EPiC Series in Computing

Volume 94, 2023, Pages 288–305

Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Collaborative Inference of Combined Invariants

Yurii Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich[1]

[1] Florida State University, Tallahassee, Florida, U.S.A.

Abstract

Inductive invariant inference is the fundamental problem in program verification, and specifically in verification of functional programs that use nonlinear recursion and algebraic data types (ADTs). For ADTs, it is challenging to come up with an abstract domain that is rich enough to represent program properties and a procedure for invariant inference which is effective for this domain. Although there are various techniques for different abstract domains for ADTs, they often diverge while analyzing real-life programs because of low expressivity of their abstract domains. Moreover, it is often unclear if they could complement each other, other than by running in a portfolio. We present a lightweight approach to combining any existing techniques for different abstract domains collaboratively, thus targeting a more expressive domain. We instantiate the approach and obtain an effective inductive invariant inference algorithm in a rich combined domain of elementary and regular ADT invariants essentially for free. Because of the richer domain, collaborations of verifiers are capable of solving problems that are beyond the capabilities of the collaborators running independently. Our implementation of the algorithm is a collaboration of two existing state-of-the-art inductive invariant inference engines having general-purpose first-order logic solvers as a backend. Finally, we show that our implementation is capable of solving a large amount of CHC-Comp 2022 problems obtained from Haskell verification problems, for which the existing tools diverge.

1 Introduction

Programs handling algebraic data types (ADT) are known to be challenging for inductive invariant inference, especially because of the recursive structure of ADTs. State-of-the-art approaches based on *elementary invariants* (represented in first-order logic) [28, 25, 7] aim at constructing a coarse abstraction which is not sufficient for proving functional correctness of programs with ADTs. On the other hand, a recent approach of the RInGen verifier [30] allows one to infer *nonelementary ADT invariants* by employing general-purpose automated theorem provers like cvc5 [35, 3] and Vampire [31]. In particular, nonelementary *regular invariants* are effective in capturing the recursive semantics of ADTs using automata but incapable of expressing relational properties, e.g., involving equalities among variables like in first-order logic (FOL). To mitigate these drawbacks, there is a need for a new approach to infer *combined inductive invariants*, i.e., inductive invariants in a combination of elementary and nonelementary domains. In practice, such a rich domain gives a verifier an ability to converge on a larger class of real-life programs.

Creating a verifier for such a combined domain *from scratch* is exceedingly hard. Instead, we show how to obtain it by a minor change in *any* CEGAR-based elementary invariant inference algorithm for transition systems. To that end, we introduce a novel approach for *collaborative inductive invariant inference*. It applies in a setting of any two verifiers A and B for different abstract domains, where A is an instance of CEGAR and B is an auxiliary verifier, and both A and B are called *collaborators*. Our main insight then is to modify verifier A slightly, so it could check the safety of a transition system by exchanging the information with collaborator B. This is done by computing a sequence of simpler *residual transition systems* in verifier A and checking their safety with verifier B. If at least one of the collaborators is able to verify safety of a transition system, then the whole collaboration succeeds as well. We instantiated this approach for ADTs to infer combined invariants by making use of verifier A to infer the first-order part and verifier B to infer their nonelementary part. Thus, we get an algorithm which is strictly more powerful than each collaborator alone, i.e., for many cases in practice where both verifiers diverge on their own, but the collaboration still succeeds in verifying the system.

We have implemented this algorithm on top of the Racer [22] and the RInGen verifiers with two general-purpose theorem provers at the backend, namely cvc5 [35, 3] (finite model finding engine) and Vampire [31]. We evaluated it on benchmarks obtained from Haskell verification problems that are publicly available at the Competition of solvers for satisfiability of Constrained Horn Clauses (CHC-Comp 2022). Our collaboration of two verifiers solves significantly more benchmarks than a parallel composition of the corresponding verifiers running independently. Thus, our collaboration solves many safety problems that are beyond the capabilities of both verifiers.

To sum up, our core contributions in this paper are as follows.

1. We present the novel approach to collaborative inductive invariant inference.
2. We instantiate it for combining invariant inference procedures for elementary and nonelementary domains over algebraic data types and get a novel algorithm for automated inference of inductive invariants in this combined domain.
3. We implemented this algorithm in the Racer and the RInGen verifiers.
4. We demonstrate the practical success of this approach on CHC-Comp 2022 benchmarks.

The rest of the paper is structured as follows. In Sec. 2 we give the necessary background to the problem. In Sec. 3 we describe the core idea of our approach, abstracting away all details. Sec. 4 adds the missing details, accomplishing the theoretical presentation of our approach. Sec. 5 describes our implementation, and in Sec. 6 we present our experiments with the implementation. Finally, in Sec. 7 we discuss related work and Sec. 8 concludes the paper.

2 Background

2.1 Automated Invariants Inference

We view programs as transition systems. Let $\langle \mathcal{S}, \subseteq, 0, 1, \cap, \cup, \neg \rangle$ be a complete Boolean lattice.

Definition 1. A *transition system* is a triple $TS = \langle \mathcal{S}, Init, T \rangle$, where $Init \in \mathcal{S}$ are initial states and $T : \mathcal{S} \to \mathcal{S}$ is a *transition function*, such that:

- T is monotonic, i.e., $s_1 \subseteq s_2$ implies $T(s_1) \subseteq T(s_2)$,

Input: a program TS and a property $Prop$
Output: SAFE with an inductive invariant or UNSAFE with a counterexample

```
1 ⟨α, γ⟩ ← Initial()
2 while true do
3     cex, A ← ModelCheck(TS, Prop, ⟨α, γ⟩)
4     if cex is empty then
5         return safe, A
6     if IsFeasible(cex) then
7         return unsafe, cex
8     ⟨α, γ⟩ ← Refine(⟨α, γ⟩, cex)
```

Algorithm 1: CEGAR: a classic approach to compute invariants.

- T is additive, i.e., $T(s_1 \cup s_2) = T(s_1) \cup T(s_2)$, and
- $T(0) = Init$.

A *safety problem* is a pair of a program TS and a property $Prop \in \mathcal{S}$. A program is *safe* with respect to a property iff for all n, $T^n(Init) \subseteq Prop$. Otherwise it is *unsafe*. Safety can be witnessed by a *(safe) inductive invariant* $I \in \mathcal{S}$, such that

$$Init \subseteq I, \qquad T(I) \subseteq I, \qquad I \subseteq Prop.$$

It is well known that the program is safe if and only if it has a safe inductive invariant. In order to infer invariants automatically, one usually fixes some *class of invariants* $\mathcal{L} \subseteq \mathcal{S}$. A *verifier* is an algorithm that returns an invariant in a class of invariants $\mathcal{L}$ for safe programs and counterexamples for unsafe programs. We refer to $\mathcal{L}$ as to the *domain* of verifier. Note that a verifier can diverge, e.g., in a case when the program is safe, but there is no invariant in the domain witnessing safety.

Let $\mathcal{A} = \langle A, \sqsubseteq, \bot_{\mathcal{A}}, \top_{\mathcal{A}}, \sqcap, \sqcup \rangle$ be a complete lattice of *abstract states*, which we refer to as to an abstract domain. Given posets $\langle \mathcal{S}, \subseteq \rangle$ and $\langle \mathcal{A}, \sqsubseteq \rangle$, a *Galois connection* (called *abstraction*) is a pair of maps $\langle \alpha, \gamma \rangle$ such that:

$$\alpha : \mathcal{S} \to \mathcal{A}, \qquad \gamma : \mathcal{A} \to \mathcal{S}, \qquad \forall x \in \mathcal{S}. \forall y \in \mathcal{A}.\ \alpha(x) \sqsubseteq y \Leftrightarrow x \subseteq \gamma(y).$$

An abstract transition function $\hat{T} : \mathcal{A} \mapsto \mathcal{A}$ lifts the transition function in the abstract domain with a possible overapproximation, i.e., for all $a \in \mathcal{A}$, $\alpha(T(\gamma(a))) \sqsubseteq \hat{T}(a)$. An abstract domain together with a Galois connection uniquely define a class of invariants $\{\gamma(a) \mid a \in \mathcal{A}\}$. We slightly abuse the notation and call this class $\mathcal{A}$ as well. For a program $TS = \langle \mathcal{S}, Init, T \rangle$ and a property $Prop$, we assume that there exists an element $a \in \mathcal{A}$, then $\gamma(a)$ is an inductive invariant of $\langle TS, Prop \rangle$ iff

$$\alpha(Init) \sqsubseteq a, \qquad \hat{T}(a) \sqsubseteq a, \qquad \gamma(a) \subseteq Prop.$$

We also require checks of the form $\gamma(a) \subseteq Prop$ to be computable.

