# Efficiently Learning Safety Proofs from Appearance as well as Behaviours

Sumanth Prabhu(✉), Kumar Madhukar, and R. Venkatesh

TCS Research, Pune, India
sumanth.prabhu@tcs.com

**Abstract.** Proving safety of programs relies principally on discovering invariants that are inductive and adequate. Obtaining such invariants, therefore, has been studied widely from diverse perspectives, including even mining them from the input program's source in a *guess-and-check* manner [13]. However, guessing candidates based on syntactical constructions of the source code has its limitations. For one, a required invariant may not manifest on the syntactic surface of the program. Secondly, a poor *guess* may give rise to a series of expensive *checks*. Furthermore, unlike conjunctions, refining disjunctive invariant candidates is unobvious and may frequently cause the proof search to diverge. This paper attempts to overcome these limitations, by learning from both – appearance and behaviours of a program. We present an algorithm that (i) infers useful invariants by observing a program's syntactic source as well as its semantics, and (ii) looks for conditional invariants, in the form of implications, that are guided by counterexamples to inductiveness. Our experiments demonstrate its benefits on several benchmarks taken from SV-COMP and the literature.

## 1 Introduction

Arguing for program correctness is a challenging task. But it is non-optional, especially as software has permeated our lives, in forms that are many times even safety- or business-critical. Not surprisingly, this subject has been the focus of a lot of research in the last several decades, and there is a vast amount of literature covering different facets of this problem. The issue that is central to all of this is that of discovering *inductive invariants*, that are sufficient to discharge the property in question. Invariants help in over-approximating the reachable states, which can then be shown to be disjoint with the set of *bad* states to establish safety, whereas precisely computing what is reachable may be infeasible.

Numerous techniques have been proposed for inferring program invariants automatically, and even semi-automatically with human assistance. Broadly speaking, these techniques learn meaningful information about the input program from its semantics, using approaches based on abstract interpretation [5–7], constraint solving [4,16], counterexample-guided abstraction refinement [3], property directed reachability [2,17], interpolation [1,8], user-assistance [19], etc. In contrast, Fedyukovich et al. [13] recently demonstrated that invariants can

© Springer Nature Switzerland AG 2018
A. Podelski (Ed.): SAS 2018, LNCS 11002, pp. 326–343, 2018.
https://doi.org/10.1007/978-3-319-99725-4_20

```
int n;
assume(1 <= n <= 1000);

int sum = 0, i = 1;

while(i<=n) {
  sum = sum + i;
  i = i + 1;
}

assert(2*sum == n*(n+1));
```

(a)

```
int LRG = nondet();
assume(LRG > 0);

int x = 0, y = LRG;

while(x < 2*LRG) {
  if (x < LRG) {
    y = y;
  } else {
    y = y + 1;
  }
  x = x + 1;
}

assert(y == 2*LRG);
```

(b)

**Fig. 1.** Motivating examples

often be caught on the surface, i.e. the invariants many times imitate the syntactical constructions appearing in the source code. Their tool, FreqHorn, works in a *guess-and-check* manner, by sampling candidates from an appearance-guided search space built automatically from ingredients found in the program source. A follow-up work [12], and the corresponding tool FreqHorn-2, accelerates this process by computing additional candidates as interpolants from proofs of bounded safety. These candidates likely reflect the nature of the error unreachability, and thus have a semantic value. While this justifies the idea of supplementing syntactic search with behavioural[1] facts about the program, interpolants obtained from bounded proofs may not fully capture these facts. Nevertheless, an important contribution of this technique is the automatic construction of sampling space, which is particular to the input program. This can even assist template-based methods, e.g. Daikon [11], in selecting the templates carefully, instead of working with a generic one that may be needlessly more expressive.

Consider the example shown in Fig. 1a. It computes the sum of first $n$ natural numbers, and asserts that twice the computed sum equals $n$ times $(n+1)$. Since this is an arithmetic fact, the program is safe. The sum is computed by iterating over the numbers from 1 to $n$ in a loop, and by adding each number to the variable *sum*, which is 0 initially. One way to prove this program correct is to obtain the following inductive invariants for the while loop: $2 * sum = (i-1) * i$, and $i \leq (n+1)$. Along with the exit condition of the loop, $(i > n)$, these are sufficient to derive that $2 * sum = n * (n+1)$.

[1] *behaviour* refers to facts derivable from the program's meaning, not necessarily limited to its concrete runs; we use the terms *behaviours* and *semantics* interchangeably.

A merely syntactic exploration would find the invariant $i \leq (n+1)$ (it is a mutation of the loop condition), but it would fail[2] to deduce that $2 * sum = (i-1) * i$ is a loop invariant. While the latter is quite similar to an expression appearing in the program, namely the property, FreqHorn-2 does not consider mutations that alter variables. And even if it did, that would result in a number of mutants which are poor candidates. I.e. they would fail the inductiveness check, which is an expensive operation in this case because of the non-linear template. On the other hand, if we look to obtain algebraic invariants behaviourally, e.g. as proposed by Sharma et al. [24], we can get that inductive invariant almost immediately.

It is noteworthy that an execution-based approach, similar to the one stated above, would be able to verify this example even when $n$ is replaced by a concrete value, say 239, and the property becomes $2*sum = 57360$. The desired invariant, $2 * sum = (i-1) * i$, is no longer available as a mutation of the property. But it is still a valid algebraic relation between $sum$, $i$ and $i^2$, that can easily be drawn from program executions. In other words, information available from concrete runs complements the syntactic search for invariants, especially when the property does not entirely manifest at the program surface, but also lies deeper in its behaviours.

