Solving Constrained Horn Clauses Using Syntax and Data

Grigory Fedyukovich*, Sumanth Prabhu†, Kumar Madhukar†, Aarti Gupta*
*Princeton University, Princeton, USA {grigoryf, aartig}@cs.princeton.edu
†TCS Research, Pune, India {sumanth.prabhu, kumar.madhukar}@tcs.com

***Abstract*—A Constrained Horn Clause (CHC) is a logical implication involving unknown predicates. Systems of CHCs are widely used to verify programs with arbitrary loop structures: interpretations of unknown predicates, which make every CHC in the system true, represent the program's inductive invariants. In order to find such solutions, we propose an algorithm based on Syntax-Guided Synthesis. For each unknown predicate, it generates a formal grammar from all relevant parts of the CHC system (i.e., *using syntax*). Grammars are further enriched by predicates and constants guessed from models of various unrollings of the CHC system (i.e., *using data*). We propose an iterative approach to *guess and check* candidates for multiple unknown predicates. At each iteration, only a candidate for one unknown predicate is sampled from its grammar, but then it gets propagated to candidates of the remaining unknowns through implications in the CHC system. Finally, an SMT solver is used to decide if the system of candidates contributes towards a solution or not. We present an evaluation of the algorithm on a range of benchmarks originating from program verification tasks and show that it is competitive with state-of-the-art in CHC solving.**

I. Introduction

To formally prove that a program meets a given safety specification, one needs to discover inductive invariants for every loop that appears in the program. Each loop invariant safely approximates the set of program states reachable before and after the corresponding loop. However, it is hard to synthesize them in isolation: if there is a program path through two loops, then invariants for these loops are likely related. For existing approaches to invariant synthesis, the increase in complexity of loop structure enlarges the search space drastically and lowers the chances of finding a suitable system of invariants.

We view the task of program verification as an instance of a more general problem of Constrained Horn Solving (e.g., [1], [2], [3], [4], [5], [6]). It takes as input a set of logical implications, called Constrained Horn Clauses (CHCs), over a set of unknown predicates, and aims at either finding a suitable interpretation for all predicates, that makes all implications true or showing that no such interpretation exists. Therefore, a conventional formulation of the invariant synthesis task for a transition system is an instance of the CHC task itself, which involves only one unknown predicate.

In this work, we present an algorithm for solving CHC tasks of arbitrary structure. It is based on a recently proposed solution for the CHC task for transition systems [7], [8], [9]; and it relies on a paradigm of Syntax-Guided Synthesis (SyGuS) [10]. In our context, each unknown predicate of the CHC system gets its own formal grammar that encodes the search space for a solution. Then, candidate formulas are sampled from the corresponding grammars and substituted in the CHC system, and the resulting formulas are checked by a Satisfiability Modulo Theories (SMT) solver for validity.

Our central idea behind the grammar construction is to use both syntax and data. In particular, this process relies on 1) pre-computed predicates obtained by parsing the interpreted parts of the CHC system, and 2) pre-computed predicates and constants synthesized from various traces (i.e., models of unrollings) of the CHC system. With these ingredients at hand, a single grammar per unknown predicate is created. By construction, it describes all the pre-computed predicates and possibly more. The use of syntax and data to obtain grammars are complementary to one another. Using syntax makes a number of useful candidates readily available that may be computationally expensive to derive from data. Whereas using data provides meaningful semantic candidates that the CHC system may be syntactically oblivious to.

However, the need to synthesize interpretations for multiple unknowns from multiple grammars produces a bottleneck: all candidates should be consistent with each other. That is, each pair of candidates for two unknowns that might appear in one CHC should make the CHC true. It is hard to enforce this requirement in practice: usually, either one or both candidates would be withdrawn and re-synthesized – this would make our algorithm inefficient. Instead, our algorithm exploits a more accurate approach to sampling: it generates a candidate for one unknown predicate at a time, and then propagates it to candidates of the remaining unknowns through all possible implications in the CHC system.

In comparison to existing approaches to CHC solving, our approach has several unique features. First, to the best of our knowledge, it exploits data more extensively than any other tool: it allows generating candidates on the fly, for which it gets models from various formulas obtained from CHCs. Furthermore, our algorithm does not necessarily consider candidates of a fixed predetermined shape: due to the use of grammars to learn candidates, the shape of pre-computed predicates (using syntax and data) is modified during the run of the algorithm. Compared to the algorithm of generating data candidates for transition systems [9], our algorithm explores unrollings modularly (i.e., for each loop in isolation), and thus it avoids SMT solving for potentially large formulas.

Finally, our approach does not involve a potentially expen-

sive fixed-point computation. Although our propagation routine is algorithmically similar to that in Generalized Property Directed Reachability [1], [4], we do not apply it recursively. Thus, our algorithm can never diverge while unwinding loops. The tradeoff is that our approach is not guaranteed to find an invariant, but it often does due to the rich grammars we generate, as shown in our experimental evaluation.

Our algorithm has been implemented on top of FREQHORN, a SyGuS-based CHC solver [7]. We have evaluated its effectiveness on a range of benchmarks originated from the verification tasks (i.e., programs with two or more loops and their safety specifications). Compared to state-of-the-art, our prototype exhibits a competitive performance and delivers results for most of the examples where the competing tools diverge. Our tool is particularly effective while discovering complex invariants over non-linear arithmetic.

The rest of the paper is structured as follows. Sect. II gives definitions, notation, and useful lemmas. Then, Sect. III presents our algorithm for a SyGuS-based CHC solver, driven by syntax, data and the candidate propagation. Finally, Sect. IV summarizes the evaluation, Sect. V outlines the related work, and Sect. VI concludes the paper.

II. PRELIMINARIES

For a given formula φ in a first-order theory $\mathcal{T}$, the Satisfiability Modulo Theories (SMT) task is to decide whether there is an assignment m of values to variables in φ that makes φ true. If every satisfying assignment to φ is also a satisfying assignment to some formula ψ, we write $\varphi \Longrightarrow \psi$. By $\top$ and $\bot$ we denote constants true and false, respectively. By $Expr$ we denote a space of all possible quantifier-free formulas in $\mathcal{T}$ and by $Vars$ a range of possible variables in $\mathcal{T}$.

