Sampling Invariants from Frequency Distributions

Grigory Fedyukovich   Samuel J. Kaufman   Rastislav Bodík
University of Washington Paul G. Allen School of Computer Science & Engineering
{grigory, kaufmans, bodik}@cs.washington.edu

***Abstract*—We present a new SMT-based, probabilistic, syntax-guided method to discover numerical inductive invariants. The core idea is to initialize frequency distributions from the program's source code, then repeatedly sample lemmas from those distributions, and terminate when the conjunction of learned lemmas becomes a safe invariant. The sampling process gets further optimized by priority distributions fine-tuned after each positive and negative sample. The stochastic nature of this approach admits simple, asynchronous parallelization. We implemented and evaluated this approach in a tool called FreqHorn which shows competitive performance on well-known linear and some non-linear programs.**

I. Introduction

Automated formal verification of programs handling unbounded loops is reduced to finding safe inductive invariants that over-approximate the sets of reachable states, but precise enough to prove unreachability of the error state. Successful solutions to this problem include Counterexample-Guided Abstraction Refinement [1] and Property Directed Reachability (PDR) [2], [3], [4], but they are not guaranteed to deliver appropriate invariants.

We aim at learning inductive invariants in an "*enumerate-and-check*" manner [5], [6]. While this approach in general meets a lot of skepticism, there are particular synthesis tasks which can be efficiently solved using tailored heuristics. Our intuition behind applying this synthesis paradigm to discover invariants is that an invariant can often be caught on the surface, i.e., it to some degree imitates the syntactical constructions which appear in the source code.

Source code can give hints for guessing a candidate to be checked for invariance. Any information of occurrences of variables, constants, arithmetic and comparison operators, and their applications can potentially guide the search of invariants. The research question we address in this paper is whether a probability distribution constructed by processing the source code could help sampling successful candidates. This reduces the number of invariance checks and decreases the total verification time.

We contribute a framework for learning invariants using sampling from probability distributions obtained after collecting multiple facts about the given source code. Before sampling, we fix a number of features which could belong to each invariant. We then split the code in clauses, normalize each clause, and check how many of the predetermined features belong to it. The statistics collected from all normalized clauses define a number of *frequency distributions.*

The main workhorse in our framework is a repetitive process that samples different pieces of a candidate invariant from the frequency distributions and then assembles them together. Each assembled candidate has a certain feature with a probability specified in the corresponding distribution. So it is likely that our sampled candidates are in some sense *representative*. Finally, the candidate is checked using an off-the-shelf SMT solver. If it is proven to be an actual invariant, our algorithm stores it and proceeds to discovering other invariants. The search continues until all invariants that are needed to verify safety are discovered, or until the search space is exhausted.

Our second contribution is an algorithm that combines sampling from frequency distributions and sampling from *priority distributions* created on the fly and adjusted after each positive and negative sample. That is, once a candidate is checked, some "*likely unrelated*" candidates get higher priorities for being sampled in the coming iterations. We show how this strategy can be made aggressive, i.e., the completeness of the search space exploration is traded off for widening of candidate diversity.

The approach has been implemented in a tool called FreqHorn which naturally admits parallelization. FreqHorn uses an SMT solver to check each sample for invariance. The learning strategy with priority distributions is shown to be extremely effective in practice, despite for some pathological situations it affects the convergence. As expected, our tool is competitive to the closely related machine-learning-based tools for invariants learning, and in some cases it is more effective than PDR-based tools.

The rest of the paper is structured as follows. Sect. II briefly discusses the fundamentals of our verification problem. Then, in Sect. III, we introduce the framework to learn numerical invariants, describe its optimizations and drawbacks, and in Sect. IV, we describe our parallel implementation, evaluation, and comparison with other tools. Sect. V has an overview of the related work, and Sect. VI concludes the paper.

II. Background

A. Programs and their inductive invariants

We use vector notation to denote sequences (e.g., of variables or constants). We assume the first-order formulas $\varphi(\vec{x}) \in \mathit{Expr}$ in the paper. For simplicity, we write φ when the arguments are clear from the context. 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. If φ is satisfiable, we write $\varphi \not\Longrightarrow \bot$ (and $\varphi \implies \bot$ otherwise). Formula ψ is called a tautology if $\neg\psi \implies \bot$.

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. For every input variable $x \in V$, there is a corresponding output variable $x' \in V'$ (i.e., the value of x in the next state).

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

$$Init(V) \implies Inv(V) \quad (1)$$

$$Inv(V) \wedge Tr(V, V') \implies Inv(V') \quad (2)$$

Example 1. *Consider Fig. 1 showing program* `Bradley` *named after its appearance in [2]. It has counter* `x` *that gets repeatedly added to variable* `y`*. Examples of inductive invariants include $x \geqslant 0$, $x \geqslant 0 \wedge y \geqslant 0$, $x \geqslant 0 \wedge x + y \geqslant 0$, and $x \geqslant 0 \wedge y - x \geqslant 0$.* □

Lemma 1. *Given program P, if Inv_1 and Inv_2 are inductive invariants for P, then $Inv_1 \wedge Inv_2$ is also an inductive invariant.*

Lemma 1 does not work in the reverse direction: if a conjunction of formulas is an inductive invariant then each conjunct in isolation could not be an inductive invariant. For example, $x \geqslant 0 \wedge y \geqslant 0$ is an inductive invariant for `Bradley`, but $y \geqslant 0$ is not an inductive invariant.

