Collaborative Inference of Combined Invariants

Yurii Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich

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 $\langle S, \subseteq, 0, 1, \cap, \cup, \neg \rangle$ be a complete Boolean lattice.

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

- $T$ is monotonic, i.e., $s_1 \subseteq s_2$ implies $T(s_1) \subseteq T(s_2)$,
Collaborative Inference of Combined Invariants  
Kostyukov, Mordvinov and Fedyukovich

Input: a program \(TS\) and a property \(Prop\)
Output: safe with an inductive invariant or UNSAFE with a counterexample

1. \(\langle \alpha, \gamma \rangle \leftarrow \text{INITIAL}()\)
2. while true do
   3. \(cex, A \leftarrow \text{MODELCHECK}(TS, Prop, \langle \alpha, \gamma \rangle)\)
   4. if \(cex\) is empty then
      5. return SAFE, \(A\)
   6. if \(\text{ISFEASIBLE}(cex)\) then
      7. return UNSAFE, \(cex\)
   8. \(\langle \alpha, \gamma \rangle \leftarrow \text{REFINE}(\langle \alpha, \gamma \rangle, cex)\)

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
- \(T(0) = \text{Init}\).

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

\[
\text{Init} \subseteq I, \quad T(I) \subseteq I, \quad I \subseteq Prop.
\]

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

\[
\alpha : S \rightarrow A, \quad \gamma : A \rightarrow S, \quad \forall x \in S. \forall y \in A. \alpha(x) \sqsubseteq y \iff x \sqsubseteq \gamma(y).
\]

An abstract transition function \(\hat{T} : A \rightarrow A\) lifts the transition function in the abstract domain with a possible overapproximation, i.e., for all \(a \in A\), \(\alpha(T(\gamma(a))) \sqsubseteq \hat{T}(a)\). An abstract domain together with a Galois connection uniquely define a class of invariants \(\{\gamma(a) \mid a \in A\}\).

We slightly abuse the notation and call this class \(A\) as well. For a program \(TS = (S, \text{Init}, T)\) and a property \(Prop\), we assume that there exists an element \(a \in A\), then \(\gamma(a)\) is an inductive invariant of \(\langle TS, Prop \rangle\) iff

\[
\alpha(\text{Init}) \subseteq a, \quad \hat{T}(a) \subseteq a, \quad \gamma(a) \subseteq Prop.
\]

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 \(\forall s. \alpha(s) = \bot, A\) and \(\forall a. \gamma(a) = 1\). From a concrete program and its abstraction, the algorithm builds a finite sequence of abstract states \(\vec{a} = \langle a_0, \ldots, a_n \rangle\) such that:

\[
a_0 = \alpha(\text{Init}) \quad \text{and} \quad a_{i+1} = a_i \cup \hat{T}(a_i) \quad \forall i \in \{0, \ldots, n - 1\}\]  \hspace{1cm} (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 \(\hat{T}(a_n) \subseteq a_n\), then \(\gamma(a_n)\) is an inductive invariant, so MODELCHECK returns it as \(A\).
and no \emph{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:

\[ A = 0 \quad \text{or} \quad \text{Init} \subseteq A \subseteq \text{Prop}. \]  

(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 \( \langle a, \gamma \rangle \) to exclude counterexample \( \text{cex} \).

### 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 \( \Sigma_S \) is a set of sorts and \( \Sigma_F \) is a set of function symbols (constructors)\(^1\). Each function symbol \( f \in \Sigma_F \) has an associated sort \( \sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma \) (possibly with \( n = 0 \)), where \( \sigma_i, \sigma \in \Sigma_S \). From now on, we consider \( \Sigma \) fixed.

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

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

### 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 \( \sigma_1 \times \cdots \times \sigma_m \) from \( \Sigma \). Predicate symbols from \( \mathcal{R} \) are called \emph{uninterpreted}.