For another limitation of the existing technique, let us consider the program shown in Fig. 1b, chosen from the benchmarks used in [12]. The program has a positive constant LRG, denoting a large value perhaps, and two variables, $x$ and $y$. The while loop in the program has two distinct *phases* – first in which only $x$ gets incremented, till it becomes LRG (and equal to $y$'s initial value), and the second where both $x$ and $y$ are incremented as long as $x$ is less than twice the large constant. The assertion holds because $x$ and $y$ are equal after every iteration in the second phase. A formal proof of correctness can be derived from the following inductive invariants: $(((x < \texttt{LRG}) \Rightarrow (y = \texttt{LRG})) \wedge ((x \geq \texttt{LRG}) \Rightarrow (y = x)))$, and $(x \leq 2*\texttt{LRG})$.

FreqHorn-2 rarely converges to a proof for this program (only once in 20 runs in our experiments, with a timeout of 600 s); the reason being lack of structured search, particularly for disjunctive invariant candidates. For example, in order to get $((x \geq \texttt{LRG}) \vee (y = \texttt{LRG}))$, FreqHorn-2 has to choose the candidate's arity as 2, and then sample the parts $(x \geq \texttt{LRG})$ and $(y = \texttt{LRG})$ separately. If any of the choices turn out to be bad, the inductiveness check fails, and a subsequent refinement may even replace disjuncts that are useful or necessary. Analyzing behaviours may not work for such programs either. There must be enough runs representing all the *phases* in order to deduce the algebraic relations, even if they exist.

We propose a method to solve this problem by extracting conditional invariants, which are implications with antecedents that are derived from conditions appearing in the program. Whether a conditional invariant needs to be sampled or not is decided by inspecting the counterexamples to inductiveness, or CTIs, of the candidates explored thus far. We check if the counterexamples can be

[2] FreqHorn-2 times out after 600 s, in an experimental set-up similar to [12,13].

made to fit into a polynomial over program variables, to determine if they are of the same *kind*. Intuitively, if there are different kinds of counterexamples, it may be worthwhile to look for an invariant for each kind. I.e. implications of the form $cond_i \Rightarrow inv_i$, where $cond_i$ qualifies the kind of CTIs, and $inv_i$ denotes the invariant that gets rid of those.

Given the invariants that are needed to prove safety of the example in Fig. 1b, it is evident that this enhancement allows us to get them quickly. Note that the restriction to sample the antecedents from a very small space (of conditions appearing in the program, and their mutations) prevents us from divergence in many cases. However, at the same time, it is expressive enough to work in a number of cases. In particular, it enables our approach to solve examples with multi-phase loops that require phase-specific invariants [23].

The core contributions of this paper are summarized as follows:

- A technique that combines learning from a program's behaviours, with that from its syntactic source, for inferring useful invariants.
- A heuristic to determine whether conditional invariants could be useful, and a method to obtain them by analyzing implications whose antecedents are chosen to be (possibly, conjunctions and/or mutations of) conditions appearing in the program, or negations thereof.
- An implementation that extends FreqHorn-2[3] – the tool used for evaluation in [12], which forms the basis of this work.
- Experimental evaluation that illustrates the usefulness of our approach on several benchmarks from SV-COMP and the literature.

*Outline of the Paper.* We start with a survey of the related work in the next section (Sect. 2), before moving over to some of the closely related ones in details, seeing that they serve as the necessary background (Sect. 3). Section 4 describes the core contributions of this work, and is followed by a discussion of the experimental results (Sect. 5). Section 6 concludes the paper, and includes our thoughts on several interesting directions of pursuing this further.

## 2 Related Work

Invariant synthesis is an essential step in program verification. Abstract interpretation [5,6] is a prominent technique which iteratively computes approximations until a fix point is reached. The assertion generated at fix point is an inductive invariant. In order to overcome the difficulty of choosing widening heuristics in abstract interpretation, template-based techniques were proposed. For example, [4] assumes the invariants to be in a fixed template over program variables. Inductiveness conditions are translated to nonlinear constraints such that the solutions of constraints are invariants. However, this technique relies on the efficiency of nonlinear constraint solving.

[3] Thanks to Grigory Fedyukovich, the sources of FreqHorn-2 are available at https://github.com/grigoryfedyukovich/aeval/tree/rnd.

A somewhat related technique for invariant discovery is that of *guess-and-check*, which repeatedly guesses candidate invariants from a known language represented by a grammar, and checks them for invariance. Automatic construction of an adequate grammar, tractable search among candidates, and inductiveness check of candidates are the main challenges of this technique. In general, an SMT solver that can decide the underlying theory is used for the inductiveness check. The other two challenges are addressed using data computed through static and dynamic analysis techniques. For instance, the technique presented in [24] uses concrete program runs as data to discover invariants. Invariants are assumed to have the form of a fixed-degree polynomial equation over program variables. The execution traces are used to solve for coefficients of the polynomial. It uses an SMT solver to check inductiveness of the solutions. A similar dynamic analysis technique to discover polynomial and array invariants has been proposed in [18]. The drawbacks of these techniques are high computational complexity for discovering invariants with inequality [18], and inability to derive disjunctions that are not polynomial equations.

Counterexamples to consecution, along with the information available on unreachable and reachable states (referred to as ICE), are used for guiding the search for invariant candidates in [15]. An invariant is assumed to be boolean combinations of atomic formulas of a particular form, e.g. an octagon. The problem of guessing a candidate is modeled as problem of generating a formula that separates reachable and unreachable states. Techniques from learning theory are used on the available data to solve this problem.

