Accelerating Syntax-Guided Invariant Synthesis

Grigory Fedyukovich[1] and Rastislav Bodik[2]

[1] Princeton University, USA, grigoryf@cs.princeton.edu
[2] University of Washington, USA, bodik@cs.washington.edu

Abstract. We present a fast algorithm for syntax-guided synthesis of inductive invariants which combines enumerative learning with inductive-subset extraction, exploits counterexamples-to-induction and interpolation-based bounded proofs. It is a variant of a recently proposed probabilistic method, called FreqHorn, which is however less dependent on heuristics than its predecessor. We present an evaluation of the new algorithm on a large set of benchmarks and show that it exhibits a more predictable behavior than its predecessor, and it is competitive to the state-of-the-art invariant synthesizers based on Property Directed Reachability.

1 Introduction

Syntax-guided techniques [1] recently earned significant success in the field of synthesis of inductive invariants [12] for a given program. Invariants are needed to represent over-approximations of sets of reachable program states, such that from their empty intersection with sets of error states one could conclude that the program is safe. While searching for invariants, it is intuitive to collect various statistics from the syntactical constructions, which appear in the program's source code, and use them as a guidance.

This work continues the track of FreqHorn, a completely automatic approach for 1) construction of the formal grammar based on the symbolic program encoding, and 2) probabilistic search through formulas belonging to that grammar. FreqHorn utilizes an SMT solver for checking inductiveness of each generated formula and iteratively constructs a suitable invariant based on the successful attempts (those formulas are called lemmas). Since based on finite number of expressions, the formal grammar is sufficiently small, which enables a relatively quick enumeration of 1) formulas directly extracted from the program's encoding (called seeds) and 2) their slight mutations.

The actual novelty of the approach is believed to be in combination of seeds and mutants, but the original paper [12] does not provide a witness for it. Furthermore, it turns a blind eye to some algorithmic and practical details which are required for making the approach actually efficient. Among the downsides are 1) the treatment of all syntactic expressions equally and ignorance to whether the candidates have any semantic value; and 2) inability to predict a more-or-less appropriate order of candidates to be sampled and checked.

Luckily, elements of the Property Directed Reachability (PDR) [4,9] can be adapted in various stages of FreqHorn's workflow and can mitigate the downsides of the original algorithm. In particular, we propose to check candidates in

batches, and we show that in practice it helps discovering larger amounts of lemmas. Additionally, we propose to keep a history of counterexamples-to-induction (CTI) which blocked FreqHorn from learning a lemma. With some periodicity, our new algorithm checks if there is a CTI which is invalidated by the currently learned lemmas, and this triggers the re-check of that failed lemma.

Last but not least, we integrate our new algorithm with the classic techniques based on Bounded Model Checking [3]. We propose to compute additional candidates by Craig interpolation [6] from proofs of bounded safety. We show that it is often sufficient to obtain some fixed amount of candidates from interpolants in the beginning of the synthesis process, and further to *bootstrap* the initial set of learned lemmas by the inductive subset extracted from the combination of the syntactic seeds and interpolants. In contrast to the entirely randomized workflow of the original version of FreqHorn, the behavior of our revised implementation at the bootstrapping is predictable. The randomized search is used by the new algorithm only for discovering mutants; and in our experiments, it was required in about one third of cases only.

To sum up, the paper contributes to the previous knowledge in the following main respects:

- New revision and new implementation of the FreqHorn algorithm which is split into the bootstrapping and the sampling stages. In the first stage, it deterministically exploits the seeds only. In the second stage, it keeps generating and checking only the mutants, and it is by design nondeterministic.
- In the bootstrapping stage, interpolation-based proofs of bounded safety that replenish the set of seeds by the candidates which likely reflect the nature of the error unreachability.
- In the sampling stage, the routine to extract inductive subsets which mitigates the effect of an unpredictably chosen sampling order.
- A more accurate strategy for the search space pruning and the efficient counterexample-guided method to give some failed candidates a second chance.

The rest of the paper is structured as follows. In Sect. 2, we briefly formulate the inductive synthesis problem, and in Sect 3 we sketch the basic FreqHorn algorithm that attempts to solve it. With the help of techniques from Sect. 4, in Sect. 5 the FreqHorn algorithm gets augmented and reformulated. In Sect. 6, we show the experimental evidence that it indeed outperforms its predecessor and is competitive to state-of-the-art. Finally, the future and related work, conclusion, and acknowledgments complete the paper in Sects. 7-9.

2 Background and Notation

Let $\mathcal{X}$ be a set of variables, $\mathcal{F}$ be a set of *function symbols*, and $\mathcal{P}$ be a set of *predicate symbols*, a union $\mathcal{X} \cup \mathcal{F}$ as a *signature* Σ. Consider a first-order language with equality (i.e., "$=$"$\in \mathcal{P}$) and signature Σ. A *Σ-structure* S consists of a *domain of interpretation*, denoted as $|S|$, and an *interpretation function* that assigns elements of $|S|$ to variables, and functions and predicates on $|S|$ to the

symbols of Σ. Given a Σ-structure S, a Σ-assignment s is a function mapping each Σ-term to an element in $|S|$. Given a formula φ in the first-order language, we call φ satisfied by a Σ-structure S, if there exists a Σ-assignment m (called *model*) under which φ evaluates to $\top$ (denoted $m \models \varphi$).

A first-order theory $\mathcal{T}$ consists of a signature Σ and a set of Σ-sentences $\mathcal{S}$. Let a set of Σ-formulas be denoted *Expr*; $\varphi \in \mathit{Expr}$ is called $\mathcal{T}$-satisfiable is there exists a Σ-structure $S \in \mathcal{S}$ such that φ satisfied by S; and φ is called $\mathcal{T}$-valid if $\neg\varphi$ is $\mathcal{T}$-unsatisfiable (denoted $\varphi \implies \bot$). The Satisfiability Modulo Theory (SMT) problem [7] for a given theory $\mathcal{T}$ and a quantifier-free formula φ aims at determining whether φ is $\mathcal{T}$-satisfiable. In this work, we formulate the tasks arising in program verification by encoding them to the SMT problems.

