# Specification Synthesis with Constrained Horn Clauses

Sumanth Prabhu
sumanth.prabhu@tcs.com
TCS Research
Indian Institute of Science
India

Grigory Fedyukovich
grigory@cs.fsu.edu
Florida State University
USA

Kumar Madhukar
kumar.madhukar@tcs.com
TCS Research
India

Deepak D'Souza
deepakd@iisc.ac.in
Indian Institute of Science
India

## Abstract

The problem of synthesizing specifications of undefined procedures has a broad range of applications, but the usefulness of the generated specifications depends on their quality. In this paper, we propose a technique for finding maximal and non-vacuous specifications. Maximality allows for more choices for implementations of undefined procedures, and non-vacuity ensures that safety assertions are reachable. To handle programs with complex control flow, our technique discovers not only specifications but also inductive invariants. Our iterative algorithm lazily generalizes non-vacuous specifications in a counterexample-guided loop. The key component of our technique is an effective non-vacuous specification synthesis algorithm. We have implemented the approach in a tool called HornSpec, taking as input systems of constrained Horn clauses. We have experimentally demonstrated the tool's effectiveness, efficiency, and the quality of generated specifications on a range of benchmarks.

***CCS Concepts:*** • **Theory of computation** → **Invariants**; **Program specifications**; **Logic and verification**; **Automated reasoning**.

***Keywords:*** specification synthesis, automated verification, inductive invariants, SMT solvers.

**ACM Reference Format:**
Sumanth Prabhu, Grigory Fedyukovich, Kumar Madhukar, and Deepak D'Souza. 2021. Specification Synthesis with Constrained Horn Clauses. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454104

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*PLDI '21, June 20–25, 2021, Virtual, Canada*
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454104

## 1 Introduction

Specification synthesis is a challenging and important problem because of its multiple applications. A direct application is the problem of finding specifications of functions with unknown bodies in the verification of *open* programs [3, 15, 48]; it can also be applied in the problem of inferring safe preconditions for a program [14, 40, 44, 45], and winning strategy synthesis in safety games [9]. One practically useful formulation of this task is concerned with a *maximal* and *non-vacuous* interpretation of unknown procedures under which a safety property (a.k.a. assertion) holds. Maximal specifications summarize the largest set of behaviors for unknown procedures, and thus are logically weakest. Non-vacuous specifications guarantee that the safety property does not hold vacuously by becoming unreachable.

Maximal specifications make minimal assumptions about undefined procedures, hence they are very valuable. The fewer the assumptions, the more choices are available for implementations of undefined procedures. However, maximality should be considered along with the program's structure. In programs with loops, the maximal specifications should allow for invariants to exist to satisfy the safety property. Non-vacuous specifications are useful as they make assertions, capturing the program safety, reachable. A notable obstacle for an approach to specification synthesis is that for some tasks, there could be infinitely many maximal solutions, and some maximal solutions could be vacuous.

Multiple variants of specification synthesis have been proposed in [1, 3, 5, 8, 13–15, 26, 29, 42, 46–48]. The approaches include automata learning [3], learning patterns from program executions [5], usage of decision procedure [13], abstract interpretation [14] and even user guidance [15]. However, they rarely address the problems of maximality and non-vacuity at the same time (see Sect. 7 for more details).

A prominent approach to maximal specification synthesis [1] uses quantifier elimination to infer the weakest and non-vacuous preconditions for safety properties. It guesses specifications iteratively and uses a black-box verification oracle to find counterexample paths indicating the need to refine the specifications. Because driven by explicit counterexamples and external verification tools, this approach, however, may be ineffective on programs with control-flow divergence and loop invariants that are often difficult to find.

Our work is motivated by programs with complex control flow, i.e., with (possibly nested) loops and recursive functions. In such programs, specifications and invariants are tightly connected, thus allowing us to discover them at the same time. To guarantee non-vacuity, the search for specifications needs to be "global": although a specification for some function is discovered with respect to only its calling context, we have to check that it does not lead to undesired consequences in the rest of the program.

As outlined in Fig. 1, our framework addresses the challenges of finding maximal and non-vacuous specifications for a wide class of programs with complicated control flow. It can be parameterized by choosing a constraint solver to produce some (possibly, non-maximal) specifications iteratively. Our approach lazily collects specifications and combines them in a single specification, which eventually becomes maximal. To accelerate convergence, we additionally require our framework to yield non-vacuous solutions – otherwise, combining specifications may have no effect. However, this non-vacuity requirement for the constraint solver appeared to be too strict for the existing solvers based on Syntax-Guided Synthesis (SyGuS) [2] and Satisfiability Modulo Theory of Uninterpreted Functions [16]. In particular, only one SyGuS-solver [43] is currently capable of taking the vacuity constraints into account – others cannot even parse them. Our key contribution is thus twofold:

- an effective approach to yield non-vacuous specifications (the upper block in Fig. 1), which is a key component in:
- an iterative generalization procedure (the rightmost block in Fig. 1) to combine various non-vacuous specifications to yield a maximal one.

Our novel algorithm to find non-vacuous specifications is based on the idea of *alternating backward and forward reasoning* over the program. Backward reasoning attempts to *propagate* the safety property towards the initial states of the program, similar to weakest precondition inference. On the other hand, forward reasoning resembles strongest postcondition inference, and is useful for inductive invariants (e.g., when an unknown function is called in a loop). While both propagation techniques are widely used in program analysis, our contribution is in their effective combination. Specifically, the alternation of these techniques lets us use some part of already learned inductive invariants to discover

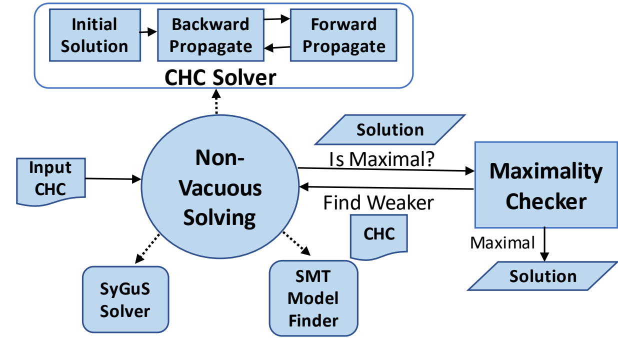


**Figure 1.** Architecture diagram of the framework.

specifications and vice versa. Furthermore, to facilitate the synthesis of inductive invariants and specifications, our approach adopts SyGuS and Houdini [24] to find helper lemmas, without which the propagated invariants are not inductive.

Because our algorithm makes use of inductive invariants, it is convenient to view the verification conditions (with the safety property and the missing specification) as a set of *constrained Horn clauses* (CHCs), implications in first order logic using uninterpreted predicates. Our third contribution in the paper is the formulation of the maximal and non-vacuous specification synthesis problems as (sequences of) existential CHC problems, where the quantifiers are imposed by the vacuity constraints. Existing CHC solvers rarely support existential quantifiers, thus our paper introduces a new specialized CHC solving algorithm as well. CHCs are used as an intermediate representation in a variety of verification and synthesis tools, [12, 19, 25, 27, 30, 33–35, 38, 39], to name a few, thus our approach can be integrated with the existing verification frontends.

We have implemented our algorithm in a tool called HornSpec on top of the FreqHorn [20, 21] framework. We evaluated HornSpec on a set of 65 CHC benchmarks with Linear Integer Arithmetic (LIA) operations, largely derived from the competition of CHC solvers (CHC-Comp) [23]. Due to the lack of solvers that can get both maximal and non-vacuous specifications, we instantiated our non-vacuous synthesis algorithm with CVC4 (a SyGuS solver) and Z3 (an SMT solver) in our tool and compared their performance. When the tool used our non-vacuous CHC solving algorithm, it found maximal specifications for twice the number of benchmarks than the other two techniques combined (54/65 vs. 21/65 and 5/65). Even when we compared only non-vacuous CHC solving, our algorithm performed significantly better, exhibiting that it efficiently generates general solutions.

The rest of the paper is organized as follows. After an illustrating example in Sect. 2, we define the notation and terms in Sect. 3. Then, Sect. 4 and Sect. 5 give an overview of our solvers for non-vacuity and maximality, respectively. Experimental evaluation is presented in Sect. 6. Finally, we discuss related work in Sect. 7 and conclude in Sect. 8.

## 2 Illustrating Example

Fig. 2 gives a program and its verification conditions. The problem is to find specifications for functions $f$ and $g$, whose bodies are unknown, such that the assertion is not violated. The generated verification conditions can be represented by three (implicitly universally quantified) implication constraints (called constrained Horn clauses or CHCs, formally defined in Sect. 3.1). In the CHCs, the uninterpreted relation $\boldsymbol{inv}(x)$ represents an inductive (loop) invariant, and $\boldsymbol{f}(z)$ and $\boldsymbol{g}(y)$ represent the specifications for the functions $f$ and $g$, respectively. The first CHC requires that the initial state be part of the inductive invariant; the second captures that the inductive invariant is closed under the body of the loop (and outputs of $f$); and the third CHC constrains the states in the inductive invariant such that they (and outputs of $g$) do not violate the assertion. To solve the problem, we need to find interpretations for $\boldsymbol{inv}$, $\boldsymbol{f}$, and $\boldsymbol{g}$ under which every CHC is valid. Notice that, due to function calls, the last two CHCs have two relations on their left-hand sides, making them *non-linear* CHCs.

The existing non-linear CHC solving techniques, [18, 36, 49] to name a few, generate vacuous specifications for the given CHCs, which makes the assertion unreachable. For example, Spacer [36] yields $\boldsymbol{inv} \mapsto \lambda x\,.\,\top$, $\boldsymbol{f} \mapsto \lambda z\,.\,\bot$, and $\boldsymbol{g} \mapsto \lambda y\,.\,\bot$. This solution satisfies the CHCs, but allows for no behaviors of $\boldsymbol{f}$ and $\boldsymbol{g}$, hence a meaningless solution. In order to get useful non-vacuous solutions, our technique augments CHCs with existentially quantified constraints (defined in Sect. 3.4).

The technique presented in [1] is not designed to directly handle inductive CHCs, like the second CHC in our example. Inductive CHCs are necessary to encode programs with a complex control flow, e.g., having loops and/or recursive functions. Inductive CHCs are challenging for SMT solvers that handle uninterpreted functions. For instance, the Z3 SMT solver (version 4.8.8) takes around ten minutes to come up with a solution.

We present an algorithm that not only approaches inductive CHCs, but also generates non-vacuous specifications using the alternation of forward and backward reasoning. For example, the algorithm finds the solution $M_2 = \{\boldsymbol{inv} \mapsto \lambda x\,.\,x \leq 19$, $\boldsymbol{g} \mapsto \lambda y\,.\,y \geq 19$, and $\boldsymbol{f} \mapsto \lambda z\,.\,z \leq 0\}$ for the CHCs in Fig. 2. It iterates over input CHCs to fix those implications that are not valid under the current solution.

| Program | CHC |
|---|---|
| `int x = 19;` | $x = 19 \implies \boldsymbol{inv}(x)$ |
| `while (*) {` | |
| `  int z = f();` | |
| `  x = x + z;` | $\boldsymbol{inv}(x) \wedge \boldsymbol{f}(z) \wedge x' = x + z \implies \boldsymbol{inv}(x')$ |
| `}` | |
| `int y = g();` | |
| `assert(y >= x);` | $\boldsymbol{inv}(x) \wedge \boldsymbol{g}(y) \wedge \neg(y \geq x) \implies \bot$ |

**Figure 2.** An example program (with a nondeterministic loop condition) and its CHC encoding.

An implication can be made valid either by weakening the relation on the right-hand side and/or strengthening the left-hand side's some relations. We refer to these as forward and backward propagation, respectively. An implication can also be made valid, albeit vacuously, by computing a solution that makes the left-hand side $\bot$. The algorithm avoids such vacuous solutions by requiring the left-hand side of each implication (after inserting a candidate solution) to be always satisfiable.