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

\[ \varphi \land P_1(\overline{x}_1), \ldots, P_n(\overline{x}_n) \rightarrow H \]

where \( C \)-formula \( \varphi \) is called a \emph{constraint} of \( C \), each \( P_i \in \mathcal{R}, \overline{x}_1 \) are tuples of pairwise different variables, and \( H \), called a \emph{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 \emph{body} denoted \( \text{body}(C) \). If \( H = \bot \), we say that \( C \) is a \emph{query clause}. A CHC system \( P \) is a finite set of CHCs with (for presentation simplicity) a single \emph{query clause}. Let \emph{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 \( T_{\sigma_1} \times \cdots \times T_{\sigma_m} \) by \( T_P \). Now we define a complete Boolean lattice of \( \langle S, \subseteq, 0, 1, \cap, \cup, \neg \rangle \), which will serve us as a concrete domain of CHC transition semantics.

\[ S \overset{\text{def}}{=} \text{a set of all mappings from every } P \in \mathcal{R} \text{ to a subset of } T_P \]

\[ s_1 \subseteq s_2 \iff \forall P \in \mathcal{R} \quad s_1(P) \subseteq s_2(P) \]  

\[ s_1 \cap s_2 \overset{\text{def}}{=} \{ P \mapsto s_1(P) \cap s_2(P) \mid P \in \mathcal{R} \} \]  

\[ 0 \overset{\text{def}}{=} \{ P \mapsto \emptyset \mid P \in \mathcal{R} \} \]  

\[ s_1 \cup s_2 \overset{\text{def}}{=} \{ P \mapsto s_1(P) \cup s_2(P) \mid P \in \mathcal{R} \} \]  

\[ 1 \overset{\text{def}}{=} \{ P \mapsto T_P \mid P \in \mathcal{R} \} \]  

\[ \neg s \overset{\text{def}}{=} \{ P \mapsto T_P \setminus s(P) \mid P \in \mathcal{R} \} \]

\(^1\)For simplicity, we omit the selectors and testers from the assertion language because they do not increase its expressiveness.
By \( \mathcal{H}\{P_1 \rightarrow X_1, \ldots, P_n \rightarrow X_n\} \) we denote the \( \Sigma \cup \mathcal{R} \)-expansion of \( \Sigma \)-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 \( C \in \mathcal{P} \), \( \mathcal{H}\{P_1 \rightarrow s(P_1), \ldots, P_n \rightarrow s(P_n)\} \models C \). 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 \( \langle \mathcal{S}, \text{Init}, T \rangle \), where \( \text{Init} \defeq T(0) \) and

\[
T(s)(P) \defeq \{ \tilde{t} \mid (B \rightarrow P(\tilde{t})) \text{ is a ground instance of some clause from } \mathcal{P}, s \models B \}.
\]

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

\[
\mathcal{P}' \defeq \text{rules}(\mathcal{P}) \cup \{ \text{body}(C)(\pi) \rightarrow Q(\pi) \mid C \text{ is a query clause of } \mathcal{P} \} \cup \{ Q(\pi) \rightarrow \bot \}.
\]

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

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

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

\[
p = \text{Seq} \rightarrow \text{ok}(p)
\]

\[
p' = \text{Fork(Join(p))} \land \text{ok}(p) \rightarrow \text{ok}(p')
\]

\[
p = \text{Seq} \land t = \text{Fork}(p') \rightarrow \text{tr}(p,t)
\]

\[
t = \text{Join(Seq)} \rightarrow \text{tr}(p,t)
\]

\[
p' = \text{Fork(Join(p))} \land t' = \text{Fork(Join(t))} \land \text{tr}(p,t) \rightarrow \text{tr}(p',t')
\]

\[
\text{ok}(p) \land \text{tr}(p,p) \rightarrow \bot
\]

In this system, \( \mathcal{R} = \{ \text{ok}, \text{tr} \} \), rules are all clauses except the last one, which is a query clause. This system is satisfied by \( s \), where \( s(\text{ok}) = \mathcal{E} \) and \( s(\text{tr}) = \{(p,t) \mid p \neq t \lor t \not\in \mathcal{E}\} \), where \( \mathcal{E} \) is the least fixpoint of the \( \text{ok} \) 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 Hoise [7]. More recent Racer [22] is also based on the enrichment of this domain with recursive functions.