Definition 1. *A* transition system *P is a tuple $\langle V \cup V', \mathit{Init}, \mathit{Tr}\rangle$, where V' is a primed copy of a set of variables V; Init and Tr are $\mathcal{T}$-encodings of respectively the* initial states *and the* transition relation.

We view *programs* as *transition systems* and throughout the paper use both terms interchangeably. *Verification task* is a pair $\langle P, \mathit{Bad}\rangle$, where $P = \langle V \cup V', \mathit{Init}, \mathit{Tr}\rangle$ is a program, and *Bad* is a $\mathcal{T}$-encoding 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 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 $P = \langle V \cup V', \mathit{Init}, \mathit{Tr}\rangle$; a formula Inv is a* safe invariant *if the following conditions (respectively called initiation, consecution, and safety) hold:*

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

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

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

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

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 a model m satisfying $\mathit{Inv}(V) \land \mathit{Tr}(V, V') \land \neg\mathit{Inv}(V')$. Model m is called *counterexample-to-induction* (CTI).

Example 1. The loop in program in Fig. 1a iterates `N` times, and in each iteration it nondeterministically picks a value `M`, adds it to `x` (conditionally) and to `c`, and assigns the sum of `x` and `c` to `k`. We wish to prove that after the loop terminates, `x` $\geq$ `N`. An invariant for the program is defined non-uniquely, e.g., both the conjunction ($k \ \mathtt{mod}\ 2 = 0 \land x = c$) and conjunction ($k = x + c \land x \geq c$) are the solutions for this verification task. $\square$

```
int x, k, c = 0;
int N = NONDET();
while (c < N) {
  int M = NONDET();
  if (k mod 2 == 0)
    x = x + M;
  c = c + M;
  k = x + c;
}
assert (x ≥ N);
```

(a)

$x = 0$
$c = 0$
$k = 0$
$c < N$
$k \texttt{ mod } 2 = 0$
$k = x + c$
$c \geq N$
$x \geq N$

(b)

$$\alpha ::= 1 \mid -1$$
$$\beta ::= 0 \mid 2$$
$$\gamma ::= x \mid y \mid k \mid N$$
$$\delta ::= \alpha \cdot \gamma + \ldots + \alpha \cdot \gamma \mid \gamma \texttt{ mod } \beta$$
$$cnd ::= \delta = \beta \mid \delta > \beta \mid \delta \geq \beta$$

(c)

$$\begin{aligned} x = 0 \wedge k = 0 \wedge c = 0 &\implies \boldsymbol{Inv}(x, k, c, N) \\ \boldsymbol{Inv}(x, k, c, N) \wedge c < N \wedge x' = \texttt{ite } (k \texttt{ mod } 2 = 0, x + M, x) \wedge & \\ c' = c + M \wedge k' = x' + c' &\implies \boldsymbol{Inv}(x', k', c', N) \\ \boldsymbol{Inv}(x, k, c, N) \wedge c \geq N \wedge \neg(x \geq N) &\implies \bot \end{aligned}$$

(d)

Fig. 1: Loopy program **(a)**, its encoding **(d)**, subexpressions extracted from the encoding **(b)**, and grammar that generalizes the subexpressions **(c)**.

3 Syntax-Guided Invariant Synthesis

In this work, we aim at discovering invariants in an enumerative way, i.e., by guessing a candidate formula, substituting it for conditions (1), (2), and (3), and checking their validity. Here we present a moderately reformulated and simplified view of an algorithm recently proposed in [12].[3] The pseudocode of the algorithm, called FreqHorn, is shown in Alg. 1. The key insight behind the algorithm is the automatic construction of the grammar G based on the fixed set of expressions obtained by traversing parse trees of *Init*, *Tr*, and *Bad* (line 2). The algorithm further uses G for generating the candidates (line 5) and populates the set of lemmas until their conjunction is an invariant. The algorithm learns from each positive an negative attempt (line 8). That is, the sampling grammar gets adjusted, such that the candidate (and some of its close relatives) is not going to be considered in any of the following iterations.

[3] The original description [12] focuses on the probabilistic routines. In the interest of this work, we do not discuss them here but restrict our attention on describing and exemplifying the pre-processing steps.

Algorithm 1: FREQHORN: Sampling inductive invariants, **cf.** [12].

Input: $\langle P, Bad\rangle$: verification task, where $P = \langle V \cup V', Init, Tr\rangle$
Output: $Lemmas \subseteq 2^{Expr}$

1 $Seeds \leftarrow$ GETSUBEXPRS($Init$, Tr, Bad);
2 $G \leftarrow$ GETGRAMMARANDDISTRIBUTIONS($Seeds$);
3 $Lemmas \leftarrow \varnothing$;
4 **while** $Bad(V) \wedge \bigwedge_{\ell \in Lemmas} \ell(V) \not\Longrightarrow \bot$ **do**
5 $\quad cnd \leftarrow$ SAMPLE(G);
6 $\quad res \leftarrow \big(Init(V) \implies cnd(V)\big) \wedge \big(cnd(V) \wedge \bigwedge_{\ell \in Lemmas} \ell(V) \wedge Tr(V, V') \implies cnd(V')\big)$;
7 $\quad$ **if** res **then** $Lemmas \leftarrow Lemmas \cup \{cnd\}$;
8 $\quad G \leftarrow$ ADJUST(G, cnd, res);

