Inductive Approach to Spacer

TAKESHI TSUKADA, Chiba University, Japan
HIROSHI UNNO, Tohoku University, Japan

The constrained Horn clause satisfiability problem is at the core of many automated verification methods, and Spacer is one of the most efficient solvers of this problem. The standard description of Spacer is based on an abstract transition system, dividing the whole procedure into small rules. This division makes individual rules easier to understand but, conversely, makes it difficult to discuss the procedure as a whole. As evidence of the difficulty in understanding the whole procedure, we point out that the claimed refutational completeness actually fails for several reasons, some of which were not present in the original version and subsequently added. It is also difficult to grasp the differences between Spacer and another procedure, such as GPDR.

This paper aims to provide a better understanding of Spacer by developing a Spacer-like procedure defined by structural induction. We first formulate the problem to be solved inductively, then give its naïve solver and transform it to obtain a Spacer-like procedure. Interestingly, our inductive approach almost unifies Spacer and GPDR, which differ in only one respect in our understanding. To demonstrate the usefulness of our inductive approach in understanding Spacer, we examine Spacer variants in the literature in terms of inductive procedures and discuss why they are not refutationally complete and how to fix them. We also implemented the proposed procedure and evaluated it experimentally.

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

Additional Key Words and Phrases: constrained Horn clause, model-based projection, tree interpolation, refutational completeness

ACM Reference Format:
Takeshi Tsukada and Hiroshi Unno. 2024. Inductive Approach to Spacer. *Proc. ACM Program. Lang.* 8, PLDI, Article 227 (June 2024), 24 pages. https://doi.org/10.1145/3656457

1 INTRODUCTION

The satisfiability problem for *constrained Horn clauses* (or *CHCs*) is the problem of asking whether a given finite set of logical formulas with predicate variables has a solution, *i.e.* an assignment to predicate variables that makes all formulas in the given set valid. Many verification problems are reducible to this problem. The most important problem is the safety verification of a while language and a language with first-order functions, which is actually equivalent to the satisfiability problem for CHCs. Other more complicated problems may not be completely reducible to the CHC satisfiability problem, but many sound though incomplete translations have been proposed and implemented: Examples include refinement type inference [Rondon et al. 2008; Unno and Kobayashi 2009] and validity checking of fixpoint logic formulas [Kobayashi et al. 2019].

Because of its practical significance, the study of efficient solvers is quite vast, particularly if a software model checker is regarded as a CHC solver through the above-mentioned equivalence.

As for the subclass known as *linear CHC*, an approach called *property-directed reachability* [Bradley 2011; Een et al. 2011] (or *PDR*) has been recognized as a quite efficient procedure. PDR was originally

Authors' addresses: Takeshi Tsukada, Chiba University, Chiba, Japan, t.tsukada@acm.org; Hiroshi Unno, Tohoku University, Sendai, Japan, hiroshi.unno@acm.org.

ACM 2475-1421/2024/6-ART227
https://doi.org/10.1145/3656457

proposed as a solver of finite model-checking that corresponds to linear CHCs over finite data domain, and Hoder and Bjørner [2012] applied the idea to non-linear CHCs over infinite data domain. Their procedure is often referred to as *GPDR*.

Furthering these ideas, Komuravelli et al. [2014, 2016] developed Spacer, which is currently one of the most efficient CHC solvers. A motivation for the development is the lack of refutational completeness of GPDR. Spacer is based on several new ideas, but the key to refutational completeness is a technique called *model-based projection*. It is used to divide the set of local candidates of counterexamples into a finite number of classes, and the finiteness of the classes allows an exhaustive search for candidates of global counterexamples in a finite number of steps.

Unfortunately, the behavior of Spacer is quite difficult to understand properly. This fact is indicated by the confusion about its refutational completeness. First, Spacer has been proved to be refutationally complete independent of the choice of the backend model-based projection procedure in Komuravelli et al. [2016], but Tsukada and Unno [2022] have shown that Spacer is not refutationally complete for a badly chosen model-based projection procedure.[1] Second, Komuravelli et al. [2015] discussed a variant of Spacer and claimed its refutational completeness without examining the differences, but the procedure of Komuravelli et al. [2015] has other sources of incompleteness in addition to that of the original spacer [Komuravelli et al. 2014, 2016] as we shall see later. These problems and related subtleties of Spacer have been overlooked even though the Spacer papers [Komuravelli et al. 2014, 2016] had many followers.

This paper aims to improve our understanding of Spacer. The ultimate goal is to improve the performance of Spacer, but this ambitious goal is left for future work. We demonstrate the usefulness of our approach by making the aforementioned arguments on refutational completeness understandable to readers who are not necessarily familiar with Spacer.