To solve the CHCs in Fig. 2, the algorithm begins with the weakest solution $\top$ for all relations. It finds that the third CHC is not satisfiable, hence performs backward propagation to strengthen $\boldsymbol{inv}$ and $\boldsymbol{g}$. For example, $\boldsymbol{inv} \mapsto \lambda x\,.\,x \leq 0$ and $\boldsymbol{g} \mapsto \lambda y\,.\,y \geq 0$. It then proceeds to the second CHC and backward propagates the solution of $\boldsymbol{inv}$ to get $\boldsymbol{f} \mapsto \lambda z\,.\,z \leq 0$. At this point, the algorithm realizes that the current solution for $\boldsymbol{inv}$ (i.e., $\boldsymbol{inv} \mapsto \lambda x\,.\,x \leq 0$) can not satisfy the first CHC as it is too strong. It now changes the direction of propagation to forward and finds a weaker solution $\boldsymbol{inv} \mapsto \lambda x\,.\,x \leq 19$. It verifies that the solutions of $\boldsymbol{inv}$ and $\boldsymbol{f}$ also satisfy the second CHC. However, the third CHC is not satisfiable as $\boldsymbol{g} \mapsto \lambda y\,.\,y \geq 0$. It uses backward propagation again, strengthening to $\boldsymbol{g} \mapsto \lambda y\,.\,y \geq 19$. This gives the solution $M_2$, which satisfies all the three CHCs, so the algorithm terminates (details follow in Sect. 4).

Usefulness of specifications depends on the number of choices available for program implementations of undefined procedures. For example, when compared to another non-vacuous solution $N_2 = \{\boldsymbol{inv} \mapsto \lambda x\,.\,x \leq 19, \boldsymbol{f} \mapsto \lambda z\,.\,z = -1$, and $\boldsymbol{g} \mapsto \lambda y\,.\,y = 19\}$, $M_2$ allows more behaviors for $\boldsymbol{f}$ and $\boldsymbol{g}$, hence more choices to implement procedures $f$ and $g$. In fact, $M_2$ is the *maximal* solution for the CHCs in a sense that weakening of any of $M_2(\boldsymbol{inv})$, $M_2(\boldsymbol{f})$, or $M_2(\boldsymbol{g})$, makes at least one CHC invalid.

To guarantee the maximality, we introduce a counterexample driven loop that lazily weakens the solution by reduction to another non-vacuous CHC task. For each (possibly non-maximal) solution, it first checks if *any* current interpretation could be weakened without sacrificing the validity of all CHCs. If not, the current solution is maximal. Otherwise, we obtain a *counterexample-to-maximality* (CTM) to formulate a new CHC task to be solved by our algorithm. By construction, the new CHCs are guaranteed to have a non-vacuous solution, which is weaker than the current one.

For instance, for a candidate $N_2 = \{\boldsymbol{inv} \mapsto \lambda x\,.\,x \leq 19$, $\boldsymbol{f} \mapsto \lambda z\,.\,z = -1$, and $\boldsymbol{g} \mapsto \lambda y\,.\,y = 19\}$, the loop first constructs an SMT formula that is satisfiable (with a CTM) if $N_2$ can be weakened, while also satisfying all the CHCs. It then uses the CTM and the current solution to determine which of the relations can be weakened. For instance, to weaken any of our relations, it constructs additional CHCs, including: 1) $z = -1 \implies \boldsymbol{f}(z)$ and 2) $z \neq -1 \wedge \boldsymbol{p_f}(z) \implies \boldsymbol{f}(z)$. Intuitively, the two CHCs constrain a new solution to $\boldsymbol{f}$ to be strictly weaker than the current one, when relation

$\boldsymbol{p_f}$ has a non-vacuous solution (i.e., it does not conflict with $z \neq -1$). A similar pair of CHCs are added for the relation $\boldsymbol{g}$. These four CHCs along with the input CHCs are provided as input to the non-vacuous CHC solving algorithm, which in turn generates the maximal solution $M_2$ (details in Sect 5).

## 3 Background

This paper approaches the task of specification synthesis by reduction to *Satisfiability Modulo Theories* (SMT) tasks. SMT aims to determine the existence of an assignment to variables of a first-order logic formula making it true. We write $m \models \varphi$ to denote that an assignment $m$ satisfies $\varphi$ (also called a *model*). Formula $\varphi$ is logically stronger than formula $\psi$ (denoted $\varphi \implies \psi$), if every model of $\varphi$ also satisfies $\psi$. The unsatisfiability of formula $\varphi$ is denoted $\varphi \implies \bot$.

We will be dealing with first-order logic over a finite set of relation symbols $\mathcal{R}$. We assume that each symbol $r$ in $\mathcal{R}$ comes with an arity $a_r$. We write $\varphi(x_1, \ldots, x_n)$ to denote a formula $\varphi$ with free variables in $\{x_1, \ldots, x_n\}$. For a formula $\varphi$ with a free variable $x$, and a term $t$, we write $\varphi[t/x]$ to denote the formula obtained by replacing each free occurrence of $x$ in $\varphi$ by $t$. We define an *interpretation* for a relation symbol $r \in \mathcal{R}$ to be a map of the form $\lambda x_1 \ldots \lambda x_{a_r}.\varphi(x_1, \ldots, x_{a_r})$, where $\varphi$ is a quantifier-free formula that does not contain any symbols from $\mathcal{R}$. An interpretation for $\mathcal{R}$ itself is a map $M$ which associates with each symbol $r \in \mathcal{R}$ an interpretation for $r$. Given a formula $\varphi$ and an interpretation $M$ for $\mathcal{R}$, we define $\varphi[M/\mathcal{R}]$ to be the formula obtained from $\varphi$ by replacing each occurrence of a term of the form $r(x_1, \ldots, x_{a_r})$ by $M(r)(x_1, \ldots, x_{a_r})$, where $r \in \mathcal{R}$.

As interpretations for a relation symbol are essentially logical formulas, we can treat them as such and define, for instance, $M(r) \implies N(r)$ to mean that for all $x_1, \ldots, x_{a_r}$ we have $M(r)(x_1, \ldots, x_{a_r}) \implies N(r)(x_1, \ldots, x_{a_r})$.

By *Expr* we denote the space of all possible quantifier-free formulas in our background theory and by *Vars* a range of possible variables. Since we will mainly deal with conjunctive formulas (which are created by adding/dropping conjuncts), we will sometimes interpret a finite subset $s$ of *Expr* as a conjunction of all its elements, i.e., $\bigwedge_{c \in s} c$.

**Definition 3.1 (CHC).** A *CHC* over a set of uninterpreted relation symbols $\mathcal{R}$ is a formula in first-order logic that has the form of one of the following three implications:

$$\varphi(\vec{x}_0) \implies \boldsymbol{r}_0(\vec{x}_0) \quad (1)$$

$$\bigwedge_{0 \le i \le n} \boldsymbol{r}_i(\vec{x}_i) \land \varphi(\vec{x}_0, \ldots, \vec{x}_{n+1}) \implies \boldsymbol{r}_{n+1}(\vec{x}_{n+1}) \quad (2)$$

$$\bigwedge_{0 \le i \le n} \boldsymbol{r}_i(\vec{x}_i) \land \varphi(\vec{x}_0, \ldots, \vec{x}_n) \implies \bot \quad (3)$$

where:

- for every $i$, $\boldsymbol{r}_i \in \mathcal{R}$ is an uninterpreted symbol;
- for some $i$ and $j$, such that $i \neq j$, it could be (though not necessary) that $\boldsymbol{r}_i = \boldsymbol{r}_j$;
- for every $i$, $\vec{x}_i$ is a vector of variables of length $a_{r_i}$;
- for every $i, j$ with $i \neq j$, the vectors $\vec{x}_i$ and $\vec{x}_j$ have no variables in common; and
- $\varphi$ is a satisfiable quantifier-free formula that does not contain any uninterpreted symbols.

For a CHC $C$, we will make use of the following notation:

- $body(C)$ (resp. $head(C)$) denotes the left (resp. right) side of the implication in $C$;
- $rels(body(C))$ denotes the set of uninterpreted symbols $\boldsymbol{r}_i \in \mathcal{R}$ that appear in $body(C)$;
- $rels(head(C))$ denotes the singleton set containing the uninterpreted symbol in $head(C)$ when $C$ is of type (1) or (2), and $\{\bot\}$ otherwise;
- For $C$ of type (1), $args(body(C))$ denotes $\vec{x}_1$, while for $C$ of type (2) and (3), $args(body(C))$ denotes $\bigcup_{i=0}^{n} \vec{x}_i$;
- $args(head(C))$ denotes $\vec{x}_1$ when $C$ is of type (1), $\vec{x}_{n+1}$ when $C$ is of type (2), and $\varnothing$ when $C$ is of type (3);
- $args(C)$ denotes $args(head(C)) \cup args(body(C))$;
- $args(\boldsymbol{r}_i, body(C))$ (resp. $args(\boldsymbol{r}_i, head(C))$ denotes $\vec{x}_i$, where $\boldsymbol{r}_i(\vec{x}_i)$ appears in $body(C)$ (resp. in $head(C)$); and
- $\varphi_C$ is used as a shortcut for the formula $\varphi$ in $C$.

A CHC of type (1) is called a *fact*, of type (2) is called *inductive*, and of type (3) is called a *query*. A CHC $C$ is *linear* if $|rels(body(C))| \le 1$; otherwise it is *non-linear*. Linear CHCs can be used to model safety proofs for transition systems (i.e., programs with only one loop), while one typically needs non-linear CHCs to model programs with possibly recursive functions. In this paper, we consider systems that may contain both linear and non-linear CHCs at the same time, thus supporting programs with complex control flow.

**Example 3.2.** In Fig. 2, $S$ consists of three CHCs over $\mathcal{R} = \{\boldsymbol{inv}, \boldsymbol{f}, \boldsymbol{g}\}$. The first CHC is a fact, the second is inductive and the third is a query. The first CHC is linear and the last two CHCs are non-linear.

**Definition 3.3 (Solution).** A system $S$ of CHCs over $\mathcal{R}$ is said to be *satisfiable* if there exists an interpretation $M$ for $\mathcal{R}$ which makes all implications in $S$ valid, i.e., for all $C \in S$, it holds that $body(C)[M/\mathcal{R}] \implies head(C)[M/\mathcal{R}]$. We call such an interpretation $M$ a *solution* to $S$.