Example 2. The verification condition for the program in Fig. 1a is represented by three implications in Fig. 1d. All the implications are syntactically split into a set *Seeds* of equalities and inequalities (or disjunctions of equalities and inequalities if there exist some) such that there are either only primed or only unprimed variables (shown in Fig. 1b). In particular, equalities $x' = x + M$ and $c' = c + M$ are excluded from *Seeds*. The grammar containing all the subexpressions, in which all primed variables are replaced by the corresponding unprimed ones, is shown in Fig. 1c. It easy to see that all lemmas consisting in both invariants $(k \texttt{ mod } 2 = 0 \wedge x = c)$ and $(k = x + c \wedge x \geq c)$ can be generated by recursively applying the grammar's production rules. □

Definition 3. *Each formula containing in set Seeds, which is used for constructing grammar G (in line 2), is called* seed. *Formula cnd produced by G is called* mutant *if $cnd \notin Seeds$.*

The downside of Alg. 1 is that it is hard to choose a sampling order for each individual lemma at the final invariant. Suppose, $cnd = (x = c)$ is sampled and checked in the first iteration of Alg. 1. Consequently, condition (2) is not fulfilled, and it is witnessed by the following CTI: $[x \leftarrow 0; k \leftarrow 1; c \leftarrow 0; N \leftarrow 10; x' \leftarrow 0; k' \leftarrow 7; c \leftarrow 7; M \leftarrow 7]$. Suppose in the second iteration of Alg. 1, $cnd = (k \texttt{ mod } 2 = 0)$. It passes checks (1) and (2), gets inserted to set *Lemmas*, and thus it is going to be taken into account in the following iterations (see implications in lines 4 and 6). Note that if in the third iteration $cnd = (x = c)$ was sampled again, the algorithm would terminate. However, it is impossible since the sampling grammar was adjusted after both negative and positive attempts, i.e., in the first and the second iteration respectively.

The opposite sampling order (i.e., $cnd = (k \texttt{ mod } 2 = 0)$ first, and then $cnd = (x = c)$) would lead to a faster convergence of the algorithm. Since it is hard to decide which order to choose, in [12], we equipped the grammar's production rules with probability distributions, and thus allowed both orders

under certain probability. In this paper, we propose a strategy which is less dependent on an order – to check candidates in batches – and we describe it in Sect. 5 in more detail.

4 Old Friends Are Best

In this section, we rehash two ideas widely used in symbolic model checking that can be adapted to accelerate syntax-guided invariant synthesis.

4.1 Interpolation-based proofs of bounded safety

Bounded Model Checking (BMC) [3] is a formal technique, primarily used for bug finding. Given a transition system $\langle V \cup V', \mathit{Init}, \mathit{Tr} \rangle$, set of error states Bad, and a non-negative integer number k, the *BMC task* is to check if there exists a path of length k ending in an error state. The idea is to unroll Tr k times, conjoin it with Init and with the negation of Bad, and to check satisfiability of the resulting formula (called BMC formula):

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

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 1. *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. Extraction of the proofs is typically done with the help of Craig interpolation [6].

Definition 4. *Given two formulas A and B, such that $A \wedge B \implies \bot$, an interpolant of 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 $\{\ell_i\} \leftarrow \textsc{getItp}(A, B)$. Alg. 2 shows an algorithm to generate interpolation-based proofs of bounded safety for BMC formulas. It iteratively unrolls the transition relation and applies interpolation for the entire BMC formula. In addition, in spirit of Lazy Annotation [24], while decrementing i, the algorithm applies backward reasoning and checks if an error state is reachable by $(k - i)$ steps from an empty state (line 4). It triggers interpolation to be applied over smaller formulas, and in some cases fastens the proof search (line 5).

Algorithm 2: BMCITP: Obtaining bounded proofs, **cf.** [23,24].

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 $\quad unr \leftarrow unr \wedge Tr(V^{(i-1)}, V^{(i)})$;
4 $\quad$ **if** $unr \wedge \neg Bad(V^{(k)}) \implies \bot$ **then**
5 $\quad\quad proof \leftarrow$ GETITP$(unr, Bad(V^{(k)}))$;
6 $\quad\quad$ **return**;
7 $unr \leftarrow unr \wedge Bad(V^{(k)})$;
8 **if** $Init(V^{(0)}) \wedge unr \implies \bot$ **then** $proof \leftarrow$ GETITP$(Init(V^{(0)}), unr)$;

Algorithm 3: HOUDINI: Calculating an inductive subset, **cf.** [13] and keeping counterexamples-to-induction.

Input: $P = \langle V \cup V', Init, Tr\rangle$: program; $Cnds \subseteq 2^{Expr}$;
$CTI \subseteq 2^{V \to \mathbb{R}}$; $CTImap$: $CTI \to 2^{Expr}$
Output: inductive $Cnds \subseteq 2^{Expr}$; updated $CTImap$

1 **while** $\bigwedge_{cnd' \in Cnds} cnd'(V) \wedge Tr(V, V') \not\Longrightarrow \bigwedge_{cnd' \in Cnds} cnd'(V')$ **do**
2 $\quad$ **for** $cnd \in Cnds$ **do**
3 $\quad\quad$ **if** $\exists \pi$, s.t. $\pi \models \big(\bigwedge_{cnd' \in Cnds} cnd'(V) \wedge Tr(V, V') \wedge \neg cnd(V')\big)$ **then**
4 $\quad\quad\quad Cnds \leftarrow Cnds \setminus \{cnd\}$;
5 $\quad\quad\quad CTI \leftarrow CTI \cup \{\pi\big|_V\}$;
6 $\quad\quad\quad CTImap(\pi\big|_V) \leftarrow CTImap(\pi\big|_V) \cup cnd$;

Example 3. Let the program in Fig. 1a is unrolled 0 times, then its BMC formula is constructed as follows: $\underbrace{x = 0 \wedge k = 0 \wedge c = 0}_{Init} \wedge \underbrace{c \geq N \wedge \neg(x \geq N)}_{Bad}$. It is unsatisfiable, and since interpolants are not unique, function GETITP$(Init, Bad)$ could return $proof_1 = \{x \geq 0, c \leq 0\}$, $proof_2 = \{x = c\}$, or $proof_3 = \{x \geq c\}$. □

4.2 Inductive subset extraction

When checking inductiveness of a set of candidate formulas "*one-by-one*" (i.e., like in Alg. 1), the order of checks is crucial, and the chance to miss some important lemma is high. It can be overcome by checking all candidate formulas at once, identifying which ones brake validity of implication (2), removing them from the set, and repeating the "*all-at-once*" check. Alg. 3 shows a simple implementation of this iterative algorithm, also known as HOUDINI [13].