A. Constrained Horn Clauses

Definition 1. *A* linear constrained Horn clause *(CHC) over a set of uninterpreted relation symbols $\mathcal{R}$ is a formula in first-order logic that has the form of one of three implications (called respectively a* fact*, an* inductive clause*, and a* query*):*

$$\varphi(\vec{x_1}) \Longrightarrow \boldsymbol{inv}_1(\vec{x_1})$$
$$\boldsymbol{inv}_1(\vec{x_1}) \wedge \varphi(\vec{x_1}, \vec{x_2}) \Longrightarrow \boldsymbol{inv}_2(\vec{x_2})$$
$$\boldsymbol{inv}_1(\vec{x_1}) \wedge \varphi(\vec{x_1}) \Longrightarrow \bot$$

where $\boldsymbol{inv}_1, \boldsymbol{inv}_2 \in \mathcal{R}$ are uninterpreted symbols, $\vec{x_1}, \vec{x_2}$ are vectors of variables, and φ, called a body*, is a fully interpreted formula (i.e., φ does not have applications of $\boldsymbol{inv}_1$ or $\boldsymbol{inv}_2$).*

For a CHC C, by $src(C)$ we denote an application of $\boldsymbol{inv} \in \mathcal{R}$ in the premise of C (if C is a fact, we write $src(C) \stackrel{\text{def}}{=} \top$). Similarly, by $dst(C)$ we denote an application of $\boldsymbol{inv} \in \mathcal{R}$ in the conclusion of C (if C is a query, we write $dst(C) \stackrel{\text{def}}{=} \bot$). We define functions rel and $args$, such that for each $\boldsymbol{inv}(\vec{x})$, $rel(\boldsymbol{inv}(\vec{x})) \stackrel{\text{def}}{=} \boldsymbol{inv}$ and $args(\boldsymbol{inv}(\vec{x})) \stackrel{\text{def}}{=} \vec{x}$. For a CHC C, by $body(C)$ we denote the body (i.e., φ) of C.

Example 1. *Fig. 1 shows a small C-like program[1] with three loops and its CHC-encoding. Each loop corresponds to one of the uninterpreted relation symbols $\mathcal{R} = \{\boldsymbol{inv}_1, \boldsymbol{inv}_2, \boldsymbol{inv}_3\}$. CHC **A** encodes the initial assignments to variables (including a nondeterministic choice for m and n) and assumptions over values of m and n. CHCs **B**, **D**, and **F** encode bodies of the first, the second, and the third loops, respectively. In order to represent a nondeterministic conditional in the first loop, CHC **B** contains the disjunction of encodings of both branches. CHCs **C** and **E** encode the fragments of the program between loops. Importantly, they include negations of the guards of preceding loops. Finally, CHC **G** encodes the negation of the assertion and the negation of the guard of the last loop.*

Linear CHCs can encode programs with nested loops, but cannot encode programs with non-inlined function calls[2]. For simplicity of presentation, the paper considers systems of CHCs that have only one query.

Definition 2. *Given a set of uninterpreted relation symbols $\mathcal{R}$ and a set S of CHCs over $\mathcal{R}$ we say that S is satisfiable if there exists an interpretation for each $\boldsymbol{inv} \in \mathcal{R}$ that makes all implications in S valid.*

Strictly speaking, an *interpretation* assigns to each symbol $\boldsymbol{inv} \in \mathcal{R}$ with arity n a relation over n-tuples. This relation can be represented by a formula φ over (at most) n free variables, denoted $fv(\varphi) \subseteq Vars$. In a specific application of $\boldsymbol{inv}$ to arguments $\vec{x}$, the free variables of φ are substituted by $\vec{x}$.

Example 2. *The system of CHCs in Fig. 1 is satisfiable (which means the program is safe), and a possible solution maps uninterpreted symbols to their interpretations as follows: $\boldsymbol{inv}_1 \mapsto x + y + n = m$, $\boldsymbol{inv}_2 \mapsto (x + y + n = m \wedge n = 0)$, and $\boldsymbol{inv}_3 \mapsto (x + y + n = m \wedge n = 0 \wedge x = 0)$.* □

B. Unrolling of CHCs

The following is built on ideas from Bounded Model Checking (BMC) [11] which aims at exploring finite length traces of programs.

Definition 3. *Given a system S of CHCs over $\mathcal{R}$, an unrolling of S of length k is a conjunction $\pi_{\langle C_0,\ldots,C_k\rangle} \stackrel{\text{def}}{=} \bigwedge_{0 \le i \le k} body(C_i)(\vec{x}_i, \vec{x_{i+1}})$, such that 1) each $C_i \in S$, 2) for each pair C_i and C_{i+1}, $rel(dst(C_i)) = rel(src(C_{i+1}))$, and variables of each $\vec{x_i}$ are shared only between $body(C_{i-1})(\vec{x}_{i-1}, \vec{x_i})$ and $body(C_i)(\vec{x}_i, \vec{x_{i+1}})$.*

Note that Def. 3 gives a more general notion of unrolling than it is customary for BMC. In particular, it allows the first step C_0 to be taken from an arbitrary place of the CHC system, i.e., C_0 is not necessarily a fact. We can consider unrollings, search for their models, and generate so called *behavioral*

[1]Because the presentation of our approach in terms of CHCs could be difficult to comprehend (e.g., notation is heavyweight in parts), here and throughout the paper we bring the analogy with program verification.

[2]We elaborate on the case with *nonlinear* CHCs in Sect. III-F.

```
int x = 0, y = 0;
int m = n = nondet();
assume (m >= 0);
while (n != 0) {
  n--;
  if (nondet()) x++;
  else y++;
}
while (x != 0) { m--; x--; }
while (y != 0) { m--; y--; }
assert (m == 0);
```

(A) $x' = 0 \wedge y' = 0 \wedge m' = n' \wedge m' \geq 0 \implies \boldsymbol{inv}_1(x',y',m',n')$

(B) $\boldsymbol{inv}_1(x,y,m,n) \wedge \neg(n=0) \wedge n' = n-1 \wedge m' = m \wedge \Big((x' = x+1 \wedge y' = y) \vee (x' = x \wedge y' = y+1)\Big) \implies \boldsymbol{inv}_1(x',y',m',n')$

(C) $\boldsymbol{inv}_1(x,y,m,n) \wedge (n=0) \wedge n' = n \wedge m' = m \wedge x' = x \wedge y' = y \implies \boldsymbol{inv}_2(x',y,'m',n')$

(D) $\boldsymbol{inv}_2(x,y,m,n) \wedge \neg(x=0) \wedge n' = n \wedge m' = m-1 \wedge x' = x-1 \wedge y' = y \implies \boldsymbol{inv}_2(x',y,'m',n')$

(E) $\boldsymbol{inv}_2(x,y,m,n) \wedge x = 0 \wedge n' = n \wedge m' = m \wedge x' = x \wedge y' = y \implies \boldsymbol{inv}_3(x',y,'m',n')$

(F) $\boldsymbol{inv}_3(x,y,m,n) \wedge \neg(y=0) \wedge n' = n \wedge m' = m-1 \wedge x' = x \wedge y' = y-1 \implies \boldsymbol{inv}_3(x',y,'m',n')$

(G) $\boldsymbol{inv}_3(x,y,m,n) \wedge y = 0 \wedge \neg(m=0) \implies \bot$

Fig. 1: Example program: (**left**) source code, and (**right**) its CHC encoding.

candidates for interpretations of unknown symbols that appear in the unrollings. We elaborate on this in Sect. III-C.

The following lemma provides yet another use of unrollings (for which C_0 is required to be a fact, and C_k – the query). We can enumerate various such unrollings and check satisfiability of the resulting formulas. Once a satisfiable formula is found, it does not make any sense to search for interpretations of any symbols in $\mathcal{R}$.

