Accelerating Syntax-Guided Invariant Synthesis

Grigory Fedyukovich[1] and Rastislav Bodík[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, leverages 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 [13] for a given program and a given safety specification. Invariants are needed to represent over-approximations of the set of reachable program states, such that from their empty intersection with the set 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 the candidate 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 a finite number of expressions, the formal grammar is sufficiently small, and thus the candidate formulas can be enumerated relatively quickly. We distinguish two types of candidates: 1) formulas directly extracted from the program's encoding (called *seeds*) and 2) formulas which are syntactically close to seeds (called *mutants*).

The conceptual novelty of FreqHorn is believed to be in the combined use of seeds and mutants, but the original paper [13] is largely silent on the matter. 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 relevance to the given safety specification; 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) [4,10] 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:

- A new revision and a new implementation of the FreqHorn algorithm which is split into the bootstrapping and the sampling stages. In the first stage, it deterministically exploits seeds only. In the second stage, it keeps generating and checking only 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 that likely reflect the nature of the error unreachability and consequently affect the grammar-based generation of mutants.
- 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 an efficient counterexample-guided approach 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 related work, conclusion, and acknowledgments complete the paper in Sect. 7 and 8.

2 Background and Notation

A first-order theory $\mathcal{T}$ consists of a signature Σ, which gathers variables, function and predicate symbols, and a set *Expr* of Σ-formulas. Formula $\varphi \in \mathit{Expr}$ is called $\mathcal{T}$-satisfiable if there exists an interpretation m of each element (i.e., a variable, a function or a predicate symbol), under which φ evaluates to $\top$ (denoted $m \models \varphi$); otherwise φ is called $\mathcal{T}$-unsatisfiable (denoted $\varphi \implies \bot$). The Satisfiability Modulo Theory (SMT) problem [8] for a given theory $\mathcal{T}$ and a 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 inductive invariant*, a formula that covers every initial state, is closed under the transition relation, and does not cover any of the error states.