**Definition 3.4 (Vacuous Solution).** Let $S$ be a system of CHCs over $\mathcal{R}$ and let $M$ be a solution to $S$. We say that $M$ is *vacuous* if 1) for some $r \in \mathcal{R}$, $M(r) \implies \bot$, or 2) for some CHC $C \in S$ which is not a query, $body(C)[M/\mathcal{R}] \implies \bot$.

Existing CHC solvers cannot guarantee to output non-vacuous solutions (which are necessary for our specification synthesis task). Note that to avoid vacuous solutions, a system of CHCs can be augmented by *existentially-quantified* constraints. In particular, for both points of Def. 3.4, this would require:

```
void main(){
  int x = f();
  int y = f();
  if (x == 5) return;
  assert(x == y);
}
```

$f(x) \land f(y) \land x \neq 5 \implies \boldsymbol{main}(x, y)$

$\boldsymbol{main}(x, y) \land (x \neq y) \implies \bot$

**Figure 3.** Example of maximal vacuous solution.

$$\boldsymbol{h}(x) \land \boldsymbol{h}(y) \land x' = x + y \implies \boldsymbol{f}(x')$$
$$\boldsymbol{h}(x) \land \boldsymbol{h}(y) \land x' = x + y \implies \boldsymbol{g}(x')$$
$$\boldsymbol{f}(x) \land \boldsymbol{g}(y) \land \neg(x \geq y) \implies \bot$$

**(a)**

$$\boldsymbol{f}(x) \land x' = x + 2 \implies \boldsymbol{f}(x')$$
$$\boldsymbol{f}(x) \land \boldsymbol{f}(y) \land \neg(y \neq x + 1) \implies \bot$$

**(b)**

$$z = 0 \implies \boldsymbol{g}(z)$$
$$\boldsymbol{f}(x) \land \boldsymbol{g}(z) \land \textit{if } (z = 0) \textit{ then } x' = 2021 \textit{ else } x' = x \implies \boldsymbol{f}(x')$$
$$\boldsymbol{f}(x) \land \boldsymbol{f}(y) \land \neg(x = y) \implies \bot$$

**(c)**

**Figure 4.** Non-linear CHCs for specification synthesis.

- for each $r \in \mathcal{R}$, $\exists \vec{x} \,.\, r(\vec{x})$, and
- for each CHC $C \in S$, $\exists \vec{x} \,.\, body(C)(\vec{x})$.

We call such a CHC system $\exists$-*extended*, where these added constraints cannot be described by Def. 3.1, and thus need to be handled by an external solver.

**Definition 3.5 (Strength Order).** Let $S$ be a system of CHCs over a set of relation symbols $\mathcal{R}$. We define a partial order (modulo logical equivalence) on the set of solutions of $S$, as follows. For solutions $M$ and $M'$ to $S$, we say $M'$ is *weaker* than $M$, written $M \preceq M'$, if for each $r \in \mathcal{R}$ we have $M(r) \implies M'(r)$. We say $M'$ is *strictly weaker* than $M$, written $M \prec M'$, if $M \preceq M'$ and there is an $r \in \mathcal{R}$ such that $M'(r) \not\Longrightarrow M(r)$.

**Definition 3.6 (Maximal Solution).** Let $S$ be a system of CHCs over $\mathcal{R}$. We call a solution $M$ to $S$ *maximal* if there is no solution $M'$ strictly weaker than it.

It is possible that a maximal solution is vacuous. For example, in a program in Fig. 3, $\boldsymbol{main}(x, y)$ represents the values of $x$ and $y$ that reach the assertion. But CHCs are ignorant of the assertion reachability in the original program. All formulas of the form $x = n$, where $n$ is an integer, are maximal solutions for $\boldsymbol{f}$. However, one of them, $x = 5$, is vacuous.

Interestingly, some CHC task could have many maximal solutions, where some of them could be vacuous.

# 4 Non-Vacuous Specification Synthesis

In this section, we describe our algorithm that overcomes the challenges of specification synthesis. A set of examples in Fig. 4 represents independent specification-synthesis problems. All of them have some uninterpreted predicate symbols that allow for vacuous solutions. In particular, in the CHC system in Fig. 4a the interpretation $M_{vac} = \{\boldsymbol{f} \mapsto \top, \boldsymbol{h} \mapsto \bot, \boldsymbol{g} \mapsto \bot\}$ satisfies all the CHCs, but it is vacuous. A non-vacuous solution is

$$M_{4a} = \{\boldsymbol{f} \mapsto \lambda x \,.\, x \geq 0, \boldsymbol{g} \mapsto \lambda y \,.\, y \leq 0, \boldsymbol{h} = \lambda x \,.\, x = 0\}.$$

Similarly, non-vacuous solutions for the systems in Fig. 4b and Fig. 4c are

$$M_{4b} = \{\boldsymbol{f} \mapsto \lambda x \,.\, x \bmod 2 = 0\}; \text{ and}$$
$$M_{4c} = \{\boldsymbol{f} \mapsto \lambda x \,.\, x = 2021, \boldsymbol{g} \mapsto \lambda z \,.\, z = 0\}.$$

## 4.1 Key Conceptual Insights

The challenges in finding non-vacuous and maximal solutions arise in the non-linearity of CHCs. For a linear CHC system, an approach based on *abduction* [17], would apply a backward reasoning beginning with the query CHC, and it would systematically find the weakest precondition. In contrast, for non-linear CHCs (such as all queries in Fig. 4), *multi-abduction* [1] can be used to get interpretations for predicates[1] in a query, which then can be propagated backwards to eventually yield the weakest preconditions for all unknown predicates upon reaching their invocations. However, this might lead to vacuous solutions for CHCs because multi-abduction is driven by the SMT models obtained from a single CHC and ignores constraints from other CHCs[2].

For Fig. 4a, an interpretation for $\boldsymbol{f}(x)$ and $\boldsymbol{g}(y)$ depends on some model $m \models x \geq y$. If $m_1 = \{x \mapsto 0, y \mapsto 0\}$, the solution is non-vacuous, as shown previously, but if $m_2 = \{x \mapsto 1, y \mapsto 1\}$, we get the mapping $M_{vac}$, such that $M_{vac}(\boldsymbol{f}) = \lambda x \,.\, x \geq 1$ and $M_{vac}(\boldsymbol{g}) = \lambda y \,.\, y \leq 1$. When this is propagated to the first two CHCs to get $M_{vac}(\boldsymbol{h})$, using *isomorphic decomposition*, it leads to the only solution: $M_{vac}(\boldsymbol{h}) = \lambda x \,.\, x > 0 \land x < 0$, which is vacuous. To prevent learning such predicates, our algorithm uses backtracking and iteratively re-solves multi-abduction using different models, while exploiting positive and negative results from the previous runs.

Abduction is not sufficient for the discovery of inductive interpretations in the case of loops, motivating us to *alternate backward and forward reasoning*. For instance, for Fig. 4b and 4c, the decomposition alone can respectively diverge, or be restarted many times. The former case, mentioned as a

[1] Technically, we discover specifications as interpretations of predicates from CHCs. Thus, throughout the paper, we use the terms specifications and interpretations interchangeably.

[2] Note that this side effect is specific only to our CHC settings, which in some sense modular. The original procedure in [1] can be applied to a "global" counterexample path-program, thus avoiding vacuous solutions.

**Algorithm 1:** SOLVECHCS

**Input:** CHCs $S$ over $\mathcal{R}$
**Output:** $res \in \{(\textsc{sat}, Lemmas : \mathcal{R} \to 2^{Expr}), (\textsc{unknown}, \varnothing)\}$

```
1  forward ← ⊥;
2  for each r ∈ R do Σ(r) ← ∅;
3  while ⊤ do
4      if ∃C ∈ S . (body(C)[Σ/R] ⟹ ⊥ ∧ head(C) ≠ ⊥) ∨
         (∃r1 ∈ rels(body(C)) . Σ(r1) = ⊥ ∧ head(C) = ⊥) then
5          Σ ← WEAKEN(C, Σ);
6          forward ← ⊤;
7      else if ∃C ∈ S . body(C)[Σ/R] ⟹̸ head(C)[Σ/R]
         then
8          if rels(body(C)) = ∅ then
9              forward ← ⊤;
10         if forward = ⊥ then
11             Σ ← STRENGTHEN(C, Σ);
12         else
13             Σ ← PROPAGATE(S, C, Σ, ∅);
14             forward ← ⊥;
15     else
16         return (SAT, Σ);
17 return (UNKNOWN, ∅);
```

challenge in [1], is solved quickly by our algorithm. In particular, the algorithm takes into account the inductive CHC, guesses candidate decompositions for $\boldsymbol{f}$, having the form of divisibility constraints, and propagates them forward (with the help of a HOUDINI-style algorithm [24]). After an inductive subset is found, the query is checked, and (if needed) the multi-abduction is solved. For Fig. 4c, there could be infinitely many decompositions for $\boldsymbol{f}$: $\lambda x\,.\,x = 0$, $\lambda x\,.\,x = 1$, $\lambda x\,.\,x = 2$,... but only one of them is essentially useful. Instead of taking a significant amount of time to enumerate all solutions, our approach learns that $\boldsymbol{g} \mapsto \lambda z\,.\,z = 0$ satisfies the first CHC, then forward-propagates it to the second CHC to get $\boldsymbol{f} \mapsto \lambda x\,.\,x = 2021$, which satisfies the third CHC, allowing for speedy and successful termination.

These and other features of the algorithm are described in detail in the following sections.

## 4.2 Algorithm Overview

In this section, we give an overview of our core algorithm that synthesizes non-vacuous specifications assuming a first order theory that admits quantifier elimination, e.g., Linear Integer Arithmetic (LIA).

**Basic Rules.** Algorithm 1 maintains a set $\Sigma$ of candidate specifications and initiates it (line 2) by empty sets (which corresponds to $\top$ formulas) and refines them based on the following intuitive rules:

**R1** If *vacuity* (line 4) holds for some $C$, then it is likely because some specifications are $\bot$ or in conflict with the body. The algorithm picks some relation $r$ from the body of $C$ and creates $\Sigma'(r)$ which is logically weaker than $\Sigma(r)$ (line 5). Thus, $body(C)[\Sigma'/\mathcal{R}]$ becomes weaker than $body(C)[\Sigma/\mathcal{R}]$ and is more likely to pass the vacuity check in the next iteration.

**R2** If *validity* (line 7) does not hold for some $C$, then candidate specifications in $rels(body(C))$ need to be strengthened and/or the candidate specification for $rels(head(C))$ needs to be weakened. The algorithm acts according to the current direction (backward or forward) of the CHC traversal.

It is important that the rules are applied in the specific order: **R2** assumes that $\Sigma$ is non-vacuous, otherwise the checks would trivially succeed. If neither **R1** nor **R2** is applicable, then $\Sigma$ satisfies all constraints in the system and is returned to the user as a final result.