Lemma 1. *Given a system of CHCs S, let $\pi_{\langle C_0,\ldots,C_k\rangle}$ be one of its unrollings, such that C_0 is a fact, and C_k is the query. Then if $\pi_{\langle C_0,\ldots,C_k\rangle}$ is satisfiable then S is unsatisfiable.*

C. Polynomial behavioral candidates

We recall a few basic definitions from linear algebra that are needed for the generation of behavioral candidates. Given a vector space $\mathbf{V}$ over a field $\mathbf{F}$, 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 $\sum\limits_{1 \leq i \leq n} a_i \cdot \mathbf{v}_i = 0$, then $\bigwedge\limits_{1 \leq i \leq n} a_i = 0$.
2) $\forall \mathbf{v} \in \mathbf{V}, \exists a_1, \ldots, a_n \in \mathbf{F}$ such that $\mathbf{v} = \sum\limits_{1 \leq i \leq n} a_i \cdot \mathbf{v}_i$.

Consider the following fixed-degree polynomial equation:

$$c_1 \cdot \alpha_1 + c_2 \cdot \alpha_2 + \cdots + c_n \cdot \alpha_n = 0 \tag{1}$$

where $\alpha_i = x_1^{k_1} \cdots x_l^{k_l}$ are *monomials*, $c_i \in \mathbb{Q}$ are *coefficients*, and $x_1, \ldots, x_n$ are the variables from $Vars$. The *degree* of a monomial is the sum $\sum\limits_{1 \leq i \leq n} k_i$, and the degree of a polynomial equation is the highest degree among its monomials.

Given the values of variables from $Vars$, let a *data matrix* contain values of monomials for $Vars$ up to degree d. We rely on [12] to obtain equations of form (1) over $Vars$ using a data matrix. When these values are substituted for monomials, we get a system of linear equations over $c_1, \ldots, c_n$. Solutions to these equations form a vector space, and the basis of this vector space, computed by the well-known Gauss-Jordan elimination algorithm, gives coefficients of polynomial equations.

III. CHC Solving as Enumerative Search

In this section, we first give a general idea of our setup, then proceed to describe details that make the search procedure effective in practice and finally summarize everything in one algorithm.

A. Basic idea

A solution for a system of CHCs S with uninterpreted symbols $\mathcal{R}$ is a mapping ℓ from each symbol to a formula (written as $\ell : \mathcal{R} \to \mathit{Expr}$) that makes each CHC in S true. For a synthesis of ℓ, suppose that every $\boldsymbol{inv} \in \mathcal{R}$ has its grammar $G(\boldsymbol{inv})$ that describes a set of possible candidate formulas for $\boldsymbol{inv}$. In a naive scenario, in each iteration of a synthesis loop, a candidate formula for each $\boldsymbol{inv}$ gets sampled from $G(\boldsymbol{inv})$. All candidates are substituted in S, and if at least one of the implications is invalid then the entire system of candidates is *failing* and the synthesis loop iterates.

Clearly, this naive approach has a large search space. For example, if for the system of CHCs in Fig. 1, the candidate for all three uninterpreted symbols $\boldsymbol{inv}_1$, $\boldsymbol{inv}_2$, and $\boldsymbol{inv}_3$ is $x + y + n = m$, then all of them will be rejected because the candidate for $\boldsymbol{inv}_3$ is too coarse to prove the query (i.e., it needs to be conjoined with $x = 0 \wedge n = 0$). However, following [7] and [8], we can optimize the search by synthesizing conjunction-free lemmas for each $\boldsymbol{inv}_i$ separately and then by conjoining them together.

Definition 4. *For a system of CHCs S over $\mathcal{R}$ and a mapping $\ell : \mathcal{R} \to \mathit{Expr}$, we say that ℓ is a set of* lemmas *for S if it makes every CHC in S (except the query) valid.*

Example 3. *For the system of CHCs in Fig. 1, a mapping from all $\boldsymbol{inv}_1$, $\boldsymbol{inv}_2$, and $\boldsymbol{inv}_3$ to $x + y + n = m$ is one set of lemmas. A mapping $\boldsymbol{inv}_1 \mapsto \top$, $\boldsymbol{inv}_2 \mapsto n = 0$, and $\boldsymbol{inv}_3 \mapsto n = 0$ is another set of lemmas.* □

Lemma 2. *Given a system of CHCs S over $\mathcal{R}$ and two sets of lemmas ℓ_1 and ℓ_2, let a mapping $\ell_3 : \mathcal{R} \to \mathit{Expr}$ be such that for each $\boldsymbol{inv} \in \mathcal{R}$. $\ell_3(\boldsymbol{inv}) \stackrel{\text{def}}{=} \ell_1(\boldsymbol{inv}) \wedge \ell_2(\boldsymbol{inv})$. Then ℓ_3 is a set of lemmas for S.*

Our algorithm generates grammars based on a set of formulas, called *seeds* [8]. By construction, grammars should be able to describe all seeds and, as a side effect, also formulas which are syntactically close to seeds (called *mutants*). In the next two subsections, we outline the process of determining seeds automatically.

B. Collecting seeds from syntax

Given a system S of CHCs over $\mathcal{R}$, let $\boldsymbol{inv} \in \mathcal{R}$ be an uninterpreted symbol for which we wish to generate a

formal grammar. Perhaps, the most obvious sources of seeds are the bodies of CHCs in S that have applications of $\boldsymbol{inv}$. First, the body of a CHC C that has applications of $\boldsymbol{inv}$ is parsed, and clauses that contain only variables in $args(src(C))$ or only variables in $args(dst(C))$ are extracted. Then, the obtained formulas are rewritten in terms of variables $\vec{x} \subseteq Vars$ (practically, it is convenient to specify $\vec{x} \stackrel{\text{def}}{=} args(src(C'))$ of some CHC C' with $\boldsymbol{inv} = rel(src(C'))$).

Formally, for a formula φ in Conjunctive Normal Form, let $Cnjs(\varphi)$ be a set of its clauses. For sets of variables $\vec{x}$ and $\vec{y}$, let a set $F_{\vec{x},\vec{y}}(\varphi)$ be defined as $F_{\vec{x},\vec{y}}(\varphi) \stackrel{\text{def}}{=} \{\psi \mid \exists \phi \in Cnjs(\varphi)\,.\,\psi = \phi[\vec{x}/\vec{y}] \wedge fv(\phi) \subseteq \vec{x}\}$, where $\phi[\vec{x}/\vec{y}]$ denotes the result of substitutions of variables $\vec{x}$ in ϕ by variables $\vec{y}$. Thus, a set of seeds obtained from bodies of CHCs can be defined as follows.