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

\[
\begin{align*}
\text{a}_1 \sqsubseteq \text{a}_2 & \iff \text{for all } P, \quad \mathcal{H} \models \text{a}_1(P) \rightarrow \text{a}_2(P) \\
\begin{array}{ll}
\text{Init}_{\text{Elem}} & \defeq \{ P \rightarrow \bot \} \\
\text{T}_{\text{Elem}} & \defeq \{ P \rightarrow \top \}
\end{array} \\
\text{a}_1 \sqcup \text{a}_2 & \defeq \{ P \rightarrow \text{a}_1(P) \lor \text{a}_2(P) \} \\
\text{a}_1 \sqcap \text{a}_2 & \defeq \{ P \rightarrow \text{a}_1(P) \land \text{a}_2(P) \}.
\end{align*}
\]

### 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 $\text{Prop}$
Output: safe with a combined invariant in $A \cup B$ or unsafe with cex

1. $\langle \alpha, \gamma \rangle \leftarrow \text{INITIAL}()$
2. $A \leftarrow 0$
3. while true do
4. async call $\text{COLLABORATE}(TS, \text{Prop}, \langle \alpha, \gamma \rangle, A)$
5. $\text{cex}, A \leftarrow \text{MODELCHECK}(TS, \text{Prop}, \langle \alpha, \gamma \rangle)$
6. if $\text{cex}$ is empty then
7. return safe, $A$
8. if $\text{ISFEASIBLE}($cex$)$ then
9. return unsafe, cex
10. $\langle \alpha, \gamma \rangle \leftarrow \text{REFINE}($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 $A$ and $B$ respectively. CEGAR($\mathcal{O}$) allows us to construct invariants in the union of these classes.

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

$$A \cup B \triangleq \{ A \cup B \mid A \in A, B \in B \}.$$ 

A combined inductive invariant in classes $A$ and $B$ is an inductive invariant in $A \cup B$.

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

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

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 $\text{Init}'$ are $T(A) \setminus A$, one-step image of
non-inductive states. $T'(\text{Init}') = T(T(A) \setminus A)$, 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
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 $O$ halts with inductive invariant $B$, then we yield the combined invariant $A \cup B$. If $O$ returns the concrete counterexample $cex$, $CEGAR$ recovers a new abstract counterexample $\overline{cex}$ from it and then proceeds by refining the domain with $\overline{cex}$.

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 $O$, the solution to which gives only one part of an answer.

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

**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 $O$, $B$ is an inductive invariant of $(\langle S, Init', T' \rangle, Prop)$, 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 $T(A \cup B) \subseteq A \cup B$.

Property. $A \subseteq Prop$ from (2) and $B \subseteq Prop$ by soundness of $O$, 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. $CEGAR(O)$ is parametrized by the $RECOVERCEX$ procedure, which recovers an abstract counterexample to $TS$ from a concrete 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 $\overline{cex} = RECOVERCEX(TS, Prop, (\alpha, \gamma), a, cex)$, then $\overline{cex}$ is an abstract counterexample to $(TS, Prop)$ with respect to $\langle \alpha, \gamma \rangle$.

**Theorem 1.** If verifier $O$ is sound, then $CEGAR(O)$ is sound.

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

**Theorem 2.** If $CEGAR$ or verifier $O$ terminate on $(TS, Prop)$, then $CEGAR(O)$ terminates on $(TS, Prop)$.

\^2under informed guess assumption: we assume that abstract counterexamples from $O$ do not “mislead” refinement.
Recall that 4.1 CEGAR( here we adapt This section presents our main approach. The core idea is described in the previous section, so

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( for CHCs over ADTs and thus achieve two goals:

- to infer inductive invariants represented in first-order theory of ADTs enriched with membership constraints \( \pi \in L \);
- 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 \( A \) abstract domains as an expanded first-order language. For every \( a \in A \) with \( \gamma(a) = L \subseteq T_{\sigma_1} \times \cdots \times T_{\sigma_m} \) 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 \( H \) with \( H(\in L) = L \). 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 \( \text{ELEM}(A) \) of mappings from \( \mathcal{R} \) to its formulas.