**Theorem 4.1.** *The candidate specifications remaining after Algorithm 1 terminates are non-vacuous.*

*Proof.* Note that after new candidate specifications are added in Algorithm 1, either by STRENGTHEN (line 11) or by PROPAGATE (line 13), **R1** (line 4) is checked in the next iteration. So if the SMT solver proves that candidates are non-vacuous in **R2** (line 7), the algorithm terminates. We prove it by contradiction. Suppose there exists a CHC $C$ with vacuous candidates in $\Sigma$. By definition of vacuous candidates, there are two cases: (1) for some $\boldsymbol{r} \in rels(body(C))$, $\Sigma(\boldsymbol{r})$ is unsatisfiable, (2) $body(C)[\Sigma/\mathcal{R}]$ is unsatisfiable (if $C$ is not a query). In either cases, WEAKEN guarantees to drop some part of $\Sigma(\boldsymbol{r})$ of some $\boldsymbol{r}$ and the check is repeated (at most, a finite number of times). Note that for two relations $\boldsymbol{r}_1, \boldsymbol{r}_2 \in rels(body(C))$ if the conjunction $\Sigma(\boldsymbol{r}_1) \land \Sigma(\boldsymbol{r}_2)$ is unsatisfiable, one of them must be unsatisfiable (since they do not share any arguments). □

**Backward / Forward Alternation.** A distinguishing feature of our algorithm is that it alternates between backward and forward reasoning over CHCs. In particular, it exploits the implications in each CHC and either generates (1) new candidate specifications of the predicates in $rels(body(C))$ based on a specification for $rels(head(C))$ (backward, see more details in Sect. 4.4), or (2) new candidate interpretation for $rels(head(C))$ based on given interpretations for $rels(body(C))$ (forward, see more details in Sect. 4.3). The algorithm has therefore a flag *forward* that indicates whether forward reasoning is enabled (and otherwise, backward reasoning is enabled).

Initially, backward reasoning is enabled (line 1), and the algorithm initializes the specifications from the queries. When updated, specifications in $\Sigma(r)$ of some $r$ require to perform the **R2** check for every CHC $C$, such that $r \in rels(head(C))$. Whenever a vacuity constraint **R1** (for current $\Sigma$) is violated or the candidates cannot be updated further, the direction of reasoning gets changed (lines 6, 9, or 14).

**Example 4.2.** Recall the example in Sect. 2. To eventually end up with the solution, Algorithm 1 begins with the empty $\Sigma$, i.e., $\Sigma(\boldsymbol{inv}) = \Sigma(\boldsymbol{f}) = \Sigma(\boldsymbol{g}) = \lambda x\,.\,\top$ that is not vacuous. The algorithm finds that the third CHC is not satisfiable while checking rule **R2**:

$$\top \land \top \land \neg(y \geq x) \;\not\Longrightarrow\; \bot.$$

Since the propagation direction is backward, candidates are strengthened to $\Sigma(\boldsymbol{inv}) = \lambda x\,.\,x \leq 0$ and $\Sigma(\boldsymbol{g}) = \lambda y\,.\,y \geq 0$ (line 11) using the abductive strengthening (to be explained in Sect. 4.4).

In the second iteration of the algorithm, **R2** fails again as the second CHC is not satisfiable under $\Sigma(\boldsymbol{inv}) = \lambda x\,.\,x \leq 0$ and $\Sigma(\boldsymbol{f}) = \lambda x\,.\,\top$:

$$x \leq 0 \land \top \land x' = x + z \;\not\Longrightarrow\; x' \leq 0.$$

In this case, the algorithm decides to strengthen only $\boldsymbol{f}$, though the second CHC also has $\boldsymbol{inv}$ in its *body*. This, too, distinguishes our algorithm from [1] and is achieved by the *fairness* heuristic (to be explained in Sect. 4.5). The solution returned is $\Sigma(\boldsymbol{f}) = \lambda z\,.\,z \leq 0$. But again, **R2** fails as the first CHC (a fact) is not satisfiable under $\Sigma(\boldsymbol{inv}) = \lambda x\,.\,x \leq 0$:

$$x = 19 \;\not\Longrightarrow\; x \leq 0$$

Since the failed CHC is a fact, the direction of propagation changes (line 6). The candidate for $\boldsymbol{inv}$ is repaired to $\lambda x\,.\,x \leq 19$ by the inductive weakening (to be explained in Sect. 4.3), and the direction of reasoning is reversed (lines 13, 14). However, the query is still not satisfiable under $\Sigma(\boldsymbol{inv}) = \lambda x\,.\,x \leq 19$ and $\Sigma(\boldsymbol{g}) = \lambda y\,.\,y \geq 0$:

$$x \leq 19 \land y \geq 0 \land \neg(y \geq x) \;\not\Longrightarrow\; \bot.$$

Then, the abductive strengthening yields $\Sigma(\boldsymbol{g}) = \lambda y\,.\,y \geq 0 \land y \geq 19$. In the final iteration neither **R1** nor **R2** is applicable, so the current candidate $M_2$ is returned. □

To search for solutions of a system of CHCs, as done for example in [21], it is useful to define an ordering between CHCs. To keep the presentation simpler, our pseudocode does not specify which particular CHCs are selected for further processing. If either of checks **R1** or **R2** fails for more than one CHC – intuitively, it picks the one *closer* to a fact CHC (in the case of forward) or the one closer to a query (in the case of backward reasoning). Given $C', C, C'' \in S$, we say $C'$ is *closer* to $C$ than $C''$, if there is a shorter sequence of CHCs connecting $C'$ to $C$, where a sequence of CHCs $C_1, \ldots, C_n \in S$ connects if for each pair $C_i$ and $C_{i+1}$, $\mathit{rels}(\mathit{head}(C_i)) \subseteq \mathit{rels}(\mathit{body}(C_{i+1}))$. For instance, in Example 4.2, in the second iteration, when the algorithm was performing backward propagation, both the first and the second CHC fails, but the algorithm chooses the second CHC, which is closer to the query. However, any other ordering heuristics can in principle be used.

**Algorithm 2:** PROPAGATE

**Input:** CHCs $S$ over $\mathcal{R}$, $C \in S$, $\Sigma' : \mathcal{R} \to 2^{Expr}$, $\mathit{visited} \subseteq S$
**Output:** $\Sigma' : \mathcal{R} \to 2^{Expr}$

1 **if** $\mathit{rels}(\mathit{head}(C)) = \bot$ or $C \in \mathit{visited}$ **then return** $\Sigma$;
2 $\mathit{visited} \leftarrow \mathit{visited} \cup \{C\}$;
3 let $r_h$ and $\vec{x}_h$ be such that $r_h(\vec{x}_h) = \mathit{head}(C)$;
4 **if** $\mathit{body}(C)[\Sigma/\mathcal{R}] \;\not\Longrightarrow\; \Sigma(r_h)(\vec{x}_h)$ **then**
5     **while** $\exists m.m \models \mathit{body}(C)[\Sigma/\mathcal{R}] \land \neg\Sigma(r_h)(\vec{x}_h)$ **do**
6         **for each** $d \in \Sigma(r_h)$ **do**
7             **if** $m \models \neg d(\vec{x}_h)$ **then**
8                 $\Sigma(r_h) \leftarrow \Sigma(r_h) \setminus \{d\}$;
9 $\Sigma(r_h) \leftarrow \Sigma(r_h) \cup$ OVER$(QE(\exists\left[\mathit{args}(\mathit{body}(C)) \setminus \vec{x}_h\right].\mathit{body}(C)[\Sigma/\mathcal{R}]))$;
10 **for each** $C' \in S$ **do**
11     **if** $C' \neq C$ and $r_h \in \mathit{rels}(\mathit{body}(C')) \cup \mathit{rels}(\mathit{head}(C'))$ **then**
12         $\Sigma \leftarrow$ PROPAGATE$(S, C', \Sigma)$;
13 **return** $\Sigma$;

### 4.3 Inductive Weakening and Propagation

Each fact in the CHC system is potentially useful for generation of interpretations. A formula from its body forms a candidate which is checked in the next stages of a CHC-solving algorithm. Whenever the inductiveness check fails for some of the CHCs, the candidate for the predicate in its body can be weakened, eventually leading to an *inductive subset*. Such a weakening loop is commonly referred to as HOUDINI [24] and used in the many CHC solving approaches, such as [36] and [21]. In addition, [21] forward-propagates successful candidates through bodies of CHCs to be considered candidates for the interpretations of predicates from the heads. Algorithm 2 demonstrates how we combine inductive weakening and forward propagation.

The algorithm begins with finding an inductive subset of candidates by dropping conjuncts from the interpretations of $\mathit{rels}(\mathit{head}(C))$ (denoted in the pseudocode as $r_h$). The basic idea is to use so-called *counterexamples to induction* that identify non-inductive conjuncts (line 7). Note that the loop always terminates, and in the worst case it ends up with dropping all conjuncts in $\Sigma(r_h)$, making the implication easily valid.

The second phase of Algorithm 2 aims at strengthening $\Sigma(r_h)$ based on bodies of other CHCs $C'$ that involve $r_h$ (line 11). In particular, in the recursive call of Algorithm 2, the fact CHC $C'$, whose head involves $r_h$, can be considered (line 9), and a strengthening of $\Sigma(r_h)$ is obtained by quantifier elimination (QE) over the body of $C'$ (i.e., by eliminating variables that do not appear in $\mathit{head}(C)$).

We also rely on various heuristics to obtain potentially inductive interpretations from precise results of quantifier elimination (referred to as OVER at line 9). Given the formula $\varphi$, we collect a finite set of its over-approximations, and then:

$$\text{OVER}(\varphi) \stackrel{\text{def}}{=} \{\varphi\} \cup \{\psi_i \mid \varphi \implies \psi_i\}$$

where $\psi_i$ can be obtained by Syntax-Guided Synthesis (SyGuS) (as shown in Sect. 4.7) or by any post-condition inference techniques. We also use a set of simple rules that break equalities into inequalities, merge inequalities, etc. Any heuristics can be used to add more elements to the candidate specifications set – all inappropriate ones will be dropped by the weakening loop (lines 5-8) in the next iterations of the algorithm.

**Example 4.3.** Recall Example 4.2 for the system of CHCs in Fig. 2. The main algorithm is in the forward propagation mode, and it aims first at weakening the candidate $x \leq 0$ given the first CHC. The Houdini loop immediately discovers the model $m = \{x \mapsto 19\}$, such that $m \models \neg(x \leq 0)$, thus leading to dropping the only conjunct. We now have $\Sigma(\boldsymbol{inv}) = \lambda x\,.\,\top$ at line 9, and $x = 19$ is passed to QE with no existentially-quantified variables (as $Vars(body(C)) \setminus \vec{x}_h$ is $\{x\} \setminus \{x\} = \varnothing$)), yielding $x = 19$ as the solution. The method over generates $x \geq 19$ and $x \leq 19$ as potential candidates. This is followed by the recursive calls to other CHCs (line 11). For the second CHC, the check at line 4 fails:

$$x \geq 19 \wedge x \leq 19 \wedge z \leq 0 \wedge x' = x + z \not\Longrightarrow x' \geq 19 \wedge x' \leq 19.$$

Here, $x \geq 19 \wedge x \leq 19$ is $\boldsymbol{inv}$'s candidate and $z \leq 0$ is $\boldsymbol{f}$'s candidate (recall Example 4.2). Any model $m$ computed at line 5 for $x \geq 19 \wedge x \leq 19 \wedge z \leq 0 \wedge x' = x + z \wedge \neg(x' \geq 19 \wedge x' \leq 19)$ will be such that $m(x') \leq 19$ as $z \leq 0$. Hence, only $x \leq 19$ satisfies this model, and $x \geq 19$ is dropped (lines 7-8). Since there are no more CHCs that are not visited, the algorithm returns with $\Sigma(\boldsymbol{inv}) = \lambda x\,.\,x \leq 19$. Note here that it retained $\Sigma(\boldsymbol{f}) = \lambda z\,.\,z \leq 0$ due to the usage of inductive weakening, which will be the final solution for $\boldsymbol{f}$ as seen from Example 4.2. □

### 4.4 Abductive Strengthening

Backward propagation in our algorithm is responsible for lifting the property (i.e., the negated body of the query) towards the initial states of the program. It is implemented in Algorithm 3, which takes as input a CHC $C$ and sets of current candidate interpretations $\Sigma$ and returns updated sets as output. It is mainly based on [17] and further on [1].