Definition 5. *Given a system S of CHCs over $\mathcal{R}$, let $\boldsymbol{inv} \in \mathcal{R}$. Then*

$$SyntSeeds(\boldsymbol{inv})(\vec{x}) \stackrel{\text{def}}{=} \bigcup_{C \in S \text{ s.t. } rel(src(C)) = \boldsymbol{inv}} F_{args(src(C)),\vec{x}}(body(C)) \cup \bigcup_{C \in S \text{ s.t. } rel(dst(C)) = \boldsymbol{inv}} F_{args(dst(C)),\vec{x}}(body(C))$$

Example 4. *For the system of CHCs in Fig. 1, all four conjuncts of $body(\boldsymbol{A})$ give seeds $\{x = 0, y = 0, m = n, m \geq 0\}$ for $\boldsymbol{inv}_1$ and $\vec{x} = \langle x, y, m, n\rangle$. Furthermore, seeds $\neg(n = 0)$ and $n = 0$ are obtained from $body(\boldsymbol{B})$ and $body(\boldsymbol{C})$ respectively.* □

C. Collecting seeds from data

We bootstrap the grammar generation by seeds that are learned from the concrete values of variables produced while checking satisfiability of various unrollings of CHCs. If a CHC system S encodes some program, then an unrolling $\pi_{\langle C_0,\ldots,C_k\rangle}$ would correspond to a program trace whose sequentially executed statements are encoded by bodies of each C_i. If such an unrolling is unsatisfiable, then the corresponding program trace is infeasible. Otherwise, a model of the unrolling gives the concrete values of program variables at each execution step. We follow the ideas of the generation of behavioral seeds from models of program unrollings recently presented in [9].

The CHC task makes our setting different from [9], which considers CHCs with one uninterpreted relation symbol only. First, the presence of multiple symbols (and consequently, multiple loops) drastically complicates the creation of unrollings: the resulting formulas become too large and might become difficult for SMT solving. Second, it might be difficult to find a satisfiable unrolling since an unwinding number suitable for one loop might not be suitable for another loop. For example in Fig. 1, if the first and the second loops are unrolled n times, then to get a satisfiable unrolling, the third loop should be unrolled only zero times.

To overcome these two challenges, we propose to explore unrollings modularly: for each cycle in isolation. Recall that Def. 3 allows an unrolling $\pi_{\langle C_0,\ldots,C_k\rangle}$ to start from the body of some CHC C_0, where C_0 is not a fact. Thus, when determining behavioral seeds for some $\boldsymbol{inv}$ (e.g., when there is no fact in S with an application of $\boldsymbol{inv}$), we are free to consider any unrolling that starts from an arbitrary C_0, as long as $rel(dst(C_0)) = \boldsymbol{inv}$. In addition, we must ensure that $\boldsymbol{inv}$ is visited often enough, and the cycle has been terminated after C_k; otherwise, the collected data would not be sufficient for generating meaningful seeds. Def. 6 reflects these conditions formally.

Definition 6. *Given a system S of CHCs over $\mathcal{R}$, let $\boldsymbol{inv} \in \mathcal{R}$. If an unrolling $\pi_{\langle C_0,\ldots,C_k\rangle}$ is such that 1) $rel(src(C_0)) \neq \boldsymbol{inv}$, 2) $rel(dst(C_0)) = \boldsymbol{inv}$, 3) $rel(src(C_k)) = \boldsymbol{inv}$, and 4) $rel(dst(C_k)) \neq \boldsymbol{inv}$, and $\big|\{C_i \in \langle C_0,\ldots,C_k\rangle \text{ s.t. } rel(dst(C_i)) = \boldsymbol{inv}\}\big| = n$, we call it* modular *for $\boldsymbol{inv}$ and denote it $\pi^n_{\boldsymbol{inv}}$.*

For practical reasons, we are interested in minimal unrollings $\pi^n_{\boldsymbol{inv}}$ satisfying Def. 6 for some n and $\boldsymbol{inv} \in \mathcal{R}$. Then we obtain a model $m_{\boldsymbol{inv}}$ of $\pi^n_{\boldsymbol{inv}}$ and compute the data matrix using the values in $m^n_{\boldsymbol{inv}}$ for every $args(dst(C_i)) \in \langle C_0,\ldots,C_k\rangle$, such that $rel(dst(C_i)) = \boldsymbol{inv}$. This data matrix is then used to discover behavioral seeds for $\boldsymbol{inv}$, denoted $BehavSeeds(\boldsymbol{inv})$, that have the fixed-degree polynomial form (1) (recall Sect. II-C).

Example 5. *For CHCs in Fig. 1, $\pi^3_{\boldsymbol{inv}_1} \stackrel{\text{def}}{=} body(\boldsymbol{A})(\vec{x}_0) \wedge body(\boldsymbol{B})(\vec{x}_0,\vec{x_1}) \wedge body(\boldsymbol{B})(\vec{x}_1,\vec{x_2}) \wedge body(\boldsymbol{C})(\vec{x}_2,\vec{x}_3)$. We are interested in values of variables in $\vec{x}_0$, $\vec{x}_1$ and $\vec{x}_2$ (which correspond to program variables $\langle x, y, m, n\rangle$ at the beginning of each loop iteration) that make $\pi^3_{\boldsymbol{inv}_1}$ true. For instance:*

x	y	m	n
0	0	2	2
0	1	2	1
1	1	2	0

Using this data matrix, we can generate a set $BehavSeeds(\boldsymbol{inv})(\langle x, y, m, n\rangle) = \{x + y - m + n = 0\}$. It is easy to see that this equality holds for every row of the data matrix. □

D. Candidate propagation

In practice, seeds obtained using methods from Sect. III-B and Sect. III-C are often insufficient for generating rich enough formal grammars. Consequently, candidate formulas that are sampled from these grammars, are often insufficient for the discovery of useful lemmas. Recall a solution of the system of CHCs in Fig. 1, as shown in Ex. 2. It requires a set of lemmas that have conjunct $n = 0$ in interpretations of $\boldsymbol{inv}_2$ and $\boldsymbol{inv}_3$. However, the set of formulas shown in Ex. 4, can offer $n = 0$ only for $\boldsymbol{inv}_1$. Our main idea, described formally in the rest of this subsection, is to exploit that every CHC C with $rel(dst(C)) = \boldsymbol{inv}_2$ or $rel(dst(C)) = \boldsymbol{inv}_3$ has a clause $n' = n$ in its body (i.e., it merely reuses an old value of n), and thus the candidate $n = 0$ of $\boldsymbol{inv}_1$ can be pushed forward to become a candidate of $\boldsymbol{inv}_2$ and $\boldsymbol{inv}_3$.

Before propagating candidates, we need to ensure that they are *self-consistent* in the following sense.

Definition 7. *Given a system of CHCs S over $\mathcal{R}$ and a subset $\mathcal{R}' \subseteq \mathcal{R}$. A mapping $Cand : \mathcal{R}' \to Expr$ is called* self-consistent *if it makes every CHC in $S' \stackrel{\text{def}}{=} \{C \in S \mid (src(C) = \top \vee rel(src(C)) \in \mathcal{R}') \wedge rel(dst(C)) \in \mathcal{R}'\}$ valid.*

Clearly, if the candidates are not self-consistent, they cannot be extended to a set of lemmas. Alg. 1 gives a simple routine to check the self-consistency of candidates with respect to CHCs S' that have applications of symbols from $\mathcal{R}'$ only. If the algorithm finds an invalid CHC C, then it weakens the candidate for $rel(dst(C))$ and repeats the self-consistency check. Intuitively, if C has the form (2), then (3) is invalid.

$$\boldsymbol{inv}_i(\vec{x_i}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \implies \boldsymbol{inv}_j(\vec{x_j}) \quad (2)$$

$$Cand(\boldsymbol{inv}_i)(\vec{x_i}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \implies Cand(\boldsymbol{inv}_j)(\vec{x_j}) \quad (3)$$

Alg. 1 weakens $Cand(\boldsymbol{inv}_j)$ to $\top$, and thus (3) becomes trivially valid. Continuing such operation for other CHCs from S' guarantees discovering a self-consistent set of candidates. Note that Alg. 1 takes as additional input a set of formulas which are already proved to be lemmas (recall Def. 4).