Our approach. Whereas Spacer is usually described as an abstract transition system, this paper describes Spacer as an inductive procedure. In the traditional description, Spacer is expressed as a collection of transition rules, each of which has an intuitive exposition. However, the understandability of each rule does not necessarily imply the understandability of the entire system. We provide an inductive description, from which one can grasp the entire structure.

Our development proceeds as follows. We first formulate the problem that our procedures solve by induction. The CHC solving does not suit induction (as is perhaps well-known; see Section 4.1), so we introduce an alternative, named the *generalized refinement problem*, suitable for inductive produces. It has a naïve inductive solver (Algorithm 3), and other more efficient procedures (Algorithm 5 and Algorithm 6) can be obtained by rewriting the naïve one. In the rewriting, we replace the quantifier elimination in the naïve solver with model-based projection and then make the procedure as lazy as possible (i.e. deferring computations that are not immediately necessary to later). The deferring process needs much care since we exchange effectful instructions.

The resulting procedures (Algorithm 5 and Algorithm 6), in particular Algorithm 5, is close to Spacer [Komuravelli et al. 2015, 2014, 2016]. The correspondence is discussed in Section 5 at the intuitive level. A more formal argument may be possible by deriving a transition system for Algorithm 5, but we omit the details by the space limitation.

Our procedure (Algorithm 5) still differs from existing implementations of Spacer in many respects. First, our procedure only captures the "skeleton" of Spacer and lacks various optimizations of practical significance. This point will be discussed in Section 5.3. Second, our procedure is

[1]They only showed that there exists a model-based projection procedure with which Spacer is not refutationally complete. An implementation of Spacer employs a particular model-based projection procedure, and the current implementation of Spacer may be refutationally complete due to a still unrevealed property of the model-based projection procedure employed by the current implementation.

refutationally complete, but Spacer is not. The refutational completeness proof of Spacer requires quite subtle arguments: in our setting, the keys are (1) the loop invariance of the arguments of the model-based projection and (2) the finiteness of the variety of possible return values.[2] We discuss the refutational (in)completeness of the original Spacer [Komuravelli et al. 2014, 2016] and a variant [Komuravelli et al. 2015] from our inductive perspective.

Contributions. The contributions of this paper can be summarized as follows.

- This paper provides a novel description of Spacer-like procedures based on structural induction. A refutationally complete variant of Spacer can be obtained as a modification of a naïve procedure based on quantifier elimination. Interestingly, our procedure unifies Spacer and GPDR [Hoder et al. 2011] (Remark 16).
- We discuss the completeness of Spacer implementations in the literature from the viewpoint of our inductive description. Our inductive approach clarifies the subtleties of completeness of Spacer, which even experts have overlooked for a decade.
- We discuss optimizations to fill the gaps between our theoretical development and Spacer.
- We give an implementation of proposed procedures and empirically evaluate solvers using CHC-comp benchmarks. In particular, we discuss the practical significance of tricks to retain refutational completeness.

2 PRELIMINARIES

This section defines the CHC satisfiability problem and introduces the notion of model-based projection, a key technique for refutationally complete CHC solving.

2.1 Constraint Language and Constrained Horn Clauses

We assume a first-order signature σ and a structure $\mathcal{S}$ of the signature σ, fixed in the sequel. Let $\mathcal{L}$ be a fragment of first-order logic (over the signature σ), called the *constraint language*. We shall study logical formulas with predicate variables, but formulas in $\mathcal{L}$ are assumed to have no predicate variables. As usual, we assume $\mathcal{L}$ consists of quantifier-free formulas.

The language $\mathcal{L}$ admits *quantifier elimination* if for every $\varphi(\vec{x}, \vec{y}) \in \mathcal{L}$ with free variables $\vec{x}$ and $\vec{y}$, one can effectively construct a formula $\psi(\vec{y}) \in \mathcal{L}$ such that $\mathcal{S} \models \psi \Leftrightarrow \exists \vec{x}.\varphi$. We assume that the constraint language $\mathcal{L}$ is closed under boolean operations and admits quantifier elimination.

For formulas $\varphi(\vec{x}, \vec{y})$ and $\psi(\vec{x}, \vec{z})$ such that $\models \varphi \Rightarrow \psi$, an *interpolation* is a formula $\vartheta(\vec{x})$ such that (1) the free variables $\{\vec{x}\}$ of ϑ are free variables of both φ and ψ and (2) $\models \varphi \Rightarrow \vartheta$ and $\models \vartheta \Rightarrow \psi$. We assume a procedure $\textsc{Itp}(\varphi, \psi)$ that returns an interpolation of φ and ψ.