Classically, verification is achieved using Counter-Example Guided Abstraction Refinement (CEGAR) [10] pseudocode of which is shown in Algorithm 1. It begins with building an initial abstraction $\langle \alpha, \gamma \rangle$, e.g., by trivial mappings like $\forall s.\alpha(s) = \bot_{\mathcal{A}}$ and $\forall a.\gamma(a) = 1$. From a concrete program and its abstraction, the algorithm builds a finite sequence of abstract states $\overline{a} = \langle a_0, \ldots, a_n \rangle$ such that:

$$a_0 = \alpha(Init) \quad \text{and} \quad a_{i+1} = a_i \sqcup \hat{T}(a_i) \quad \forall i \in \{0, \ldots, n-1\} \tag{1}$$

If at some point $\gamma(a_i) \not\subseteq Prop$ then an *abstract counterexample cex* is returned with either $A = 0$ (if $i = 0$) or $A = \gamma(a_{i-1})$, which still satisfies $\gamma(a_{i-1}) \subseteq Prop$. If for all i, $\gamma(a_i) \subseteq Prop$, and at some step $\hat{T}(a_n) \sqsubseteq a_n$, then $\gamma(a_n)$ is an inductive invariant, so ModelCheck returns it as A

and no *cex*. The notion of abstract counterexample is defined by each particular instantiation of CEGAR. Observe from the above, that a returned value of ModelCheck satisfies the following:

$$A = 0 \quad \text{or} \quad \mathit{Init} \subseteq A \subseteq \mathit{Prop}. \tag{2}$$

If no abstract counterexample exists, then the program is safe and $\gamma(a_n)$ is an inductive invariant. Otherwise, an abstract counterexample should be checked for feasibility. If there exists a corresponding *concrete* counterexample, CEGAR halts with a counterexample, and otherwise it proceeds to iteratively refining abstraction $\langle \alpha, \gamma \rangle$ to exclude counterexample *cex*.

2.2 First-Order Theory of Algebraic Data Types

A many-sorted ADT signature is a pair $\Sigma = \langle \Sigma_S, \Sigma_F \rangle$, where Σ_S is a set of sorts and Σ_F is a set of function symbols (constructors)[1]. Each function symbol $f \in \Sigma_F$ has an associated sort $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ (possibly with $n = 0$), where $\sigma_i, \sigma \in \Sigma_S$. From now on, we consider Σ fixed.

We denote a set of ground Σ-terms with sort σ by $\mathbb{T}_\sigma$. By $\mathcal{C}$ we denote an *assertion language*, the many-sorted first-order language of Σ-formulas over predicates $\{=_\sigma |\ \sigma \in \Sigma_S\}$ (throughout the paper, we omit subscripts of $=$ since they are clear from the context). By $\mathcal{H}$ we denote a structure with domains $\mathbb{T}_\sigma$ for each sort $\sigma \in \Sigma_S$ that interprets every ground term with itself, and every symbol $=_\sigma$ with $\mathcal{H}(=_\sigma) = \{(x, x) \mid x \in \mathbb{T}_\sigma\}$. A formula φ in the assertion language is *satisfiable modulo theory of ADTs*, if $\mathcal{H} \models \varphi$.

A formula φ' is a *ground instantiation* of a formula φ if φ' is obtained from φ by substituting every free variable in φ with some ground term. A *relation, defined by formula* φ, denoted $\mathcal{H}(\varphi)$, is a set $\{\overline{a} \mid \mathcal{H} \models \varphi(\overline{a})\}$. By $\forall\varphi$ we denote the universal closure of φ.

2.3 Constrained Horn Clauses

Let $\mathcal{R} = \{P_1, \ldots, P_n\}$ be a finite set of predicate symbols. Each symbol $P \in \mathcal{R}$ is associated with some sort $\sigma_1 \times \cdots \times \sigma_n$ from Σ. Predicate symbols from $\mathcal{R}$ are called *uninterpreted*.

Definition 2. A *Constrained Horn Clause* (CHC) C is a $\Sigma \cup \mathcal{R}$-formula of the form:

$$\varphi \wedge P_1(\overline{x}_1), \ldots, P_n(\overline{x}_n) \to H$$

where $\mathcal{C}$-formula φ is called a *constraint* of C, each $P_i \in \mathcal{R}$, $\overline{x}_i$ are tuples of pairwise different variables, and H, called a *head*, is either $\bot$ or an atomic formula $P(\overline{x})$ for some $P \in \mathcal{R}$ and variables $\overline{x}$. The premise of a clause C is called its *body* denoted $\mathit{body}(C)$. If $H = \bot$, we say that C is a *query clause*. A CHC system $\mathcal{P}$ is a finite set of CHCs with (for presentation simplicity) a *single query clause*. Let $\mathit{rules}(P)$ be the set of clauses with P in head ($P \in \mathcal{R}$).

Satisfiability of CHCs. Let $P \in \mathcal{R}$ be associated with sort $\sigma_1 \times \cdots \times \sigma_m$. We denote the set $\mathbb{T}_{\sigma_1} \times \ldots \times \mathbb{T}_{\sigma_m}$ by $\mathbb{T}_P$. Now we define a complete Boolean lattice of $\langle \mathcal{S}, \subseteq, 0, 1, \cap, \cup, \neg \rangle$, which will serve us as a concrete domain of CHC transition semantics.

$$\mathcal{S} \stackrel{\text{def}}{=} \text{a set of all mappings from every } P \in \mathcal{R} \text{ to a subset of } \mathbb{T}_P$$

$$s_1 \subseteq s_2 \Leftrightarrow \forall P \in \mathcal{R} \quad s_1(P) \subseteq s_2(P) \qquad s_1 \cap s_2 \stackrel{\text{def}}{=} \{P \mapsto s_1(P) \cap s_2(P) \mid P \in \mathcal{R}\}$$

$$0 \stackrel{\text{def}}{=} \{P \mapsto \varnothing \mid P \in \mathcal{R}\} \qquad s_1 \cup s_2 \stackrel{\text{def}}{=} \{P \mapsto s_1(P) \cup s_2(P) \mid P \in \mathcal{R}\}$$

$$1 \stackrel{\text{def}}{=} \{P \mapsto \mathbb{T}_P \mid P \in \mathcal{R}\} \qquad \neg s \stackrel{\text{def}}{=} \{P \mapsto \mathbb{T}_P \setminus s(P) \mid P \in \mathcal{R}\}$$

[1] For simplicity, we omit the selectors and testers from the assertion language because they do not increase its expressiveness.

By $\mathcal{H}\{P_1 \mapsto X_1, \ldots, P_n \mapsto X_n\}$ we denote the $\Sigma \cup \mathcal{R}$-expansion of Σ-structure $\mathcal{H}$ from Sec. 2.2, which interprets every symbol P_i with relation X_i. We say that a system of CHCs $\mathcal{P}$ is *satisfied by* $s \in \mathcal{S}$ modulo theory of ADTs, written as $s \models \mathcal{P}$, if for each $C \in \mathcal{P}$, $\mathcal{H}\{P_1 \mapsto s(P_1), \ldots, P_n \mapsto s(P_n)\} \models \forall C$. We say that system $\mathcal{P}$ is *satisfiable* iff $s \models \mathcal{P}$ for some $s \in \mathcal{S}$.

Transition semantics of CHCs. A CHC system $\mathcal{P}$ defines the transition system $\langle \mathcal{S}, Init, T \rangle$, where $Init \stackrel{\text{def}}{=} T(0)$ and

$$T(s)(P) \stackrel{\text{def}}{=} \left\{ \bar{t} \mid \left(B \to P(\bar{t})\right) \text{is a ground instance of some clause from } \mathcal{P}, s \models B \right\}.$$

Without loss of generality, we assume that each CHC system $\mathcal{P}$ is transformed to have a single query predicate Q, that is,

$$\mathcal{P}' \stackrel{\text{def}}{=} rules(\mathcal{P}) \cup \{body(C)(\overline{x}) \to Q(\overline{x}) \mid C \text{ is a query clause of } \mathcal{P}\} \cup \{Q(\overline{x}) \to \bot\}.$$

A property is thus defined as $Prop(Q) \stackrel{\text{def}}{=} \bot$ and for each $P \in \mathcal{R}$, $Prop(P) \stackrel{\text{def}}{=} \top$.