An important prerequisite for the algorithm is that all formulas should be consistent with each other. In practice, HOUDINI applies to a preprocessed set of formulas, such that condition (1) holds for each of them. It is trivial to derive that the prerequisite for the algorithm is fulfilled in this case.

Lemma 2. *Given a set of formulas Cnds, if for each $cnd \in Cnds$ it holds that $Init(V) \implies cnd(V)$ then $\bigwedge_{cnd \in Cnds} cnd(V)$ is satisfiable.*

Example 4. Conjunction of formulas from set *Seeds* in Fig. 1b is unsatisfiable, and its minimal unsatisfiable core is $c < N \land c \geq N$. Thus, Alg. 3 would immediately return the entire set *Seeds*. Let set *Cnds* is constructed from *Seeds* by removing all elements, for which condition (1) does not hold. Conjunction of the elements in *Cnds* is satisfiable: $\{x = 0, c = 0, k = 0, k \texttt{ mod } 2 = 0, k = x + c\}$. Applying Alg. 3 to *Cnds* gives the inductive subset $\{k \texttt{ mod } 2 = 0, k = x+c\}$. □

Note that we extended Alg. 3 with a routine to extract a counterexample-to-induction $m \in CTI$ for each element dropped from *Cnds* (lines 3-6). We restrict each m to only the assignments to unprimed variables (i.e., from V) and group all non-inductive formulas from *Cnds* by the particular m that killed them. This routine is important for optimizing the syntax-guided invariant synthesis algorithm, and it is discussed in more detail in Sect. 5.

5 Reconsidering Syntax-Guided Invariant Synthesis

The lesson we learned when running the FreqHorn algorithm is that the program encoding gives many hints on how the shape of lemmas should look like. However, the encoding itself can barely give any information about the sampling order. Our main idea to revise the FreqHorn algorithm is to treat the seeds and mutants separately. Indeed, as we saw in Ex. 2, both seeds and mutants are needed for constructing an invariant, but the seeds do *not* actually need to be re-sampled – these candidates are ready to be checked prior to any sampling.

We present a new revision of the FreqHorn algorithm which is split into two main stages, the *bootstrapping* and the *sampling*. In the first stage, it exploits only the seeds. The idea is to terminate this stage as quickly as possible and to populate the set of lemmas with (preferably, the maximal) inductive subset of seeds. If this subset is not enough for an invariant, the algorithm should proceed to the next stage, in which it should keep generating and checking only the mutants.

The pseudocode of the new FreqHorn's revision is shown in Alg. 4. In the bootstrapping, the algorithm relies on Alg. 2 to replenish the set of seeds by semantically-meaningful candidates, and in the sampling stage, it relies on Alg. 3 to mitigate the effect of an unpredictably chosen sampling order. Another algorithmic advantage against Alg. 1 lies also in the more accurate strategy for the search space pruning and the efficient counterexample-guided method to give some failed candidates a *second chance*.

The algorithm takes as input a verification task and values of important configuration parameters N, M, and K (to be explained further). Like Alg. 1, it starts with obtaining a set of expressions *Seeds* from *Init*, *Tr*, and *Bad* (line 1). Then, *Seeds* gets merged with sets of formulas obtained by Craig interpolation from proofs of bounded safety for a range of bounds $0 \ldots N$. Note that if there

Algorithm 4: 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: $Lemmas \subseteq 2^{Expr}$

```
1  Seeds ← getSubExprs(Init, Tr, Bad);
2  for (k ← 0; k < N; k ← k + 1) do
3      proof ← bmcItp(⟨P, Bad⟩, k);
4      if proof = ∅ then return;
5      else Seeds ← Seeds ∪ proof;
6  G ← getGrammarAndDistributions(Seeds);
7  CTI, CTImap ← ∅;
8  #learned ← 0;
9  Cnds ← Seeds;
10 while Bad(V) ∧ ⋀_{ℓ∈Lemmas} ℓ(V) ⇏ ⊥ do
11     while |Cnds| < M do Cnds ← Cnds ∪ {sample(G)};
12     for cnd ∈ Cnds do
13         if (Init(V) ⇏ cnd(V)) then
14             G ← adjust(G, cnd, false);
15             Cnds ← Cnds \ {cnd}
16     Lemmas′, CTI, CTImap ← Houdini(P, Cnds ∪ Lemmas, CTI, CTImap);
17     for ℓ ∈ Lemmas′ \ Lemmas do G ← adjust(G, ℓ, true);
18     NewLemmas = {ℓ′ | ℓ′ ∈ Lemmas′, s.t. ⋀_{ℓ∈Lemmas} ⇏ ℓ′};
19     #learned ← #learned + |NewLemmas|;
20     Lemmas = Lemmas ∪ NewLemmas;
21     Cnds ← ∅;
22     if #learned > K then
23         #learned ← 0;
24         for m ∈ CTI do
25             if m ⊭ ⋀_{ℓ∈Lemmas} ℓ(V) then
26                 CTI ← CTI \ m;
27                 Cnds ← Cnds ∪ CTImap(m);
```

is a counterexample of length $k < N$ discoverable by the BMC engine then an invariant does not exist (recall Lemma 1), and Alg. 4 terminates (line 4).