A *constrained Horn clause* (or *CHC*) is a formula of one of the following forms

$$\forall \vec{x}. \quad P_1(\vec{t}_1) \land \cdots \land P_n(\vec{t}_n) \land \varphi \Longrightarrow Q(\vec{s}) \qquad \forall \vec{x}. \quad P_1(\vec{t}_1) \land \cdots \land P_n(\vec{t}_n) \land \varphi \Longrightarrow \bot,$$

where $\vec{x}$ is the sequence of object variables appearing in the formula, $\vec{t}_1, \dots, \vec{t}_n$ and $\vec{s}$ are sequences of terms, φ is a formula in the constraint language, and $P_1, \dots, P_n$ and Q are predicate variables. The position of Q is called the *head position*. We often omit the quantifiers and just write as $P_1(\vec{t}_1) \land \cdots \land P_n(\vec{t}_n) \land \varphi \Longrightarrow Q(\vec{s})$ and $P_1(\vec{t}_1) \land \cdots \land P_n(\vec{t}_n) \land \varphi \Longrightarrow \bot$.

A *CHC system* is a finite set of CHCs, regarded as their conjunction. A *solution* of a CHC system is an interpretation of predicate variables that makes all formulas in the system true. For a solution ξ, we write $\xi(P)$ for the interpretation of P under ξ. The solutions are naturally ordered: $\xi \leq \zeta$ if and only if $\forall \vec{x}.\xi(P)(\vec{x}) \Longrightarrow \zeta(P)(\vec{x})$ for every predicate variable P. The problem of deciding if a given CHC system has a solution is called the *CHC satisfiability problem*.

[2]These points have already been presented in the (wrong) proof of the termination of Spacer [Komuravelli et al. 2016]. The flaw was that an argument of model-based projection was, in fact, not a loop invariant (in terms of our framework).

Under a mild condition on $\mathcal{L}$ and $\mathcal{S}$, a given CHC system can be effectively transformed into

$$\{\ \iota(\vec{x},\vec{y}) \Longrightarrow P(\vec{x}), \qquad P(\vec{x}) \land P(\vec{y}) \land \tau(\vec{x},\vec{y},\vec{z},\vec{u}) \Longrightarrow P(\vec{z}), \qquad P(\vec{x}) \land \beta(\vec{x},\vec{y}) \Longrightarrow \bot\ \},$$

where $\iota, \tau, \beta \in \mathcal{L}$, without changing the satisfiability of CHC systems. We shall often refer ι, τ, β as the *inital states*, *transition relation* and *bad states*. For the above CHC system, $\alpha(\vec{x}) := \neg\exists\vec{y}.\beta(\vec{x},\vec{y})$ is called the *assertion* in this paper.

2.2 Model Based Projection

Model-based projection [Komuravelli et al. 2014, 2016] is a way to handle quantified formulas. It is closely related to *quantifier elimination*, which asks to find a formula $\psi(\vec{y})$ equivalent to a given existentially quantified formula $\exists\vec{x}.\varphi(\vec{x},\vec{y})$ (where $\varphi(\vec{x},\vec{y}) \in \mathcal{L}$). Model-based projection is, in a sense, a partial calculation of quantifier elimination.

Definition 1 (Model-based projection [Komuravelli et al. 2014, 2016])**.** Let $\varphi(\vec{x},\vec{y})$ be a quantifier-free formula with free variables $\vec{x},\vec{y}$. A function $\textsc{Mbp}(\exists\vec{x}.\varphi(\vec{x},\vec{y}),-)$ from models $\mathcal{M} \models \varphi$ of φ to a quantifier-free formula $\psi_{\mathcal{M}}(\vec{y}) := \textsc{Mbp}(\exists\vec{x}.\varphi(\vec{x},\vec{y}),\mathcal{M})$ over $\{\vec{y}\}$ is a *model-based projection (of φ w.r.t. $\vec{x}$)* if it satisfies, for every $\mathcal{M} \models \varphi$,

$$\models \psi_{\mathcal{M}}(\vec{y}) \Rightarrow \exists\vec{x}.\varphi(\vec{x},\vec{y}) \qquad \text{and} \qquad \mathcal{M} \models \psi_{\mathcal{M}}(\vec{y}),$$

and furthermore it satisfies the *image finiteness*, *i.e.* $\{\textsc{Mbp}(\exists\vec{x}.\varphi,\mathcal{M}) \mid \mathcal{M} \models \varphi\}$ is a finite set. □

A model-based projection can be performed linear in time and space for certain theories [Komuravelli et al. 2014, 2016] such as linear real arithmetic (LRA) and linear integer arithmetic (LIA). Hereafter we assume that a model-based projection $\textsc{Mbp}(\exists\vec{x}.\varphi,-)$ exists for every $\varphi \in \mathcal{L}$ and $\vec{x}$ and that the mapping $(\varphi,\vec{x},\mathcal{M}) \mapsto \textsc{Mbp}(\exists\vec{x}.\varphi,\mathcal{M})$ is computable (where $\mathcal{M} \models \varphi$).