In [22], the invariant candidates are sampled as boolean combinations of linear inequalities, whose coefficients and constants are taken from a data set that is populated from constants occurring in the source code, and their sums and differences. It also incorporates those counterexamples in the data that disqualify a candidate as an invariant. The entire program source may also be considered as data, e.g. [13]. A frequency distribution obtained from the input program's source guides the automatic construction of grammar. Moreover, failed candidates are used to prune the search space of candidates. This technique was found to be competitive to other machine-learning techniques. However, pruning can cause divergence in the algorithm. This problem is partially addressed in [12], which performs consecution checks in batches, and uses the counterexamples to induction effectively. It also supplements the method with candidates of semantic values, obtained as interpolants from bounded proofs. Our work further enhances this by mining candidates from program behaviours, and enabling discovery of conditional invariants.

## 3 Notations and Background

We begin with a description of the notations that are used in Fedyukovich et al. [12,13], which we also follow.

**Definition 1.** *A program $P$ is defined as a transition system, or a tuple $\langle V \cup V', Init, Tr \rangle$, where*

- *$V$ denotes the set of variables, and the corresponding primed set $V'$ represents their next-state copies,*
- *$Init$ is a set of initial states encoded as a formula over $V$, and*
- *$Tr(V, V')$ is a transition relation encoded as a formula over $V$ and $V'$.*

We assume that the formulas belong to a fixed first order language $\mathcal{L}$. A *state* is an assignment of values to all variables in $V$ or $V'$. For a formula $\phi$ over $V$, a state $s$ satisfies it, $s \models \phi$, when the assignment of values to all variables as per $s$ satisfies the formula $\phi$. A state $s_k$ is *reachable* if either $s_k \models Init$ or $\exists s_{k-1}, (s_{k-1}, s'_k) \models Tr$, where $s_{k-1}$ is a reachable state and $s'_k$ assigns same values as $V$ for corresponding primed set $V'$.

Given $\langle P, Bad \rangle$, where $Bad$ is an undesirable set of states encoded as a formula over $V$, verification of $P$ is the task of deciding whether a state from $Bad$ is reachable or not. An $\mathcal{L}$-formula $Inv$ which is disjoint from $Bad$ and includes all the reachable states is called a safe inductive invariant, or henceforth simply an invariant. If we assume that an invariant exists in $\mathcal{L}$, then verification of $P$ reduces to finding an invariant $Inv$, such that the following hold:

$$Init(V) \Rightarrow Inv(V) \qquad \text{initiation}$$
$$Inv(V) \wedge Tr(V, V') \Rightarrow Inv(V') \qquad \text{consecution}$$
$$Inv(V) \wedge Bad(V) \Rightarrow \bot \qquad \text{safety}$$

Note that $\bot$ denotes *false*. These validity checks can be transformed into equivalent unsatisfiability checks, to be discharged by an SMT solver e.g. Z3 [9]. The models corresponding to the consecution check failure are referred to as *CTIs*. More formally, CTIs is a set of pair of states $(s_k, s'_{k+1})$, such that $s_k \models Inv$ and $(s_k, s'_{k+1}) \models Tr$, but $s'_{k+1} \not\models Inv'$.

We also recall a few basic definitions from linear algebra that we use.

Given a vector space $\mathbf{V}$, over a field $\mathbf{F}$ with its additive identity denoted as 0, its *basis* $\mathbf{B} = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ is a minimal subset of $\mathbf{V}$ satisfying:

1. $\forall\, a_1, \ldots, a_n \in \mathbf{F}$, if $a_1\mathbf{v}_1 + \cdots + a_n\mathbf{v}_n = 0$, then $a_1 = 0, \ldots, a_n = 0$.
2. $\forall\, \mathbf{v} \in \mathbf{V}, \exists\, a_1, \ldots, a_n \in \mathbf{F}$ such that $\mathbf{v} = a_1\mathbf{v}_1 + \cdots + a_n\mathbf{v}_n$.

The cardinality of $\mathbf{B}$ is called *dimension* of $\mathbf{V}$. For a matrix $\mathbf{A}$, the dimension of the vector space generated by its columns is called its *rank*. The *nullspace* of a matrix $\mathbf{A}$ is a set of all vectors $\mathbf{v}$ such that $\mathbf{A}\mathbf{v} = 0$. The dimension of a matrix's nullspace is also called its *nullity*.

### 3.1 Syntax-Guided Invariant Synthesis

An important contribution of [13] is the automatic generation of production rules for the sampling grammar $G$, guided by the structure of encoding of $Init$, $Tr$ and $Bad$. Candidate invariants are guessed using these production rules, and then checked for invariance and safety using an SMT solver. The candidates sampled from $G$ are disjunctions of linear inequalities. The final invariant is assumed to be

**Algorithm 1.** FreqHorn: Syntax-guided invariant generation

**Input:** *Init*, *Tr*, *Bad* and $V$
**Output:** *lemmas*

1: $\mathcal{P} \leftarrow$ computeDistribution(*Init*, *Tr*, *Bad*)
2: $G \leftarrow$ constructGrammar($\mathcal{P}$)
3: $L \leftarrow \emptyset$ ▷ the set of lemmas
4: **while** $\bigwedge_{l \in L} l \wedge Bad(V)$ is SAT **do**
5: $\quad init \leftarrow false$, $consec \leftarrow false$
6: $\quad cand \leftarrow$ newCandidate($G$)
7: $\quad$ **if** $Init(V) \wedge \neg cand(V)$ is UNSAT **then** $init \leftarrow true$
8: $\quad$ **if** $cand(V) \bigwedge_{l \in L} l(V) \wedge Tr(V, V') \wedge \neg cand(V')$ is UNSAT **then** $consec \leftarrow true$
9: $\quad$ **if** $init \wedge consec$ **then** $L \leftarrow L \cup cand$
10: $\quad$ adjust($cand, G, \mathcal{P}$)
11: **return** $L$

a conjunction of these candidates, also called *lemmas*. I.e. $Inv \Leftrightarrow l_0 \wedge l_1 \wedge \cdots \wedge l_n$, where the lemmas $l_i \in G$.