Further reasoning of the candidate propagation, given self-consistent formulas $Cand$ for some $\mathcal{R}' \subseteq \mathcal{R}$, boils down to recursive post- and precondition inference: for any CHC in S that has the form (2), where $\boldsymbol{inv}_i \in \mathcal{R}'$ and $\boldsymbol{inv}_j \notin \mathcal{R}'$, we wish to identify a formula $Cand(\boldsymbol{inv}_j)$, such that (3) holds. Symmetrically, if $\boldsymbol{inv}_i \notin \mathcal{R}'$ and $\boldsymbol{inv}_j \in \mathcal{R}'$, we wish to identify a formula $Cand(\boldsymbol{inv}_i)$, such that again (3) holds.

The method of candidate propagation is based on quantifier elimination.

Definition 8. *Given a formula that has the form* (4)*.*

$$Cand(\boldsymbol{inv}_i)(\vec{x_i}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \implies \boldsymbol{inv}_j(\vec{x_j}) \quad (4)$$

Forward propagation *of $Cand(\boldsymbol{inv}_i)$ gives a formula $Cand(\boldsymbol{inv}_j)$, such that:*

$$Cand(\boldsymbol{inv}_j)(\vec{x_j}) \stackrel{\text{def}}{=} \exists \vec{x_i} \,.\, Cand(\boldsymbol{inv}_i)(\vec{x_i}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \quad (5)$$

Intuitively, if $\varphi(\vec{x_i}, \vec{x_j})$ encodes a transition from a program state $\vec{x_i}$ to a program state $\vec{x_j}$, then $Cand(\boldsymbol{inv}_j)(\vec{x_j})$ encodes a set of all possible states that are reachable from $Cand(\boldsymbol{inv}_i)(\vec{x_i})$ by making the $\varphi(\vec{x_i}, \vec{x_j})$ step. Note that in case $Cand(\boldsymbol{inv}_i)(\vec{x_i}) = \top$, propagating $\top$ can still give meaningful candidates, if e.g., the dst-arguments do not depend on the src-arguments. On the other hand, if $Cand(\boldsymbol{inv}_i)(\vec{x_i}) = \bot$, propagating $\bot$ ends up with $\bot$ again.

Note that the result of forward propagation (5) can be substituted back to implication (4) and make it true. Interestingly, the operation of *backward propagation* (defined below) does not have such property; and to enforce it, we should apply an additional weakening of the propagated formula.

Definition 9. *Given a formula that has the form* (6)*.*

$$\boldsymbol{inv}_i(\vec{x_i}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \implies Cand(\boldsymbol{inv}_j)(\vec{x_j}) \quad (6)$$

Backward propagation *of $Cand(\boldsymbol{inv}_j)$ gives a formula $Cand(\boldsymbol{inv}_i)$, such that:*

$$Cand(\boldsymbol{inv}_i)(\vec{x_i}) \stackrel{\text{def}}{=} \exists \vec{x_j} \,.\, Cand(\boldsymbol{inv}_j)(\vec{x_j}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \quad (7)$$

Algorithm 1: WEAKEN: establishing self-consistency.

Input: CHCs S' over $\mathcal{R}'$, set of candidates $Cand : \mathcal{R}' \to Expr$; learned $Lemmas : \mathcal{R} \to 2^{Expr}$
Output: weakened $Cand$

```
1 allGood ← ⊤;
2 for all C ∈ S' do
3     if ⋀_{ℓ ∈ Lemmas(rel(src(C)))} ℓ(args(src(C))) ∧
          Cand(rel(src(C)))(args(src(C))) ∧ body(C) ⟹
              Cand(rel(dst(C)))(args(dst(C))) then
4         Cand(rel(dst(C))) ← ⊤;
5         allGood ← ⊥;
6         break;
7 if allGood then return Cand;
8 else return WEAKEN(Cand, R', S', Lemmas);
```

Algorithm 2: EXTEND: recursive propagation.

Input: CHCs S over $\mathcal{R}$; $\mathcal{R}' \subseteq \mathcal{R}$, set of candidates $Cand : \mathcal{R}' \to Expr$; learned $Lemmas : \mathcal{R} \to 2^{Expr}$
Output: $res \in \{\top, \bot\}$, extended $Cand$

```
1  Cand ← WEAKEN(Cand, R', S', Lemmas);
2  if ∀inv ∈ R' . Cand(inv) = ⊤ then return ⟨⊥, _⟩;
3  for all C ∈ S s.t. rel(src(C)) ∈ R' and rel(dst(C)) ∉ R' do
4      Cand(rel(dst(C))) ← PROPAGATEFORWARD(C, Cand);
5      ⟨positive, Cand⟩ ←
           EXTEND(S, R' ∪ {rel(dst(C))}, Cand, Lemmas);
6      if ¬positive then return ⟨⊥, _⟩;
7  for all C ∈ S s.t. rel(dst(C)) ∈ R' and rel(src(C)) ∉ R' do
8      Cand(rel(src(C))) ← PROPAGATEBACKWARD(C, Cand);
9      ⟨positive, Cand⟩ ←
           EXTEND(S, R' ∪ {rel(src(C))}, Cand, Lemmas);
10     if ¬positive then return ⟨⊥, _⟩;
11 return ⟨⊤, Cand⟩;
```

Both forward and backward propagation can be applied recursively for any set of candidates $Cand$ and a subset $\mathcal{R}' \subseteq \mathcal{R}$. This is shown formally in Alg. 2. After establishing the self-consistency of candidates (line 1), Alg. 2 extends $Cand$ by adding *inferred* candidates using forward propagation (line 4) for all CHCs C that have $rel(src(C)) \in \mathcal{R}'$ and $rel(dst(C)) \in \mathcal{R} \setminus \mathcal{R}'$, and *inferred* candidates using backward propagation (line 8) for all CHCs C that have $rel(dst(C)) \in \mathcal{R}'$ and $rel(src(C)) \in \mathcal{R} \setminus \mathcal{R}'$. Each round of propagation enlarges the set of symbols annotated by candidates $\mathcal{R}'$ as well as $Cand$, and Alg. 2 is called recursively (lines 5 and 9). If $\mathcal{R}' = \mathcal{R}$ then it is enough to check self-consistency of $Cand$ (and weaken it if needed) before returning $Cand$ as a set of lemmas.

Theorem 1. *Assuming termination of the quantifier elimination procedure and termination of each implication check, Alg. 2 always terminates.*

For theories which do not admit a terminating quantifier-elimination procedure, Alg. 2 can be safely modified by replacing the results of calling the propagation methods on lines 4 and 8 by constant $\top$.

Algorithm 3: SOLVECHCS: overall algorithm.