Lemma 2. *Given program $P = \langle Var, Init, Tr\rangle$, let Inv_1 be an inductive invariant for P, program P_1 be $\langle Var, Init, Tr \wedge Inv_1\rangle$, and Inv_2 be an inductive invariant for P_1; then $Inv_1 \wedge Inv_2$ is an inductive invariant for P.*

Lemma 2 enables incremental invariant discovery. For example, for `Bradley`, one could find an inductive invariant $x \geqslant 0$ first and then conjoin it to the transition relation. It remains to find an inductive invariant satisfying the strengthened transition relation, e.g., $y \geqslant 0$. Finally, conjunction $x \geqslant 0 \wedge y \geqslant 0$ is an invariant for `Bradley`.

Definition 3. *A* 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 formula encoding the* error states *over V.*

A verification task has a solution if the set of error states is not reachable. We call the program *safe* in this case. Safety is decided by discovering a safe inductive invariant, a formula that covers the initial state, is closed under the transition relation, and does not cover the error state.

```
int x = y = 0;
while (*) {
  x = x + 1;
  y = y + x;
}
assert(y >= 0);
```

Figure 1: Possibly infinite loop over algebraic integers (cf. [2]).

Definition 4. *Let $P = \langle V \cup V', Init, Tr\rangle$ and $\langle P, Bad\rangle$ be a verification task; an inductive invariant Inv for P is called* safe *if:*

$$Inv(V) \wedge Bad(V) \implies \bot \quad (3)$$

Examples of safe inductive invariants for `Bradley` include $x \geqslant 0 \wedge y \geqslant 0$ and $x \geqslant 0 \wedge y - x \geqslant 0$.

B. Sampling from probability distributions

Definition 5. *A probability distribution on a set A is a function $p : A \to \mathbb{R}$, such that $\forall a \in A \,.\, 0 \leqslant p(a) \leqslant 1$ and $\sum\limits_{a \in A} p(a) = 1$.*

In this paper, we consider a process which, given a set of formulas and a probability distribution, chooses at each iteration an element from the set with a probability determined by the distribution.

Example 2. *Given four formulas over x and y, a probability distribution $p^{\texttt{Bradley}}$ could be defined as follows:*

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

In order to prove program `Bradley` *safe, it could be sufficient to sample from distribution $p^{\texttt{Bradley}}$ two times and to check invariance incrementally for each sample. Assuming that formula $x \geqslant 0$ was sampled at the first round (with probability 0.4), it is enough to sample either $y \geqslant 0$ or $y - x \geqslant 0$ (with probability $0.3 + 0.1$). Thus, the probability of discovering a safe inductive invariant in two steps equals* $0.4 \cdot (0.3 + 0.1) = 0.16$. □

Consider now a scenario that rejects all samples that were already checked for invariance (e.g., by nudging the distribution accordingly). It is easy to see that the probability of discovering a safe inductive invariant (in two steps) increases.

In the next section, we discuss a practical way of creating both, the sets of samples, and their probability distributions.

III. Learning Numerical Invariants

A. Grammar and probabilistic production rules

Fig. 2 shows a grammar for generating the candidate inductive invariants (also referred to as the *sampling*

$$
\begin{aligned}
c &::= c_1 \mid c_2 \mid \ldots \mid c_\ell \\
k &::= k_1 \mid k_2 \mid \ldots \mid k_m \\
x &::= x_1 \mid x_2 \mid \ldots \mid x_n \\
\mathit{lincom} &::= k \cdot x + k \cdot x + \ldots + k \cdot x \\
\mathit{ineq} &::= \mathit{lincom} > c \mid \mathit{lincom} \geqslant c \\
\mathit{cand} &::= \mathit{ineq} \vee \mathit{ineq} \vee \ldots \vee \mathit{ineq}
\end{aligned}
$$

Figure 2: Sampling grammar.

grammar). The formulas are generated using *probabilistic production rules*. In contrast to standard non-deterministic production rules, each choice is in line with a particular *probability distribution*, making the samples more predictable.

The sampling works in a top-to-bottom manner. Given a probability distribution $p_\vee$ for the arities of the *or*-operator, we sample a value n from $p_\vee$ and reserve n slots for operands of $\vee$ (linear inequalities). Then, for each $1 \leqslant i \leqslant n$, we sample a non-empty subset $\vec{x} \subseteq V$ of variables from a given probability distribution p_+. Then, given a sequence of probability distributions $\{p_{k_j}\}$ for scalar coefficients for each variable $x_j \in V$, we sample a value $k_j \in K \subseteq \mathbb{R}\backslash\{0\}$. Summing products of each k_j with x_j, we get a linear combination (denoted $\{x_j, k_j\}$). Lastly, for each inequality, we sample a binary comparison operator (either $>$ or $\geqslant$) and a constant $c \in C \subseteq \mathbb{R}$ from given probability distributions p_{op} and p_c, respectively.

Each conjunction-free sample is individually checked for invariance. Following Lemma 2, each successful sample is conjoined to the transition relation, and thus it will be used while checking invariance of samples in the future. This lets us discover conjunctive invariants without having the conjunction operator in the sampling grammar.