The bootstrapping ends when the merged set *Seeds* is taken as input by Alg. 3, and it extracts an inductive subset (line 16). However, prior to it, the lemma-consistency prerequisite of Alg. 3 should be fulfilled. Thus, the algorithm checks the initiation condition for all elements of the merged set (recall Lemma 2), and the set is filtered accordingly (lines 12-15).

Example 5. Let set *Seeds* be as in Fig. 1b, and set *Cnds* is constructed from *Seeds* by removing all elements, for which condition (1) does not hold. Assume

that a proof of bounded safety for $k = 0$ is $\{x = c\}$ (as one of the options in Ex. 3). Applying Alg. 3 to $Cnds \cup \{x = c\}$, we get the inductive subset $\{k \texttt{ mod } 2 = 0, k = x + c, x = c\}$. Since the conjunction of these lemmas is a invariant, and the algorithm terminates just after the bootstrapping. □

Checking the candidate formulas in batches is an important improvement over Alg. 1. This way, the algorithm becomes less dependent of the heuristics for prioritizing the search-space traversal. The size of the batch M is configurable, and if the size of set $Cnds$ is less than M, then the set gets additional mutants (line 11). Mutants are sampled from the grammar, which is powered by both, the program's encoding (similar to Alg. 1) and the proofs of bounded safety (new in Alg. 4). This enlarges the search space for the further mutants.

If the initial batch of candidates still misses some lemmas necessary for an invariant, then Alg. 4 proceeds to a new iteration. In particular, the extracted inductive subset gets merged with the set of lemmas (line 20), and the assembly of a new batch of candidates starts from scratch (line 21).

Example 6. Assume that a proof of bounded safety is $proof_1 = \{x \geq 0, c \leq 0\}$ (as in Ex. 3). However, the initiation condition is fulfilled for none of the elements of $proof_1$, so none of them contains in the set of formulas $Cnds$ taken as input by Alg. 3. Thus, $proof_1$ does not bring any additional value to the set of seeds, and (contrary to the case in Ex. 5) the algorithm does not terminate after the bootstrapping. Instead, it proceeds to sampling fresh mutants. □

A substantial distinction between the FreqHorn's revisions is how they react to the positive and negative attempts. In Alg. 1, the search space gets adjusted after each individual check. The grammar adjustments are performed by changing the probabilities assigned to the production rules. In addition to zeroing the probability of sampling a candidate *cnd* itself, after each positive check, Alg. 1 zeroes the probabilities of sampling some formulas which are weaker than *cnd*, and after each negative check – the probabilities of sampling some formulas which are stronger than *cnd* (see [12] for more details).

In contrast, Alg. 4 reacts just to the failed candidates after the initiation check (line 14) and the successful candidates after the inductiveness check (line 17). Otherwise, if the inductiveness check failed for a candidate *cnd* (inside Alg. 3), Alg. 4 does not disqualify *cnd* from being checked again in the future, and this is done by keeping *cnd* locally and periodically seeking an opportunity to give *cnd* a *second chance*.

To efficiently exploit the second-chance candidates, we rely on the extension of Alg. 3 by the routine to extract counterexamples-to-induction. That is, for each failed *cnd* there exists $m \in CTI$ that killed it. To maintain this information, every application of Alg. 3 updates the map $CTImap$ between CTI and failed candidates. In Alg. 4, it remains to periodically check whether some m is eliminated (line 25), and it would increase chances of all candidates killed by m (line 27) to succeed the inductiveness check in the next iteration. On the other hand, if some m still models the conjunction of learned lemmas then it

is guaranteed that candidates in $CTImap(m)$ will fail the inductiveness check again.

Finally, to make sure the CTI-check happens not too often, we run it only when at least K new lemmas are learned. For this, Alg. 4 has a redundancy check (line 18): a new lemma ℓ gets learned only when the conjunction of all lemmas does not imply ℓ. Obviously, when no new lemmas (after the redundancy check) are added, it does make sense to run the CTI-check as all CTIs are still valid.

Theorem 1. *If Alg. 4 terminates, then either an actual bug is found, or an invariant is synthesized.*

5.1 Optimizations

The following tricks are omitted from the algorithm's pseudocode to simplify reading, but they are important for the efficiency of our implementation.

- As a consequence of calculating frequencies, in the original FreqHorn algorithm, the seeds were given priorities, but the mutants were considered with a relatively small probability. In contrast, the new FreqHorn's revision forces the seeds to be checked in the bootstrapping. So while doing sampling, it gives priorities to mutants, and for that it ignores frequencies.
- The initiation checks (lines 12-15) for proofs of bounded safety are omitted since by definition of interpolant (Def. 4) they are already fulfilled. The initiation checks for the second-chance candidates are omitted as well.
- In case a candidate fails the inductiveness check, and it is queued for a second chance, it is still possible that Alg. 4 samples it again in the next iterations. Re-sampling is avoided by additional adjustments of the probabilities for the sampling grammar in line 6 of Alg. 3.
- Alg. 3 could be optimized if solved *with assumptions*. However, in our experience, it may lead to dropping more candidates than needed. Ideas for getting a maximal inductive subset from [22] could be applied here as well.
- For getting proofs of bounded safety for various bounds, an incremental SMT solver could be used. That is, it could reuse parts of BMC formulas for bound k to encode a BMC formula for bound $k+1$. Potentially, other tricks (e.g., [31,5]) could also be applied here.

6 Evaluation

We implemented FreqHorn-2 on top of our prior implementation Freq-Horn[4]. The tool takes as input a verification task in a form of linear 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.

[4] The source code and benchmarks are available at https://github.com/grigoryfedyukovich/aeval/tree/rnd.

We evaluated FreqHorn-2 on various safe and buggy programs taken SV-COMP[5] and literature (e.g., [8,14]). Since most of benchmarks, proposed by [8], appeared to be solvable during the bootstrapping of FreqHorn-2 (more details in Sect. 6.1) within (fractions of) seconds, we crafted additional harder benchmarks by ourselves.