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

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

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

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

To simplify reading, in the rest of the paper *safe inductive invariants* are referred to as just *invariants*. We assume that an invariant *Inv* has the form of conjunction, i.e., $\mathit{Inv} = \ell_0 \wedge \ldots \wedge \ell_n$, and each ℓ_i is called a *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 an interpretation m satisfying $\mathit{Inv}(V) \wedge \mathit{Tr}(V, V') \wedge \neg \mathit{Inv}(V')$, to which we refer to as a *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, $\texttt{x} \geq \texttt{N}$. An invariant for the program is defined non-uniquely, e.g., both the conjunction ($k \texttt{ mod } 2 = 0 \wedge x = c$) and conjunction ($k = x + c \wedge x \geq c$) are the solutions for this verification task. □

```
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)

$$x = 0 \quad c = 0 \quad k = 0 \quad c < N \quad k \bmod 2 = 0 \quad k = x + c \quad c \geq N \quad x \geq N$$

(b)

$$\alpha ::= 1\mid -1$$
$$\beta ::= 0 \mid 2$$
$$\gamma ::= x \mid y \mid k \mid N$$
$$\delta ::= \alpha \cdot \gamma + \ldots + \alpha \cdot \gamma \mid \gamma \bmod \beta$$
$$cand ::= \delta = \beta \mid \delta > \beta \mid \delta \geq \beta$$

(c)

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

(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 [13].[3] The pseudocode of the algorithm, called FreqHorn, is shown in Alg. 1. The key insight behind the algorithm is the automatic construction of a grammar G (line 2) based on a fixed set *Seeds* of expressions obtained by traversing parse trees of *Init*, *Tr*, and *Bad* (line 1).

To create G from *Seeds*, we drop all expressions that contain variables from both, V and V', and deprime all variables in the remaining expressions. Then, we normalize elements of *Seeds* to have the form of equalities, inequalities, or disjunctions of equalities and inequalities. Finally, formulas are rewritten, such

[3] The original description [13] 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.** [13].

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 $\quad cand \leftarrow$ sample(G);
6 $\quad res \leftarrow \big(Init(V) \implies cand(V)\big) \wedge \big(cand(V) \wedge \bigwedge_{\ell \in Lemmas} \ell(V) \wedge Tr(V, V') \implies cand(V')\big)$;
7 $\quad$ **if** res **then**
8 $\quad\quad Lemmas \leftarrow Lemmas \cup \{cand\}$;
9 $\quad G \leftarrow$ adjust(G, $cand$, res);

that all terms are moved to the left side, and the subtraction, $<$, and $\leq$ are rewritten respectively as the addition, $>$, and $\geq$.

The algorithm uses G for generating the candidate formulas (line 5) and populates the set of lemmas until their conjunction is an invariant. The algorithm learns from each positive and negative attempt (line 9). That is, G gets adjusted, such that the candidate (and some of its close relatives) is not going to be sampled in any of the following iterations.

Example 2. The verification condition for the program in Fig. 1a is represented by three implications in Fig. 1d. They are syntactically split into the set *Seeds* of expressions over V, elements of which used to contain only primed or only unprimed variables (shown in Fig. 1b). In particular, equalities $x' = x + M$ and $c' = c + M$ are excluded from *Seeds*, and equality $k' = x' + c'$ is rewritten to $k = x + c$. The grammar containing the normalized expressions from *Seeds* is shown in Fig. 1c. It is easy to see that all lemmas consisting in both invariants from Ex. 1, $(k \texttt{ mod } 2 = 0 \wedge x = c)$ and $(k = x + c \wedge x \geq c)$, can be generated by applying the grammar's production rules recursively. □

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

The main downside of Alg. 1 is that it is hard to choose a sampling order for each individual lemma at the final invariant. Suppose, $cand = (x = c)$ 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: $[x \leftarrow 0; k \leftarrow 1; c \leftarrow 0; N \leftarrow 10; x' \leftarrow 0; k' \leftarrow 7; c \leftarrow 7; M \leftarrow 7]$. The grammar is then adjusted, such that $x = c$ (and some syntactically relevant, stronger or equivalent formulas, e.g. $-x = -c$) do not belong to the grammar anymore.

Suppose, in the second iteration of Alg. 1, $cand = (k = x + c)$. 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). The grammar is then adjusted again, such that $k = x + c$ (and some weaker or equivalent formulas, e.g. $k \geq x+c$) do not belong to the grammar anymore. Note that if in the third iteration $cand = (x = c)$ was sampled again, the algorithm would terminate. However, it is impossible since the sampling grammar was adjusted after both negative and positive attempts.

The opposite sampling order (i.e., $cand = (k = x+c)$ first, and $cand = (x = c)$ then) would lead to a faster convergence of the algorithm. Since it is hard to decide which order to choose, the production rules are equipped with probability distributions that allow both orders under certain probabilities. In this paper, we propose to use 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 the satisfiability of the resulting formula (called a 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. The extraction of 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 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.*

Algorithm 2: BMCITP: Obtaining bounded proofs, **cf.** [24,25].

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 ← ⊤;
2  for (i ← k; i > 0; i ← i − 1) do
3      unr ← unr ∧ Tr(V^(i−1), V^(i));
4      if unr ∧ Bad(V^(k)) ⟹ ⊥ then
5          proof ← GETITP(unr, Bad(V^(k)));
6          return;
7  unr ← unr ∧ Bad(V^(k));
8  if Init(V^(0)) ∧ unr ⟹ ⊥ then
9      proof ← GETITP(Init(V^(0)), unr);
```

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 $\{\ell_i\} \leftarrow \text{GETITP}(A, B)$. Alg. 2 shows an algorithm to generate interpolation-based proofs of bounded safety for BMC formulas. It iteratively unrolls the transition relation and applies the interpolation to the entire BMC formula. In addition, in spirit of Lazy Annotation [25], while decrementing i, the algorithm applies a backward reasoning and checks if an error state is reachable by $(k - i)$ steps from an empty state (line 4). It triggers the interpolation to be applied to smaller formulas, and in some cases fastens the proof search (line 5).

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 the consecution 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 the 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, which is extensively used in PDR and also known as HOUDINI [14], Note that HOUDINI is only meaningful for the candidate formulas which are already implied by the initial states.