4.1 CEGAR( for CHCs: Generating Residual System

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

\[
(\text{Init}', T') = (T(A) \cap \neg A, \lambda B, (T(B) \cap \neg A))
\]

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

First, it substitutes each atom \( P(\overline{t}) \) in both heads and bodies of the CHC system with disjunction \( a(P)(\overline{t}) \lor P(\overline{t}) \) (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 \( C_0 = P(x) \land \varphi(x, x') \rightarrow P(x') \) would become \( (a(P)(x) \lor P(x)) \land \varphi(x, x') \rightarrow (a(P)(x') \lor P(x')) \), which after conversion to Conjunctive Normal Form (CNF, line 2) would be split into a clause system \( P' = \{C_1, C_2\} \), where

\[
C_1 = a(P)(x) \land \varphi(x, x') \land \neg a(P)(x') \rightarrow P(x') \\
C_2 = P(x) \land \varphi(x, x') \land \neg a(P)(x') \rightarrow P(x').
\]

Let us compute \( \text{Init}' = T'(0) = \{x' \mid C \equiv (B \rightarrow P(x')) \land C \subseteq P', 0 \models B\} \). Recall that \( 0(P) = \bot \), so \( 0 \not\models C_2 \), as there is a predicate call \( P(x) \) in \( C_2 \) body. Thus \( \text{Init}' = \{x' \mid C \equiv C_1, 0 \models a(P)(x) \land \varphi(x, x') \land \neg a(P)(x')\} = \{x' \mid H = a(P)(x) \land \varphi(x, x') \land \neg a(P)(x')\} \).
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 $a(tr)(p,t) \equiv \neg(p = t) \lor t = Join(Seq)$, $a(ok)(p) \equiv p = Seq$ for Example 1, RESIDUALCHCs first produces a formula

$$p = Seq \rightarrow (p = Seq \lor ok(p))$$

$$p' = Fork(Join(p)) \land (p = Seq \lor ok(p)) \rightarrow (p' = Seq \lor ok(p'))$$

$$t = Join(Seq) \rightarrow (\neg(p = t) \lor t = Join(Seq) \lor tr(p,t))$$

$$p' = Fork(Join(p)) \land t' = ForK(Join(t)) \land (\neg(p = t) \lor t = Join(Seq) \lor tr(p,t)) \rightarrow (\neg(p' = t') \lor t' = Join(Seq) \lor tr(p',t'))$$

$$(p = Seq \lor ok(p)) \land (\neg(p = p) \lor p = Join(Seq) \lor tr(p,p)) \rightarrow \bot$$

which is simplified to

$$p = Fork(Join(Seq)) \rightarrow ok(p)$$

$$p' = Fork(Join(p)) \land ok(p) \rightarrow ok(p')$$

$$t = Fork(Join(Seq)) \rightarrow tr(p,t)$$

$$p' = Fork(Join(p)) \land t' = Fork(Join(t)) \land p' = t' \land tr(p,t) \rightarrow tr(p',t')$$

$$(p = Seq \lor ok(p)) \land (p = Join(Seq) \lor tr(p,p)) \rightarrow \bot$$

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

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 \( \mathcal{P}' = \text{RESIDUALCHCs}(\mathcal{P}, a) \).

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

\[ \varphi \land a(P_1)(\overline{x_1}) \land \ldots \land a(P_n)(\overline{x_n}) \land \neg a(\overline{P})(\overline{x}) \rightarrow P(\overline{x}), \]

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

\[ C' \equiv \varphi \land P_1(\overline{x_1}) \land \ldots \land P_n(\overline{x_n}) \rightarrow P(\overline{x}) \quad \text{and} \quad \Phi' \equiv \varphi \land a(P_1)(\overline{x_1}) \land \ldots \land a(P_n)(\overline{x_n}), \]

with \( n \) leaf children \( \langle C'_1, a(P_1)(\overline{x_1}) \rangle, \ldots, \langle C'_n, a(P_n)(\overline{x_n}) \rangle \), where \( C'_i \equiv a(P_i)(\overline{x_i}) \rightarrow P_i(\overline{x_i}) \). Definition of a refutation tree is trivially satisfied. Note that \( \mathcal{H} \models C' \rightarrow \Phi' \).