The algorithm first applies abduction via universal quantifier elimination to obtain a formula $\psi$ (line 3) over arguments of predicates that appear in the body of $C$. If $C$ is linear, then $\psi$ gets an interpretation to the only predicate $\boldsymbol{r}$ in $rels(body(C))$, and the only action the algorithm needs to perform before termination is to update $\Sigma(\boldsymbol{r})$[3]. Otherwise, $\psi$ has to be decomposed among relations in $rels(body)$. For this purpose, we adapt the multi-abduction technique, originated from [1] (lines 6-11).

The simplest use of multi-abduction assumes that the algorithm proceeds with all the predicates from the body of $C$, i.e., $R_f = rels(body(C))$ at line 4 (the other use is discussed in Sect. 4.5). The algorithm gets a model of $\psi$ (at line 5) and creates under-approximations of candidates (called *elementary*) as equalities between arguments of each $r \in R_f$ and their values from the model (at line 7). These elementary solutions get further weakened (at line 9) by subsequent runs of abduction [17]. Each relation under consideration is assumed to be uninterpreted, while formulas are substituted for others. Weakened solutions for the relations covered in earlier iterations, and elementary interpretations for the rest (lines 8-9). The algorithm performs the isomorphic decomposition (line 11; to be described in Sect. 4.6) for those relations that are invoked multiple times in the body of $C$.

[3]Our pseudocode does not have this for brevity, but the general functionality produces the same result.

**Algorithm 3:** strengthen

**Input:** CHC $C \in S$ and $\Sigma : R \to 2^{Expr}$
**Output:** $\Sigma^{new} : R \to 2^{Expr}$

1 $\Sigma^{new} \leftarrow \Sigma$;
2 let $rels(body(C)) = \{\boldsymbol{r}_0, \ldots, \boldsymbol{r}_n\}$;
3 $\psi \leftarrow \mathrm{QE}(\forall \big[args(body(C)) \setminus args(\boldsymbol{r}_0, \ldots, \boldsymbol{r}_n, body(C))\big]\,.\,(body(C)[\Sigma/\mathcal{R}]) \implies \bigwedge_{c \in \Sigma(rels(head(C)))} c)$;
4 let $R_f \subseteq rels(body(C))$;
5 let $m \models \psi$;
6 **for each** $r \in R_f$ **do**
7 $\quad\Sigma^{new}(r) \leftarrow \vec{x} = m(\vec{x})$, s.t. $\vec{x} = args(r, body(C))$;
8 **for each** $r \in R_f$ **do**
9 $\quad\Sigma^{new}(r) \leftarrow \mathrm{QE}(\forall \big[\bigcup_{r' \neq r} args(r', body(C))\big]\,.\,\bigwedge_{r' \neq r} \Sigma^{new}(r') \implies \psi)$;
10 **for each** $r \in R_f$ if invoked several times in $rels(body(C))$ **do**
11 $\quad\Sigma^{new}(r) \leftarrow$ isoDecompose$(\Sigma^{new}(r), r, C)$;
12 **return** $\Sigma^{new}$;

**Example 4.4.** Consider the first iteration from Example 4.2, where Algorithm 1 is running in the backward direction on CHCs from Fig. 2. The third CHC with $\Sigma(\boldsymbol{inv}) = \Sigma(\boldsymbol{g}) = \lambda x\,.\,\top$ has failed. The algorithm starts with multi-abducing $y \geq x$. For the QE at line 3, $y \geq x$ is passed without any quantified variables (as $\{x, y\} \setminus \{x, y\} = \varnothing$). Since there are two relations, the algorithm continues with $\psi = y \geq x$, to decompose the following:

$$\boldsymbol{inv}(x) \wedge \boldsymbol{g}(y) \implies y \geq x.$$

Suppose, the algorithm computes a model $m = \{x \mapsto 0, y \mapsto 0\}$ at line 7, and $R_f = \{\boldsymbol{inv}, \boldsymbol{g}\}$, using which, we construct $\Sigma^{new} = \{\boldsymbol{inv} \mapsto \lambda x\,.\,x = 0, \boldsymbol{g} \mapsto \lambda y\,.\,y = 0\}$. The first QE call (line 9) is made with $\forall y\,.\,y = 0 \implies y \geq x$ for $\boldsymbol{inv}$, which yields $x \leq 0$. In the second QE call, $\boldsymbol{g}$ gets $y \geq 0$ corresponding to $\forall x\,.\,x \leq 0 \implies y \geq x$. Since there are no multiple invocations of any relation, the algorithm returns with this solution. □

### 4.5 Fairness

Our approach makes use of a useful *fairness* heuristic that allows for decomposing a property into candidate interpretations (as in Sect. 4.4), but only of a subset of relations. Fairness prioritizes relations that currently have no candidates. Intuitively, it drops those relations for which the same candidates are repeated from a previous iteration. During the decomposition, non-chosen relations are replaced by their existing candidate specifications.

In Algorithm 3, fairness is applied in two ways: (1) if there are relations without any candidates (i.e., $\Sigma(r) = \top$), then other relations are excluded from $R_f$; (2) after finding a decomposition, if a candidate is rediscovered for the second time, then this candidate is dropped, the relation is excluded from $R_f$, and the backward propagation restarts. This requires maintaining the history of candidates, which is intuitive but not shown in the algorithm for brevity.

**Example 4.5.** Recall Example 4.2 (Algorithm 1 running on Fig. 2), where in the second iteration, the second CHC has failed for $\Sigma(\boldsymbol{inv}) = \lambda x \,.\, x \le 0$ and $\Sigma(\boldsymbol{f}) = \lambda x \,.\, \top$. The decomposition query is therefore as follows:

$$\boldsymbol{inv}(x) \wedge \boldsymbol{f}(z) \implies (x \le 0 \wedge x' = x + z \implies x' \le 0).$$

In this case, Algorithm 3 computes $\psi = x + z \le 0$ after applying QE (line 3) to $\forall x' \,.\, x \le 0 \wedge x' = x + z \implies x' \le 0$. There are two relations ($\boldsymbol{f}$ and $\boldsymbol{inv}$) in the *body*, however it is fair to let $R_f = \{\boldsymbol{f}\}$. Suppose the algorithm computes a model $m = \{x \mapsto 0, z \mapsto 0\}$ for $x + z \le 0$ (line 7). Initially, $\Sigma^{new}(\boldsymbol{f}) = \lambda z \,.\, z = 0$, whereas $\Sigma^{new}(\boldsymbol{inv}) = \lambda x \,.\, x \le 0$. QE (line 9) is passed with $\forall x \,.\, x \le 0 \implies (x + z) \le 0$, for $\boldsymbol{f}$, which returns $\lambda z \,.\, z \le 0$. □

**Example 4.6.** Let us consider the case when the third CHC failed with $\Sigma(\boldsymbol{inv}) = \lambda x \,.\, x \le 19$ and $\Sigma(\boldsymbol{g}) = \lambda y \,.\, y \ge 0$:

$$\boldsymbol{inv}(x) \wedge \boldsymbol{g}(y) \implies (x \le 19 \wedge y \ge 0 \implies y \ge x).$$

Without the fairness heuristic, this abduction query would yield the strengthening for $\boldsymbol{inv}$ (as in Example 4.4). However, the candidate $x \le 0$ is identified as repeated for $\boldsymbol{inv}$, the algorithm drops it, lets $R_f = \{\boldsymbol{g}\}$, and restarts. This results in a different initial solution: $\Sigma^{new}(\boldsymbol{g}) = \lambda y \,.\, y = 19$ and $\Sigma^{new}(\boldsymbol{inv}) = \lambda x \,.\, x \le 19$. Further, on QE with $\forall x \,.\, x \le 19 \implies y \ge x$ (line 9), $\boldsymbol{g}$ gets $\lambda y \,.\, y \ge 19$. □

### 4.6 History-based Isomorphic Decomposition

Finding a decomposition if some relation from the body of a CHC has two or more invocations (with different arguments) is achieved by the isomorphic decomposition algorithm in [1]. For each such relation, the algorithm tries to further decompose the interpretation obtained in the previous step (recall Sect. 4.4) by progressively weakening interpretations until the required solution is found. In this section, we present this algorithm (Algorithm 4) with the following significant modification to [1].

**Algorithm 4:** isoDecompose

**Input:** $cand \in Expr$, $r \in \mathcal{R}$ and CHC $C \in S$
**Output:** $soln \in Expr$

1 **for each** $r(\vec{x}_i) \in body(C)$ **do**
2 let $\vec{p}_{x_i}$ be fresh copies of $\vec{x}_i$;
3 $soln \leftarrow \bot$;
4 $\psi \leftarrow \bigwedge_{\vec{x}_i} (\bigvee_{\vec{p}_{x_j}} \vec{x}_i = \vec{p}_{x_j})$;
5 **while** $\exists m \,.\, m \models \bigwedge_{\vec{p}_{x_i}} \neg soln[\vec{p}_{x_i}/\vec{x}] \wedge (\forall \vec{x}_i \ldots \vec{x}_n \,.\, \psi \implies cand)$ **do**
6 $soln \leftarrow soln \vee \bigvee_{\vec{p}_{x_i}} \vec{x} = m(\vec{p}_{x_i})$;
7 **for each** $\vec{x}_i$ **do**
8 $soln_i(\vec{x}_i) \leftarrow soln[\vec{x}_i/\vec{x}]$
9 **for each** $\vec{x}_i$ **do**
10 $soln_i \leftarrow \mathrm{QE}(\forall [\vec{x}_{j \ne i}] \bigwedge_{j \ne i} soln_j \implies cand)$
11 $soln \leftarrow \bigwedge_{\vec{x}_i} soln_i[\vec{x}/\vec{x}_i]$;
12 $\psi \leftarrow \bigwedge_{\vec{x}_i} (\bigvee_{\vec{p}_{x_j}} \vec{x}_i = \vec{p}_{x_j}) \vee soln[\vec{x}_i/\vec{x}]$;
13 **return** $soln$;

Algorithm 4 receives a relation $r$ that occurs multiple times in $body(C)$ and the corresponding candidate (represented by $cand$) that has been computed by considering $r$ as a single instance with all its arguments (i.e., $\Sigma^{new}(r)$ in Algorithm 3). The algorithm begins by creating placeholder variables $\vec{p}_{x_i}$. These are used to make sure that the solution is unaffected by exchange of argument instances. This constraint is encoded by the formula $\psi$ (line 4). The algorithm iteratively strengthens the solution ($soln$) in a loop (line 5- 12). In each iteration of this loop, the algorithm begins by computing an initial solution that has not been considered previously. This is achieved by getting a model for the formula that is not part of current solution (i.e in $\neg soln$), but implies $cand$ even when the arguments are exchanged (line 6). The model is projected over each argument instance to get an initial solution ($soln_i$) (lines 7,8). These initial solutions are weakened using quantifier elimination (line 9), similar to Algorithm 3. Finally, a conjunction of all these solutions is considered as the interpretation of current iteration (line 11). This loop continues until no interpretation can be added.

