagannathan S (2018) A data-driven CHC solver. In: PLDI. ACM, London, pp 707–721