**Step.** \( T \) is a node \( \langle C, \Phi \rangle \) with children \( \langle C_1, \Phi_1 \rangle, \ldots, \langle C_n, \Phi_n \rangle \). \( C \equiv \varphi \land a(R_1)(\overline{y_1}) \land \ldots \land a(R_m)(\overline{y_m}) \land \neg a(R)(\overline{y}) \land P_1(\overline{x_1}) \land \ldots \land P_n(\overline{x_n}) \rightarrow R(\overline{y}) \)

\[ \Phi \equiv \varphi \land a(R_1)(\overline{y_1}) \land \ldots \land a(R_m)(\overline{y_m}) \land \neg a(R)(\overline{y}) \land P_1(\overline{x_1}) \land \ldots \land \Phi_n(\overline{x_n}). \]

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

\[ C'' \equiv \varphi \land R_1(\overline{y_1}) \land \ldots \land R_m(\overline{y_m}) \land P_1(\overline{x_1}) \land \ldots \land P_n(\overline{x_n}) \rightarrow R(\overline{y}) \]

\[ \Phi'' \equiv \varphi \land a(R_1)(\overline{y_1}) \land \ldots \land a(R_m)(\overline{y_m}) \land \Phi'_1(\overline{x_1}) \land \ldots \land \Phi'_n(\overline{x_n}). \]

We add new children for \( R_j \): \( \langle C''_{n+j}, a(R_j)(\overline{y_j}) \rangle \), where \( C''_{n+j} \equiv a(R_j)(\overline{y_j}) \rightarrow R_j(\overline{y_j}) \).

For each \( i \), \( \mathcal{H} \models \Phi_i \rightarrow \Phi_i' \) by induction, so for their conjunction we have \( \mathcal{H} \models \Phi \rightarrow \Phi' \).

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 \( \mathcal{H} \models \Phi \rightarrow \Phi' \). As \( \Phi \) is a satisfiable \( \Sigma \)-formula, so is \( \Phi' \), 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(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:

\[ ok(p) \Leftrightarrow p \in E, \quad tr(p, t) \Leftrightarrow t \notin E, \]

where \( E \) 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:

\[ ok(p) \Leftrightarrow p \in E, \quad tr(p, t) \Leftrightarrow \neg (p = t) \lor \neg (t \in E). \]

5 Implementation

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

**RInGen(\( V \)).** We implemented the COLLABORATE procedure in RInGen(\( V \)) with the following generalization in the ResidualCHCs\( (P, a) \) procedure (see Sec. 4.1). We employ a conjunctive form of Racer lemmas to infer invariants from Elem(\( A \)) of form \( \bigwedge_i (\varphi_i(\tau) \lor \tau \in L_i) \).

Having \( a(P) = \bigwedge_i \varphi_i \), we replace all atoms \( P(\tilde{t}) \) with conjunctions of disjunctions \( \bigwedge_i (\varphi_i(\tilde{t}) \lor L_i(\tilde{t})) \) with fresh predicate symbols \( L_i \). This allows us to infer more general invariants than Elem \( \cup A \) (see Defn. 3), which consists only of formulas of the form \( \varphi(\tau) \lor \tau \in L \).

After transformations, RInGen(\( V \)) calls a general-purpose background solver \( V \) 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(\( V \)) relies on a sound background solver \( V \) 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
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 \(\mathcal{V} = \text{cvc5} [3]\) and \(\mathcal{V} = \text{Vampire} [24]\). 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 \(\text{Elem}\)?**

**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
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).

Collaborative inference 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
Collaborative Inference of Combined Invariants

Kostyukov, Mordvinov and Fedyukovich

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

Collaborative Inference of Combined Invariants
Kostyukov, Mordvinov and Fedyukovich


Collaborative Inference of Combined Invariants

Kostyukov, Mordvinov and Fedyukovich