In the original algorithm (as presented in [1]), the formula $\psi$ is not updated in each iteration. However, to consider previously generated interpretations, our algorithm disjoins it with current interpretation $soln[\vec{x}_i/\vec{x}]$. This helps in termination of our algorithm when there are infinitely many maximal solutions (like shown in Example 4.7), where the original algorithm does not terminate.

**Example 4.7.** Consider the system of CHCs from Fig. 4c. At the first iteration, for the third CHC (with two invocations of $\boldsymbol{f}$), the abductive strengthening calls Algorithm 4 with

$cand = (x = y)$ as the candidate for $\boldsymbol{f}$. Note that $cand$ has infinitely many decompositions (recall Sect. 4.1). Algorithm 4 gets a model $m = \{p_x \mapsto 0, p_y \mapsto 0\}$, constructs a formula:

$$\top \wedge \forall x, y \,.\, (x = p_x \vee x = p_y) \wedge (y = p_x \vee y = p_y) \implies x = y,$$

initializes $soln_1(x) = x = 0$, $soln_2(y) = y = 0$ for two invocations of $\boldsymbol{f}$, and constructs candidates as $soln = x = 0$ after QE (line 11). Now, $\psi = ((x = p_x \vee x = p_y \vee x = 0) \wedge (y = p_x \vee y = p_y \vee y = 0))$ (line 12). This makes the algorithm terminate at the next iteration since the following equation at line 5 is unsatisfiable:

$$\begin{aligned}(p_x \neq 0) \wedge (p_y \neq 0) \wedge \\ \forall x, y \,.\, (x = p_x \vee x = p_y \vee x = 0) \wedge \\ (y = p_x \vee y = p_y \vee y = 0) \implies x = y\end{aligned}$$

The original algorithm [1] does not terminate as the above equation without $x = 0$ and $y = 0$ is satisfiable. Moreover, it is not able to find any solution as all the solutions are satisfiable in the equation above without adding the current candidate.

Algorithm 1 continues with $\Sigma(\boldsymbol{f}) = \lambda x \,.\, x = 0$, backward propagates $\Sigma(\boldsymbol{g}) = \lambda z \,.\, \neg(z = 0)$ using the second CHC and fairness, weakens $\Sigma(\boldsymbol{g}) = \lambda x \,.\, \top$ and forward-propagates $\Sigma(\boldsymbol{g}) = \lambda z \,.\, z = 0$ using the first CHC, and finally recursively forward-propagates $\Sigma(\boldsymbol{f}) = \lambda x \,.\, x = 2021$. □

### 4.7 Vacuity Weakening and Additional Candidates

The candidates obtained from previous methods can lead to vacuous solutions (recall Def. 3.4). In order to consider only non-vacuous solutions, Algorithm 1 performs weakening (line 4). This method drops the candidates using the unsat core until the vacuity check passes.

Our technique can easily accommodate candidate specifications derived from other CHC solving techniques like FreqHorn [21] and Spacer [36]. is instantiated in Algorithm 1 (before line 3) to derive additional candidates. A subset of CHCs (fact, query, and inductive CHCs) are passed to it and its solution is used to produce initial candidates. In our implementation, we use a simple grammar in LIA with divisibility constraints:

$$\begin{aligned} lincom &::= const \cdot var + \ldots + const \cdot var \\ cand &::= lincom > const \;\big|\; lincom \geq const \\ &\;\big|\; lincom \;\texttt{mod}\; const = const \end{aligned}$$

More importantly, values for $const$ in the grammar are obtained automatically from the results of the syntax analysis of CHCs, as also done in [21].

**Example 4.8.** Consider the CHCs from Fig. 4b. The technique presented in [1] is not able to solve them as the solution ($f(x) = \lambda x \,.\, \bigvee_{k \in \mathbb{Z}} x = 2 \cdot k$) requires infinite expressions at linear arithmetic (integer or real). However, a technique based on SyGuS generates candidates $\lambda x \,.\, x \bmod 2 = 0$ and $\lambda x \,.\, x \bmod 2 = 1$ based on the transition relation of first CHC as $x'$ is getting incremented by 2. These candidates are then passed to Algorithm 2 which filters out only inductive ones. Finally, they leak into the multi-abduction query, and our algorithm just needs to confirm that it is a solution. □

**Algorithm 5:** maximalSoln

**Input:** CHCs $S$ over $\mathcal{R}$, ordered relations $\mathcal{R}' \subseteq \mathcal{R}$
**Output:** $res \in \{(\text{sat}, MaximalSoln : \mathcal{R} \to 2^{Expr})\}$

```
1 isMax ← ⊥, W ← ∅, Σ ← ∅;
2 while ¬isMax do
3     S′ ← weakeningRules(S, W, Σ);
4     (res, Σ) ← solveCHCs(S′);
5     (isMax, W, CTM) ← isMaximal(S, Σ);
6 return (sat, Σ);
```

**Algorithm 6:** isMaximal

**Input:** CHCs $S$ over $\mathcal{R}$, $\Sigma : \mathcal{R} \to 2^{Expr}$
**Output:** $res \in \{\top, \bot\}$, $\mathcal{W} \subseteq \mathcal{R}$, CTM

```
1 for each r(x⃗) ∈ R do
2     let p⃗_x⃗ be fresh copies of x⃗;
3     Σ^ext(r) ← Σ(r)(x⃗) ∨ p⃗_x⃗ = x⃗;
4 if ∃m . m ⊨ ⋁_{r(x⃗)∈R} ¬Σ(r)(p⃗_x⃗) ∧
    ⋀_{C∈S} ∀args(C) . (body(C)[Σ^ext/R] ⟹ head(C)[Σ^ext/R])
  then
5     W ← {r ∈ R | m ⊨ ¬Σ(r)(p⃗_x⃗)};
6     return (⊥, W, m);
7 return (⊤, ∅, ∅);
```

## 5 Maximal Specification Synthesis

Specifications discovered by our approach from the previous section are guaranteed to be non-vacuous but could be non-maximal. A likely reason for non-maximality is the modularity of CHCs: an interpretation of predicate discovered in one context (even, with abduction), may be unnecessarily strengthened in another context. To check if it is the case, a global reasoning is needed. The technique in this section finds so-called counterexamples-to-maximality (CTM) by checking if there is an interpretation for at least on of the predicates that can be weakened by at least one model. We present our algorithm that takes as input a set of CHCs $S$ over uninterpreted relations $\mathcal{R}$ and discovers maximal solutions automatically.

**Algorithm Overview.** Algorithm 5 begins with initializing auxiliary variables that keep track of the current state of the algorithm. $\Sigma : \mathcal{R} \to 2^{Expr}$ stores the current solution for each relation, $isMax$ denotes whether the current solution is maximal, and $\mathcal{W} \subseteq \mathcal{R}$ keeps track of relations that have to be weakened. Algorithm 5 has a weakening loop (lines 2-5), which runs till the current non-vacuous solution $\Sigma$ is

not maximal. In the weakening loop, method solveCHCs returns a non-vacuous solution for a set of CHCs $S'$ (line 4) and method isMaximal checks if the current non-vacuous solution $\Sigma$ is maximal (line 5).

If the current solution is not maximal, isMaximal returns a CTM and a set of relations $\mathcal{W}$ that have to be weakened in the next iteration. Method weakeningRules adds new CHCs (line 3), such that any solution to $S'$ will be weaker than the current solution. In the first iteration, however, weakeningRules returns the original CHCs $S$ as $\mathcal{W}$ is empty. Hence, the first call to solveCHCs solves the original set of CHCs ($S$). In the subsequent iterations, solveCHCs returns a weaker solution for all the relations in $\mathcal{W}$, while fixing the current solution for other relations. This continues until isMaximal confirms that the current solution is maximal.

Note that $S'$ always has a solution imposed by the CTM. Thus, whenever solveCHCs returns unknown, the algorithm disjoins the equalities produced from satisfying assignments from the CTM to the corresponding relation's current interpretation. If isMaximal fails to decide whether the current solution is maximal, then the algorithm returns unknown.

**The isMaximal Algorithm.** Algorithm 6 describes the method that checks if the current solution $\Sigma$ can be weakened further w.r.t. CHCs $S$. It constructs an SMT formula, which is satisfiable if the current solution is not maximal. Recall that a solution is not maximal if there is another solution that is strictly weaker for at least one relation (Def. 3.5). Based on the satisfying assignment of the formula (which is a CTM), this method also returns a set of relations which can make the solution weaker.

It begins by extending the current solution by disjoining with a new set of placeholder variables (lines 1-3). These variables represent the values that are not part of the current solution but satisfy all the CHCs. If there is at least one such value, then the solution is not maximal. These extended formulas are substituted in place of relations in the input CHCs. This results in a set of universally quantified implications over all the arguments in the corresponding CHCs (note that the placeholder variables are not quantified in them). These implications and a formula that asserts that at least one set of values are not in the current solution, is checked for satisfiability (line 4). If this formula is not satisfiable, then the method returns that current solution is maximal. Otherwise, it checks which placeholder variables have values outside the current solution and returns the corresponding relations (line 6).

**Example 5.1.** Recall the set of CHCs in Fig 2 and the non-vacuous solution $N_2$ in Sect. 2, which has $\Sigma(\boldsymbol{inv}) = \lambda x\,.\,x \leq 19$, $\Sigma(\boldsymbol{f}) = \lambda z\,.\,z = -1$ and $\Sigma(\boldsymbol{g}) = \lambda y\,.\,y = 19$. isMaximal decides that $\Sigma$ is not maximal and can return $\mathcal{W} = \{\boldsymbol{f}, \boldsymbol{g}\}$ by using the model $m(p_y) = 20$, $m(p_z) = 0$ for the following formula:

$$
\begin{aligned}
\forall x\,.\,x = 19 &\implies x \leq 19 \vee p_x = x \wedge \\
\forall x, z, x'\,.\,(x \leq 19 \vee p_x = x) \wedge (z = -1 \vee p_z = z)& \\
\wedge x' = x + z &\implies (x' \leq 19 \vee p_x = x') \wedge \\
\forall x, y\,.\,(x \leq 19 \vee p_x = x) \wedge (y = 19 \vee p_y = y)& \\
\wedge \neg(y \geq x) &\implies \bot \wedge \\
\neg p_x \leq 19 \vee \neg p_z = -1 \vee \neg p_y = 19 & \quad \square
\end{aligned}
$$

**Theorem 5.2** (Correctness of isMaximal)**.** *Algorithm 5 returns true iff the solution is maximal (assuming SMT solver terminates on our formulas).*

*Proof.* $\Rightarrow$ We prove by contradiction. Suppose isMaximal decides that $\Sigma$ is maximal, but it is not. By definition of maximality (Def. 3.6) there exists a solution $\Sigma'$ that is weaker than $\Sigma$. Hence, $\Sigma(r) \implies \Sigma'(r)$ for all the relations and there is at least one relation $r$ for which $\Sigma'(r) \not\Longrightarrow \Sigma(r)$. This implies that $\neg\Sigma(r)(\vec{p}_x)$ is satisfiable. Also, $\Sigma'$ is a solution to all the CHCs (i.e., all implications are satisfiable when relations are substituted by interpretation in $\Sigma'$). However, isMaximal found that this formula is unsatisfiable as it decided that $\Sigma$ is maximal. This contradicts our initial assumption.