Example 2. Consider the theory of real arithmetic and let $\varphi(b,x) := (x^2 + bx + 1 = 0)$. Then $(\exists x.\varphi(b,x)) \Leftrightarrow (b^2 - 4 \geq 0) \Leftrightarrow (b \leq -2 \lor 2 \leq b) \Leftrightarrow (b \leq -2 \lor (2 \leq b \leq 7) \lor 3 \leq b)$. So the map

$$\mathcal{M} \mapsto \begin{cases} \psi'_1 & \text{if } \mathcal{M}(b) < 0 \\ \psi'_2 & \text{if } 0 \leq \mathcal{M}(b) < 4 \\ \psi'_3 & \text{if } 4 \leq \mathcal{M}(b) \end{cases} \qquad \text{where} \qquad \begin{array}{l} \psi'_1(b) := (b \leq -2) \\ \psi'_2(b) := (2 \leq b \leq 7) \\ \psi'_3(b) := (3 \leq b). \end{array}$$

is a model-based projection $\textsc{Mbp}(\exists x.\varphi(b,x),-)$ for φ (note that $-2 < \mathcal{M}(b) < 2$ is impossible since $\mathcal{M} \models \varphi$). Of course, this is not a unique choice: one can provide a model-based projection based on $(\exists x.\varphi(b,x)) \Leftrightarrow (b \leq -2 \lor 2 \leq b)$ (yielding $b \leq -2$ or $2 \leq b$ depending on the input model) or on $(\exists x.\varphi(b,x)) \Leftrightarrow (b^2 - 4 \geq 0)$ (yielding $b^2 - 4 \geq 0$ independent of the input model). □

Example 3. Assume that $\mathcal{L}$ admits quantifier elimination. Then, the following procedure provides a model-based projection. Given a formula $\varphi(\vec{x},\vec{y})$ and variables $\vec{x}$, the procedure first performs quantifier elimination for $\exists\vec{x}.\varphi(\vec{x},\vec{y})$, yielding $\psi(\vec{y}) = \bigvee_{i=1}^{n} \psi_i(\vec{y})$. Then, given a model $\mathcal{M} \models \varphi$, the procedure chooses i such that $\mathcal{M} \models \psi_i$ and returns ψ_i.[3] However, this procedure is inefficient, and the procedures by Komuravelli et al. [2014, 2016] do not invoke quantifier elimination. □

There are many ways of understanding what a model-based projection does: (a) It computes an *under-approximation* of $\exists\vec{x}.\varphi$, guided by a model $\mathcal{M} \models \varphi$; (b) It is a *lazy quantifier-elimination*; (c) It compute a *generalization of a point* $\mathcal{M}(\vec{y})$ in the denotation of $\exists\vec{x}.\varphi$. The viewpoints (a) and (b) can be found in the Spacer paper [Komuravelli et al. 2014, 2016], and (c) will be discussed in Remark 16 comparing Spacer with GPDR [Hoder and Bjørner 2012].

[3]There is also a simpler model-based projection procedure that returns ψ independent of the input model $\mathcal{M}$. However, the intuition of the model-based projection is closer to choosing a disjunct from the formula obtained by quantifier elimination.

Algorithm 1 Quantifier elimination by model-based projection

```
1: function Qe($\varphi(\vec{x},\vec{y}), \{\vec{x}\}$)
2:     $\psi(\vec{y}) := \bot$
3:     while $\exists\mathcal{M}.\ \mathcal{M} \models \varphi(\vec{x},\vec{y}) \wedge \neg\psi(\vec{y})$ do
4:         $\vartheta(\vec{y}) := \textsc{Mbp}(\exists\vec{x}.\varphi, \mathcal{M})$
5:         $\psi := \psi \vee \vartheta$
6:     return $\psi$
```

Here, we explain the perspective (b). This is essentially explained in Example 3: a model-based projection procedure produces a disjunct of the formula obtained by quantifier elimination. Conversely, a model-based projection procedure provides a quantifier elimination procedure (*cf.* Algorithm 1). The formula $\psi(\vec{y})$ records the current under-approximation of $\exists\vec{x}.\varphi$. Line 3 checks if the current approximation is exhaustive, and if not, a new under-appoximation ϑ is computed and added (lines 4 and 5). The procedure terminates because $\{\vartheta \mid \mathcal{M} \models \varphi \text{ and } \vartheta = \textsc{Mbp}(\exists\vec{x}.\varphi, \mathcal{M})\}$ is a finite set and the same ϑ does not appear twice in the computation.