Example 4. Conjunction of formulas from set *Seeds* in Fig. 1b is unsatisfiable, and its minimal unsatisfiable core is $c < N \wedge c \geq N$. Thus, Alg. 3 would immediately return the entire set *Seeds*. Let a set *Cands* be constructed from *Seeds* by re-

Algorithm 3: HOUDINI: Calculating an inductive subset, **cf.** [14] and keeping counterexamples-to-induction.

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

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

moving all elements, for which condition (1) does not hold. Conjunction of the elements in *Cands* is satisfiable: $\{x = 0, c = 0, k = 0, k \bmod 2 = 0, k = x + c\}$. Applying Alg. 3 to *Cands* gives the inductive subset $\{k \bmod 2 = 0, k = x + c\}$. □

Note that we extended Alg. 3 with a routine to extract a counterexample-to-induction π for each element dropped from *Cands* (lines 3-6). We restrict each π to only assignments to variables from V (denoted $\pi\big|_V$) and group all non-inductive formulas from *Cands* by the particular π that killed them. This routine is important for optimizing the syntax-guided invariant synthesis algorithm, and it is discussed in more detail in Sect. 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 seeds and mutants separately. Indeed, as we have seen in Ex. 2, both seeds and mutants are needed for constructing an invariant, but seeds do *not* actually need to be re-sampled – these candidates are ready to be checked prior to any sampling.

5.1 Overview

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 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 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