$\Leftarrow$ We prove the contrapositive: if isMaximal decides that the $\Sigma$ is not maximal then it is not maximal. It is easy to see that the $\Sigma^{ext}$ is a weaker solution than $\Sigma$ when the placeholder variables are substituted by corresponding satisfying assignment of first SMT formula. isMaximal decides that $\Sigma$ is not maximal only when this formula is satisfiable. $\square$

**Weakening Rules.** Our method for weakening returns a set of CHCs, such that a non-vacuous solution to them is weaker than the current solution $\Sigma$. When $\mathcal{W}$ is empty, it returns the original set of CHCs $S$. Otherwise,

- in $S$, it substitutes each relation outside of $\mathcal{W}$ by its interpretation in $\Sigma$ and then rewrites $S$ to conform to the CHC structure;
- it adds the following two additional CHCs for each relation $r(\vec{x})$ in $\mathcal{W}$:

$$
\begin{aligned}
\Sigma(r)(\vec{x}) &\implies r(\vec{x}) \\
\neg\Sigma(r)(\vec{x}) \wedge p_r(\vec{x}) &\implies r(\vec{x}).
\end{aligned}
$$

In the last additional CHC, $p_r$ is a placeholder relation that does not appear in $\mathcal{R}$. Intuitively, both additional CHCs ensure that the new solution is strictly weaker than the current solution $\Sigma(r)$. This is because by definition of non-vacuous solution (Def. 3.4) the solution of $p_r(\vec{x})$ does not conflict with $\neg\Sigma(r)(\vec{x})$. Hence, $p_r(\vec{x})$ should have at least one satisfying assignment outside of $\neg\Sigma(r)(\vec{x})$. By semantics of implication, a new solution for $r$ should contain this assignment, hence it is strictly weaker than $\Sigma(r)(\vec{x})$. It is easy to see in the following theorem.

**Theorem 5.3** (Correctness of weakeningRules)**.** *Let $S$ be a system of CHCs over $\mathcal{R}$, $\mathcal{W} \subseteq \mathcal{R}$ be a non-empty set of relations*

*to be weakened, $\Sigma$ is a solution, and $S'$ is a set of CHCs obtained from* WEAKENINGRULES. *Then a non-vacuous solution $S'$ is weaker than the solution $\Sigma$ over relations in $\mathcal{W}$.*

**Example 5.4.** Recall the set of CHCs in Fig 2. Let $\Sigma = \{\boldsymbol{inv} \mapsto \lambda x\,.\,x \leq 19, \boldsymbol{f} \mapsto \lambda z\,.\,z = -1, \boldsymbol{g} \mapsto \lambda y\,.\,y = 19\}$, and $\mathcal{W} = \{\boldsymbol{f}, \boldsymbol{g}\}$. Then the new set of CHCs obtained from WEAKENINGRULES is shown below.

$$\begin{aligned}
z = -1 &\implies \boldsymbol{f}(z)\\
z \neq -1 \land \boldsymbol{p_f}(z) &\implies \boldsymbol{f}(z)\\
y = 19 &\implies \boldsymbol{g}(y)\\
y \neq 19 \land \boldsymbol{p_g}(y) &\implies \boldsymbol{g}(y)\\
x \leq 19 \land \boldsymbol{f}(z) \land x' = x + z \land \neg(x \leq 19) &\implies \bot\\
x \leq 19 \land \boldsymbol{g}(y) \land \neg(y \geq x) &\implies \bot
\end{aligned}$$

When these CHCs are given as input to SOLVECHCs, it returns the maximal solution $M_2 = \{\boldsymbol{inv} \mapsto \lambda x\,.\,x \leq 19, \boldsymbol{g} \mapsto \lambda y\,.\,y \geq 19, \boldsymbol{f} \mapsto \lambda z\,.\,z \leq 0\}$. □

Our algorithm can be parameterized by a subset of relations ($R_u \subseteq \mathcal{R}$) on which the maximality constraint should be imposed. Then the method checks if the current non-vacuous solution is maximal for $R_u$, while ensuring there exists a solution for relations in $\mathcal{R} \setminus R_u$.

## 6 Evaluation

**Implementation.** We have implemented the maximality specification synthesis algorithm (Algorithm 5) as a tool HORNSPEC on top of the FREQHORN [20, 21] framework[4]. The tool has an implementation of Algorithm 1 as the default non-vacuous CHC solving. It also supports SyGuS and SMT based non-vacuous CHC solving by using the tools CVC4 [43] and Z3 [16], respectively. HORNSPEC uses Z3 for maximality checks. For quantifier elimination, it uses the implementation of model-based projection [10] for LIA available in Z3.

Our evaluation goals are to confirm that:

- HORNSPEC can generate maximal specifications while exhibiting good performance and
- non-vacuous specifications are generated efficiently and can be extended to maximal specifications.

Towards this, we experimented on 65 non-linear CHC benchmarks with integer variables and LIA operations that represent open programs. Of these benchmarks, 43 were derived from CHC-COMP [23] and 22 from common patterns in many benchmarks from verification competitions [6]. The CHC-COMP benchmarks are CHC-encodings of real programs automatically generated from various tools and adheres to a common format. They are available in public and generally challenging. The original benchmarks represented verification tasks of closed programs, which is not in the scope of this work. Hence, we modified them by removing arbitrary predicates and CHCs, thus generating specification synthesis tasks. Overall, the benchmarks have 7 CHCs and 4 unknown relations in average with a standard deviation of 1 and 0.95, respectively. The benchmark with maximum CHCs has 28 CHCs and 12 unknown relations. Unknown relations in these benchmarks correspond to loop invariants, preconditions and open functions, expressible in LIA. We performed this experiment on an Ubuntu 20.04 machine with 2.5 GHz processor and 16 GB memory. Each benchmark was given a timeout of 100 seconds.

To the best of our knowledge, there are no tools that can discover maximal specifications for a set of CHCs. The CHCs in our benchmarks have inductive structure, hence the technique presented in [1] is not directly applicable. However, our tool is parameterized by a non-vacuous CHC solving technique and that problem can be encoded as SMT of uninterpreted functions and SyGuS constraints (with existential quantification for the vacuity check; see Def. 3.4). So, for comparison, we replaced our non-vacuous CHC solving technique by the Z3 (v4.8.8) SMT solver, and the CVC4 SyGuS solver (v1.8 with the option `--sygus-add-const-grammar`), that can accept existential quantification and is the winner of multiple categories in SyGuS-Comp 2019 [4].

**Non-Vacuous Specifications.** We first report the results of the tools for generating non-vacuous specifications. Here, Algorithm 5 was exited after the generation of first solution (i.e., exit after first execution of line 4 in Algorithm 5). This experiment was done to provide a baseline in performance of non-vacuous specification generation. Out of 65 benchmarks, HORNSPEC was able to solve 60, CVC4 was able to solve 51, and Z3 solved 23. HORNSPEC was able to solve 56 benchmarks within 30 seconds (and the rest 4 within 45 seconds), which shows that HORNSPEC can get non-vacuous solution quickly. The scatter plot in Fig. 5 shows the time taken by different tools. CVC4 was better than HORNSPEC on 9 benchmarks and Z3 on 3 with a time difference of 5 or more seconds. CVC4 was able to solve 3 benchmarks that HORNSPEC could not.

**Maximal Specifications.** We finally report the results for complete runs of Algorithm 5, hence the discovered specifications are maximal. Out of 65 benchmarks, HORNSPEC solved 54, CVC4 solved 23, and Z3 only 5. We have observed that the initial non-vacuous solution generated by HORNSPEC was maximal in 39/54 benchmarks, whereas the remaining 15/54 were solved within one (12/15) or two (3/15) iterations. This demonstrates effectiveness of our CHC solving algorithm in finding initial and weaker solutions. Z3, on the other hand, was stuck in the weakening loop in most cases, and CVC4 was unable to find a weaker specification to the initial non-maximal one. The scatter plot in Fig. 6 shows the time taken by different tools. In terms of time, CVC4 outperformed HORNSPEC in 9 benchmarks and Z3 in 4, however, the *time difference was less than 2 seconds* in all but one benchmark.

[4]The source code and benchmarks can be found at https://github.com/freqhorn/hornspec.

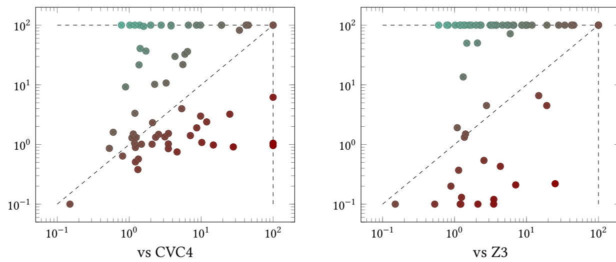


**Figure 5.** HornSpec vs competitors in synthesizing non-vacuous solution. Each round in a plot represents the run times of HornSpec (x-axis) and a competitor (y-axis) in seconds. Timeouts are placed on the boundaries.

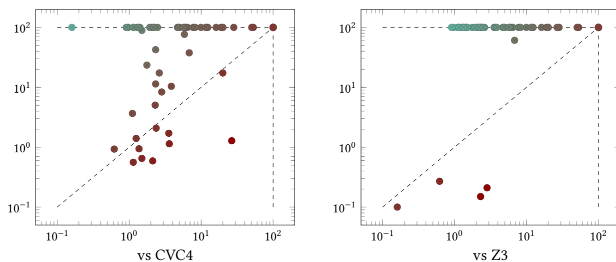


**Figure 6.** HornSpec vs competitors while proving maximal specifications. Legends are same as in Fig. 5.

There was no benchmark on which CVC4 or Z3 was able to find a maximal specification, but HornSpec could not.

## 7 Related Work

Specification synthesis is closely related to the task of logical abduction, which is the inference of missing hypotheses for a given premise and conclusion. It has been used for inferring inductive invariants [17], and also to infer preconditions [11]. In the context of specification synthesis, a generalization is needed to consider multiple abducibles over different vocabularies, which [1] terms as *multi-abduction*. The subject of [1] comes closest to our work as they also address the problem of deriving most general specifications of multiple unknown procedures to ensure a desired safety property. However, their technique is not directly applicable for recursive CHCs, which often arise due to complex control flow, and require additional verification oracles. In contrast, we solve CHCs with inductive structures as well, by the use of an iterative algorithm capable of forward and backward reasoning.

The work on angelic verification [15] [37]) also addresses the problem of open program verification, but depends on a user-supplied set of acceptable specifications, and neither guarantees maximality, nor infers inductive invariants. Our work automatically generates maximal specifications and inductive invariants.