A high level description of their technique is presented in Algorithm 1. The procedure computeDistribution computes a frequency distribution of arities of operations, program variables and constants used, from the *Init*, *Tr* and *Bad*. This distribution is used to construct production rules for the sampling grammar resulting in an initial grammar $G$ in the second step. After this step the algorithm enters a loop where candidate lemmas, as per the grammar $G$, are guessed and checked until a safe invariant is found. The SAT checks in lines 4, 7, and 8 are, respectively, the checks for safety, initiation, and consecution. If a candidate fails one of the last two checks, the grammar $G$ is adjusted so that syntactically similar candidates are not sampled immediately. Otherwise the candidate is added to the set of lemmas.

### 3.2 Bootstrapping and Batch Checking

The tool FreqHorn that implements Algorithm 1 outperforms other data-based tools. However, in a follow-up paper [12], Fedyukovich et al. mitigate two downsides of this technique, namely (i) the candidates being ignorant to the program semantics, and (ii) a useful candidate failing the inductiveness check, even though it is inductive relative to some other candidates that may get sampled in due course. They propose an improved algorithm (shown as Algorithm 2) that works in two phases: *bootstrapping* and *sampling*. During bootstrapping they add additional candidates obtained as interpolants, from proofs of bounded safety, as *seeds* (line 1). This adds semantically valuable candidates, unlike its predecessor where candidate sampling was purely syntactic. The seeds themselves may be safe invariants, or they may assist in constructing safe invariants in the sampling phase. The sampling phase works in a similar manner as before, except that the consecution check is done for a batch of candidates at once,

instead of a single candidate (line 12). This is to address the latter issue, i.e. to avoid rejecting candidates that are relatively inductive to other lemmas. This check is similar to the algorithm used in HOUDINI tool [14].

**Algorithm 2.** FREQHORN-2: Bootstrapping and Batch Checking

**Input:** *Init*, *Tr*, *Bad* and $V$
**Output:** *lemmas*

1: $candidates \leftarrow$ BOOTSTRAPINTERPOLANTS($Init$, $Tr$, $Bad$)
2: $\mathcal{P} \leftarrow$ COMPUTEDISTRIBUTION($Init$, $Tr$, $Bad$)
3: $G \leftarrow$ CONSTRUCTGRAMMAR($\mathcal{P}$)
4: $L \leftarrow \emptyset$
5: **while** $\bigwedge_{l \in L} l \wedge Bad(V)$ is SAT **do**
6:     **while** $|candidates| < BatchSize$ **do** ▷ for a pre-decided $BatchSize$
7:         $cand \leftarrow$ NEWCANDIDATE($G$)
8:         **if** $Init(V) \wedge \neg cand(V)$ is UNSAT **then**
9:             $candidates \leftarrow candidates \cup \{cand\}$
10:        **else** ADJUST($cand, G, \mathcal{P}$)
11:    **for** $cand \in candidates$ **do**
12:        **if** $\bigwedge_{c \in candidates} c \wedge \bigwedge_{l \in L} l \wedge Tr(V, V') \wedge \neg cand(V')$ is SAT **then**
13:            $candidates \leftarrow candidates \setminus \{cand\}$
14:            ADJUST($cand, G, \mathcal{P}$)
15:            $candidates.reset$ ▷ start the loop afresh
16:    **for** $cand \in candidates$ **do**
17:        $L \leftarrow L \cup \{cand\}$
18: **return** $L$

## 4 Combining Syntax and Behaviours

The semantic information added by interpolants in the bootstrapping phase of [12] certainly accelerates the task. However, we have seen that interpolants from bounded proofs may fail to capture certain behavioural facts. Making the sampling grammar richer is one solution, but without any guidance irrelevant candidates will become a bottleneck during the checking phase. We propose an enhancement to the semantic guidance – from candidates that are not available on surface, but can be discovered by analyzing behaviours. We also show how CTIs may be used to detect the need for conditional invariants and how this can be useful for a certain class of programs.

### 4.1 Behaviours

Recall the example in Fig. 1a, which needed, along with the inequality ($i \leq n+1$), an algebraic invariant ($2 * sum = i^2 - i$) which was not available from syntax.

We aim to discover lemmas such as these, by sampling candidates that have the following fixed degree polynomial equation form:

$$c_1 * m_1 + c_2 * m_2 + \cdots + c_n * m_n = 0$$

where $m_i = x_1^{k_1} \dots x_l^{k_l}$ are *monomials* and $c_i \in \mathbb{Q}$ are *coefficients*. The *degree* of a monomial is the sum $\sum_i k_i$, and the degree of a polynomial equation is the highest degree among its monomials. In our technique, we consider that $x_i$*'s* come from the set of variables $V$. For instance, $2 * sum - i^2 + i = 0$ is a polynomial equation of degree 2 for the program in the variables *sum* and $i$, with the monomials *sum*, $i^2$ and $i$. One may sample such candidate lemmas by guessing the monomials and their coefficients. However, the probability of obtaining a poor candidate is very high, resulting in a number of expensive checks. Instead, we rely on the following theorem from [24] to discover them.

**Theorem 1.** *If an invariant is a conjunction of k polynomial equations each of degree d and nullity of A is k, where A is a data matrix, then any basis for nullspace of A forms an invariant.*

A *data matrix* is a matrix of values of monomials up to degree $d$. Each row of the data matrix corresponds to values of monomials computed by using concrete values of corresponding variables from $V$. The concrete values of variables are obtained from behaviours. For example, Table 1 shows a data matrix computed with $d = 2$ for the program in Fig. 1a. The first three columns shows the values of variables $i$, $n$ and *sum* at loop head for five iterations of the loop. The value of $n$ is a non-deterministic assignment as it is not initialized in the program.