Input: CHCs S over $\mathcal{R}$
Output: $res \in \{\text{SAT}, \text{UNKNOWN}\}$, $Lemmas : \mathcal{R} \to 2^{Expr}$

1 **for all** $\boldsymbol{inv} \in \mathcal{R}$ **do**
2 $\quad Seeds \leftarrow SyntSeeds(\boldsymbol{inv}) \cup BehavSeeds(\boldsymbol{inv})$;
3 $\quad G(\boldsymbol{inv}) \leftarrow \text{GETGRAMMAR}(Seeds)$;
4 $\quad Lemmas(\boldsymbol{inv}) \leftarrow \varnothing$;
5 **while** $\forall C \in S\,.\,(dst(C) = \bot) \implies \Big(\bigwedge_{\ell \in Lemmas(rel(src(C)))} \ell(args(src(C))) \wedge body(C) \implies \bot \Big)$ **do**
6 $\quad$ **if** $\forall \boldsymbol{inv} \in \mathcal{R}\,.\,\text{ALLBLOCKED}(G(\boldsymbol{inv}))$ **then**
7 $\quad\quad$ **return** $\langle \text{UNKNOWN}, \varnothing \rangle$;
8 $\quad \boldsymbol{inv} \leftarrow \text{PICKRELATIONALSYMBOL}(\mathcal{R})$;
9 $\quad Cand(\boldsymbol{inv}) \leftarrow \text{SAMPLE}(G(\boldsymbol{inv}))$;
10 $\quad \langle positive, Cand \rangle \leftarrow \text{EXTEND}(S, \{\boldsymbol{inv}\}, Cand, Lemmas)$;
11 $\quad$ **for all** $\boldsymbol{inv} \in \mathcal{R}$ **do**
12 $\quad\quad$ **if** *positive* **then**
13 $\quad\quad\quad Lemmas(\boldsymbol{inv}) \leftarrow Lemmas(\boldsymbol{inv}) \cup \{Cand(\boldsymbol{inv})\}$;
14 $\quad\quad G(\boldsymbol{inv}) \leftarrow \text{BLOCK}(G(\boldsymbol{inv}), Cand(\boldsymbol{inv}), positive)$;
15 **return** $\langle \text{SAT}, Lemmas \rangle$;

E. Core algorithm

Our main contribution is an effective search strategy for a solution of a given system of CHCs S over a set of uninterpreted symbols $\mathcal{R}$. The search is over a set of candidate formulas for each $\boldsymbol{inv} \in \mathcal{R}$ which is described by a formal grammar $G(\boldsymbol{inv})$. In this section, we instantiate the setup outlined in Sect. III-A by the components that make the entire procedure practical. The pseudocode of the algorithm is shown in Alg. 3.

Alg. 3 starts by creating the sampling grammars $G(\boldsymbol{inv})$ for each $\boldsymbol{inv} \in \mathcal{R}$. Grammars are constructed automatically: first (line 2), by collecting *Seeds* as described in Sect. III-B and Sect. III-C; and then (line 3) by creating production rules that would be able to produce all *Seeds* recursively. We do not impose any restrictions on the implementation of this routine, and in practice, one could additionally add a normalization pass over all *Seeds* before processing them. Note that various unrollings, considered for constructing the behavior candidates, can be enhanced with the bodies of the query (and of other clauses if necessary) to be checked for the existence of counterexamples (recall Lemma 1). If no counterexamples are found, the algorithm starts *guessing and checking* candidate formulas $Cand(\boldsymbol{inv})$ for each $\boldsymbol{inv} \in \mathcal{R}$.

Simultaneous sampling from multiple grammars might lead to many iterations of Alg. 3. To be turned to a set of lemmas, each set of candidate formulas should be self-consistent. But if the candidates are sampled without taking into account any relationship among loops, the weakening by Alg. 1 might be too aggressive and might withdraw many good candidates. Instead, we propose to fix precisely one grammar (say, $G(\boldsymbol{inv})$ for some $\boldsymbol{inv} \in \mathcal{R}$) per iteration, to sample a candidate formula $Cand(\boldsymbol{inv})$ from $G(\boldsymbol{inv})$, and to propagate $Cand(\boldsymbol{inv})$ recursively to candidate formulas $Cand(\boldsymbol{inv}')$ for all $\boldsymbol{inv}' \in \mathcal{R}$ through all implications in S (lines 8-10).

In particular, at each iteration, Alg. 3 picks $\boldsymbol{inv} \in \mathcal{R}$ (in our implementation, we use Weak Topological Ordering [13], but any other heuristic can be used instead). Then the algorithm samples a formula $Cand(\boldsymbol{inv})$ – it could either be one of *Seeds* or a syntactically mutated formula. The goal now is to find candidate formulas for all other $\boldsymbol{inv}' \in \mathcal{R} \setminus \{\boldsymbol{inv}\}$ and to check all implications in CHCs. The algorithm performs inference of preconditions and postconditions using the routine described in Sect. III-D (Alg. 2).

Recall that Alg. 2 not only populates $Cand$ with candidate formulas for some symbols but also drops some unsuccessful candidate formulas due to weakening. Note that Alg. 1 implements a simple strategy, in which a candidate formula $Cand(\boldsymbol{inv}_j)$ can only be dropped to $\top$ – this helps when $Cand(\boldsymbol{inv}_j)$ is conjunction-free. However, in case $Cand(\boldsymbol{inv}_j)$ is conjunctive (which could be due to quantifier elimination), a more careful weakening (e.g., [14], [15] or [16]) can be used. In the worst-case scenario, weakening ends up with an empty candidate, which means that nothing was learned at this iteration, and a new candidate formula should be sampled.

In the case when a sequence of weakening-propagation calls has converged, the entire $Cand$ is learned as a lemma (line 13). The process is repeated until the conjunction of lemmas is strong enough to be a solution for the entire system (apply Lemma 2). Finally, for the progress of the algorithm, both failed and positive attempts are noted, and the algorithm ensures that the candidates are not sampled again in the future (line 14). If all candidates of all grammars are blocked, the algorithm terminates with an unknown result (line 6). The facts that each formal grammar admits only a finite number of candidates and that each candidate is considered only once enable us to prove the following theorem.

Theorem 2. *Alg. 3 always makes a finite number of iterations, and if it converges with* SAT*, the set of all learned lemmas constitutes a solution of the CHC system.*

Similarly to [8], the algorithm can be optimized by introducing *bootstrapping* and *sampling* stages, candidate batching and exploiting counterexamples-to-induction, and thus it can be effectively integrated with the elements of Generalized Property Directed Reachability (GPDR) [1], [4].

F. Extension to nonlinear CHCs

Definition 10. *A* nonlinear *CHC is a formula in first-order logic that has the form of one of three implications:*

$$\varphi(\vec{x_1}) \implies \boldsymbol{inv}_1(\vec{x_1})$$
$$\bigwedge_{0 \le i \le n} \boldsymbol{inv}_i(\vec{x_i}) \wedge \varphi(\vec{x_0}, \ldots, \vec{x_{n+1}}) \implies \boldsymbol{inv}_{n+1}(\vec{x_{n+1}})$$
$$\bigwedge_{0 \le i \le n} \boldsymbol{inv}_i(\vec{x_i}) \wedge \varphi(\vec{x_0}, \ldots, \vec{x_n}) \implies \bot$$

Our synthesis algorithm can be adapted to solve systems of nonlinear CHCs with limited backward propagation. The rest

of the components operate in the same way: each $\boldsymbol{inv} \in \mathcal{R}$ gets its grammar, and candidates are iteratively sampled from them.