It is worth noting that the use of model-based projection in Algorithm 1 is far from typical. Model-based projection $\textsc{Mbp}(\exists\vec{x}.\varphi(\vec{x},\vec{y}), \mathcal{M})$ generates a formula $\vartheta(\vec{y})$, which is a part of the formula ψ obtained by quantifier elimination, *i.e.* $\psi(\vec{y}) = \vartheta(\vec{y}) \vee \cdots$ with an unknown part $\cdots$. The unknown part $\cdots$ can be obtained by further calling Mbp with different models as in Algorithm 1, but typically, one suspends the calculation of $\cdots$ and infers of consequences of the fact that θ is an under-approximation of $\exists\vec{x}.\varphi$. The calculation of the $\cdots$ part will be resumed if necessary or discarded (and we would be in trouble if a discarded part was actually necessary).

3 A BASIC STRATEGY AND PROCEDURES FOR CHC SOLVING

This section briefly reviews a common strategy to solve CHCs based on finite approximations of CHCs and tree interpolation. This is a fairly common strategy used in GPDR [Hoder and Bjørner 2012], Spacer [Komuravelli et al. 2015, 2014, 2016] and a procedure proposed by Unno and Kobayashi [2009] among others. Our procedures also follow this strategy.

For simplicity, let us focus on the following non-linear CHC system

$$S \;=\; \{\; \iota(x) \Rightarrow P(x), \quad P(x) \wedge P(y) \wedge \tau(x,y,z) \Rightarrow P(z), \quad P(z) \Rightarrow \alpha(z) \;\} \tag{1}$$

which has a single predicate variable P and a unique non-linear clause $P(x) \wedge P(y) \wedge \tau(x,y,z) \Rightarrow P(z)$ with two occurrences of the predicate variable on the left-hand-side of $\Rightarrow$.

3.1 Finite Approximation and Tree Interpolation

This subsection presents an approach to CHC solving based on finite approximations, which is analogous to bounded model-checking in program verification. Let us first explain the idea in terms of program verification, which we believe is more intuitive.

We consider the safety verification problem for transition systems. Formally, a transition system is a quadruple $\mathcal{X} = (X, I, R, A)$ where X is the set of states, $I \subseteq X$ is the inital states, $R \subseteq X \times X$ is the transition relation and $A \subseteq X$ is the assertion that we expect to hold for all reachable states. A state $x \in X$ is *reachable* if there exists a sequence $x_0, x_1, \dots, x_k$ of states $x_i \in D$ such that $x_0 \in I$, $x_k = x$ and $(x_i, x_{i+1}) \in R$ for every $i < k$. A state $x \in X$ is a *bad state* if $x \notin A$, and the transition system $\mathcal{X}$ is *safe* if no bad state is reachable. The safety checking is undecidable in general due to the unboundedness of the lengths of transitions. However, if the length of transitions is bounded, the problem becomes decidable. That means, for every $k \in \mathbb{N}$, whether there exists a bad state reachable within k steps is decidable (provided that I, R and A are definable in $\mathcal{L}$). A classic idea in verification [McMillan 2003] is to solve bounded problems by gradually increasing the bound k and to use the solutions for bounded problems as a hint to solve the unbounded problem.

Example 4. The transition system $\mathcal{X} = (\mathbb{Z}, \{x \mid 2 \leq x \leq 8\}, \{(x, 2x-3) \mid x \in \mathbb{Z}\}, \{x \mid x \geq -5\})$ is unsafe. This system is safe within k steps for $k \leq 3$ and unsafe within k steps for $k > 3$. The bounded unsafety of $\mathcal{X}$ (for $k > 3$) shows the (undounded) unsafety of $\mathcal{X}$. □

Example 5. The transition system $\mathcal{X}' = (\mathbb{Z}, \{x \mid 2 \leq x \leq 8\}, \{(x, 2x) \mid x \in \mathbb{Z}\}, \{x \mid x \geq -5\})$ is safe (and safe within k steps for every k). The safety of $\mathcal{X}'$ within 5 steps can be witnessed by an overapproximation of the set of states reachable within 5 steps, such as $\{x \mid 2 \leq x \leq 256\}$ and $\{x \mid 0 \leq x\}$. The latter is closed under the transition, so it shows the unbounded safety of $\mathcal{X}'$. □