**Table 1.** Monomials up to degree 2 for the program in Fig. 1a

| $i$ | $n$ | $sum$ | $i^2$ | $i * n$ | $i * sum$ | $n^2$ | $n * sum$ | $sum^2$ | $const$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 36 | 0 | 1 | 36 | 0 | 1296 | 0 | 0 | 1 |
| 2 | 36 | 1 | 4 | 72 | 2 | 1296 | 36 | 1 | 1 |
| 3 | 36 | 3 | 9 | 108 | 9 | 1296 | 108 | 9 | 1 |
| 4 | 36 | 6 | 16 | 144 | 24 | 1296 | 216 | 36 | 1 |
| 5 | 36 | 10 | 25 | 180 | 50 | 1296 | 360 | 100 | 1 |

The central idea of Theorem 1 is that if invariants are assumed to be polynomial equations of degree $d$ over $V$, then one can obtain coefficients of these equations using the data matrix. This is because the values from data matrix, when substituted for monomials, gives us a system of linear equations in $c_1 \dots c_n$. The solutions to these equations form a vector space, and the basis of this vector space gives coefficients of polynomial equations. The basis of a system of linear equations can be computed by the well-known Gauss-Jordan elimination algorithm. The computational complexity of this algorithm is $\mathcal{O}(m^2 n)$ for an $m \times n$ matrix.

**Algorithm 3.** getAlgebraicCandidates: Learning algebraic invariants from behaviours

1: **procedure** getAlgebraicCandidates(*behaviours*)
2: $candidates \leftarrow \emptyset$
3: $M \leftarrow$ computeMonomials($behaviours$, $d_{poly}$)
4: $B \leftarrow$ GaussJordan($M$)
5: **for** $coefficients \in B$ **do**
6: $candidates \leftarrow candidates \cup$ ConstructPolynomial($coefficients$, $d_{poly}$)
7: **return** *candidates*

Algorithm 3 presents the procedure getAlgebraicCandidates, which takes behaviours as input, provided either by user or computed using an SMT solver, and returns a set of candidates. It starts with computing the values of all monomials up to pre-decided degree $d_{poly}$ using behaviours, and stores them in a data matrix $M$. The basis of nullspace of this data matrix $B$ is computed using Gauss-Jordan algorithm in the next step. Each vector of the basis is used as coefficients $c_1 \dots c_n$ to construct a polynomial equation following Theorem 1. Thus computed polynomial equations are returned as candidates. For instance, if we use the values from Table 1 we get basis $B = \{(0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ -36),\ (-1\ 0\ -2\ 1\ 0\ 0\ 0\ 0\ 0\ 0),\ (-36\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0),$ $(0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ -1296),\ (0\ 0\ -36\ 0\ 0\ 0\ 0\ 1\ 0\ 0)\}$. When they are substituted as coefficients we get the following polynomials as candidates: $n - 36 = 0$, $-i - 2 * sum + i^2 = 0$, $-36 * i + i * n = 0$, $n * n - 1296 = 0$ and $-36 * sum + sum * n = 0$. Among these candidates $-i - 2 * sum + i^2 = 0$ passes both initiation and consecution checks.

### 4.2 Counterexamples to Induction (CTIs)

In this subsection we present a heuristic to solve programs like Fig. 1b. We observe that invariants of such programs may have different lemmas that hold in different blocks of the loop, i.e. the lemmas may only be conditional. Hence, the technique presented in previous section will not be able to generate necessary invariants. To address this, we first need to check whether a given program requires conditional invariants. A naive solution is to traverse the transition relation $Tr$ and look for *if* conditions in loops. However, this will not work always and may even miss simple invariants. Consider the program shown in Fig. 2 which is taken from the benchmarks of FreqHorn-2. Even though this program has an *if* condition, a simple assertion $i + j = n$ itself is a safe invariant. This invariant can be discovered from the technique mentioned in Sect. 4.1.

We call $Tr$, a *polynomial relation* if it is possible to represent all variables from $V'$ in a fixed degree polynomial equation over $V$. This polynomial is of the form:

$$f(x_i') = c_1 * m_1 + c_2 * m_2 + \dots + c_n * m_n$$

where $x_i' \in V'$, $m_i$ are all possible monomials over $V$ up to a certain degree $d$ and $c_i \in \mathbb{Q}$ are coefficients.

```
main() {
  int i=0,j=0,k=100,n=0,b;
  assume(b == 0 || b == 1);
  while(n < 2*k) {
    if (b == 0) {
      i++; b = 1;
    } else {
      j++; b = 0;
    }
    n++;
  }
  assert(i+j == n);
}
```

**Fig. 2.** A benchmark program from [12]

Our idea is that if $Tr$ is not a polynomial relation then the loop requires conditional invariants. For example, consider the program in Fig. 1b. Both $x'$ and $y'$ are getting modified by two relations: $y' = y, x' = x + 1, \mathtt{LRG}' = \mathtt{LRG}$ and $y' = y + 1, x' = x + 1, \mathtt{LRG}' = \mathtt{LRG}$, which are from *if* and *else* blocks respectively. It is not possible to find a polynomial function for $y'$. Hence, we need an implication. Whereas if we consider the example from Fig. 2, all the variables in $V'$ can be represented by the following polynomial equations over $V$: $n' = n + 1$, $b' = 1 - b$, $i' = i + 1 - b$ and $j' = j + b$. Hence, this program does not require conditional invariant.