Note that the sampling grammar does not contain the comparison operators other than $>$ and $\geqslant$. The expressiveness of formulas is achieved by providing large enough sets of K and C for numerical coefficients and constants: if an element is in a set, its additive inverse is also in the set. Thus, instead of generating formula $k_1 \cdot x_1 + \ldots + k_n \cdot x_n < c$, we generate an equivalent formula $(-k_1) \cdot x_1 + \ldots + (-k_n) \cdot x_n > -c$ (see more details in Sect. III-B).

In practice, there could be dependencies among ingredients of a sample. For instance, the number of disjuncts in a sample could affect the variables, coefficients, constants, and the comparison operators appearing in each disjunct. Because our sampling is hierarchical, the dependencies propagate top-to-bottom. To formally address this, we allow the production rules operate over conditional probability distributions.

B. Value ranges and frequency distributions

The sampling grammar imposes the fixed structure on the candidate invariants. The key to success while assembling each candidate is to fix the sets of numerical constants K and C. Our contribution is the technique that 1) automatically constructs these sets and 2) supplies each production rule in the grammar with the probability distribution. We achieve both targets via exploring the *Init*, *Tr*, and *Bad* formulas, included in the verification task, and calculating the frequencies of appearances of particular constants.

The algorithm of frequency calculations is informally described below. It starts with converting the *Init*, *Tr*, and *Bad* formulas to the Conjunctive Normal Form, splitting them into *clauses* (i.e., disjunctions of linear inequalities) and inserting the clauses to two sets, denoted respectively A_V and $A_{V \cup V'}$. Set A_V contains elements which have appearances of input variables V only (i.e., all elements obtained from *Init* or *Bad* and possibly some elements from *Tr*). Set $A_{V \cup V'}$ contains elements which have appearances of input and output variables V and V' at the same time (e.g., $x' = x + 2$ which can be originated from *Tr*).

Then, for each clause $a \in A_V$, an application of $\neq$, $=$, $<$, or $\leqslant$ is replaced by application(s) of $>$ or $\geqslant$:

$$
\frac{A < B}{-A > -B} \qquad \frac{A \leqslant B}{-A \geqslant -B}
$$

$$
\frac{A = B}{A \geqslant B \wedge -A \geqslant -B} \qquad \frac{A \neq B}{A > B \vee -A > -B}
$$

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., $A_V \leftarrow A_V \backslash \{a\} \cup \{a_+, a_-\}$. After this rewriting, we assume that each clause $a \in A_V$ matches the sampling grammar. Thus, it is straightforward to determine the arity of the $\vee$-operator (and include them to set N), numerical coefficients (and include them to set K), and constants (and include them to set C).

Additionally, we collect constants which appear in clauses $A_{V \cup V'}$ and include them to K. The last trick is to include products $c \cdot k$ to K, for any $c \in C$ and $k \in K$; and products $c_1 \cdot c_2$ to C, for any $c_1, c_2 \in C$.

Definition 6. *The set of formulas specified by the grammar in Fig. 2, in which the sets of arities of the $\vee$-operator N, numerical coefficients K, and constants C are obtained from A_V and $A_{V \cup V'}$ is called an* appearance-guided search space*.*

Finally, we are ready to calculate various statistics, in particular:

- how often $a \in A_V$ has arity $i \in N$,
- how often each combination of variables $\vec{x} \subseteq V$ appears among the inequalities,
- how often a variable $x \in V$ has a coefficient $k \in K$,
- how often a constant $c \in C$ appears among the inequalities,

Algorithm 1: Sampling inductive invariants.

```
Input: $P = \langle V \cup V', Init, Tr \rangle$: program;
       $\langle P, Bad \rangle$: verification task
Output: $learnedLemmas$: set of $Expr$
1  $A_V, A_{V \cup V'} \leftarrow$ NORMALIZE($P$);
2  $C, K, N \leftarrow$ GETRANGES($A_V, A_{V \cup V'}$);
3  $\{p_*\} \leftarrow$ GETFREQUENCIES($A_V, A_{V \cup V'}$);
4  $learnedLemmas \leftarrow \varnothing$;
5  while $(Bad(V) \wedge \bigwedge_{\ell \in learnedLemmas} \ell(V) \not\Longrightarrow \bot)$ do
6      $Cand \leftarrow \bot$;
7      $n \leftarrow$ SAMPLE($p_\vee$);
8      for $(i \in [1, n])$ do
9          $\vec{x}_i \leftarrow$ SAMPLE($p_+ \mid n$);
10         $\vec{k}_i \leftarrow$ SAMPLE($p_k \mid n, \vec{x}$);
11     for $(i \in [1, n])$ do
12         $c_i \leftarrow$ SAMPLE($p_c \mid i, n, \{\vec{x}_i, \vec{k}_i\}$);
13         $op_i \leftarrow$ SAMPLE($p_{op} \mid i, n, c_i, \{\vec{x}_i, \vec{k}_i\}$);
14         $Cand \leftarrow Cand \vee$ ASSEMBLEINEQ($\vec{x}_i, \vec{k}_i, c_i, op_i$);
15     if $(\neg Cand(V) \Longrightarrow \bot)$ then continue;
16     if $(Init(V) \wedge \neg Cand(V) \not\Longrightarrow \bot)$ then continue;
17     if $(Cand(V) \wedge Tr(V, V') \wedge \neg Cand(V') \wedge$
           $\bigwedge_{\ell \in learnedLemmas} \ell(V) \not\Longrightarrow \bot)$ then continue;
18     $learnedLemmas \leftarrow learnedLemmas \cup Cand$;
```

- how often an operator $op \in \{>, \geqslant\}$ appears among the inequalities.