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
5          return;
6      else
7          Seeds ← Seeds ∪ proof;
8  G ← getGrammarAndDistributions(Seeds);
9  CTI, CTImap ← ∅;
10 #learned ← 0;
11 Cands ← Seeds;
12 while Bad(V) ∧ ⋀_{ℓ∈Lemmas} ℓ(V) ⇏ ⊥ do
13     while |Cands| < M do
14         Cands ← Cands ∪ {sample(G)};
15     for cand ∈ Cands do
16         if Init(V) ⇏ cand(V) then
17             G ← adjust(G, cand, false);
18             Cands ← Cands \ {cand};
19     ⟨Lemmas', CTI, CTImap⟩ ← Houdini(P, Cands ∪ Lemmas, CTI, CTImap);
20     for ℓ' ∈ Lemmas' \ Lemmas do
21         G ← adjust(G, ℓ', true);
22     NewLemmas = {ℓ' | ℓ' ∈ Lemmas', s.t. ⋀_{ℓ∈Lemmas} ⇏ ℓ'};
23     #learned ← #learned + |NewLemmas|;
24     Lemmas = Lemmas ∪ NewLemmas;
25     Cands ← ∅;
26     if #learned > K then
27         #learned ← 0;
28         for m ∈ CTI do
29             if m ⊭ ⋀_{ℓ∈Lemmas} ℓ(V) then
30                 CTI ← CTI \ m;
31                 Cands ← Cands ∪ CTImap(m);
```

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 (to be explained in Sect. 5.2) lies in a 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 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 5).

The bootstrapping ends when the merged set *Seeds* is taken as input by Alg. 3, and it extracts an inductive subset (line 19). However, prior to it, the algorithm checks the initiation condition for all elements of the merged set, and the set is filtered accordingly (lines 15-18).

Example 5. Let set *Seeds* be as in Fig. 1b, and set *Cands* be 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 $\mathit{Cands} \cup \{x = c\}$, we get the inductive subset $\{k \texttt{ mod } 2 = 0, k = x + c, x = c\}$. Since the conjunction of these lemmas is an invariant, 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 *Cands* is less than M, then the set gets additional mutants (lines 13-14). 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 24), and the assembly of a new batch of candidates starts from scratch (line 25).

Example 6. Assume that a proof of bounded safety is $\mathit{proof}_1 = \{x \geq 0, c \leq 0\}$ (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 *Cands* 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. □

Theorem 1. *If Alg. 4 terminates, then either an actual bug is found (line 5), or an invariant is synthesized (after the while-loop).*

5.2 Learning strategy

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 (recall the example after Def. 3). The grammar adjustments are performed by changing the probabilities assigned to the production rules. In addition to zeroing the probability of sampling a candidate *cand*

itself, after each positive check, Alg. 1 zeroes the probabilities of sampling some formulas which are weaker than *cand*, and after each negative check – the probabilities of sampling some formulas which are stronger than *cand* (see [13] for more details).

In contrast, Alg. 4 reacts just to the failed candidates after the initiation check (line 17) and to the successful candidates after the consecution check (line 21). Otherwise, if the consecution check failed for a candidate *cand* (inside Alg. 3), Alg. 4 does not disqualify *cand* from being checked again in the future, and this is done by keeping *cand* locally and periodically seeking an opportunity to give *cand* 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 *cand* there exists $m \in CTI$ that killed it. To maintain this information, every application of Alg. 3 updates the map *CTImap* from *CTI* to failed candidates. In Alg. 4, it remains to periodically check whether some m is eliminated (line 29), and it would increase chances of all candidates killed by m (line 31) to succeed the consecution 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 consecution check again.