One approach to check if $Tr$ is a polynomial relation is to encode a constraint whose satisfiability implies that $V'$ can be represented by $V$. However, this approach will not scale with larger degrees and variables. We propose an efficient technique by using concepts of linear algebra and CTIs. Recall that the models corresponding to the consecution check failure are referred to as CTIs. In a nutshell, we try to look for coefficients $c_1 \dots c_n$ that are consistent with CTIs. We substitute values for $f(x'_i)$ and $m_i$ in polynomial equations by using values of $V'$ and $V$ respectively from CTIs. If there are $l$ CTIs this results in $l$ linear equations over $c_1 \dots c_n$. These equations can be represented in matrix form as $\mathbf{M}\boldsymbol{c} = \boldsymbol{f}_{\boldsymbol{x}'_i}$, where $\mathbf{M}$ is the matrix of values for $m_i$, $\boldsymbol{c}^T = \big(c_1 \, \dots \, c_n\big)$ and ${\boldsymbol{f}_{\boldsymbol{x}'_i}}^T = \big(x'_{i_1} \, \dots \, x'_{i_l}\big)$. The following standard theorem from linear algebra [21] helps to determine if these equations have a solution for the $c_i$'s or not.

**Theorem 2.** *A system of linear equations is consistent if and only if the rank of the matrix of the system is equal to the rank of its augmented matrix.*

In our case the matrix of the system is $\mathbf{M}$ and the augmented matrix is $\mathbf{M}|\boldsymbol{f}_{\boldsymbol{x}'_i}$, i.e. $\mathbf{M}$ augmented with $\boldsymbol{f}_{\boldsymbol{x}'_i}$. As per Theorem 2 if $rank(\mathbf{M})$ and $rank((\mathbf{M}|\boldsymbol{f}_{\boldsymbol{x}'_i}))$ are not equal then it is not possible to have a solution for $c_i$.

The procedure CHECKFORIMPL is presented in Algorithm 4. It takes the *CTIs* as input. In the first step, it computes $\mathbf{M}$ using *CTIs* up to degree $d_{poly}$. It then checks for each variable $x'_i$ in $V'$ whether the rank of its augmented matrix

**Algorithm 4.** CHECKFORIMPL: Deciding the need for implications from CTIs

```
1: procedure CHECKFORIMPL(CTIs)
2:     M ← COMPUTEMONOMIALS(CTIs, d_poly)
3:     for x'_i ∈ V' do
4:         f_{x'_i} ← CTIs[x_i']
5:         M_aug ← AUGMENT(M, f_{x'_i})
6:         if RANK(M) ≠ RANK(M_aug) then
7:             return true
```

is equal to rank of the matrix $M$. If this is not the case for any of the variables, the procedure returns with the decision that implications will be sampled as candidates. The complexity of computing the rank of a $m \times n$ matrix is $\mathcal{O}(m^2 n)$.

We get the candidates for implication by sampling antecedents and consequents from different sampling grammar. The sampling grammar for antecedent is constructed by considering only conditions of *if* statements in $Tr$. This consideration ensures that candidates for antecedents are sampled from the syntax that is causing implications. For consequent, the $Init$, $Tr$ and $Bad$ is considered, like in the FREQHORN algorithm.

A class of programs that this technique can successfully address is the one with *multi-phase* loops, as mentioned in [23]. *Splitter-predicates* are used to identify the different phases of the loop, based on when these predicates, or their negations hold. A loop may start its iteration in one of the phases and then move to new phases as it progresses. Owing to these, such programs require disjunctive invariants. The solution presented in [23] is to compute invariants for each phase separately. The splitter-predicates are either conditions of *if* statements, or their weakest preconditions w.r.t. statements in the loop. Similarly, we derive antecedents from a grammar constructed using the encoding of conditions. In principle, this enables our technique to work for programs where splitter-predicates helps in discovering disjunctive invariants; in fact, even in cases when the phases are not syntactically evident.

### 4.3 Combining Behaviours and CTIs

Algorithm 5 shows the complete algorithm, which combines the techniques illustrated above. We skip the description of steps that are already explained in Sects. 3.1 and 3.2. The algorithm begins by generating behaviours using an SMT solver, if they are not provided as input. This is done by unwinding $Tr$ to a certain bound and then computing models for $V$ at each unwinding. These behaviours are used to compute algebraic candidate lemmas as described earlier. The next two steps create a frequency distribution $\mathcal{P}$ using $Init$, $Tr$ and $Bad$, and a grammar $G$ using $\mathcal{P}$. The grammar $G$ is used to get candidates when algebraic lemmas are not found, or are insufficient to prove the property. We create a new frequency distribution $\mathcal{P}_a$ based on conditions in loop body and its negations, and a grammar $G_a$ using $\mathcal{P}_a$. These are used to sample antecedents, if required.