These statistics are used to construct frequency distributions, respectively: $p_\vee$, p_+, p_{k_0}, ..., p_{k_n}, p_c, and p_{op}, and to guide the sampling process. To enlarge the search space, an artificial ϵ-frequency representing appearances that never happened in the actual code (e.g., by connecting a variable and a constant that never appear together) could be introduced. The value of ϵ-frequency could be chosen heuristically based on values of other frequencies, as long as it stays sufficiently small and positive.

C. Core algorithm

Alg. 1 shows the routines of the *sampler*, the invariance *checker*, and their interaction. As a preprocessing step (lines 1-3), the algorithm normalizes formulas, collects sets N, K, and C, and calculates frequencies as described in Sect. III-B.

The sampler generates a formula (line 14) from the appearance-guided search space using the frequency distributions. In a naive scenario, the sampler deals with distributions $p_\vee$, p_+, p_{k_0},..., p_{k_n}, p_c, and p_{op} directly. However, in order to make the sampling more predictable, the algorithm creates conditional distributions and samples from them (lines 9-13), in particular:

- how often each combination of variables $\vec{x} \subseteq V$ appears among inequalities which are contained in a clause of the given arity n,
- how often a constant $c \in C$ appears in *ineq*, given *ineq* is the i-th inequality among n inequalities $\{ineq_j\}$ and each $ineq_j$ is over particular $\{\vec{x}_j, \vec{k}_j\}$,
- etc ...

As mentioned in Sect. III-A, the choice of conditional distributions is justified by the order of sampling of each ingredient of a candidate *Cand*. Thus, any change in this order may affect the conditional distributions used for sampling. At the same time, any conditions for the distributions could be made optional (depending on the problem in hand). Evidently, this does not affect soundness of the entire approach, but affects the speed of convergence.

If the sampled *Cand* is a tautology (performed by an SMT solver, line 15), then it is known to be an inductive invariant, but it does not make any progress towards completing the verification; thus *Cand* should be withdrawn. It would also make sense to withdraw all unsatisfiable candidates, but by construction, the number of products $k_i \cdot x_i$ in each disjunct is always positive, which makes *Cand* always satisfiable.

The checker decides a number of local SMT queries per each *Cand*. A negative result – i.e., whenever *Cand* is a tautology or the initiation or the consecution check fails (line 16 or 17, respectively) – is called an *invariance failure*. Otherwise, the result is positive and is called a *learned lemma*. Each new learned lemma is book-kept (line 18) for the safety check (line 5), and also for the consecution checks (line 17) of candidates coming in the next iterations (recall Lemma 2). The sampler and the checker alternate until a safe inductive invariant is found.

Theorem 1. *If a safe inductive invariant can be expressed by a conjunction of formulas within the appearance-guided search space; then the probability that Alg. 1 eventually discovers it tends to 1.*

D. Prioritizing the search space

The success of techniques based on syntax-guided synthesis (SyGuS) depends on how effectively the search space is pruned after a positive or negative sample is examined. To avoid repeatedly appearing learned lemmas and invariance failures in the future, we introduce a pool of all samples and refer to it whenever a new candidate is sampled. Furthermore, we employ a lightweight analysis to identify and block some closely related candidates from being sampled and prioritize some likely unrelated candidates to being sampled and checked for invariance.

Definition 7. *The* $priorMap_{\langle \vec{x}, \vec{k} \rangle}$ *function maps linear combinations* $\{\vec{x}_i, \vec{k}_i\}$ *to joint probability distributions for* op_i *and* c_i *(called* priority distributions*).*

One natural goal of $priorMap_{\langle \vec{x}, \vec{k} \rangle}$ is to set to zero probabilities of sampling the candidates which are 1) already checked, 2) stronger than failures, and 3) weaker than learned lemmas. Consequently, the probabilities of other candidates should be increased, and we achieve it by exploiting the ordering of constants and comparison operators in the sampling grammar. Alg. 2 shows a version of Alg. 1, augmented with prioritizing capabilities (the pseudocode inherited from Alg. 1 is decoloured).