The CHC systems corresponding to safety verification of transition systems are of the form

$$L \;=\; \{\, \iota(x) \Rightarrow P(x), \quad P(x) \land \tau(x,y) \Rightarrow P(y), \quad P(x) \Rightarrow \alpha(x) \,\}. \tag{2}$$

The satisfiability of L in Equation (2) is equivalent to the safety of $\mathcal{X} = (D, I, R, A)$ where D is the range of the variables x and y, $I = \{\, d \in D \mid\; \models \iota(d) \,\}$, $R = \{\, (d, d') \in D \times D \mid\; \models \tau(d, d') \,\}$ and $A = \{\, d \in D \mid\; \models \alpha(d) \,\}$. The bounded version also has a CHC representation. For example, the CHC system $L^{(2)}$ corresponding to the bounded version with $k = 2$ is given by

$$\begin{array}{rrr} & P_2(x) \land \tau(x,y) \Rightarrow P_1(y), & P_1(x) \land \tau(x,y) \Rightarrow P_0(y), \\ \iota(x) \Rightarrow P_2(x), & \iota(x) \Rightarrow P_1(x), & \iota(x) \Rightarrow P_0(x), \\ P_2(x) \Rightarrow \alpha(x), & P_1(x) \Rightarrow \alpha(x), & P_0(x) \Rightarrow \alpha(x). \end{array} \tag{3}$$

The first and second lines require that P_2 to be an overapproximation of initial states, P_1 to be an overapproximation of states reachable by 0 or 1 step, and P_0 to be an overapproximation of states reachable within 2 steps. The third line requires that P_2, P_1 and P_0 contain no bad state. For general $k \in \mathbb{N}$, the CHC system $L^{(k)}$ is the set of constraints over predicate variables $P_k, \ldots, P_0$ given by

$$L^{(k)} \;=\; \{\, P_{i+1}(x) \land \tau(x,y) \Rightarrow P_i(y) \;\mid 0 \leq i < k \,\} \cup \{\, \iota(x) \Rightarrow P_i(x),\; P_i(x) \Rightarrow \alpha(x) \;\mid 0 \leq i \leq k \,\}.$$

The CHC system $L^{(k)}$, which we call the *k-th approximation* of L, has some remarkable properties. The most notable property is the decidability of satisfiability, coming from the acyclicity of the dependencies between predicate variables. The *dependency graph* of a CHC system has predicate variables as nodes, and it has an edge from P to Q if the system has a CHC of the form $(\cdots \land Q \land \cdots) \Rightarrow P$. For example, the dependency graphs of L and of $L^{(k)}$ are

$$\begin{tikzcd} P \arrow[loop right] \end{tikzcd} \quad \text{and} \quad P_0 \longrightarrow P_1 \longrightarrow \cdots \longrightarrow P_k,$$

respectively. Intuitively, the infinite path in the former graph causes the unboundedness of transitions, and the length of the longest path in the latter is the bound k.[4] The acyclicity allows us to construct a solution by induction, going from leaf to root (see Sections 4.1 and 4.2). A solution ζ of $L^{(k)}$ may provide a solution of L. So, we can apply the same strategy as in verification: it solves $L^{(k)}$ with increasing k, expecting that solutions of approximations help to solve the original problem L.

Example 6. The CHC system L' corresponding to $\mathcal{X}'$ in Example 5 consists of $2 \leq x \leq 8 \Rightarrow P(x)$, $P(x) \Rightarrow P(2x)$ and $P(x) \Rightarrow x \geq -5$. Then $L'^{(5)}$ is

$$\{\, P_{i+1}(x) \Rightarrow P_i(2x) \;\mid 0 \leq i < 5 \,\} \cup \{\, 2 \leq x \leq 8 \Rightarrow P_i(x),\; P_i(x) \Rightarrow x \geq -5 \;\mid 0 \leq i \leq 5 \,\},$$

and a solution ξ is $\xi(P_5)(x) = (0 \leq x \leq 10)$, $\xi(P_4)(x) = (0 \leq x \leq 100)$, $\xi(P_3)(x) = (0 \leq x \leq 1000)$, $\xi(P_2)(x) = (0 \leq x \leq 10000)$ and $\xi(P_1) = \xi(P_0) = (0 \leq x)$. In this case, $\varphi(x) := \xi(P_0)(x)$ is a solution of L. □

[4]However, the direction of the edges is opposite to the direction of the transition.