**Algorithm 5.** ELABOR: Learning from Behaviours and CTIs

**Input:** *Init*, *Tr*, *Bad* and *V*
**Output:** *lemmas*

1: $behaviours \leftarrow \textsc{execute}(Init, Tr, Bad)$
2: $candidates \leftarrow \textsc{getAlgebraicCandidates}(behaviours)$
3: $\mathcal{P} \leftarrow \textsc{computeDistribution}(Init, Tr, Bad)$
4: $G \leftarrow \textsc{constructGrammar}(\mathcal{P})$
5: $\mathcal{P}_a \leftarrow \textsc{computeDistribution}(Tr_{conds})$
6: $G_a \leftarrow \textsc{constructGrammar}(\mathcal{P}_a)$
7: $L \leftarrow \emptyset$ ▷ the set of lemmas
8: $disjunct \leftarrow false$
9: **while** $\bigwedge_{l \in L} l(V) \wedge Bad(V)$ is SAT **do**
10: **if** $\neg disjunct$ **then** $disjunct \leftarrow \textsc{checkForImpl}(CTIs)$
11: **if** $disjunct$ **then** $antecedent \leftarrow \textsc{newCandidate}(G_a)$
12: **while** $|candidates| < BatchSize$ **do** ▷ for a pre-decided $BatchSize$
13: $cand \leftarrow \textsc{newCandidate}(G)$
14: **if** $init \leftarrow Init(V) \wedge \neg cand(V)$ is UNSAT **then**
15: **if** $disjunct$ **then** $candidates \leftarrow candidates \cup \{antecedent \Rightarrow cand\}$
16: **else** $candidates \leftarrow candidates \cup \{cand\}$
17: **else** $\textsc{adjust}(cand, G, \mathcal{P})$
18: **for** $cand \in candidates$ **do**
19: **if** $\bigwedge_{c \in candidates} c(V) \bigwedge_{l \in L} l(V) \wedge Tr(V, V') \wedge \neg cand(V')$ is SAT **then**
20: $candidates \leftarrow candidates \setminus \{cand\}$
21: $\textsc{adjust}(cand, G, \mathcal{P})$
22: $CTIs \leftarrow CTIs \cup \{\textsc{getModel}(V)\} \cup \{\textsc{getModel}(V')\}$
23: $candidates.reset$ ▷ start the loop afresh
24: **if** $disjunct \wedge |candidates| > 0$ **then** $\textsc{adjust}(antecedent, G_a, \mathcal{P}_a)$
25: **for** $cand \in candidates$ **do** $L \leftarrow L \cup \{cand\}$
26: **return** $L$

The algorithm proceeds to sample and check candidates in a loop, similar to FREQHORN-2. This loop is modified to check if sampling implications is necessary. In the beginning of each iteration, the procedure CHECKFORIMPL is called. If it suggests that an implication is needed then we get them by sampling antecedents from $G_a$, and consequents from $G$. This is followed by a check for inductiveness and safety. If the consecution check fails, we store the corresponding models (CTIs) in a matrix. This check is unmodified from FREQHORN-2. The grammar $G_a$ is adjusted when the inductiveness check passes for candidates with existing antecedents, to ensure different antecedents for new candidates. In our experiments, we unwound the transition relation up to bound of 10 for getting the behaviours. We also put a threshold on the number of CTIs collected before checking the need for implications, and bounded the degree of polynomials to 2.

## 5 Experiments

The aim of our experiments was to evaluate the effectiveness of our ideas. In particular, we were looking to answer the following questions:

1. Does the proposed strategy, of adding behaviours and implications, help improve the performance of FreqHorn-2 - (a) w.r.t. the number of benchmarks solved, and (b) w.r.t. the average time taken to solve a benchmark?
2. Does our CTI-based heuristic hamper the tool's performance in cases when a conditional invariant may not necessarily be required?

*Implementation and Set-Up.* We have implemented our ideas as an extension of FreqHorn-2. We have named it as ELABor, which stands for Efficiently Learning from Appearance and Behaviour. Like its predecessors, the input program and the property are assumed to be in the form of linear constrained Horn clauses. Additionally, loop head states observed from behaviours may be provided as input. If that is missing, ELABor automatically generates behavioural data by unrolling the input program to a certain bound and evaluating models for program variables at loop head using Z3. Candidate lemmas are computed from loop head states using the Gauss-Jordan algorithm. For matrix operations, we use Armadillo [20], a C++ library for linear algebra.

**Table 2.** Comparison on programs for which FreqHorn-2 timed out in more than half of the runs; the values show the mean execution time taken (in *seconds*)

| Program | FreqHorn-2 | ELABor | Reason |
|---|---|---|---|
| exact_iters_5 | $\infty$ | 0.7 | B |
| s_mutants_22 | $\infty$ | 24.6 | B |
| s_mutants_21 | $\infty$ | 0.7 | B |
| dillig22-6 | $\infty$ | 229 | I |
| dillig22-4 | $\infty$ | 7.2 | B |
| dillig22-3 | $\infty$ | 13.6 | B |
| nonlin_gauss_sum | $\infty$ | 49.9 | B |
| abdu_03 | 312.3 | 0.9 | B |
| exact_iters_4 | 272.2 | 0.7 | B |
| menlo_park_term_orig | 373.2 | 188.1 | B |
| s_mutants_20 | 252.6 | 2.2 | B |
| dillig18 | 344.4 | 45.8 | I |
| dillig22-5 | 224.7 | 13.1 | B |
| phases_true-unreach-call1 | 445.7 | 256.9 | I |
| gj2007_true-unreach-call | 342.8 | 150.1 | I |
| half_true_modif | 476.6 | 0.7 | B |

We experimented with the benchmarks that are provided with FreqHorn-2. These benchmarks have been taken from SV-COMP and the literature. There were a total of 172 safe programs, of which we excluded 6 programs that had nested conditions and function call which our tool does not support. We only compare ELABor with its predecessor FreqHorn-2, as the latter has been shown to outperform other data-driven tools on these benchmarks [12]. Our experiments were performed by running 4 tasks in parallel, on a system with 16 cores of 2.40 GHz speed each, and total memory of 20 GB. We used a timeout of 600 s for each task. The tasks were run 10 times each, on both the tools, to handle the stochastic nature of the tools. We ran FreqHorn-2 with the interpolants option and a bound of 3. ELABor, on the other hand, was run without the interpolants option (it is turned off by default), as it might be unnecessary to employ multiple ways of getting behavioural candidates. The artifact submitted with this paper contains both the tools, the benchmarks, and the instructions and scripts to reproduce the results.