Algorithm 2: Sampling inductive invariants from priority distributions.

```
Input: $P = \langle V \cup V', Init, Tr\rangle$: program;
       $\langle P, Bad\rangle$: verification task
Output: $learnedLemmas$: set of $Expr$
1  $A_V, A_{V \cup V'} \leftarrow$ NORMALIZE$(P)$;
2  $C, K, N \leftarrow$ GETRANGES$(A_V, A_{V \cup V'})$;
3  $\{p_*\} \leftarrow$ GETFREQUENCIES$(A_V, A_{V \cup V'})$;
4  $learnedLemmas \leftarrow \varnothing$;
5  while $(Bad(V) \wedge \bigwedge_{\ell \in learnedLemmas} \ell(V) \not\Longrightarrow \bot)$ do
6      $Cand \leftarrow \bot$;
7      $n \leftarrow$ SAMPLE$(p_\vee)$;
8      for $(i \in [1, n])$ do
9          $\vec{x}_i \leftarrow$ SAMPLE$(p_+ \mid n)$;
10         $\vec{k}_i \leftarrow$ SAMPLE$(p_k \mid n, \vec{x})$;
11     if $(priorMap_{\langle \vec{x}, \vec{k}\rangle} = \varnothing)$ then
12         $priorMap_{\langle \vec{x}, \vec{k}\rangle} \leftarrow uniform_{\langle \vec{x}, \vec{k}\rangle}$;
13         for $(i \in [1, n])$ do
14             $c_i \leftarrow$ SAMPLE$(p_C \mid i, n, \{\vec{x}_i, \vec{k}_i\})$;
15             $op_i \leftarrow$ SAMPLE$(p_{Op} \mid i, n, c_i, \{\vec{x}_i, \vec{k}_i\})$;
16             $Cand \leftarrow Cand \vee$ ASSEMBLEINEQ$(\vec{x}_i, \vec{k}_i, c_i, op_i)$;
17     else if $(priorMap_{\langle \vec{x}, \vec{k}\rangle} =$ undefined$)$ then continue;
18     else
19         for $(i \in [1, n])$ do
20             $c_i, op_i \leftarrow$ SAMPLE$(priorMap_{\langle \vec{x}, \vec{k}\rangle}(\{\vec{x}_i, \vec{k}_i\}))$;
21             $Cand \leftarrow Cand \vee$ ASSEMBLEINEQ$(\vec{x}_i, \vec{k}_i, c_i, op_i)$;
22     if $(\neg Cand(V) \Longrightarrow \bot)$ then
23         UPDATE$(priorMap_{\langle \vec{x}, \vec{k}\rangle}$, PRIORITIZE$\,{}^{\uparrow}_{\vec{op},\vec{c}}(\langle \vec{x}, \vec{k}\rangle))$;
24         continue;
25     if $(Init(V) \wedge \neg Cand(V) \not\Longrightarrow \bot)$ then
26         UPDATE$(priorMap_{\langle \vec{x}, \vec{k}\rangle}$, PRIORITIZE$\,{}^{\vec{op},\vec{c}}_{\downarrow}(\langle \vec{x}, \vec{k}\rangle))$;
27         continue;
28     if $(Cand(V) \wedge Tr(V, V') \wedge \neg Cand(V') \wedge \bigwedge_{\ell \in learnedLemmas} \ell(V) \not\Longrightarrow \bot)$ then
29         UPDATE$(priorMap_{\langle \vec{x}, \vec{k}\rangle}$, PRIORITIZE$\,{}^{\vec{op},\vec{c}}_{\downarrow}(\langle \vec{x}, \vec{k}\rangle))$;
30         continue;
31     $learnedLemmas \leftarrow learnedLemmas \cup Cand$;
32     UPDATE$(priorMap_{\langle \vec{x}, \vec{k}\rangle}$, PRIORITIZE$\,{}^{\uparrow}_{\vec{op},\vec{c}}(\langle \vec{x}, \vec{k}\rangle))$;
```

For each sequence of linear combinations $\langle \vec{x}, \vec{k}\rangle \overset{\text{def}}{=} \{\{\vec{x}_i, \vec{k}_i\}\}$, that has been sampled for the first time: 1) $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ gets assigned a sequence of *uniform* joint distributions for each $\vec{x}_i$ and $\vec{k}_i$ (line 12), and 2) the remaining ingredients for assembling *Cand* are sampled from the frequency distributions (lines 13-16, as in Alg. 1). Before proceeding to the next iteration, each positive or negative result nudges distributions at $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ (line 23, 26 29, or 32).

For each $\langle \vec{x}, \vec{k}\rangle$, that has been sampled not for the first time, there exists a sequence of *non-uniform* distributions at $priorMap_{\langle \vec{x}, \vec{k}\rangle}$. Further, the sampler produces candidates from that distribution instead of the frequency distributions (lines 19-21), and again, distributions at $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ get nudged after each positive and negative result.

Once for some $\langle \vec{x}, \vec{k}\rangle$ the search is exhausted then the distributions at $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ are said to be undefined (line 17), and the sampler proceeds to exploring another sequence of linear combinations. In the rest of the subsection, we clarify how distributions at $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ get nudged and how it might make them undefined.

Definition 8. *Given $\langle \vec{x}, \vec{k}\rangle$ and $ineq_1 \vee \ldots \vee ineq_n$, such that each $ineq_i$ is over a linear combination $\{\vec{x}_i, \vec{k}_i\}$, operator op_i, and constant c_i, we write:*

- PRIORITIZE$\,{}^{\uparrow}_{\vec{op},\vec{c}}(\langle \vec{x}, \vec{k}\rangle)$ *– to produce a joint probability distribution for each $1 \leqslant i \leqslant n$ 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;*
- PRIORITIZE$\,{}^{\vec{op},\vec{c}}_{\downarrow}(\langle \vec{x}, \vec{k}\rangle)$ *– to produce a joint probability distribution for each $1 \leqslant i \leqslant n$ 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* PRIORITIZE$\,{}^{\uparrow}_{\vec{op},\vec{c}}(\langle \vec{x}, \vec{k}\rangle)$*).*

Example 3. *Let $C = \{-5, 0, 5\}$, and $ineq_1 \vee ineq_2 = x > -5 \vee x + y \geqslant 5$ be a learned lemma for some program P. Then, any disjunction $ineq'_1 \vee ineq'_2$ where $ineq'_1 \in \{x \geqslant -5, x > -5\}$, and $ineq'_2 \in \{x+y \geqslant -5, x+y > -5, x + y \geqslant 0, x + y > 0, x + y \geqslant 5\}$ is weaker or equal to $ineq_1 \vee ineq_2$, and checking its invariance would not affect our verification process. The* PRIORITIZE$\,{}^{\uparrow}_{\vec{op},\vec{c}}(\langle \vec{x}, \vec{k}\rangle)$ *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:*