All the programs were encoded using the theories of linear and nonlinear integer arithmetic. 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.

6.1 The bootstrapping experiment

In total, we considered **151** safe programs. For **90** of them, the seeds, generated by breaking the symbolic encoding to pieces, did already contain all lemmas needed for invariants. However, when we checked the seeds one-by-one, we revealed invariants for only 81, but using the inductive subset extraction helped revealing all 90. Each set of seeds contained in average 9 formulas. Non-surprisingly, an average run of Alg. 3 for these sets took 1.18 sec, while an average run of a naive one-by-one checker took 0.47 sec.

For our BMC implementation, we considered bounds 1, 2, and 3. Generated interpolants already contained all lemmas for invariants for **65** programs. Each set of proofs contained in average 2 formulas. In all these cases, the output of Alg. 2 was taken as input by Alg. 3, and the final safety check was run afterwards. An average run of Alg. 2 together with Alg. 3 took 0.72 sec.

Our most promising results were achieved while running Alg. 3 for the merged sets of seeds and proofs of bounded safety (i.e., both sets as in the two prior runs together). The merged sets already contained all lemmas for invariants for **106** programs. For two of them, Alg. 3 was unable to discover an invariant if given only the set of seeds or the set of proofs. An average run of both Alg. 2 and Alg. 3 took 1.4 sec. For comparison, out of these 106 benchmarks, the original version of FreqHorn exceeded the timeout for 9 ones, and an average run for the remaining 97 benchmarks took 4.7 sec.

This experiment lets us to conclude that *the bootstrapping is exceptionally important* for accelerating syntax-guided invariant synthesis. In contrast to FreqHorn's fully randomized workflow, FreqHorn-2's behavior at the bootstrapping is predictable. FreqHorn-2 uses the randomized search only to discover mutants, and in our experiments, it was required only in 45 out of 151 cases.

6.2 Overall statistics

Fig. 2 shows three scatter plots comparing average running times of FreqHorn-2 vs FreqHorn, μZ [16], and Spacer3 [21] respectively. Both μZ and Spacer3 are PDR-based, and despite the latter is faster than the former and can solve more benchmarks, there are 14 instances, for which the former outperforms the latter. We decided not to compare FreqHorn-2 with data-driven tools [29,14] since, in our experience [12], FreqHorn already outperforms them.

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

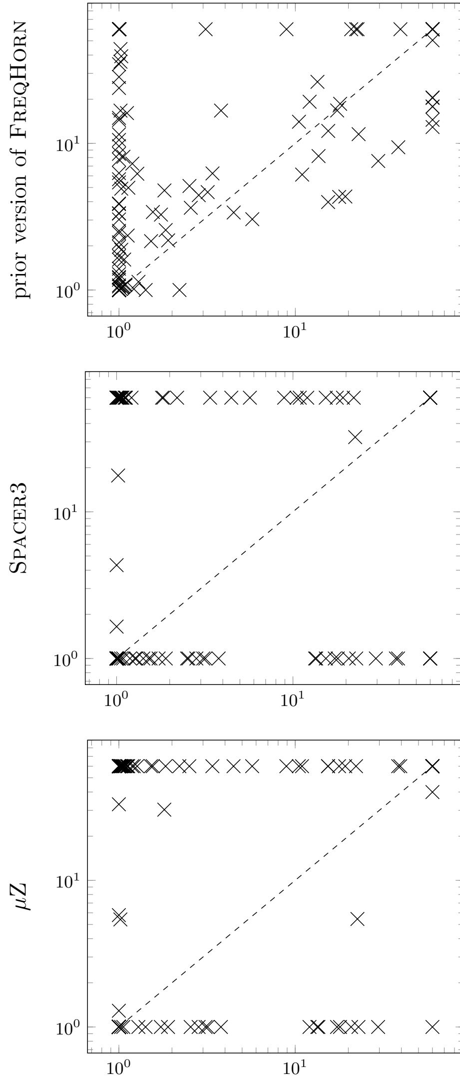

Fig. 2: FreqHorn-2 vs competitors.

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

Fig. 3: `count_by_2`: a bottleneck.