*Results.* Of the 166 benchmarks that we used, FreqHorn-2 could not generate safe invariants for 13 programs in any of the runs. Apart from these, there were 11 programs which FreqHorn-2 missed on more than half of the runs. Of these 24 programs in total, ELABor worked for 16 programs almost always (it solved 14 in all 10 runs and for 2 more in 8 runs out of 10). Table 2 lists these programs, along with the mean execution time (over successful runs) of the tools. The symbol $\infty$ indicates a time out in all runs. The last column shows the reason behind ELABor discovering a safe invariant: ‘B’ indicates the enhancement of combining behaviours, and ‘I’ indicates the one of mining implications.

W.r.t. the average execution time, we say that one of the tool did better than the other only if (i) the faster tool took less than half the time that the other one, or (ii) the time difference was more than 100 s. ELABor outperformed its predecessor on 31 programs, while for 8 programs it is FreqHorn-2 that worked better. The scatter plot on the left in Fig. 3 compares the time taken

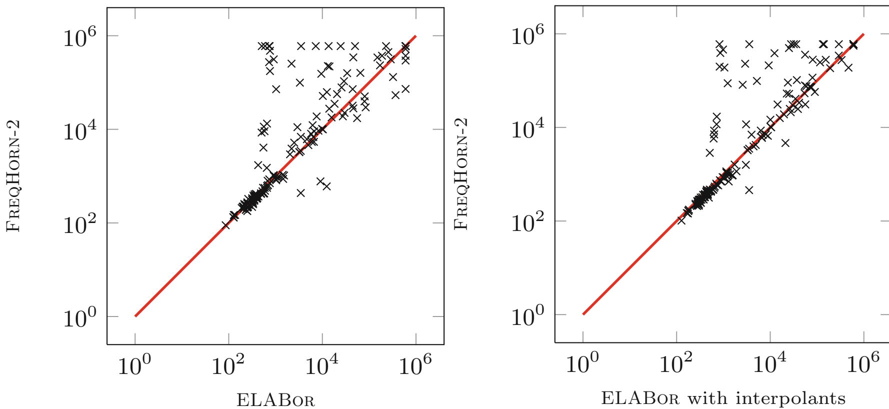


**Fig. 3.** Scatter plots comparing execution time (in ms) of the tools

(in *milliseconds*) by ELABOR (along the x-axis) and FREQHORN-2 (along the y-axis). The slack for those 8 programs was mostly due to lemmas that the interpolation engine provided upfront to FREQHORN-2, while we took a bit longer in discovering them. We confirmed this by running ELABOR with the interpolants option—now there were only 3 programs for which FREQHORN-2 outperformed us. However, the additional time taken by the interpolation engine gets reflected as points that were above the line, drifting closer to the line in the scatter plot on the right in Fig. 3.

## 6 Conclusion and Future Work

This work builds upon a recently proposed idea of inferring inductive invariants using a guess-and-check method, by sampling predicates, and its mutants, from the input program source [13]. In addition to obtaining a seed set of candidates from interpolation proofs of bounded safety [12], we show that a similar seed set can be obtained by analyzing behaviours of the program. We also propose a method to overcome a limitation of this guess-and-check method w.r.t. disjunctive invariants, by looking for conditional invariants in the form of implications.

There are a number of interesting directions in which this work may be extended. In particular, it would be worthwhile to explore the following:

– *Guidance from counterexamples to adequacy.* The current approach to deal with the inadequacy of discovered lemmas is to simply look for more. It would be useful to see how the property and the lemmas may together guide the search for additional facts, e.g. using ideas from abductive inference [10].
– *Refining candidates with disjunctions.* In the present algorithm, a disjunctive invariant candidate is either inductive, or is entirely useless. A method to find out which disjunct needs refinement, and how may it be refined, would certainly be helpful.
– *Choosing between syntax and behaviours.* Can there be some guidance in deciding, at every stage of the algorithm, whether the missing lemmas are more likely to be found through a syntactic search, or a behavioural one?
– *Machine learning to refine sampling.* Can machine learning technique be helpful in deciding when and how to nudge the probability distribution of candidates sampling?

We plan to investigate some of these research directions as we go ahead.

## References

1. Albarghouthi, A., McMillan, K.L.: Beautiful interpolants. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 313–329. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_22
2. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7

3. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
4. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_39
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM (1977)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1979, pp. 269–282. ACM, New York (1979). https://doi.org/10.1145/567752.567778
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1978, pp. 84–96. ACM, New York (1978). http://doi.acm.org/10.1145/512760.512770
8. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symbolic Logic **22**(3), 269–285 (1957). https://projecteuclid.org:443/euclid.jsl/1183732824
9. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
10. Dillig, I.: Abductive inference and its applications in program analysis, verification, and synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, 27–30 September 2015, p. 4 (2015)
11. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: Proceedings of the 22nd International Conference on Software Engineering, pp. 449–458. ACM (2000)
12. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 251–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_14
13. Fedyukovich, G., Kaufman, S.J., Bodík, R.: Sampling invariants from frequency distributions. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 100–107 (2017)
14. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45251-6_29
15. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 499–512. ACM, New York (2016)
16. Gupta, A., Rybalchenko, A.: InvGen: an efficient invariant generator. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_48
17. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13

18. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to discover polynomial and array invariants. In: Proceedings of the 34th International Conference on Software Engineering, pp. 683–693. IEEE Press (2012)
19. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, pp. 614–630. ACM, New York (2016)
20. Sanderson, C., Curtin, R.: Armadillo: a template-based C++ library for linear algebra. J. Open Source Softw. (2016)
21. Shafarevich, I.R., Remizov, A.O.: Linear Algebra and Geometry. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-30994-6
22. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. Form. Methods Syst Des. **48**(3), 235–256 (2016)
23. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 703–719. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_57
24. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 574–592. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_31