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

□

Distributions $priorMap_{\langle \vec{x}, \vec{k}\rangle}$ block the sampler from producing more formulas than needed, i.e., not only the ones which are strictly weaker (or stronger) that the learned lemmas (or invariance failures). Thus there is a risk to miss some invariants and to meet divergence. However, the intention of our synthesis procedure is to encourage exploring a wide range of unrelated samples. While having the risk to miss a "*next door*" invariant, we *aggressively* increase probabilities of "*far away*" candidates to being sampled. Our experiments confirm that such aggressive-pruning strategy in many cases accelerates the invariant discovery (see Sect. IV-B).

Note that in Alg. 2, we prioritize the search space after a tautology in a similar fashion to how we proceed after a learned lemma (but as in Alg. 1, we do not add it to *learnedLemmas*). This trick helps blocking some more tautologies from being sampled.

Example 4. *Let $C = \{-5, 0, 5\}$, and $ineq_3 \vee ineq_4 = x \geqslant 5 \vee x + y > 5$ be an invariance failure for P. The* $\text{PRIORITIZE}^{\vec{op}, \vec{c}}_{\downarrow}(\langle\vec{x}, \vec{k}\rangle)$ *function outputs two probability distributions p'_x and p'_{x+y}:*

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

□

Definition 9. *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 $\forall a \in A \,.\, p_m(p, p')(a) \overset{\text{def}}{=} \frac{min(p(a), p'(a))}{s}$.*

We write $\text{UPDATE}(priorMap_{\langle\vec{x},\vec{k}\rangle}, \{p_i\})$ to produce a distribution $p_m(priorMap_{\langle\vec{x},\vec{k}\rangle}(\{\vec{x}_i, \vec{k}_i\}), p_i)$ for each i and to store all of them in $priorMap_{\langle\vec{x},\vec{k}\rangle}$.

Example 5. *Given p_x and p_{x+y}, p'_x and p'_{x+y} obtained in Examples 3-4 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, and the condition of Def. 5 is violated.*

$x > 5 \mapsto 0$	$x + y > 5 \mapsto$ undefined
$x \geqslant 5 \mapsto 0$	$x + y \geqslant 5 \mapsto$ undefined
$x > 0 \mapsto {}^1\!/_2$	$x + y > 0 \mapsto$ undefined
$x \geqslant 0 \mapsto {}^1\!/_2$	$x + y \geqslant 0 \mapsto$ undefined
$x > -5 \mapsto 0$	$x + y > -5 \mapsto$ undefined
$x \geqslant -5 \mapsto 0$	$x + y \geqslant -5 \mapsto$ undefined

This way, since one of the two distributions at $priorMap_{\langle\vec{x},\vec{k}\rangle}$ 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.* □

E. Invariants over non-linear arithmetic

Our approach has a limited support for learning numerical invariants for programs with non-linear arithmetic computations. It naturally extends the idea of collecting and exploiting features of the program source code. In addition to populating sets N, K, and C from *Init*, *Tr*, and *Bad* (described in Sect. III-B), our algorithm populates set W by applications of either 1) the modulo operator, 2) the division operator, or 3) the multiplication operator, the list of arguments of which contains more than one variable.

The grammar in Fig. 3 enhances the sampling grammar with elements of W, which are treated as fresh variables. That is, the sampler may end up with candidates having elements of W, possibly multiplied by numeric constants and appeared in linear combinations.

$$\begin{aligned}
&\ldots \\
x ::= & x_1 \mid \ldots \mid x_n \mid x_i \texttt{ div } k_i \mid x_i \texttt{ mod } k_j \\
& \mid x_i \cdot x_j \mid x_i \texttt{ div } x_j \mid x_i \texttt{ mod } x_j \\
&\ldots
\end{aligned}$$

Figure 3: Non-linear sampling grammar (see Fig. 2 for the omissions).

```
int x = y = 0;
int z = *;
while (*) {
  x = x + z;
  y = y + 1;
}
assert(x == y * z);
```

Figure 4: Program with non-linear computations.

Example 6. *For program shown in Fig. 4, $Bad = (x = y \cdot z)$, thus $W = \{y \cdot z\}$. This lets our sampler generate candidates such as $-1 \cdot x + y \cdot z \geqslant 0$ and $x + -1 \cdot y \cdot z \geqslant 0$ which would pass the invariance check by a theory solver over non-linear arithmetic.* □

F. Further extensions

Extending frequencies with redundant clauses. Before populating the set of clauses A_V from *Init*, *Tr*, and *Bad*, these formulas could be enhanced by conjoining with some redundant clauses. A straightforward approach would consider pairs of conjuncts of *Init* (respectively, *Tr*, and *Bad*), infer a new clause and conjoin it back to *Init*. For instance, if $Init = (x = 0) \wedge (y = 0)$ then it implies $x = y$; and thus we can rewrite *Init* to be $(x = 0) \wedge (y = 0) \wedge (x = y)$. After the frequency distributions are created and used for sampling, the probability of getting formulas $-1 \cdot x + y \geqslant 0$ and $x + -1 \cdot y \geqslant 0$ increases. In practice, there could be many possible ways of inferring redundant clauses, and we leave the investigation of which way is the best for the future work.