Proposition 1. *A CHC system $\mathcal{P}$ is satisfiable iff the corresponding transition system $\langle \mathcal{S}, Init, T \rangle$ is safe with respect to Prop.*

Example 1 ($ForkJoin$)**.** Consider CHCs over an ADT $Prog ::= Seq \mid Fork(Prog) \mid Join(Prog)$, describing a concurrent program transformation.

$$\begin{aligned} p = Seq &\to ok(p) \\ p' = Fork(Join(p)) \wedge ok(p) &\to ok(p') \\ p = Seq \wedge t = Fork(p') &\to tr(p,t) \\ t = Join(Seq) &\to tr(p,t) \\ p' = Fork(Join(p)) \wedge t' = Fork(Join(t)) \wedge tr(p,t) &\to tr(p',t') \\ ok(p) \wedge tr(p,p) &\to \bot \end{aligned}$$

In this system, $\mathcal{R} = \{ok, tr\}$, rules are all clauses except the last one, which is a query clause. This system is satisfied by s, where $s(ok) = \mathcal{E}$ and $s(tr) = \{(p,t) \mid p \neq t \vee t \notin \mathcal{E}\}$, where $\mathcal{E}$ is the least fixpoint of the ok predicate.

2.4 Elementary Domain

The elementary domain is commonly used in the CHC solving community, e.g., by state-of-the-art tools like Spacer [29], Eldarica [25], and Hoice [7]. More recent Racer [22] is also based on the enrichment of *this* domain with recursive functions.

We say that a relation $X \subseteq \mathbb{T}_{\sigma_1} \times \ldots \times \mathbb{T}_{\sigma_m}$ is $\mathcal{C}$-definable (*elementary*) if there is a $\mathcal{C}$-formula φ such that $\mathcal{H}(\varphi) = X$. A model $s \in \mathcal{S}$ is $\mathcal{C}$-definable (*elementary*) if for each $P \in \mathcal{R}$, $s(P)$ is elementary. We denote the class of all $\mathcal{C}$-definable models with Elem. *An elementary abstract domain* is $\mathcal{A}_{\text{Elem}} = \langle \text{Elem}, \sqsubseteq, \bot_{\text{Elem}}, \top_{\text{Elem}}, \sqcap, \sqcup \rangle$, where:

$$a_1 \sqsubseteq a_2 \Leftrightarrow \text{for all } P, \quad \mathcal{H} \models a_1(P) \to a_2(P) \qquad \gamma(a) \stackrel{\text{def}}{=} \{P \mapsto \mathcal{H}(a(P))\}$$

$$\bot_{\text{Elem}} \stackrel{\text{def}}{=} \{P \mapsto \bot\} \qquad \top_{\text{Elem}} \stackrel{\text{def}}{=} \{P \mapsto \top\}$$

$$a_1 \sqcup a_2 \stackrel{\text{def}}{=} \{P \mapsto a_1(P) \vee a_2(P)\} \qquad a_1 \sqcap a_2 \stackrel{\text{def}}{=} \{P \mapsto a_1(P) \wedge a_2(P)\}.$$

3 Collaborative Invariant Inference: Core Idea

For programs with ADTs, the expressiveness issue of verifier domains is a typical source of verifier divergence besides the undecidability of verification itself. As motivated in [30], invariant inference approaches for ADT problems rely on verifier domains, which lack expressivity

Parameters: Verifier $\mathcal{O}$ over domain $\mathcal{B}$
Input: a program TS and a property $Prop$
Output: SAFE with a combined invariant in $\mathcal{A} \uplus \mathcal{B}$ or UNSAFE with cex

```
1  ⟨α, γ⟩ ← Initial()
2  A ← 0
3  while true do
4      async call Collaborate(TS, Prop, ⟨α, γ⟩, A)
5      cex, A ← ModelCheck(TS, Prop, ⟨α, γ⟩)
6      if cex is empty then
7          return safe, A
8      if IsFeasible(cex) then
9          return unsafe, cex
10     ⟨α, γ⟩ ← Refine(⟨α, γ⟩, cex)
```

Algorithm 2: CEGAR($\mathcal{O}$).

themselves. However, they might be fruitfully combined. Yet there has not been an approach to infer invariants by combining verifier efforts, as it is not straightforward how to make the existing verifiers collaborate.

In this section, we give an overview of our novel approach to collaborative inference of combined invariants, and we defer the technicalities to the next section. We keep it general enough to be potentially applicable to verifiers with different abstract domains, and even to verifiers for programs with features beyond ADTs.

We call the approach CEGAR($\mathcal{O}$) as the collaboration process can be viewed as CEGAR that queries some oracle $\mathcal{O}$, which might not terminate. Let the domains of verifiers be classes $\mathcal{A}$ and $\mathcal{B}$ respectively. CEGAR($\mathcal{O}$) allows us to construct invariants in the union of these classes.

Definition 3. For classes of states $\mathcal{A} \subseteq \mathcal{S}$ and $\mathcal{B} \subseteq \mathcal{S}$, a *combined class* of states is

$$\mathcal{A} \uplus \mathcal{B} \stackrel{\text{def}}{=} \{A \cup B \mid A \in \mathcal{A}, B \in \mathcal{B}\}.$$

A *combined inductive invariant* in classes $\mathcal{A}$ and $\mathcal{B}$ is an inductive invariant in $\mathcal{A} \uplus \mathcal{B}$.

CEGAR($\mathcal{O}$) Algorithm 2 gives pseudocode of our approach. The algorithm is similar to the classic CEGAR one, as we have presented in Algorithm 1, but at the beginning of every iteration it asynchronously queries the collaborating verifier $\mathcal{O}$ by calling the Collaborate routine (line 4). Asynchronous calls prevent the algorithm from getting stuck while continuously refining the abstraction.

The **Collaborate** procedure is shown in Algorithm 3. Given the original safety problem, the current abstraction, and $A = \gamma(a)$ for some $a \in \mathcal{A}$, it forms a new *residual* transition system

$$TS' = \langle \mathcal{S}, Init', T' \rangle = \langle \mathcal{S}, T(A) \setminus A,\ \lambda B.\, (T(B) \setminus A) \rangle .$$

The safety of the residual system is then checked by the collaborating verifier $\mathcal{O}$. Here $A \setminus B$ is a shorthand for $A \cap \neg B$. Note that Algorithm 3 overwrites the abstraction used in Algorithm 2 (line 6), i.e., $\langle \alpha, \gamma \rangle$ is global and shared between two procedures.

Intuitively, the residual transition system describes states that are reachable from the states that violate the inductiveness of A. Indeed, initial states $Init'$ are $T(A) \setminus A$, one-step image of non-inductive states. $T'(Init') = T(T(A) \setminus A) \setminus A$, a set of two-step images of non-inductive states. Our insight is to use another verifier to *weaken* the non-inductive set of states A to some fixpoint in a combined class. If the second verifier finds the inductive overapproximation B of

Parameters: Verifier $\mathcal{O}$ over domain $\mathcal{B}$
Input: Program $TS = \langle \mathcal{S}, Init, T \rangle$, property $Prop$, abstraction $\langle \alpha, \gamma \rangle$, set of states A, such that $A = 0$ or $Init \subseteq A \subseteq Prop$

1 $TS' \leftarrow \langle \mathcal{S}, T(A) \setminus A, \lambda B.\, (T(B) \setminus A) \rangle$
2 $B, cex \leftarrow \mathcal{O}(TS', Prop)$
3 **if** *cex is empty* **then**
4 $\quad$ **halt with** SAFE, $A \cup B$
5 $\widehat{cex} \leftarrow$ RecoverCex $(TS, Prop, \langle \alpha, \gamma \rangle, A, cex)$
6 $\langle \alpha, \gamma \rangle \leftarrow$ Refine$(\langle \alpha, \gamma \rangle, \widehat{cex})$

Algorithm 3: Collaborate subroutine.

non-inductive states, then $A \cup B$ is an inductive invariant of the original system. Intuitively, we split the safe, yet non-inductive, part of the current invariant candidate and let the collaborating verifier to fill holes in it to accomplish fixpoint construction.