This strategy is applicable to S in Equation (1) as well. The difference from L in Equation (2) is that the rule $P(x) \land P(y) \land \tau(x,y,z) \Rightarrow P(z)$ has two occurrences of P in the left-hand side of $\Rightarrow$. In order to precisely track the occurrences, an approximation of S has predicate variables P_w indexed by words $w \in \{L,R\}^*$ over L and R. We write ϵ for the empty word and $|w|$ for the length of the word $w \in \{L,R\}^*$. The *k-th approximation of* S, written $S^{(k)}$, is a CHC system over predicate variables $\{P_w \mid w \in \{L,R\}^*, |w| \leq k\}$ given by

$$S^{(k)} \quad := \quad \begin{array}{l} \{\, P_{wL}(x) \land P_{wR}(y) \land \tau(x,y,z) \Rightarrow P_w(z) \;\mid w \in \{L,R\}^*,\ |w| < k\} \\ \quad \cup\{\, \iota(x) \Rightarrow P_w(x), \quad P_w(x) \Rightarrow \alpha(x) \;\mid w \in \{L,R\}^*,\ |w| \leq k\}. \end{array}$$

Similar to the above cases, a solution ξ of an approximation $S^{(k)}$ can provide a solution of the original problem S, for example, if $\xi(P_\epsilon) = \xi(P_L) = \xi(P_R)$.[5] So, our basic strategy is to solve approximations $S^{(k)}$ with increasing k and then to use solutions of $S^{(k)}$ to construct a solution of S.

The dependency graph of $S^{(k)}$ is a complete binary tree, namely,

$$\begin{tikzcd}
 & & P_{LL}\cdots \\
 & P_L \arrow[ru] \arrow[r] & P_{LR}\cdots \\
P_\epsilon \arrow[ru] \arrow[rd] & & P_{RL}\cdots \\
 & P_R \arrow[ru] \arrow[r] & P_{RR}\cdots
\end{tikzcd}$$

A CHC system is called *tree-like* by Rümmer et al. [2014] if its dependency graph is a tree,[6] and a solution of a tree-like CHC system coincides with a *tree interpolant* [Heizmann et al. 2010].[7] In this terminology, our strategy is to solve CHCs by iteratively computing tree interpolations.

3.2 Interpolation-Based CHC Solving

The basic strategy invokes subprocedures solving tree-like CHC systems $S^{(0)}, S^{(1)}, \ldots$. These subproblems can be solved independently, but a more efficient approach exploits the similarity between $S^{(k-1)}$ and the subset of $S^{(k)}$ consisting of the constraints for $\{P_{Lw} \mid w \in \{L,R\}^*, |w| < k\}$.

Suppose that we have solved $S^{(k-1)}$, yielding a solution ξ. Let ζ be an assignment to predicate variables $\{P_w \mid w \in \{L,R\}^*, |w| \leq k\}$ for $S^{(k)}$ given by

$$\zeta(P_\epsilon) \;:=\; \top \qquad \text{and} \qquad \zeta(P_w) \;:=\; \xi(P_v) \quad \text{if } w = Lv \text{ or } w = Rv.$$

The assignment ζ is almost a solution of $S^{(k)}$; the assignment ζ satsifies all constraints in $S^{(k)}$ except for $P_\epsilon(x) \Rightarrow \alpha(x)$. So, we do not need to find a solution of $S^{(k)}$ from scratch. It suffices to adjust the "almost-solution" ζ so that $P_\epsilon(x) \Rightarrow \alpha(x)$ is satisfied.

Such "almost-solutions" are called *traces* in IC3/PDR [Bradley 2011] and their relatives (e.g. [Hoder and Bjørner 2012]). Formally, a *trace* Φ for $S^{(k)}$ is an assignment to $\{P_w \mid w \in \{L,R\}^*, |w| \leq k\}$ satisfying $S^{(k)} \setminus \{P_\epsilon \Rightarrow \alpha(x)\}$. The value $\Phi(P_w)$ for P_w shall be written as $\Phi(w)$ and φ_w. The *domain* $\mathrm{dom}(\Phi)$ of the trace Φ of $S^{(k)}$ is $\{w \in \{L,R\}^* \mid |w| \leq k\}$. For $w \in \mathrm{dom}(\Phi)$, the *subtrace* Φ_w is a trace of $S^{(k-|w|)}$ defined by $\Phi_w(v) := \Phi(wv)$. The immediate subtraces Φ_L and Φ_R often appear in the description of algorithms. For traces Φ and Φ' of $S^{(k-1)}$ such that $\varphi_\epsilon \Rightarrow \alpha$ and $\varphi'_\epsilon \Rightarrow \alpha$, we write $(\top, \Phi, \Phi')$ for the trace Ψ of $S^{(k)}$ such that $\Psi(\epsilon) = \top$, $\Psi_L = \Phi$ and $\Psi_R = \Phi'$.