In the future, we would like to discover ways of effective backward propagation for nonlinear CHCs. In particular, a variant of (6) for nonlinear CHCs might be as follows:

$$\boldsymbol{inv}_i(\vec{x_i}) \wedge \boldsymbol{inv}_j(\vec{x_j}) \wedge \varphi(\vec{x_i}, \vec{x_j}, \vec{x_k}) \implies Cand(\boldsymbol{inv}_k)(\vec{x_k})$$

Applying quantifier elimination, we get candidates for conjunctions $Cand(\boldsymbol{inv}_i) \wedge Cand(\boldsymbol{inv}_j)$, but not necessarily for individual conjuncts $Cand(\boldsymbol{inv}_i)$ and $Cand(\boldsymbol{inv}_j)$.

IV. Implementation and Evaluation

We have implemented the algorithm from Sect. III-E on top of our previous implementation FreqHorn[3]. The tool takes a system of CHCs, automatically performs its unrolling, searches for counterexamples (if any), generates behavioral candidates, propagates and weakens candidates. To eliminate quantifiers, FreqHorn uses the technique based on Model-Based Projections [17]. For solving SMT queries, it uses Z3 [18]. For matrix operations, FreqHorn uses Armadillo [19], a C++ library for linear algebra.

We evaluated FreqHorn on **101** satisfiable CHC-systems[4] taken from the literature on program verification (e.g. [20]) and crafted by ourselves. There are 81 systems of CHCs over the theories of linear (LIA) and 20 over nonlinear integer arithmetic (NIA). All systems have two or more uninterpreted relation symbols. Because our quantifier-elimination engine has limited support for NIA, we disabled candidate propagation for the cases when the body of corresponding CHCs contains nonlinear arithmetic. In such cases, we assigned $\top$ to the propagated candidates and performed the self-consistency checks. Thus, disabling candidate propagation did not lead to incorrect results.

Among the 101 benchmarks, FreqHorn was able to solve **81** within a timeout of 5 minutes: 65 over LIA, and 16 over NIA. The remaining 20 benchmarks require disjunctive invariants which are difficult to find for FreqHorn. In order to evaluate the significance of candidate propagation, behavioral candidates, and candidates guessed from syntax, we performed controlled experiments with the corresponding features disabled. Fig. 2 gives the scatter plots that compare configurations on all benchmarks. Each point in a plot represents a pair of the runtime (sec) of the full configuration of FreqHorn (x-axis) and the runtime (sec) of the restricted configuration of FreqHorn (y-axis). In each plot, the color saturation roughly reflects the benefits of the full configuration, i.e., the delta between the runtimes.

The configuration of FreqHorn with candidate propagation disabled (thus, candidates for all unknowns had to be sampled independently) was able to solve 56 benchmarks, and it was on average three times slower than the full configuration. After disabling behavioral candidates (but with candidate propagation), FreqHorn was able to solve 60 benchmarks. Time-wise, this experiment gave less consistent results: for 15 benchmarks the restricted configuration outperformed the full one. Finally, after disabling syntactic candidates (but with candidate propagation and behavioral candidates), FreqHorn was able to solve only 37 benchmarks. The experiment confirmed that all features of our algorithm are essential for its efficacy, and it leaves room for devising heuristics to apply in specific contexts.

We also compared our tool to Spacer v.3 [4], μZ v.4.4.2 [1], and Eldarica v.1.3 [2] CHC solvers (shown in Fig. 3)[5][6]. Among the 101 benchmarks, Spacer was successful on 45, μZ on 42, and Eldarica on 71. FreqHorn solved 41 benchmarks on which Spacer diverged, 44 on which μZ diverged, 22 on which Eldarica diverged. In total, it solved **16** benchmarks on which all the competitors diverged, and 10 of them are over NIA.

In our benchmark selection, there are 8 tricky tasks which were solved by none of the tools. Investigating bottlenecks in solving them motivates our future work.

V. Related work

Conceptually, our algorithm for solving CHCs can be viewed as an extension of the syntax-guided invariant synthesizer [7] for transition systems (i.e., CHCs with one uninterpreted relation symbol). Thus, [7] is built around one sampling grammar, and does not require any candidate propagation. For arbitrary CHCs, as shown in our experiments, a naively extended approach of [7] does not scale well. Furthermore, in many cases, for convergence, it would require some symbolic constraints to be propagated across CHCs before the grammar is constructed (otherwise, the grammars might not be sufficient, and sometimes might be even empty). Our new solution is insensitive to these challenges.

Other instantiations of [7] include [8] and [9], but they still do not span beyond the transition systems. Our approach incorporates essential details of [8] and [9], namely enriching the grammars by externally created seeds. In particular, as in [9], we use polynomial equations as candidates for a relation between variables, generated after analyzing models for unrollings of CHCs. But again, [9] does not deal with multiple uninterpreted relation symbols. Our approach required solutions to several new challenges. First, a satisfiable unrolling for every loop must be found to obtain behavioral data. Second, even if we get a good candidate for interpretation of one symbol, often a weakening or a strengthening of this candidate is needed to accommodate suitable candidates for other symbols. We have addressed these issues by introducing a concept of modular unrolling of a system of CHCs, and by considering the seeds obtained from data to bootstrap the grammar generation.

[3]The source code is available at https://github.com/grigoryfedyukovich/aeval/tree/rnd.

[4]Available at https://github.com/grigoryfedyukovich/aeval/tree/rnd/bench_horn_multiple, and also contributed to CHC-COMP: http://chc-comp.github.io/.

[5]We excluded the time needed to start Java Virtual Machine from the running time of Eldarica.

[6]Full statistics are available at https://goo.gl/ADZdez.

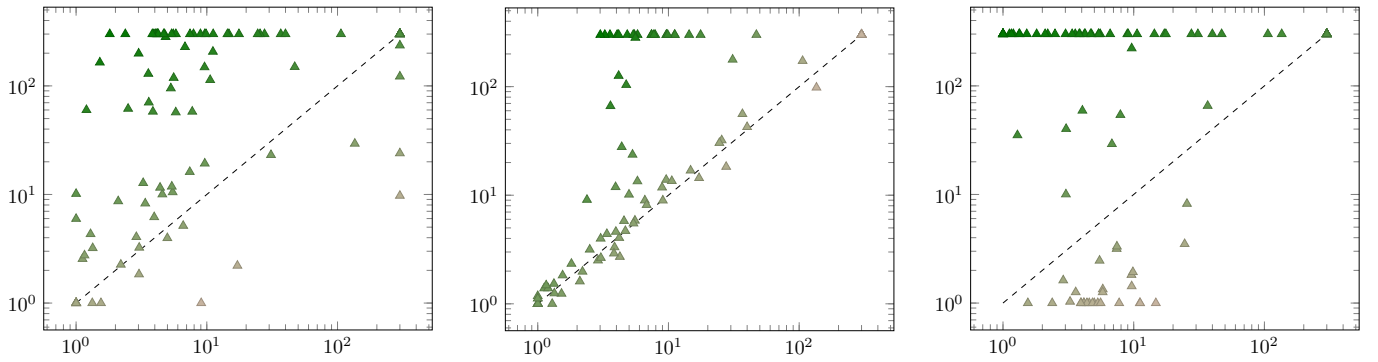

Fig. 2: Internal statistics on FreqHorn (sec×sec): points above the diagonal represent runtimes for benchmarks on which full configuration outperformed the restricted configuration.