If $\mathcal{O}$ halts with inductive invariant B, then we yield the combined invariant $A \cup B$. If $\mathcal{O}$ returns the concrete counterexample cex, Collaborate recovers a new abstract counterexample $\widehat{cex}$ from it and then proceeds by refining the domain with $\widehat{cex}$.

Note that neither set of states A nor B is sufficient to separately prove the safety of the original transition system. Intuitively, it means that the collaboration is done by delegating *simpler* problems to the verifier $\mathcal{O}$, the solution to which gives only one part of an answer.

Lemma 1. *If* Collaborate $(TS, Prop, \langle \alpha, \gamma \rangle, a)$ *halts with* $SAFE(A \cup B)$ *(line 4), then* $A \cup B$ *is a combined invariant of* $\langle TS, Prop \rangle$.

Proof. Initial. From (2) we have $Init \subseteq A \subseteq A \cup B$ or $A = 0$ and then by the definition of $T(0)$, $Init = T(0) \setminus 0 = T(A) \setminus A \subseteq B \subseteq A \cup B$.

Transition. From soundness of $\mathcal{O}$, B is an inductive invariant of $(\langle \mathcal{S}, Init', T' \rangle, Prop)$, i.e., $T(A) \setminus A \subseteq B$ ($Init'$ definition) and $T(B) \setminus A \subseteq B$ (T' definition), so $(T(A) \cup T(B)) \setminus A \subseteq B$, from which we have $T(A) \cup T(B) \subseteq A \cup B$, hence as T is additive, $T(A \cup B) \subseteq A \cup B$.

Property. $A \subseteq Prop$ from (2) and $B \subseteq Prop$ by soundness of $\mathcal{O}$, so $A \cup B \subseteq Prop$. □

Counterexamples to the residual safety problem are traces that violate the inductiveness of a current invariant candidate A. That is, a *concrete* counterexample to safety of the residual system (i.e., *cex* at line 2 of Algorithm 3) corresponds to some *abstract* counterexample. CEGAR($\mathcal{O}$) is parametrized by the RecoverCex procedure, which recovers an abstract counterexample to TS from a counterexample to the residual program (line 5). As an instance, in Sec. 4.2 we propose a procedure which builds an abstract counterexample in linear time on the size of a refutation tree for constrained Horn clauses and elementary abstract domain.

Requirement 1. *If* $\widehat{cex}$ = RecoverCex $(TS, Prop, \langle \alpha, \gamma \rangle, a, cex)$, *then* $\widehat{cex}$ *is an abstract counterexample to* $\langle TS, Prop \rangle$ *with respect to* $\langle \alpha, \gamma \rangle$.

Theorem 1. *If verifier* $\mathcal{O}$ *is sound, then* CEGAR($\mathcal{O}$) *is sound.*

Proof. Immediately follows from the soundness of the original CEGAR [10], Lemma 1 and Requirement 1. □

Theorem 2. *If* CEGAR *or verifier* $\mathcal{O}$ *terminate on* $\langle TS, Prop \rangle$, *then* CEGAR($\mathcal{O}$) *terminates*[2] *on* $\langle TS, Prop \rangle$.

[2] under *informed guess assumption*: we assume that abstract counterexamples from $\mathcal{O}$ do not "mislead" refinement

Input: CHC system $\mathcal{P}$, elementary model $a \in \mathcal{A}_{\text{ELEM}}$
Output: Residual CHC system $\mathcal{P}'$

1 $\Phi \leftarrow \mathcal{P}$ with every uninterpreted atom $P(\bar{t})$ replaced with $a(P)(\bar{t}) \vee P(\bar{t})$
2 **return** $CNF(\Phi)$

Algorithm 4: ResidualCHCs algorithm for generation of a residual CHC system.

Proof. If $\mathcal{O}$ terminates, so does the first call Collaborate $(TS, Prop, \langle \alpha, \gamma \rangle, 0)$, as $Init' = T(0) \setminus 0 = Init$ and $T' = \lambda B.(T(B) \setminus 0) = T$. If CEGAR terminates, so does CEGAR$(\mathcal{O})$, as a call to Collaborate is asynchronous. □

4 Collaborative Verification of Horn Clauses over ADTs

This section presents our main approach. The core idea is described in the previous section, so here we adapt CEGAR$(\mathcal{O})$ for CHCs over ADTs and thus achieve two goals:

- to infer inductive invariants represented in first-order theory of ADTs enriched with membership constraints $\overline{x} \in L$;
- to extend the SMT-based CHC solvers with querying generic-purpose FOL solvers like saturation-based provers [31] and finite model finders [9, 35].

Combined domain. We begin by investigating the combination of Elem and arbitrary $\mathcal{A}$ abstract domains as an expanded first-order language. For every $a \in \mathcal{A}$ with $\gamma(a) = L \subseteq \mathbb{T}_{\sigma_1} \times \cdots \times \mathbb{T}_{\sigma_m}$ we define a predicate symbol "$\in L$" with an arity $\sigma_1 \times \cdots \times \sigma_m$. A *membership constraint* is an atomic formula with a membership predicate symbol. Its semantics is defined by expanding $\mathcal{H}$ with $\mathcal{H}(\in L) = L$. A first-order ADT language over predicate symbols of equality and membership constraints is called *first-order language with membership constraints*. This language naturally defines the abstract domain Elem$(\mathcal{A})$ of mappings from $\mathcal{R}$ to its formulas.

4.1 CEGAR$(\mathcal{O})$ for CHCs: Generating Residual System

Recall that Collaborate routine in Algorithm 2 starts by constructing a new system

$$\langle Init', T' \rangle = \langle T(A) \cap \neg A, \lambda B.\, (T(B) \cap \neg A) \rangle$$

which gets passed to the off-the-shelf verifier. The procedure ResidualCHCs$(\mathcal{P}, a)$ (see Algorithm 4) performs the equivalent construction by transforming the original CHC system $\mathcal{P}$ in two steps. It takes an original system $\mathcal{P}$ and an elementary model a as input.

First, it substitutes each atom $P(\bar{t})$ in both heads and bodies of the CHC system with disjunction $a(P)(\bar{t}) \vee P(\bar{t})$ (line 1). Second, it moves the $\mathcal{C}$-formula from the head to the body with the negation and splits clauses by the disjunction. For example, a clause $C_0 \equiv P(x) \wedge \varphi(x, x') \rightarrow P(x')$ would become $(a(P)(x) \vee P(x)) \wedge \varphi(x, x') \rightarrow (a(P)(x') \vee P(x'))$, which after conversion to Conjunctive Normal Form (CNF, line 2) would be split into a clause system $\mathcal{P}' = \{C_1, C_2\}$, where

$$C_1 \equiv a(P)(x) \wedge \varphi(x, x') \wedge \neg a(P)(x') \rightarrow P(x') \tag{3}$$

$$C_2 \equiv P(x) \wedge \varphi(x, x') \wedge \neg a(P)(x') \rightarrow P(x'). \tag{4}$$

Let us compute $Init' = T'(0) = \{x' \mid C \equiv (B \rightarrow P(x')), C \in \mathcal{P}', 0 \models B\}$. Recall that $0(P) = \bot$, so $0 \not\models C_2$, as there is a predicate call $P(x)$ in C_2 body. Thus $Init' = \{x' \mid C \equiv C_1, 0 \models a(P)(x) \wedge \varphi(x, x') \wedge \neg a(P)(x')\} = \{x' \mid \mathcal{H} \models a(P)(x) \wedge \varphi(x, x') \wedge \neg a(P)(x')\}$. On the other hand,

$$(T\,(a) \cap \neg a)(P) = \{x' \mid a \models P(x) \wedge \varphi(x, x')\} \cap \neg\gamma(a) =$$
$$= \{x' \mid \mathcal{H} \models a(P)(x) \wedge \varphi(x, x'), \mathcal{H} \models \neg a(P)(x')\} = \mathit{Init}'.$$

The same steps can be performed with T'. That is, after the CNFization we get CHCs which semantically correspond to the residual system.