Benchmark	FreqHorn-2	FreqHorn	μZ	Spacer
abdu_01	ϵ	1.23	∞	ϵ
abdu_02	ϵ	28.37	∞	ϵ
abdu_03	ϵ	34.67	∞	ϵ
abdu_04	ϵ	3.85	∞	ϵ
bhmr2007	ϵ	∞	∞	ϵ
bouncy_three	1.73	3.27	ϵ	ϵ
bouncy_two	4.47	3.38	∞	∞
cggmp_iter_1	2.83	4.43	ϵ	ϵ
cggmp_iter_2	13.35	26.27	ϵ	ϵ
cggmp_iter_3	20.73	∞	ϵ	ϵ
const_div_1	ϵ	1.40	∞	ϵ
const_div_2	ϵ	ϵ	∞	ϵ
const_mod_3	1.06	8.10	ϵ	ϵ
count_by_2	∞	∞	39.97	∞
countud	2.51	5.12	∞	ϵ
css2003	ϵ	10.55	∞	ϵ
dillig02	1.02	43.95	∞	ϵ
dillig03	ϵ	ϵ	ϵ	∞
dillig05	1.84	2.56	∞	∞
dillig07	ϵ	2.53	∞	ϵ
dillig08	1.52	2.15	∞	ϵ
dillig10	2.55	3.64	ϵ	ϵ
dillig13	∞	∞	∞	ϵ
dillig14	ϵ	5.41	∞	ϵ
dillig15	1.56	3.40	∞	ϵ
dillig16	15.37	12.15	∞	ϵ
dillig18	8.91	∞	∞	∞
dillig20-3	29.52	7.58	ϵ	ϵ
dillig21	ϵ	14.91	ϵ	1.65
dillig22-1	ϵ	1.21	∞	ϵ
dillig22-2	1.27	6.21	∞	ϵ
dillig22-3	1.02	16.72	∞	ϵ
dillig22-4	ϵ	14.43	33	ϵ
dillig22-6	∞	20.55	∞	ϵ
dillig22	1.19	7.27	∞	ϵ
dillig41	ϵ	ϵ	ϵ	∞
dillig42	3.10	∞	ϵ	ϵ
dillig44	1.03	39.26	∞	ϵ
exact_iters_1	17.95	18.61	ϵ	ϵ
exact_iters_2	10.93	6.12	∞	∞
exact_iters_3	10.49	14.04	∞	∞
exact_iters_4	∞	20.41	ϵ	ϵ
exact_iters_5	∞	14.48	∞	∞
formula25	1.90	2.18	ϵ	ϵ
formula27	3.79	16.73	ϵ	ϵ
gcd_2	3.18	4.65	ϵ	ϵ
gj2007	∞	50.46	∞	ϵ
half_true_modif	∞	∞	∞	ϵ
hhk2008	ϵ	9.57	∞	ϵ
nonlin_div	22.12	∞	∞	∞
nonlin_factorial	ϵ	∞	∞	∞
nonlin_minus_1	ϵ	ϵ	∞	∞
nonlin_minus_2	ϵ	1.88	∞	∞
nonlin_mod_1	5.70	3.04	∞	∞
nonlin_mod_mult	ϵ	ϵ	∞	∞
nonlin_mult_1	ϵ	ϵ	∞	∞
nonlin_mult_2	ϵ	2.35	∞	∞
nonlin_mult_3	ϵ	4.98	∞	∞
nonlin_mult_4	ϵ	ϵ	∞	∞
nonlin_mult_5	∞	17.81	∞	∞
nonlin_mult_6	ϵ	1.61	∞	∞
nonlin_square	ϵ	ϵ	∞	∞
nonterm_01	ϵ	ϵ	∞	ϵ
phases	∞	17.81	∞	∞
recur_1	ϵ	ϵ	∞	ϵ
recur_102	1.29	1.14	ϵ	ϵ
s_seeds_05	ϵ	ϵ	∞	ϵ
s_seeds_06	ϵ	ϵ	ϵ	∞
s_mutants_01	ϵ	ϵ	ϵ	∞
s_mutants_02	ϵ	1.18	ϵ	∞
s_mutants_03	ϵ	ϵ	ϵ	∞
s_mutants_05	2.20	ϵ	∞	∞
s_mutants_06	ϵ	ϵ	ϵ	∞
s_mutants_07	1.42	ϵ	ϵ	ϵ
s_mutants_11	ϵ	1.08	5.79	ϵ
s_mutants_16	19.22	4.34	∞	∞
s_mutants_17	15.34	3.98	∞	∞
s_mutants_18	39.44	∞	∞	ϵ
s_mutants_20	ϵ	16.09	∞	∞
s_mutants_21	∞	12.92	∞	∞
s_mutants_22	17.72	4.31	∞	∞
s_mutants_23	1.22	1.02	∞	∞
s_disj_ite_01	17.26	16.76	ϵ	ϵ
s_disj_ite_02	13.54	8.19	ϵ	ϵ
s_disj_ite_03	22.84	11.56	ϵ	ϵ
s_disj_ite_04	12.05	19.22	ϵ	∞
s_disj_ite_05	22.53	∞	5.45	32.32
s_disj_ite_06	38.37	9.41	∞	ϵ
sn_1024	ϵ	ϵ	1.29	4.33
sn_2048	1.02	36.04	5.41	17.68
sn_4096	1.81	4.77	30.32	∞
sn_8192	3.39	6.24	∞	∞
trex3	1.03	4.91	ϵ	ϵ

Fig. 4: Exact timings.

Each point in a plot represents a pair of the FreqHorn-2 run (x-axis) and the competing tool run (y-axis). More precise numbers are shown in Table 4. To simplify reading, we removed non-representative "*noise*"-runs which took less that 1 sec or exceeded the timeout for all configurations. In the table ϵ means an insignificant amount of time ($\leq$ 1 sec), and ∞ means the timeout (60 sec). The FreqHorn and FreqHorn-2 timings are the means of three individual runs.

In total, the table contains **93** instances. FreqHorn-2 outperformed its predecessor in **73** out of 93 cases. We witnessed the speedup up to **43X**, and in average FreqHorn-2 was *four times* faster than FreqHorn. In **37** cases FreqHorn-2 outperformed Spacer3, and in 34 cases Spacer3 outperformed FreqHorn-2. In **55** cases FreqHorn-2 outperformed μZ, and in 22 cases μZ outperformed FreqHorn-2. Note that FreqHorn-2 still has some performance anomalies which we believe are connected to often blind grammar-construction mechanism, inability to generate large disjunctions, and possible inefficiencies of the black-box interpolation engine.

7 Future Work

Consider a simple program in Fig. 3: 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. A invariant $x = 0 \vee x = 2 \vee \ldots \vee x = 256$ could be discovered by μZ in about 40 sec (and this time grows nonlinearly when 256 gets replaced by larger constants: e.g., for 512 it is 11 minutes and a half). That said, there exists a much more compact invariant $x \texttt{ mod } 2 = 0 \wedge x \leq 256$, 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 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 [25]) in the preprocessing steps of FreqHorn.

8 Related Work

In this work we exploit a range of techniques originated from symbolic model checking, and in particular in IC3/PDR [4,9], e.g., the idea of keeping CTIs and analyzing them to push previously considered lemmas [30]. Various strategies

could be applied for making the lemma pushing more or less eager, i.e., as soon as a newly-added lemma invalidates some CTI. In some IC3 implementations (e.g., [15]), eager pushing does not pay off, but avoiding to push certain lemmas during the regular pushing stage of IC3 resulted in an improvement. Since we do not have many lemmas, eager pushing also pays off.