More aggressive pruning. Besides of aggressive priority distributions, other tricks could be applied to shrink the search space. From a set of learned lemmas $\{ineq_1 \vee \ldots \vee ineq_n, \neg ineq_1, \ldots, \neg ineq_{n-1}\}$, it follows that formula $\neg ineq_n$ is not an inductive invariant. Thus, it could be withdrawn by the algorithm and not considered for sampling. Furthermore, the sets C and K might allow equivalent formulas (e.g., $x > 1$ and $2 \cdot x > 2$). The priority distributions could be nudged to block those as well.

Compensating aggressive pruning. One could introduce a "*reincarnating*" function, which turns distributions at $priorMap_{\langle\vec{x},\vec{k}\rangle}$ back to $uniform_{\langle\vec{x},\vec{k}\rangle}$ once they become undefined, or once each next lemma is learned.

IV. Implementation and Evaluation

We implemented the proposed approach in a tool FreqHorn[1]. It takes as input a verification task in a form of linear constrained Horn clauses. Despite we described the algorithm in a setting of single-loop programs, FreqHorn also supports multiple (possibly, nested) loops, but the risk of divergence due to the search space pruning in this case is higher.

A. Parallel architecture

FreqHorn is designed to benefit from asynchronous parallelism. The *master process* takes care of the preprocessing, sampling, learning, and prioritizing steps. It is equipped with an incremental SMT solver, called safeSolver which holds the conjunction of all learned lemmas till the end of the entire verification.

The most expensive computation at the FreqHorn workflow happens to be the invariance checking. Each sampled *Cand* requires a number of isolated SMT checks, performed by *worker processes*. Thus each worker is equipped with its own SMT solver, called invSolver, which examines if *Cand* is an inductive invariant. In particular, invSolver gets reset before each tautology check, initiation check, and consecution check. After all checks are done, a worker communicates its positive or negative result back to the master, and the worker becomes available for another candidate.

When the verification starts, n workers are available. The master samples n candidates in a row and sends one sample per worker. Since each sample requires unpredictable worker's time, the communication between the master and workers is asynchronous. After a learned lemma is received, the master re-checks safety. If the safety check failed (or an invariance failure is received), the master creates / nudges the priority distributions, samples a new candidate and sends it to the available worker. If the safety check succeeded, the verification is done.

B. Evaluation

Benchmarks. We evaluated FreqHorn on a set of 76 loopy programs, taken from various sources including SVCOMP[2], literature (e.g., [2], [7]) and crafted programs. The set contains 16 benchmarks over non-linear arithmetic (i.e., with $\cdot$, `div`, and `mod` operators).

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. All solvers were instantiated with Z3 [8]. Due to the stochastic nature of our learning, the FreqHorn timings are the means of 10 independent runs. We used a timeout of 10 mins for all runs.

[1]The source code and benchmarks are available at https://github.com/grigoryfedyukovich/aeval/tree/rnd-parallel-master-slave.

[2]Software Verification Competition, http://sv-comp.sosy-lab.org/

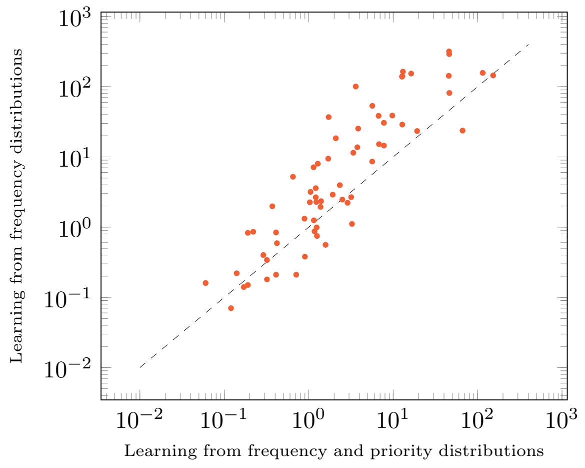

Figure 5: Effect of the search space pruning / prioritizing.

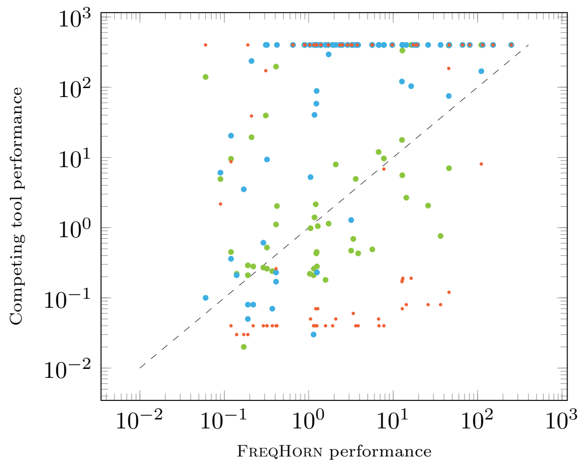

Figure 6: Comparison with μZ (red); ICE-DT (green); MCMC (blue).

Comparing FreqHorn's learning strategies. Fig. 5 shows a scatter plot comparing the average running times of FreqHorn with / without prioritizing the search space. Each point in the plot represents a pair of learning runs for the same benchmark: Alg. 2 (x-axis) and Alg. 1 (y-axis). Priority distributions accelerated the synthesis in 48 cases, and we witnessed a speedup up to 28%. As expected, there were six benchmarks, for which priority distributions helped finding an invariant (i.e., Alg. 1 diverged), but there were two another benchmarks, for which the search space was pruned too aggressively (i.e., Alg. 2 diverged).