Example 2. Given an elementary model $a(tr)(p,t) \equiv \neg(p = t) \vee t = \mathit{Join}(\mathit{Seq})$, $a(ok)(p) \equiv p = \mathit{Seq}$ for Example 1, ResidualCHCs first produces a formula

$$p = \mathit{Seq} \rightarrow (p = \mathit{Seq} \vee \mathit{ok}(p))$$
$$p' = \mathit{Fork}(\mathit{Join}(p)) \wedge (p = \mathit{Seq} \vee \mathit{ok}(p)) \rightarrow (p' = \mathit{Seq} \vee \mathit{ok}(p'))$$
$$p = \mathit{Seq} \wedge t = \mathit{Fork}(p') \rightarrow (\neg(p = t) \vee t = \mathit{Join}(\mathit{Seq}) \vee \mathit{tr}(p,t))$$
$$t = \mathit{Join}(\mathit{Seq}) \rightarrow (\neg(p = t) \vee t = \mathit{Join}(\mathit{Seq}) \vee \mathit{tr}(p,t))$$
$$p' = \mathit{Fork}(\mathit{Join}(p)) \wedge t' = \mathit{Fork}(\mathit{Join}(t)) \wedge (\neg(p = t) \vee t = \mathit{Join}(\mathit{Seq}) \vee \mathit{tr}(p,t)) \rightarrow$$
$$\rightarrow (\neg(p' = t') \vee t' = \mathit{Join}(\mathit{Seq}) \vee \mathit{tr}(p', t'))$$
$$(p = \mathit{Seq} \vee \mathit{ok}(p)) \wedge (\neg(p = p) \vee p = \mathit{Join}(\mathit{Seq}) \vee \mathit{tr}(p,p)) \rightarrow \bot$$

which is simplified to

$$p = \mathit{Fork}(\mathit{Join}(\mathit{Seq})) \rightarrow \mathit{ok}(p)$$
$$p' = \mathit{Fork}(\mathit{Join}(p)) \wedge \mathit{ok}(p) \rightarrow \mathit{ok}(p')$$
$$t = \mathit{Fork}(\mathit{Join}(\mathit{Join}(\mathit{Seq}))) \rightarrow \mathit{tr}(p,t)$$
$$p' = \mathit{Fork}(\mathit{Join}(p)) \wedge t' = \mathit{Fork}(\mathit{Join}(t)) \wedge p' = t' \wedge \mathit{tr}(p,t) \rightarrow \mathit{tr}(p', t')$$
$$(p = \mathit{Seq} \vee \mathit{ok}(p)) \wedge (p = \mathit{Join}(\mathit{Seq}) \vee \mathit{tr}(p,p)) \rightarrow \bot.$$

Note that this residual system is simpler than the original of the Example 1 as the last tr clause contains new equality $p' = t'$. That is, the elementary model a is "integrated" in the system via the residual construction.

4.2 CEGAR($\mathcal{O}$) for CHCs: Recovering Counterexamples

In this section, we show how to recover an abstract counterexample from a concrete counterexample to the residual system $\mathcal{P}' = \text{ResidualCHCs}(\mathcal{P}, a)$, i.e., instantiate RecoverCex from Algorithm 2.

Concrete counterexamples. It is well-known that unsatisfiability of CHC systems can be witnessed by a resolution refutation, i.e., counterexamples are represented by refutation trees.

Definition 4. A *refutation tree* of a CHC system $\mathcal{P}$ is a finite tree with nodes $\langle C, \Phi\rangle$, where

- $C \in \mathcal{P}$ and Φ is a $\Sigma \cup \mathcal{R}$-formula;
- in the root node C is a query of $\mathcal{P}$ and Φ is a satisfiable Σ-formula;
- node $\langle C, \Phi\rangle$ is a leaf only if $\Phi \equiv \mathit{body}\,(C)$ and Φ is a Σ-formula;
- node $\langle C, \Phi\rangle$ has children $\langle C_1, \Phi_1\rangle, \ldots, \langle C_n, \Phi_n\rangle$ only if: 1. $\mathit{body}\,(C) \equiv \varphi \wedge P_1(\overline{x}_1) \wedge \ldots \wedge P_n(\overline{x}_n)$; 2. $C_i \in \mathit{rules}\,(P_i)$; 3. $\Phi \equiv \varphi \wedge \Phi_1(\overline{x}_1) \wedge \ldots \wedge \Phi_n(\overline{x}_n)$.

A *concrete counterexample* for a CHC system $\mathcal{P}$ is a refutation tree of $\mathcal{P}$.

Abstract counterexamples. Let us introduce another transformation $Q\,(\mathcal{P}, a)$ which defines counterexamples to the abstraction. Intuitively, we have an abstract counterexample to a CHC system when we have a refutation tree with some leaves being abstract states instead of constrained facts. Thus, the transformation $Q\,(\mathcal{P}, a)$ for each $P \in \mathcal{R}$ adds to $\mathcal{P}$ new clauses

$$a(P)(\overline{x}) \rightarrow P(\overline{x}).$$

Definition 5 (**Abstract counterexample**)**.** An abstract counterexample for a CHC system $\mathcal{P}$ with respect to an abstract state a is a refutation tree of $Q(\mathcal{P}, a)$.

That is, a refutation to $Q(\mathcal{P}, a)$ may not be a refutation to the original system $\mathcal{P}$ as some refutation leaves might be abstractions $a(P)$. Yet these states may still be reachable in the original system, so their feasibility is further checked in CEGAR.

That is, CEGAR needs abstract counterexamples, so we need **RecoverCex procedure**, which recovers such *abstract* counterexamples to the *original* system from the *concrete* counterexamples to the *residual* system. Let us now instantiate this procedure. More formally, we will now show how to recursively build a refutation tree T' of $\mathcal{P}'' = Q(\mathcal{P}, a)$ from T, a refutation tree of $\mathcal{P}' = \textsc{ResidualCHCs}(\mathcal{P}, a)$.

Base. T is a leaf $\langle C, \Phi\rangle$, where $C \in \mathcal{P}'$. As $\Phi = \mathit{body}(C)$ is a Σ-formula, C is

$$\varphi \wedge a(P_1)(\overline{x}_1) \wedge \ldots \wedge a(P_n)(\overline{x}_n) \wedge \neg a(P)(\overline{x}) \to P(\overline{x}),$$

so we put the root of T' to be $\langle C', \Phi'\rangle$, where

$$C' \equiv \varphi \wedge P_1(\overline{x}_1) \wedge \ldots \wedge P_n(\overline{x}_n) \to P(\overline{x}) \quad \text{and} \quad \Phi' \equiv \varphi \wedge a(P_1)(\overline{x}_1) \wedge \ldots \wedge a(P_n)(\overline{x}_n),$$

with n leaf children $\langle C'_i, a(P_i)(\overline{x}_i)\rangle$, where $C'_i \equiv a(P_i)(\overline{x}_i) \to P_i(\overline{x}_i)$. Definition of a refutation tree is trivially satisfied. Note that $\mathcal{H} \models \Phi \to \Phi'$.

Step. T is a node $\langle C, \Phi\rangle$ with children $\langle C_1, \Phi_1\rangle, \ldots, \langle C_n, \Phi_n\rangle$, $C_i \in \mathit{rules}(P_i)$ from $\mathcal{P}'$ and

$$C \equiv \varphi \wedge a(R_1)(\overline{y}_1) \wedge \ldots \wedge a(R_m)(\overline{y}_m) \wedge \neg a(R)(\overline{y}) \wedge P_1(\overline{x}_1) \wedge \ldots \wedge P_n(\overline{x}_n) \to R(\overline{y})$$
$$\Phi \equiv \varphi \wedge a(R_1)(\overline{y}_1) \wedge \ldots \wedge a(R_m)(\overline{y}_m) \wedge \neg a(R)(\overline{y}) \wedge \Phi_1(\overline{x}_1) \wedge \ldots \wedge \Phi_n(\overline{x}_n).$$

By the recursive step, we already have corresponding nodes $\langle C'_1, \Phi'_1\rangle, \ldots, \langle C'_n, \Phi'_n\rangle$, so we define

$$C' \equiv \varphi \wedge R_1(\overline{y}_1) \wedge \ldots \wedge R_m(\overline{y}_m) \wedge P_1(\overline{x}_1) \wedge \ldots \wedge P_n(\overline{x}_n) \to R(\overline{y})$$
$$\Phi' \equiv \varphi \wedge a(R_1)(\overline{y}_1) \wedge \ldots \wedge a(R_m)(\overline{y}_m) \wedge \Phi'_1(\overline{x}_1) \wedge \ldots \wedge \Phi'_n(\overline{x}_n).$$

We add new children for R_j: $\langle C'_{n+j}, a(R_j)(\overline{y}_j)\rangle$, where $C'_{n+j} \equiv a(R_i)(\overline{y}_i) \to R_j(\overline{y}_j)$.