The idea of applying Houdini to extract invariants from proofs of bounded safety was fundamental for the first version of Spacer [22]. They, however, keep obtaining proofs along the entire verification process. In contrast, we use proofs mainly for the bootstrapping, while the remaining progress of the algorithm is entirely dictated by the success of sampling.

Most of the successful verification tools today use various combinations of different static and dynamic techniques. In particular, approaches [28,2] use invariants from abstract interpretation to force convergence of k-induction. Recently, k-induction was benefitted from lemmas obtained from PDR [20]. A promising idea to exploit the data from traces while creating and manipulating the candidates [11,14] for invariants could also be used in our syntax-guided approach: at least we could add more constants to the grammar. However no one currently knows how to avoid over-population of the grammar by too many constants.

Techniques for automatic construction of grammars were applied outside of formal verification, but in the domain of security analysis and dynamic test generation [17,18]. 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 [1] is being applied in program synthesis more frequently than in verification, e.g., [27,26,10,19]. In these tasks, a formal grammar is typically additionally provided, and it is considered a part of specification. In contrast, in our task, the specification itself is the verification condition, so it contains the encoding of the entire program, and it is rich enough to let us construct formal grammars automatically. This is in fact the main driving idea behind FreqHorn, and it leaves us a spacious room for further improvements.

9 Conclusion

We presented a new revision of the FreqHorn algorithm to synthesize safe inductive invariants based on syntactic features of the source code and the proofs of bounded safety. The new algorithm contains the deterministic bootstrapping stage and the nondeterministic sampling stage, which make it more predictable than its predecessor, allows converging more frequently and in average four times faster. 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 interchange of ideas across application domains.

Acknowledgments It is hard to underestimate the value of discussions with Alexander Ivrii, Arie Gurfinkel, Michael W. Whalen, and other attendees of the International Conference on Formal Methods in Computer-Aided Design (FM-CAD 2017) which gave rise to many interesting ideas and inspired this work.

References

1. 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*, pages 1–17. IEEE, 2013.
2. D. Beyer, M. Dangl, and P. Wendler. Boosting k-Induction with Continuously-Refined Invariants. In *CAV, Part I*, volume 9206 of *LNCS*, pages 622–640, 2015.
3. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
4. A. R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
5. G. Cabodi, P. Camurati, M. Palena, P. Pasini, and D. Vendraminetto. Interpolation-based learning as a mean to speed-up bounded model checking (short paper). In *SEFM*, volume 10469 of *LNCS*, pages 382–387, 2017.
6. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. In *J. of Symbolic Logic*, pages 269–285, 1957.
7. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
8. I. Dillig, T. Dillig, B. Li, and K. L. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, pages 443–456. ACM, 2013.
9. N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, pages 125–134. IEEE, 2011.
10. G. Fedyukovich, M. B. S. Ahmad, and R. Bodík. Gradual Synthesis for Static Parallelization of Single-Pass Array-Processing Programs. In *PLDI*, pages 572–585. ACM, 2017.
11. G. Fedyukovich, A. Callia D'Iddio, A. E. J. Hyvärinen, and N. Sharygina. Symbolic Detection of Assertion Dependencies for Bounded Model Checking. In *FASE*, volume 9033 of *LNCS*, pages 186–201. Springer, 2015.
12. G. Fedyukovich, S. Kaufman, and R. Bodík. Sampling Invariants from Frequency Distributions. In *FMCAD*, pages 100–107. IEEE, 2017.
13. C. Flanagan and K. R. M. Leino. Houdini: an Annotation Assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.
14. P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *POPL*, pages 499–512. ACM, 2016.
15. A. Gurfinkel and A. Ivrii. Pushing to the top. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 65–72. IEEE, 2015.
16. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, volume 7317, pages 157–171. Springer, 2012.
17. M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *ASE*, pages 720–725. ACM, 2016.
18. M. Höschele and A. Zeller. Mining input grammars with AUTOGRAM. In *ICSE - Companion Volume*, pages 31–34. IEEE Computer Society, 2017.
19. J. P. Inala, N. Polikarpova, X. Qiu, B. S. Lerner, and A. Solar-Lezama. Synthesis of recursive ADT transformations from reusable templates. In *TACAS, Part I*, volume 10205 of *LNCS*, pages 247–263, 2017.
20. D. Jovanovic and B. Dutertre. Property-directed k-induction. In *FMCAD*, pages 85–92. IEEE, 2016.

21. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV*, volume 8559 of *LNCS*, pages 17–34, 2014. https://bitbucket.org/spacer/code/branch/spacer3.
22. A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *CAV*, LNCS, pages 846–862. Springer, 2013.
23. K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.
24. K. L. McMillan. Lazy annotation revisited. In *CAV*, volume 8559 of *LNCS*, pages 243–259. Springer, 2014.
25. D. Mordvinov and G. Fedyukovich. Synchronizing Constrained Horn Clauses. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 338–355. EasyChair, 2017.
26. P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodík. Chlorophyll: synthesis-aided compiler for low-power spatial architectures. In *PLDI*, pages 396–407. ACM, 2014.
27. Y. Pu, R. Bodík, and S. Srivastava. Synthesis of first-order dynamic programming algorithms. In *OOPSLA*, pages 83–98. ACM, 2011.
28. P. Roux, R. Delmas, and P. Garoche. SMT-AI: an abstract interpreter as oracle for k-induction. *Electr. Notes Theor. Comput. Sci.*, 267(2):55–68, 2010.
29. R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV*, volume 8559 of *LNCS*, pages 88–105. Springer, 2014.
30. M. Suda. Triggered clause pushing for IC3. *CoRR*, abs/1307.4966, 2013.
31. Y. Vizel, A. Gurfinkel, and S. Malik. Fast interpolating BMC. In *CAV*, volume 9206 of *LNCS*, pages 641–657. Springer, 2015.