Comparing with other tools. Fig. 6 shows a scatter plot comparing the timings of Alg. 2 and μZ v.4.5.0 [4], ICE-DT [7], and MCMC [9] invariant synthesizers. With four workers, FreqHorn outperformed μZ for 37 benchmarks (including 32 for which μZ crashed or reached timeout), ICE-DT for 53 (respectively, 30), and MCMC for 67 (respectively, 49).

Compared to μZ, FreqHorn solved all non-linear tasks and the tasks requiring large disjunctive invariants. Interestingly, FreqHorn was able to deliver a compact conjunctive representation of them. Compared to its closest competitor, MCMC (see Sect. V for more details), FreqHorn exhibited more consistent performance. We compared benchmarks for which both tools succeeded by their coefficients of variation (i.e., the ratio of the standard deviation to mean of the benchmark time): FreqHorn gets 0.60, and MCMC gets 0.92. Finally, FreqHorn for most benchmarks outperformed ICE-DT. This could be possibly explained by the reliance of the latter on the actual program executions, which are hard to get for non-deterministic programs.

V. 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 [9] 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 Sect. IV-B, it is not enough to guarantee consistent results. In [10], [11], [7], 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.

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 [12], [2], [3], [4] and abductive inference [13] depend crucially on the background theory of verification conditions which should admit interpolation and / or quantifier elimination. Those approaches were shown effective for propositional logic, linear arithmetic, and arrays. Our tool, in contrast, can discover non-linear invariants since it reduces the synthesis task to only quantifier-free queries and does not require interpolation.

Some prior work considered non-enumerative invariant inference from the source code. In Formula Slicing [14], a variant of Houdini [15], candidate invariants are chosen from the *Init* formulas (not from *Tr* or *Bad*, as in our case). In Niagara [16], [17] 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.

Syntax-guided approaches to synthesis [6] in general proceed by exploring the pre-determined (and adjusted for a particular problem) grammar. Recently, the "*Divide-and-Conquer*" [18] methodology, in which the problem is being solved in small pieces, has been successfully applied to SyGuS. Our approach has a similar underlying idea – to learn each lemma individually and conjoin it to the invariant – but the implementation via frequency and priority distributions is entirely new.

VI. Conclusion

We addressed the challenge of inductive invariant synthesis. Motivated by the observation that invariants can often be learned from the "*easy-to-get*" ingredients, we proposed to guide the learning process by frequency distributions, collected after a lightweight syntactic analysis of the source code, and to further prune / prioritize the search space. We built FreqHorn, the first tool that constructs the grammars for candidate invariants and distributions completely automatically, enables parallel verification of well-known programs over linear arithmetic, and to a limited extent supports non-linear arithmetic. In the future, we plan to investigate how deeper statistics about the code can help discovering more complicated inductive invariants.

References

[1] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, vol. 1855 of *LNCS*, pp. 154–169, Springer, 2000.

[2] A. R. Bradley, "Understanding IC3," in *SAT*, vol. 7317 of *LNCS*, pp. 1–14, Springer, 2012.

[3] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, pp. 125–134, IEEE, 2011.

[4] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, vol. 7317, pp. 157–171, Springer, 2012.

[5] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, pp. 404–415, ACM, 2006.

[6] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD*, pp. 1–17, IEEE, 2013.

[7] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *POPL*, pp. 499–512, ACM, 2016.

[8] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, vol. 4963 of *LNCS*, pp. 337–340, Springer, 2008.

[9] R. Sharma and A. Aiken, "From invariant checking to invariant inference using randomized search," in *CAV*, vol. 8559 of *LNCS*, pp. 88–105, Springer, 2014.

[10] S. Gulwani and N. Jojic, "Program verification as probabilistic inference," in *POPL*, pp. 277–289, ACM, 2007.

[11] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *CAV*, vol. 8559 of *LNCS*, pp. 69–87, Springer, 2014.

[12] A. R. Bradley and Z. Manna, "Property-directed incremental invariant generation," *Formal Asp. Comput.*, vol. 20, no. 4-5, pp. 379–405, 2008.

[13] I. Dillig, T. Dillig, B. Li, and K. L. McMillan, "Inductive invariant generation via abductive inference," in *OOPSLA*, pp. 443–456, ACM, 2013.

[14] E. G. Karpenkov and D. Monniaux, "Formula slicing: Inductive invariants from preconditions," in *HVC*, vol. 10028 of *LNCS*, pp. 169–185, Springer, 2016.

[15] C. Flanagan and K. R. M. Leino, "Houdini: an Annotation Assistant for ESC/Java," in *FME*, vol. 2021 of *LNCS*, pp. 500–517, Springer, 2001.

[16] G. Fedyukovich, A. Gurfinkel, and N. Sharygina, "Incremental verification of compiler optimizations," in *NFM*, vol. 8430 of *LNCS*, pp. 300–306, Springer, 2014.

[17] G. Fedyukovich, A. Gurfinkel, and N. Sharygina, "Property directed equivalence via abstract simulation," in *CAV*, vol. 9780, Part II, pp. 433–453, Springer, 2016.

[18] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling Enumerative Program Synthesis via Divide and Conquer," in *TACAS, Part I*, vol. 10205 of *LNCS*, pp. 319–336, 2017.