For each i, $\mathcal{H} \models \Phi_i \to \Phi'_i$ by induction, so for their conjunction we have $\mathcal{H} \models \Phi \to \Phi'$.

We end up with the root of T, some $\langle C, \Phi\rangle$, where C is a query of $\mathcal{P}'$. We already have a corresponding root $\langle C', \Phi'\rangle$ in T' for it by recursion, and we know that $\mathcal{H} \models \Phi \to \Phi'$. As Φ is a satisfiable Σ-formula, so is Φ', which concludes the proof that T' is a refutation tree of $\mathcal{P}''$.

Proposition 2. *The* RecoverCex *procedure is linear-time on the number of nodes of the input refutation tree.*

4.3 Combination with Regular Domain

Recently CHC solvers, namely RInGen [30] and RCHC [23], emerged to support a *regular domain* (which we call Reg). This regular domain is based on *tree automata* and their operations [11]. It is shown that Reg and Elem domains are incomparable, i.e., they intersect yet do not consume one another [30]. The combined domain Elem(Reg) is their natural first-order generalization, and our approach can be applied to, e.g., Racer and RInGen to infer invariants in this domain automatically.

Our running example shows that Elem(Reg) is strictly more expressive than both domains.

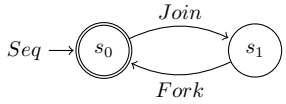
Example 3. Recall the CHC system from Example 1. It is safe yet neither Reg nor Elem invariants, and both Racer and RInGen (CHC-Comp winners; CHC solvers with Elem and Reg abstract domains) launched independently diverge on it.

However, if we extend Racer to call RInGen via our CEGAR($\mathcal{O}$) approach, we can get an invariant for this CHC system. Modified Racer infers Elem lemmas and runs RInGen

oracle with a residual CHC system from Example 2, which halts with the following Reg part of the invariant:

$$ok(p) \Leftrightarrow p \in \mathcal{E}, \quad tr(p,t) \Leftrightarrow t \notin \mathcal{E},$$

where $\mathcal{E}$ is the language of the following tree automaton.



The abstract state from the Example 2 and the regular part of the invariant above are then combined in the following Elem(Reg) invariant:

$$ok(p) \Leftrightarrow p \in \mathcal{E}, \quad tr(p,t) \Leftrightarrow \neg(p = t) \vee \neg(t \in \mathcal{E}).$$

5 Implementation

We implemented the approach in a tool called Collab($\mathcal{V}$) on top of the Racer [22] CHC solver (a successor of Spacer [28]) in Z3[3] and the RInGen($\mathcal{V}$) [30] CHC solver based on transformations and further integration with a general-purpose FOL-solver $\mathcal{V}$[4].

Z3/Racer. Racer implements an instance of the popular *Property-Directed Reachability* (PDR) [28] framework that can be seen as a complex instance of CEGAR. PDR represents an abstract set of states in a form of conjunction of formulas (called *lemmas*) of different *levels* by iteratively increasing the level in a loop. The following is maintained: if a set of lemmas $\{\varphi_i\}$ was built on level n, then $\bigwedge_i \varphi_i$ over-approximates all states reachable in less than n transition steps and under-approximates the safety property. Thus, lemmas built by Racer fulfill the requirement of the abstraction A from the Collaborate subroutine (see Algorithm 3). We modify Racer so that at the end of each iteration a set of lemmas of maximal level is asynchronously passed to a new instance of RInGen($\mathcal{V}$).

RInGen($\mathcal{V}$). We implemented the Collaborate procedure in RInGen($\mathcal{V}$) with the following generalization in the ResidualCHCs($\mathcal{P}, a$) procedure (see Sec. 4.1). We employ a conjunctive form of Racer lemmas to infer invariants from Elem($\mathcal{A}$) of form $\bigwedge_i(\varphi_i(\overline{x}) \vee \overline{x} \in L_i)$. Having $a(P) = \bigwedge_i \varphi_i$, we replace all atoms $P(\overline{t})$ with *conjunctions of disjunctions* $\bigwedge_i(\varphi_i(\overline{t}) \vee L_i(\overline{t}))$ with fresh predicate symbols L_i. This allows us to infer more general invariants than Elem $\uplus\, \mathcal{A}$ (see Defn. 3), which consists only of formulas of the form $\varphi(\overline{x}) \vee \overline{x} \in L$.

After transformations, RInGen($\mathcal{V}$) calls a general-purpose background solver $\mathcal{V}$ with a timeout significantly less than the original run (600 seconds against 30 seconds in our experiments). Its output is then passed back to Racer where it is asynchronously processed. Thus, we omit any expensive CNF transformation from Algorithm 4 as RInGen($\mathcal{V}$) relies on a sound background solver $\mathcal{V}$ with full FOL support.

6 Experiments

Benchmarks. We have empirically evaluated Collab against state-of-the-art CHC solvers on the "Tons of Inductive Problems" (TIP) [8] benchmark set which made up the majority of the ADT track of CHC-COMP 2022 solver competition[5]. The set consists of 454 CHC systems

[3] https://github.com/Columpio/z3/tree/racer-solver-interaction
[4] https://github.com/Columpio/RInGen/releases/tag/chccomp22
[5] https://github.com/chc-comp/ringen-adt-benchmarks

	Racer	RInGen(cvc5)	Collab(cvc5)
Racer	**20**	10	20
RInGen(cvc5)		**25**	23
Collab(cvc5)			**117**

Table 1: SAT results, cvc5 backend

	Racer	RInGen(Vampire)	Collab(Vampire)
Racer	**20**	19	20
RInGen(Vampire)		**135**	129
Collab(Vampire)			**189**

Table 2: SAT results, Vampire backend

	Racer	RInGen(cvc5)	Collab(cvc5)
Racer	**15**	12	14
RInGen(cvc5)		**21**	17
Collab(cvc5)			**19**

Table 3: UNSAT results, cvc5 backend

	Racer	RInGen(Vampire)	Collab(Vampire)
Racer	**15**	15	15
RInGen(Vampire)		**46**	28
Collab(Vampire)			**28**

Table 4: UNSAT results, Vampire backend

with inductive ADT problems, originally generated from Haskell programs.

Experiments were performed on the StarExec [37] platform having a cluster of machines with Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz and Red Hat Enterprise Linux 7, 600 second CPU time limit and with 16 GB memory limit.

Competing tools. The evaluation was performed against the baseline solvers: Z3 with Racer engine [22] and RInGen [30]. As both Collab($\mathcal{V}$) and RInGen($\mathcal{V}$) rely on the FOL-solver backend $\mathcal{V}$, we evaluated them with $\mathcal{V}$ = cvc5 [3] and $\mathcal{V}$ = Vampire [24]. Racer and RInGen(Vampire) took first places on CHC-COMP competition 2021 and 2022 ADT tracks [36, 16]. Thus, for each $\mathcal{V}$, we call Racer and RInGen($\mathcal{V}$) *baseline* for Collab($\mathcal{V}$).

We pose three research questions to be addressed in this comparison.

RQ 1 (Convergence)**.** Recall that our main goal is to extend the invariant inference capabilities. *Does* Collab *solve more benchmarks than baseline tools working independently? In particular, does* Collab *solve benchmarks, which cannot be solved by either of baseline tools?*

RQ 2 (Performance)**.** Collaboration can have the overhead of running multiple instances of the CEGAR oracle in parallel. *What is the performance impact of the collaborated run?*

RQ 3 (Advancement characteristic)**.** *Are the benchmarks solved uniquely by* Collab *not solved by others because of inexpressivity issue? In particular, how many benchmarks solved uniquely by* Collab *are in not in* Elem*?*

Results. The results are summarized in tables 1, 2, 3 and 4. Each table describes either SAT or UNSAT results with RInGen and Collab having either cvc5 or Vampire backend. A number in each cell stands for the amount of benchmarks solved **both** by the solver in column