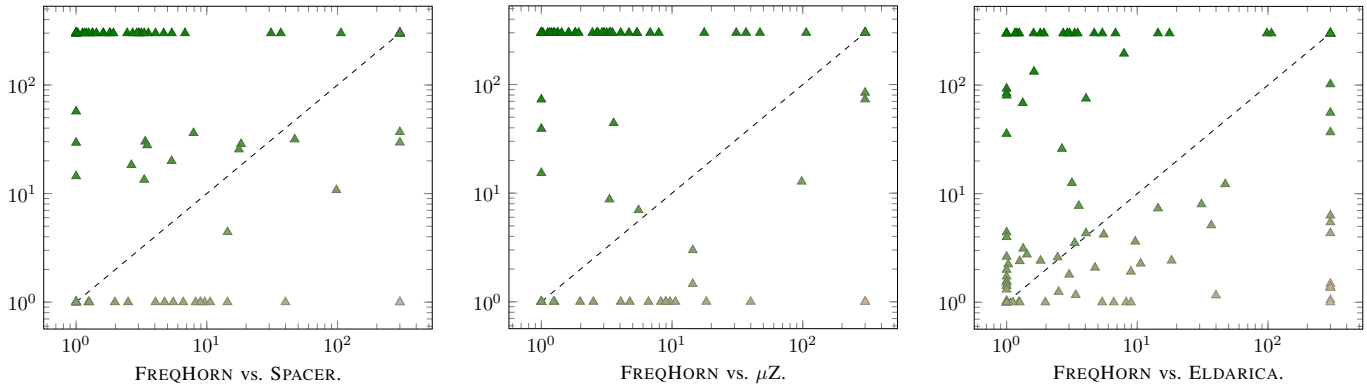

Fig. 3: Comparison of FreqHorn (sec×sec) and external tools: points above the diagonal represent runtimes for benchmarks on which the best configuration of FreqHorn outperformed the competitor.

Apart from solving unrollings as in [9], there are prominently two ways to get behavioral data – from infeasible paths using interpolation [21], and from reachable states along feasible paths using test-based executions [22], [12], [23], [24]. These techniques are not only limited by the expressiveness of their grammar, which is fixed, they also take the naive approach to dealing with multiple loops, i.e., the candidates are learned independently for all loops. In contrast, we use behavioral seeds to bootstrap the grammar. Furthermore, we propagate candidates learned for one loop to obtain constraints on those for adjacent loops.

Propagation of candidates and search for inductive subsets is at the heart of the approaches based on Generalized Property Directed Reachability (GPDR) [1], [4]. In a nutshell, they are based on implicit unrollings of loops and a monotonic fixed-point computation, driven by spurious counterexamples. However, such methods often diverge due to failures to generalize an inductive invariant from counterexamples. In contrast, our approach does not perform a fixed-point computation, and propagates candidates only through a finite number of implications, specified directly in CHCs. Failures to propagate lead to withdrawing the candidate and generating a new guess from the grammar. In practice, this makes our solution effective on many benchmarks which are difficult for GPDR.

VI. Conclusions

We have presented an algorithm for solving systems of CHCs based on Syntax-Guided Synthesis. For each unknown predicate in CHCs, our algorithm generates a formal grammar from the syntax of the CHC system and models of various unrollings of the system. A solution for the system (i.e., an interpretation of each unknown predicate that makes all CHCs true) is then guessed from the corresponding grammars and checked by an SMT solver. It is crucial for the effectiveness of the approach to use modular unrollings of CHCs and to propagate candidates through all available implications in the CHC system. We have presented the evaluation of our prototype built on top of the FreqHorn tool and have confirmed that the algorithm is effective on a range of benchmarks originating from program verification tasks and competitive with state-of-the-art CHC solvers. As we go ahead, we plan to optimize the algorithm using heuristics, to develop effective strategies for backward candidate propagation in case of nonlinear CHCs, and to extend our tool with the support of CHCs over arrays, algebraic data types and bit-vectors.

Acknowledgements: This work was supported in part by NSF Grant 1525936. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect those of the NSF.

REFERENCES

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

[2] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer, "A verification toolkit for numerical transition systems - tool paper," in *FM*, vol. 7436 of *LNCS*, pp. 247–251, Springer, 2012.

[3] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *PLDI*, pp. 405–416, ACM, 2012.

[4] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-Based Model Checking for Recursive Programs," in *CAV*, vol. 8559 of *LNCS*, pp. 17–34, 2014. https://bitbucket.org/spacer/code/branch/spacer3.

[5] H. Unno and T. Terauchi, "Inferring simple solutions to recursion-free Horn Clauses via sampling," in *TACAS*, vol. 9035 of *LNCS*, pp. 149–163, Springer, 2015.

[6] B. Kafle, J. P. Gallagher, and J. F. Morales, "Rahft: A tool for verifying Horn clauses using abstract interpretation and finite tree automata," in *CAV, Part I*, vol. 9779 of *LNCS*, pp. 261–268, Springer, 2016.

[7] G. Fedyukovich, S. Kaufman, and R. Bodík, "Sampling Invariants from Frequency Distributions," in *FMCAD*, pp. 100–107, IEEE, 2017.

[8] G. Fedyukovich and R. Bodík, "Accelerating Syntax-Guided Invariant Synthesis," in *TACAS, Part I*, vol. 10805 of *LNCS*, pp. 251–269, Springer, 2018.

[9] S. Prabhu, K. Madhukar, and R. Venkatesh, "Efficiently learning safety proofs from appearance as well as behaviours," in *SAS*, LNCS, Springer, 2018.

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

[11] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *TACAS*, vol. 1579 of *LNCS*, pp. 193–207, Springer, 1999.

[12] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A data driven approach for algebraic loop invariants," in *ESOP*, vol. 7792 of *LNCS*, pp. 574–592, Springer, 2013.

[13] F. A. Bourdoncle, "Efficient Chaotic Iteration Strategies with Widenings," in *FMPA*, vol. 735 of *LNCS*, pp. 128–141, Springer, 1993.

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

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

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

[17] G. Fedyukovich, A. Gurfinkel, and N. Sharygina, "Automated discovery of simulation between programs," in *LPAR*, vol. 9450 of *LNCS*, pp. 606–621, Springer, 2015.

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

[19] C. Sanderson and R. Curtin, "Armadillo: a template-based c++ library for linear algebra," *Journal of Open Source Software*, 2016.

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

[21] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," in *J. of Symbolic Logic*, pp. 269–285, 1957.

[22] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *ICSE*, pp. 449–458, ACM, 2000.

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

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