Finally, to ensure that the CTI-check happens not too often, we run it only when at least K new lemmas are learned. To make this happen, Alg. 4 perfoms a redundancy check (line 22) for all lemmas ℓ that have passed the initiation and the consecution checks: ℓ gets learned only when the conjunction of all lemmas learned so far does not imply ℓ. Obviously, when no new lemmas (after the redundancy check) are added, it does not make sense to run the CTI-check since all CTIs are still valid.

5.3 Optimizations

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

- As a consequence of calculating frequencies, in the original FreqHorn algorithm, seeds were given priorities, but mutants were considered with a relatively small probability. In contrast, the new FreqHorn's revision forces 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 15-18) 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 consecution 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 to the probabilities of 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 [23] 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 a BMC formula for bound k to encode the BMC formula for bound $k+1$. Potentially, other tricks (e.g., [30,5]) could also be applied here. Finally, interpolation could be replaced by the weakest precondition computation.

6 Evaluation

We implemented FreqHorn-2 on top of our prior implementation FreqHorn[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. All the symbolic reasoning is performed by the Z3 SMT solver [7].

We evaluated FreqHorn-2 on various safe and buggy programs taken from SVCOMP[5] and literature (e.g., [9,15]). Since most of benchmarks, proposed by [9], 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 (LIA) and non-linear integer arithmetic (NIA). We did run FreqHorn-2 on unsafe instances for the testing purposes only. It was able to detect a counterexample, but since no invariant exists in these cases, we do not discuss this experience here.

6.1 The bootstrapping experiment

In total, we considered **171** safe programs. For **103** 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 63, but using the inductive subset extraction helped revealing all 103. Each set of seeds contained in average 9 formulas.

For our BMC implementation, we considered bounds 1, 2, and 3. Generated interpolants already contained all lemmas for invariants for **70** programs.[6] Each set of bounded 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 performed afterwards. 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 **114** programs.

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

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

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

Table 1: Exact timings.

Benchmark	FreqHorn_2	FreqHorn	μZ	Spacer
abdu_01	ϵ	1.27	ϵ	ϵ
abdu_04	ϵ	8.64	ϵ	ϵ
bouncy_three	6.87	6.76	ϵ	ϵ
bouncy_two	2.74	2.41	∞	∞
cegar1	ϵ	2.18	ϵ	ϵ
cggmp_iter_1	3.09	8.73	ϵ	ϵ
cggmp_iter_2	10.50	24.92	ϵ	ϵ
cggmp_iter_3	18.80	55.86	ϵ	ϵ
const_div_1	ϵ	2.61	ϵ	ϵ
const_div_2	ϵ	1.40	ϵ	ϵ
const_div_3	∞	∞	21.31	∞
const_mod_3	ϵ	∞	ϵ	ϵ
count_by_2_modif	3.28	10.98	∞	∞
count_by_2	2.80	1.62	∞	∞
countud	3.60	15.96	∞	ϵ
css2003	ϵ	9.59	ϵ	ϵ
dillig02	10.91	35.48	ϵ	ϵ
dillig03	ϵ	ϵ	∞	∞
dillig05	4.74	2.04	∞	∞
dillig07	1.93	2.58	ϵ	ϵ
dillig08	ϵ	1.26	∞	ϵ
dillig10	2.28	2.72	ϵ	ϵ
dillig13_1	∞	∞	ϵ	ϵ
dillig13	∞	∞	ϵ	ϵ
dillig14	1.90	6.98	ϵ	ϵ
dillig15	3.28	3.61	∞	ϵ
dillig16	20.14	15.35	∞	ϵ
dillig20_1	ϵ	1.65	ϵ	ϵ
dillig20_2	ϵ	4.97	ϵ	ϵ
dillig20_3	34.51	4.53	ϵ	ϵ
dillig21	ϵ	4.06	7.5	3.98
dillig22_2	ϵ	8.59	ϵ	ϵ
dillig22_3	29.77	15.43	ϵ	ϵ
dillig22_4	9.99	15.20	ϵ	ϵ
dillig22_5	14.12	12.02	ϵ	ϵ
dillig22_6	25.88	13.75	∞	ϵ
dillig22	ϵ	7.25	ϵ	ϵ
dillig37	1.48	1.67	ϵ	ϵ
dillig41	ϵ	1.41	∞	∞
dillig42_1	23.88	∞	ϵ	ϵ
dillig42	52.13	∞	ϵ	ϵ
dillig44_1	ϵ	∞	ϵ	ϵ
dillig44	ϵ	∞	ϵ	ϵ
dillig46	6.20	4.86	ϵ	ϵ
ex7	2.28	8.93	ϵ	ϵ
exact_iters_1	26.55	35.58	ϵ	ϵ
exact_iters_2	∞	22.76	∞	∞
exact_iters_3	23.96	26.60	∞	∞
exact_iters_4	∞	29.19	ϵ	ϵ
fig3	ϵ	21.82	ϵ	ϵ
formula22	ϵ	11.34	ϵ	ϵ
formula25	ϵ	1.12	ϵ	ϵ
formula27	ϵ	20.31	ϵ	ϵ
gcd_2	2.15	1.84	ϵ	ϵ
gcd_3	2.40	3.58	ϵ	ϵ
gj2007	∞	∞	∞	ϵ