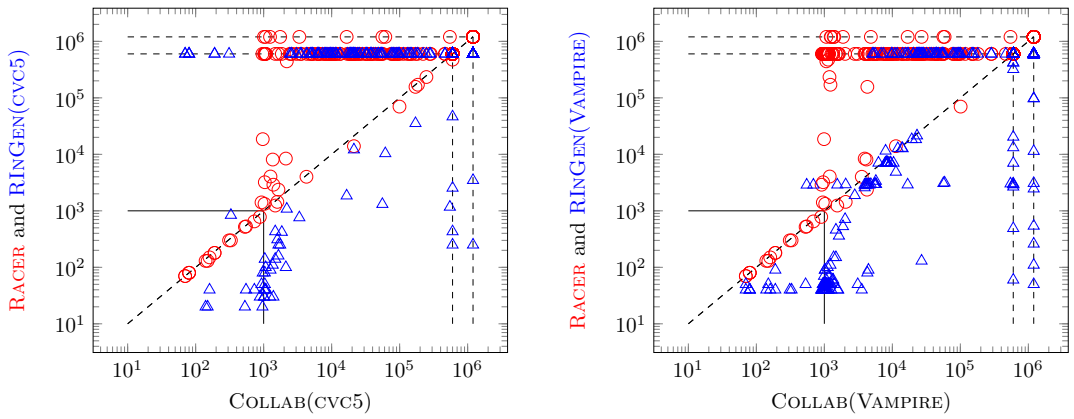


Figure 1: Comparison of runtimes (in milliseconds): Collab (x-axis) and a competitor (y-axis).

and the solver in row. Thus, the numbers on a diagonal (in bold) represent the total number of results obtained by the corresponding tool.

On safe benchmarks our tool outperformed the competitors: 117 SAT answers on Collab(cvc5) against only 20 SAT answers from Racer and 25 from RInGen(cvc5), as well as 189 SATs on Collab(Vampire) against 20 SATs on Racer and 135 from RInGen(Vampire). That is, Collab(cvc5) solved 235% more benchmarks than parallel composition of Racer and RInGen(cvc5), and 39% more benchmarks if we change the backend to Vampire. This is a substantial progress over current state-of-the-art CHC-COMP winning solvers.

On unsafe benchmarks Collab solved less than a maximum of the baseline solvers: 19 (cvc5) and 28 (Vampire) UNSAT on Collab against 21 (cvc5) and 46 (Vampire) UNSAT on RInGen. The main reason is that our approach is focused on the complex task of invariant inference and does not enhance the counterexample search. All counterexamples in Collab thus come directly from one of the baseline solvers. We do not reach some counterexamples obtained from RInGen, as it is run with 30 seconds time limit from Racer in Collab.

Importantly, all 20 SAT and 15 UNSAT answers from Racer were also obtained by Collab, except for one near-time-limit UNSAT benchmark because of starting oracle process overhead.

However, there are benchmarks which were solved by the baseline solvers but not by our tool. Collab(cvc5) did not solve 7 benchmarks, which were successfully verified by RInGen(cvc5). Two of these benchmarks could be solved by increasing the 30 second collaborating solver time limit in Collab. The remaining 5 benchmarks are solved instantly by RInGen(cvc5), yet their results cannot be fetched from inter-process communication in our implementation. The reason is that Racer diverges during SMT constraints solving queries, and thus it never reads the outputs of the collaborating solver. This problem is completely technical and can be fixed by reading the collaborating solver outputs in SMT kernel check points. We have a similar picture for Collab(Vampire): in total, 24 unsolved benchmarks are solvable by RInGen(Vampire). Only 8 of them are unsolved due to a low collaborating call time limit, and the remaining 16 are not solved because Racer diverged.

General performance plots for Collab against the baseline solvers are presented in Figure 1. Each point in a plot represents a pair of the run times (msec × msec) of Collab (x-axis) and a competitor (y-axis): triangles for Racer and circles for RInGen. Outer dashed lines represent crashes, which both Racer and Collab have due to instability of the used

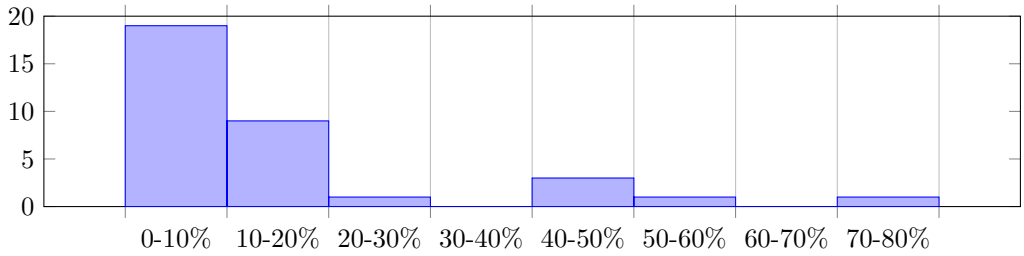


Figure 2: Number of benchmarks (y-axis) solved both by Collab and Racer (34 in total) and CPU time overhead (x-axis) of running Collab over running pure Racer. There are no runs with overhead more than 80%, so the later x-axis is not shown.

Racer branch[6]. Inner dashed lines represent cases when a solver reached the time limit. As Collab solved significantly more instances than the baseline solvers, most of the figures are on top dashed lines of both plots. Half of the rest of triangles are near the diagonal, meaning that collaboration finished after the first collaborative solver call. Another half of such triangles are near one second (which is outlined by solid lines at the bottom left angle) for a similar reason as to why some benchmarks are unsolved: the Racer engine of Collab performs complex SMT-solving and thus does not read a collaborative solver result for some time. Most of the circles not on the dashed lines are near the diagonal on both plots meaning that Collab performed comparably with Racer when a collaborating solver did not help. This overhead is to be discussed next.

Figure 2 shows the **performance overhead** which the collaboration gives. There are correspondingly 34 and 35 benchmarks solved both by Racer and by Collab with either cvc5 or Vampire, which gives us 69 runs of Collab in total. On 35 out of these 69 runs Collab outperformed Racer and on the remaining 34 it did not because no call to the collaborative solver was successful, so Collab behaved just as Racer with process calling overhead. The overhead on these 34 runs is presented in Figure 2. The plot shows how many times slower on these runs Collab is against pure Racer. Overhead of most of the runs appears to be near 10%: the mean overhead across all runs is 15% and the median is 8%. There are only 6 runs with overhead more than 20%. On three of them, Racer works from 14 to 70 seconds and Collab is 40-50% slower because of accumulated amount of simultaneously run collaborating processes. The remaining runs with the overhead over 20% are those where Racer works nearly 2 seconds and Collab works from 2 to 4 seconds too. That is why it gives high percentage and thus it can be neglected.

Answer to RQ 1. Collab solves significantly more SAT benchmarks than baseline tools working independently: 117 against $20 + 25$ and 189 against $20 + 135$ for different backends. In particular, Collab(cvc5) solves 97 benchmarks unsolved by Racer and 94 benchmarks unsolved by RInGen(cvc5). Collab(Vampire) solves 169 benchmarks unsolved by Racer and 60 benchmarks unsolved by RInGen(Vampire).

Collab solves slightly less UNSAT benchmarks than baseline tools as our collaboration approach does not introduce new counterexample building techniques and thus relies on running underlying solvers. That is, orthogonal counterexample finding enhancements can be integrated with our approach, e.g., one proposed in [6].

[6]We performed evaluation on top of it because it does not lose any results on our benchmarks compared to a stable branch, but it sometimes performs almost ten times faster.

A small amount of both SAT and UNSAT answers (2 for CVC5 as a backend and 8 for VAMPIRE) obtained by collaborating RINGEN instances were missed by COLLAB because of somewhat "hard-coded" time limit for the collaborating solver. This gives another room for improvement, however it will likely make the total overhead grow as well if we increase the time limit. Existing techniques to verification time prediction such as [15] could be applied to avoid hard coding the time limit at all.

Some SAT and UNSAT answers (5 for the CVC5 backend and 16 for the VAMPIRE backend) obtained by RINGEN were missed by COLLAB because of its underlying RACER engine divergence during solving SMT queries. This can be technically solved by introducing more fine-grained synchronization with the collaborating solver process not on the RACER end (as in Algorithm 2) but on the SMT solving end, which, on the other hand, will slightly increase the total solving overhead.

To sum up, there are possible ways to improve our result, but it already significantly outperforms state-of-the-art by the number of solved benchmarks.

Answer to RQ 2. As it was summarized in Figure 2, the median overhead of COLLAB is near 8%, which comes from starting and managing oracle processes. High ($> 50\%$) overheads come from two runs finished in nearly 3 seconds.