The problem of precondition inference [14, 40, 44, 45] can be posed as a specification synthesis problem by adding a relation over program variables and placing it at the beginning of the program. In comparison our problem is more general, as we also allow unknown functions. Inferring missing specifications is addressed in [7] and [48] as well, though limited to taint analysis. In contrast, our approach is not tied to any particular analysis and we look at safety in general.

Another problem also referred to as specification synthesis is addressed in [3, 5, 29, 46, 48]. These techniques infer so called *call sequences* under which an assertion is safe. It is done by analyzing functions' bodies [3, 29], its usage [46], or program executions [5, 48]. In these techniques, open program refers to a class-like structure with functions. Our work synthesizes logical specifications for functions without bodies (also referred to as open programs), and hence is an orthogonal work.

The technique presented in [28] infers maximal refinement types in user-specified preference order. The predicates in refinement types correspond to specifications. A maximal solution is computed in a loop by continuously improving the current solution until a fixed point is obtained, similar to ours. However, in their problem setting, a procedure body is given on which the refinement type is inferred, akin to closed program verification. Also, they depend on a template-based method to find necessary inductive invariants.

CHC solving has gained a lot of attention in recent years in synthesis of inductive invariants and verification of closed programs (i.e., programs with function calls with known bodies) [18, 21, 22, 31, 32, 36, 41, 49]. But since programs of our interest are open, the existing non-linear CHC solvers generate *vacuous* specifications. Hence, obtaining non-vacuous specifications requires imposition of additional constraints on the generated specifications.

The winning strategy synthesis of safety games [9] can be viewed as a specification synthesis problem, albeit in a richer logical formalism of existential Horn clauses, for which our present technique can be extended in future.

## 8 Conclusion

We have presented a novel approach to discover maximal and non-vacuous specifications. We treated this problem as an instance of a CHC problem and proposed an iterative generalization procedure in which non-vacuous solutions are combined until a maximal one is found. The core of our approach is a non-vacuous CHC solver that synthesises solutions by alternating forward and backward reasoning. It uses multi-abduction, SyGuS, and Houdini to discover inductive lemmas for (conjunctive) invariants, and it propagates them to the final specifications. Empirically, we demonstrated that our technique is both effective and efficient over a wide space of benchmarks originated from CHC-COMP. We are currently investigating the challenges and benefits in adapting our approach for other related problems such as quantified specifications for programs handling arrays and strategy synthesis for safety games.

## References


[1] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *POPL*. ACM, 789–801. https://doi.org/10.1145/2914770.2837628

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *FMCAD*. IEEE, 1–17. https://doi.org/10.3233/978-1-61499-495-4-1

[3] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. In *POPL*. ACM, 98–109. https://doi.org/10.1145/1040305.1040314

[4] Rajeev Alur, Dana Fisman, Saswat Padhi, Andrew Reynolds, Rishabh Singh, and Abhishek Udupa. 2019. SyGuS-Comp 2019. https://sygus.org/comp/2019/results-slides.pdf.

[5] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *POPL*. ACM, 4–16. https://doi.org/10.1145/503272.503275

[6] Ezio Bartocci, Dirk Beyer, Paul E. Black, Grigory Fedyukovich, Hubert Garavel, Arnd Hartmanns, Marieke Huisman, Fabrice Kordon, Julian Nagele, Mihaela Sighireanu, Bernhard Steffen, Martin Suda, Geoff Sutcliffe, Tjark Weber, and Akihisa Yamada. 2019. TOOLympics 2019: An Overview of Competitions in Formal Methods. In *TACAS, Part III (LNCS)*, Vol. 11429. Springer, 3–24. https://doi.org/10.1007/978-3-030-17502-3_1

[7] Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification Inference Using Context-Free Language Reachability. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 553–566. https://doi.org/10.1145/2676726.2676977

[8] Nels E. Beckman and Aditya V. Nori. 2011. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI*. ACM, 211–221. https://doi.org/10.1145/1993498.1993524

[9] Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014. A constraint-based approach to solving games on infinite graphs. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 221–234. https://doi.org/10.1145/2535838.2535860

[10] Nikolaj Bjørner and Mikolás Janota. 2015. Playing with Quantified Satisfaction. In *LPAR (short papers) (EPiC Series in Computing)*, Vol. 35. EasyChair, 15–27. https://easychair.org/publications/paper/jmM

[11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[12] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2018. ICE-Based Refinement Type Discovery for Higher-Order Functional Programs. In *TACAS, Part I (LNCS)*, Vol. 10805. Springer, 365–384. https://doi.org/10.1007/978-3-319-89960-2_20

[13] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. 2013. Parameter synthesis with IC3. In *FMCAD*. IEEE, 165–168. http://ieeexplore.ieee.org/document/6679406/

[14] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. 128–148. https://doi.org/10.1007/978-3-642-35873-9_10

[15] Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. 2015. Angelic Verification: Precise Verification Modulo Unknowns. In *CAV, Part I (LNCS)*, Vol. 9206. Springer, 324–342. https://doi.org/10.1007/978-3-319-21690-4_19

[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[17] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *OOPSLA*. ACM, 443–456. https://doi.org/10.1145/2509136.2509511

[18] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. 2018. Horn-ICE learning for synthesizing invariants and contracts. *PACMPL* 2, OOPSLA (2018), 131:1–131:25. https://doi.org/10.1145/3276501

[19] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodík. 2017. Gradual Synthesis for Static Parallelization of Single-Pass Array-Processing Programs. In *PLDI*. ACM, 572–585. https://doi.org/10.1145/3062341.3062382

[20] Grigory Fedyukovich, Samuel Kaufman, and Rastislav Bodík. 2017. Sampling Invariants from Frequency Distributions. In *FMCAD*. IEEE, 100–107. https://doi.org/10.23919/FMCAD.2017.8102247

[21] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2018. Solving Constrained Horn Clauses Using Syntax and Data. In *FMCAD*. IEEE, 170–178. https://doi.org/10.23919/FMCAD.2018.8603011

[22] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2019. Quantified Invariants via Syntax-Guided Synthesis. In *CAV, Part I (LNCS)*, Vol. 11561. Springer, 259–277. https://doi.org/10.1007/978-3-030-25540-4_14

[23] Grigory Fedyukovich, Philipp Rümmer, and Arie Gurfinkel. 2019. CHC-COMP. https://chc-comp.github.io/2019/chc-comp19.pdf.

[24] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini: an Annotation Assistant for ESC/Java. In *FME (LNCS)*, Vol. 2021. Springer, 500–517. https://doi.org/10.1007/3-540-45251-6_29

[25] Pierre-Loïc Garoche, Temesghen Kahsai, and Xavier Thirioux. 2016. Hierarchical State Machines as Modular Horn Clauses. In *HCVS (EPTCS)*, Vol. 219. 15–28. https://doi.org/10.4204/EPTCS.219.2

[26] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018), 48:1–48:28. https://doi.org/10.1145/3158136

[27] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *CAV (LNCS)*, Vol. 9206. Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

[28] Kodai Hashimoto and Hiroshi Unno. 2015. Refinement Type Inference via Horn Constraint Optimization. In *SAS (LNCS)*, Sandrine Blazy and Thomas P. Jensen (Eds.), Vol. 9291. Springer, 199–216. https://doi.org/10.1007/978-3-662-48288-9_12

[29] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2005. Permissive interfaces. In *ESEC/SIGSOFT FSE*. ACM, 31–40. https://doi.org/10.1145/1081706.1081713

[30] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. 2012. A Verification Toolkit for Numerical Transition Systems - Tool Paper. In *FM (LNCS)*, Vol. 7436. Springer, 247–251. https://doi.org/10.1007/978-3-642-32759-9_21

[31] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *FMCAD*. IEEE, 158–164. https://doi.org/10.23919/FMCAD.2018.8603013

[32] Bishoksan Kafle, John P. Gallagher, and Pierre Ganty. 2016. Solving non-linear Horn clauses using a linear Horn clause solver. In *Proceedings 3rd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2016, Eindhoven, The Netherlands, 3rd April 2016*. 33–48. https://doi.org/10.4204/EPTCS.219.4

[33] Temesghen Kahsai, Rody Kersten, Philipp Rümmer, and Martin Schäf. 2017. Quantified Heap Invariants for Object-Oriented Programs. In *LPAR (EPiC Series in Computing)*, Vol. 46. EasyChair, 368–384. https://easychair.org/publications/paper/Pmh

[34] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A Framework for Verifying Java programs. In *CAV, Part I (LNCS)*, Vol. 9779. Springer, 352–358. https://doi.org/10.

1007/978-3-319-41528-4_19

[35] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *ACM*. ACM, 222–233. https://doi.org/10.1145/1993498.1993525

[36] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *CAV (LNCS)*, Vol. 8559. 17–34. https://doi.org/10.1007/978-3-319-08867-9_2

[37] Shuvendu K Lahiri, Akash Lal, Sridhar Gopinath, Alexander Nutz, Vladimir Levin, Rahul Kumar, Nate Deisinger, Jakob Lichtenberg, and Chetan Bansal. 2020. Angelic Checking within Static Driver Verifier: Towards high-precision defects without (modeling) cost. In *FMCAD*. IEEE. https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_24 169–178.

[38] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-Based Verification for Rust Programs. In *ESOP (LNCS)*, Peter Müller (Ed.), Vol. 12075. Springer, 484–514. https://doi.org/10.1007/978-3-030-44914-8_18

[39] Dmitry Mordvinov and Grigory Fedyukovich. 2017. Verifying Safety of Functional Programs with Rosette/Unbound. *CoRR* abs/1704.04558 (2017). https://github.com/dvvrd/rosette.

[40] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices* 51, 6 (2016), 42–56. https://doi.org/10.1145/2908080.2908099

[41] Sumanth Prabhu, Kumar Madhukar, and R Venkatesh. 2018. Efficiently learning safety proofs from appearance as well as behaviours. In *SAS (LNCS)*, Vol. 11002. Springer, 326–343. https://doi.org/10.1007/978-3-319-99725-4_20

[42] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. In *PLDI*. ACM, 123–134. https://doi.org/10.1145/1250734.1250749

[43] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. 2019. cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In *CAV, Part II (LNCS)*, Vol. 11562. Springer, 74–83. https://doi.org/10.1007/978-3-030-25543-5_5

[44] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivancic, and Aarti Gupta. 2008. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. 295–306. https://doi.org/10.1145/1390630.1390666

[45] Mohamed Nassim Seghir and Daniel Kroening. 2013. Counterexample-Guided Precondition Inference. In *ESOP (Lecture Notes in Computer Science)*, Vol. 7792. Springer, 451–471. https://doi.org/10.1007/978-3-642-37036-6_25

[46] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. 2007. Static specification mining using automata-based abstractions. In *ISSTA*. ACM, 174–184. https://doi.org/10.1145/1273463.1273487

[47] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*. ACM, 282–291. https://doi.org/10.1145/1134285.1134325

[48] Haiyan Zhu, Thomas Dillig, and Isil Dillig. 2013. Automated Inference of Library Specifications for Source-Sink Property Verification. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*. 290–306. https://doi.org/10.1007/978-3-319-03542-0_21

[49] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *PLDI*. ACM, 707–721. https://doi.org/10.1145/3192366.3192416