half_true_modif	∞	58.15	ϵ	ϵ
half_true_orig	35.58	45.00	ϵ	ϵ
hhk2008	6.60	28.46	ϵ	ϵ
menlo_park_term_simpl_1	ϵ	1.39	∞	15.84
menlo_park_term_simpl_2	12.29	58.36	∞	∞
n_c11	ϵ	∞	ϵ	ϵ
nonlin_div	26.00	∞	∞	∞
nonlin_factorial	ϵ	∞	∞	∞
nonlin_minus_1	ϵ	1.78	∞	∞

Benchmark	FreqHorn_2	FreqHorn	μZ	Spacer
nonlin_minus_2	ϵ	2.60	∞	∞
nonlin_mod_1	5.78	16.13	∞	∞
nonlin_mod_2	∞	18.46	∞	∞
nonlin_mod_mult	ϵ	ϵ	∞	∞
nonlin_mult_1	ϵ	ϵ	∞	∞
nonlin_mult_2	ϵ	2.28	∞	∞
nonlin_mult_3	ϵ	2.86	∞	∞
nonlin_mult_4	ϵ	2.48	∞	∞
nonlin_mult_5	∞	3.22	∞	∞
nonlin_mult_6	ϵ	3.27	∞	∞
nonlin_power	ϵ	∞	ϵ	ϵ
nonlin_square	ϵ	ϵ	∞	∞
nonterm_01	ϵ	ϵ	∞	ϵ
phases1	∞	10.81	∞	∞
s_disj_ite_01	8.77	5.43	ϵ	ϵ
s_disj_ite_02	3.98	5.98	ϵ	ϵ
s_disj_ite_03	13.15	2.71	ϵ	∞
s_disj_ite_04	7.01	4.13	ϵ	ϵ
s_disj_ite_05	4.64	∞	18.66	40.38
s_disj_ite_06	6.43	∞	ϵ	ϵ
s_mutants_01	ϵ	ϵ	∞	∞
s_mutants_02	ϵ	2.40	∞	∞
s_mutants_03	ϵ	ϵ	∞	∞
s_mutants_05	7.86	1.90	∞	∞
s_mutants_06	ϵ	ϵ	∞	∞
s_mutants_07	1.63	1.75	ϵ	ϵ
s_mutants_09	ϵ	3.41	ϵ	ϵ
s_mutants_11	ϵ	1.86	4.72	ϵ
s_mutants_12	ϵ	4.09	ϵ	ϵ
s_mutants_13	ϵ	10.42	ϵ	ϵ
s_mutants_14	ϵ	11.47	ϵ	ϵ
s_mutants_15	ϵ	14.50	ϵ	ϵ
s_mutants_16	14.38	∞	∞	∞
s_mutants_17	14.56	5.83	∞	∞
s_mutants_18	39.85	∞	ϵ	ϵ
s_mutants_19	ϵ	1.31	1.44	ϵ
s_mutants_20	41.15	36.47	∞	∞
s_mutants_21	∞	24.68	∞	∞
s_mutants_22	14.82	16.52	∞	∞
s_mutants_23	1.61	1.11	∞	∞
s_mutants_24	1.58	2.81	∞	ϵ
s_seeds_04	ϵ	∞	20.49	ϵ
s_seeds_05	ϵ	ϵ	∞	ϵ
s_seeds_06	ϵ	1.78	ϵ	∞
s_seeds_10	ϵ	∞	ϵ	ϵ
s_triv_01	ϵ	1.33	ϵ	ϵ
s_triv_07	ϵ	1.15	ϵ	ϵ
s_triv_08	ϵ	∞	ϵ	ϵ
s_triv_09	ϵ	∞	ϵ	ϵ
s_triv_11	ϵ	∞	ϵ	ϵ
s_triv_12	ϵ	∞	ϵ	ϵ
s_triv_14	ϵ	ϵ	∞	ϵ
s_triv_16	ϵ	∞	ϵ	ϵ
s_triv_17	ϵ	∞	ϵ	ϵ
sn_1024	6.14	14.64	24.83	7.12
sn_2048	25.91	26.11	∞	30.87
sn_4096	7.57	∞	∞	∞
sn_8192	11.82	41.01	∞	∞
three_dots_moving_1	ϵ	58.48	10.28	ϵ
three_dots_moving_2	ϵ	∞	ϵ	ϵ
three_dots_moving_3	ϵ	∞	ϵ	ϵ
trex3	ϵ	8.23	ϵ	ϵ
yz_plus_minus	4.35	∞	∞	∞

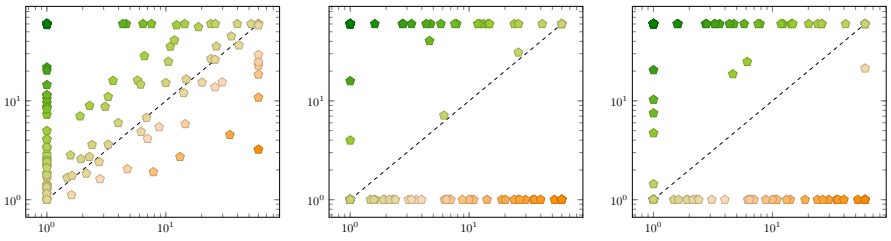

Fig. 2: FreqHorn-2 vs respectively FreqHorn, Spacer3, and μZ.

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 57 out of 171 cases.

6.2 Overall statistics

Since technically FreqHorn-2 is a CHC-solver, we compared it against other CHC-solvers, namely μZ [17,22] and Spacer3 [22]. All the tools were provided with the same CHC-encodings of verification problems (and thus, the results do not directly depend on a process of encoding a C-program to a CHC-file). Both μZ and Spacer3 are PDR-based, and despite the latter is faster than the former and can solve more benchmarks, there are 26 instances, for which the former outperforms the latter.

Table 1 shows the precise running times of FreqHorn-2, FreqHorn, μZ, and Spacer3. To simplify reading, we removed non-representative "*noise*"-runs which took less that 1 second or exceeded a timeout of 60 seconds by all tools. In the table, ϵ denotes an insignificant amount of time (≤ 1 second), and ∞ denotes the timeout. The numbers of FreqHorn and FreqHorn-2 are the means of three individual runs. In total, the table contains **128** instances. Additionally, Fig. 2 shows three scatter plots comparing running times of FreqHorn-2 vs FreqHorn, μZ, and Spacer3 respectively. Each point in a plot represents a pair of the FreqHorn-2 run (x-axis) and the competing tool run (y-axis).

FreqHorn-2 outperformed its predecessor in **90** out of 128 cases. We witnessed the speedup up to **233X**, and in average FreqHorn-2 was *four times* faster than FreqHorn. In **40** cases FreqHorn-2 outperformed Spacer3, and in 38 cases Spacer3 outperformed FreqHorn-2. In **51** cases FreqHorn-2 outperformed μZ, and in 34 cases μZ outperformed FreqHorn-2. Unfortunately, FreqHorn-2 still has some performance anomalies, which we believe are connected to the often blind grammar-construction mechanism, inability to generate large disjunctions, and possible inefficiencies of the black-box interpolation engine.

7 Related Work