Answer to RQ 3. It is hard to *precisely* calculate which of the benchmarks solved uniquely by COLLAB are not in ELEM since it is hard to formally prove the undefinability in ELEM. Although it could be estimated as the number of benchmarks where a backend either yields a tree automaton with cycles or a saturation, and all unique SATs by COLLAB are such. So, we suppose that all uniquely solved benchmarks do not belong to ELEM, and we thus blame the lack of expressivity for the divergence of baseline solvers which thus further motivates us to advance the presented approach.

7 Related Work

ADT invariant inference. Most existing techniques infer ADT invariants expressed in first-order logic, i.e., in the assertion language [28, 20, 34, 40, 17, 25]. As inductive invariants of practical programs over ADTs cannot be expressed in FOL, a complementary way of expressing ADT invariants as tree automata [11] has recently emerged in RINGEN [30] and RCHC [23]. In our work, we showed how such complementary techniques can be combined collaboratively and thus how to infer invariants in FOL with membership constraints. Recently proposed RACER [22] infers ADT invariants in an extension of FOL with catamorphism constraints. Our work is similar because tree automata can be viewed as catamorphisms yet RACER needs them to be *predefined by the user*, while with our approach they can be inferred automatically.

Correlation of diverse invariant inference techniques. There are a number of works correlating different invariant inference techniques, namely: IC3 and ICE in [39], IC3 and abstract interpretation in [19], interpolation-based methods and concept learning in [18]. Such papers investigate the theoretical similarities between compared techniques by detecting similarities in order to understand and improve them. In contrast, our approach allows one to make out of diverse techniques a new invariant inference algorithm for their combined domain.

Combined invariants. Combining abstract domains is a fruitful direction in abstract interpretation [32, 21]. For example, the ASTRÉE analyzer uses a combination of cooperative abstract domains to improve the precision of the analysis [14]. Various combinations of abstract domains are studied in [13]. Synthesis of invariants for combined SMT theories is studied in [4]. However, this work studies only the combination of EUF and LIA, while our approach is applied to ADT theory. Although, we believe that our core idea described in Sec. 3 is generic enough to

be adopted in other frameworks. The properties of the first-order ADT theory with membership constraints, which we use as combined abstract domain in Sec. 4, are studied in [12].

Collaboration of verifiers. The idea of integration of multiple verification engines into one is used in portfolio frameworks like [1]. Unlike the portfolio verifiers, we build the more complex CEGAR-specific interconnections between the tools by generating a sequence of residual systems. Our idea can be viewed as extended CEGAR, which "accomplishes" the fixpoint construction by querying the external oracle. From this point of view, our approach is similar to oracle-guided inductive program synthesis [26].

Constrained Horn Clauses. Constrained Horn clauses have been proposed to be a lingua franca for automatic program verification [5]. Since then, a number of CHC solvers have arisen [33, 22, 25, 7, 27, 23, 17, 40, 30, 2, 38]. In this domain, our work can be considered as a way of collaborating CEGAR-based CHC solvers with other ones.

8 Conclusion

In this paper, we introduced the CEGAR-based approach for collaborative inference of inductive invariants in combined domains for programs over algebraic data types. We have shown that the collaboration performs at least as well (in sense of convergence) than the collaborating verifiers working independently, and that it can converge even in cases when all collaborating verifiers diverge because it handles a richer class of invariants from a combined domain. We have applied our approach to combine the fixpoint engine in Z3 with general-purpose logical solvers wrapped into the RInGen framework. We have demonstrated that our approach can solve 235% more instances than the virtually best solver of Z3 and cvc5 and 39% more instances than Z3 + Vampire, with no substantial time overhead (8% on average).

References

[1] Zsófia Ádám, Gyula Sallai, and Ákos Hajdu. Gazer-theta: Llvm-based verifier portfolio with bmc/cegar (competition contribution). In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 433–437, Cham, 2021. Springer International Publishing.

[2] Masoud Asadzade, Martin Blicha, Antti EJ Hyvärinen, and Natasha Sharygina. The OpenSMT Solver in SMT-COMP 2021. 2021.

[3] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

[4] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 378–394, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[5] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, pages 24–51. Springer, 2015.

[6] Martin Blicha, Grigory Fedyukovich, Antti E. J. Hyvärinen, and Natasha Sharygina. Transition power abstractions for deep counterexample detection. In Dana Fisman and Grigore Rosu, editors,

Tools and Algorithms for the Construction and Analysis of Systems, pages 524–542, Cham, 2022. Springer International Publishing.

[7] Adrien Champion, Naoki Kobayashi, and Ryosuke Sato. HoIce: An ICE-based non-linear Horn clause solver. In Sukyoung Ryu, editor, *Programming Languages and Systems*, pages 146–156, Cham, 2018. Springer International Publishing.

[8] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Tip: tons of inductive problems. In *International Conference on Intelligent Computer Mathematics*, pages 333–337. Springer, Cham, 2015.

[9] Koen Claessen and Niklas Sörensson. New techniques that improve mace-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications*, pages 11–27. Citeseer, 2003.

[10] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[11] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 2008.

[12] Hubert Comon and Catherine Delor. Equational formulas with membership constraints. *Information and Computation*, 112(2):167–216, 1994.

[13] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. A survey on product operators in abstract interpretation. *Electronic Proceedings in Theoretical Computer Science*, 129:325–336, Sep 2013.

[14] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, pages 272–300, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[15] Mike Czech, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. Predicting rankings of software verification tools. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics*, SWAN 2017, page 23–26, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Emanuele De Angelis and Hari Govind Vediramana Krishnan. Report on the 2022 edition, 2022.

[17] Grigory Fedyukovich, Samuel J Kaufman, and Rastislav Bodík. Sampling invariants from frequency distributions. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 100–107. IEEE, 2017.

[18] Yotam MY Feldman, Mooly Sagiv, Sharon Shoham, and James R Wilcox. Learning the boundary of inductive invariants. *arXiv preprint arXiv:2008.09909*, 2020.

[19] Yotam MY Feldman, Mooly Sagiv, Sharon Shoham, and James R Wilcox. Property-directed reachability as abstract interpretation in the monotone theory. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–31, 2022.

[20] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer, 2014.

[21] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. *ACM SIGPLAN Notices*, 41(6):376–386, 2006.

[22] K. Hari Govind V, Sharon Shoham, and Arie Gurfinkel. Solving constrained Horn clauses modulo algebraic data types and recursive functions. *Proceedings of the ACM on Programming Languages*, 6(POPL), January 2022.

[23] Timothée Haudebourg. *Automatic verification of higher-order functional programs using regular tree languages*. PhD thesis, 2020. Thèse de doctorat dirigée par Genet, Thomas et Jensen, Thomas Informatique Rennes 1 2020.

[24] Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Invariant generation in Vampire. In

Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 60–64, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[25] Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn Solver. In *FMCAD*, pages 158–164. IEEE, 2018.

[26] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

[27] Bishoksan Kafle, John P. Gallagher, and José F. Morales. Rahft: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 261–268, Cham, 2016. Springer International Publishing.

[28] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[29] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. Automatic abstraction in SMT-based unbounded software model checking. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 846–862, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[30] Yurii Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich. Beyond the elementary representations of program invariants over algebraic data types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 451–465, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[32] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. *ACM SIGPLAN Notices*, 37(1):270–282, 2002.

[33] Kenneth L McMillan and Andrey Rybalchenko. Solving constrained Horn clauses using interpolation. *Tech. Rep. MSR-TR-2013-6*, 2013.

[34] Saswat Padhi and Todd D Millstein. Data-driven loop invariant inference with automatic feature synthesis. *CoRR*, 2017.

[35] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstić. Finite model finding in SMT. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 640–655, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[36] Philipp Rümmer and Grigory Fedyukovich. Competition report: CHC-COMP-21. *Electronic Proceedings in Theoretical Computer Science*, 344:91–108, Sep 2021.

[37] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, pages 367–373, Cham, 2014. Springer International Publishing.

[38] Takeshi Tsukada, Hiroshi Unno, Taro Sekiyama, and Kohei Suenaga. Enhancing loop-invariant synthesis via reinforcement learning. *arXiv preprint arXiv:2107.09766*, 2021.

[39] Yakir Vizel, Arie Gurfinkel, Sharon Shoham, and Sharad Malik. IC3 - flipping the E in ICE. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 521–538, Cham, 2017. Springer International Publishing.

[40] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. *ACM SIGPLAN Notices*, 53(4):707–721, 2018.