In this work we exploit a range of techniques originated from symbolic model checking, and in particular from IC3/PDR [4,10], e.g., the idea of keeping CTIs and analyzing them to push previously considered lemmas [29]. 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., [16]), eager pushing does not pay off, but avoiding to push certain lemmas during the regular pushing stage of IC3 results 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 [23]. 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 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 [21]. A promising idea to exploit the data from traces [12,15] while creating and manipulating the candidates for invariants could also be used in our syntax-guided approach: at least we could add more constants to the grammar. However we are currently unaware of a strategy to find meaningful constants and to avoid over-population of the grammar by too many constants. Our preliminary experiments resulted so far in the performance decrease.

Techniques for automatic construction of grammars were applied outside of formal verification, but in the domains of security analysis and dynamic test generation [18,19]. 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, syntax-guided techniques [1] keep being used in program synthesis more frequently than in the inductive invariant synthesis. For instance, in applications [27,26,11,20] a formal grammar is additionally provided, and it is considered a part of specification. In contrast, in our application, the verification condition contains the encoding of the entire program and the safety specification, which together are enough for construction of formal grammars completely automatically. This is in fact the main driving idea behind FREQHORN, and it leaves us a spacious room for its further adaptations, e.g., in proving and disproving program termination, automated repair of software regressions, and security analysis.

8 Conclusion

We have presented the 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 successful 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 (FMCAD 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. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
8. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
9. I. Dillig, T. Dillig, B. Li, and K. L. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, pages 443–456. ACM, 2013.
10. N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, pages 125–134. IEEE, 2011.
11. 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.
12. 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.
13. G. Fedyukovich, S. Kaufman, and R. Bodík. Sampling Invariants from Frequency Distributions. In *FMCAD*, pages 100–107. IEEE, 2017.
14. 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.
15. P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *POPL*, pages 499–512. ACM, 2016.
16. A. Gurfinkel and A. Ivrii. Pushing to the top. In *FMCAD*, pages 65–72. IEEE, 2015.
17. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, volume 7317 of *LNCS*, pages 157–171. Springer, 2012.
18. M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *ASE*, pages 720–725. ACM, 2016.
19. M. Höschele and A. Zeller. Mining input grammars with AUTOGRAM. In *ICSE - Companion Volume*, pages 31–34. IEEE Computer Society, 2017.
20. 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.
21. D. Jovanovic and B. Dutertre. Property-directed k-induction. In *FMCAD*, pages 85–92. IEEE, 2016.

22. 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`.
23. A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *CAV*, volume 8044 of *LNCS*, pages 846–862. Springer, 2013.
24. K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.
25. K. L. McMillan. Lazy annotation revisited. In *CAV*, volume 8559 of *LNCS*, pages 243–259. Springer, 2014.
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. M. Suda. Triggered clause pushing for IC3. *CoRR*, abs/1307.4966, 2013.
30. Y. Vizel, A. Gurfinkel, and S. Malik. Fast interpolating BMC. In *CAV*, volume 9206 of *LNCS*, pages 641–657. Springer, 2015.