Recall that it suffices for solving $S^{(k)}$ to adjust a trace Φ so that $P_\epsilon \Rightarrow \alpha$ is satisfied. The main problem addressed in this paper is a slight generalization, which we call *refinement* problem:

> Given a trace Φ of $S^{(k)}$ and an assertion $\neg\beta$, find a trace Φ' of $S^{(k)}$ that satisfies $\varphi'_\epsilon(x) \Rightarrow \neg\beta(x)$ and $\varphi'_w(x) \Rightarrow \varphi_w(x)$ (for every $w \in \mathrm{dom}(\Phi)$).

[5] A weaker condition suffices. See the next subsection.

[6] Strictly speaking, the class of tree-like CHCs in this sense is slightly wider than the original (for the simplicity of presentation). An approximation $S^{(k)}$ is tree-like in the sense of Rümmer et al. [2014] as well.

[7] Heizmann et al. [2010] called this notion *nested interpolant*.

Algorithm 2 CHC solving by using a refinement procedure

```
1: function CHCSolve($\iota, \tau, \alpha$)
2:     $\Phi = ()$;   $n := 0$
3:     while true do
4:         $\Phi := (\top, \Phi, \Phi)$;   $n := n + 1$
5:         $R \in \textsc{Refine}(\Phi, \alpha;\ \iota, \tau)$
6:         if $R =$ None then
7:             return UNSAT
8:         $\Phi := R$
9:         for $i := 0, \dots, n - 1$ do
10:            if $\models \forall x. \bigwedge_{j \le i} \bigwedge_{|w|=j} \varphi_w(x) \Rightarrow \bigwedge_{|v|=i+1} \varphi_v(x)$ then
11:                return $\bigwedge_{j \le i} \bigwedge_{|w|=j} \varphi_w(x)$
```

A trace that satisfies the above condition is called a *refinement* of Φ w.r.t. $\neg\beta$ (borrowing the terminology of [McMillan 2006]). Let $\textsc{Refine}(\Phi, \neg\beta; \iota, \tau)$ (or more simply $\textsc{Refine}(\Phi, \neg\beta)$ when ι and τ are clear from the context) be the set of refinements (or symbol None if there is no refinement):

$$\begin{aligned}\textsc{Refine}(\Phi, \neg\beta) := \{\, \Phi' \mid \Phi' \text{ is a refinement of } \Phi \text{ w.r.t. } \neg\beta \,\}\\ \cup \{\, \text{None} \mid \text{there is no refinement of } \Phi \text{ w.r.t. } \neg\beta \,\}.\end{aligned}$$

The refinement problem itself is a tree interpolation problem, so it is computable. The main interest of this paper is an efficient refinement procedure.

Once a refinement procedure is given, a procedure for the CHC satisfiability can be given by Algorithm 2. Starting from the trivial trace $\Phi = ()$, the procedure iterates the following process.

(1) **Line 4** extends the trace by adding a new root $\top$.
(2) **Lines 5–7** refine the trace Φ with respect to α (*i.e.* performing the bounded model-checking). If an error is found (i.e. if $R =$ None), then the CHC system is unsatisfiable.
(3) **Line 8** updates the trace Φ.
(4) **Lines 9–11** try to extract an invariant from the current trace.

Theorem 7. *Algorithm 2 is sound. That is, if it returns* UNSAT*, then the CHC system S in Equation (1) is unsatisfiable, and if it returns a formula, then it is a solution of the CHC system. It is refutationally complete if Line 5 always terminates.*

Remark 8. Here, we note some differences from the standard setting. First, a trace in our setting has the tree structure (*i.e.* a branching structure) instead of the linear structure. To get a trace depending only on the depth, simply take the conjunction of all the nodes at that depth. Second, in the standard setting, a trace is *monotone* (*i.e.* $\varphi_{wL} \Rightarrow \varphi_w$ and $\varphi_{wR} \Rightarrow \varphi_w$ in our notation), whereas we do not require monotonicity. Third, the root in our setting is the *deepest* part in the standard setting. Hence the condition in line 9 (Algorithm 2) is usually written as $O_{n-i} \Rightarrow O_{n-i-1}$ in the standard setting, where O_m is the component of the trace O at depth m. □

3.3 Spacer

Spacer [Komuravelli et al. 2015, 2014, 2016] is a well-known and highly efficient procedure for CHC solving. This paper aims to give an inductive description of Spacer to make it easier to reason about and develop its variant. Here, we briefly review the procedure using the traditional description based on abstract transition systems. The details of Spacer vary slightly in the literature, and the discussion here is based on Komuravelli et al. [2015].

(Candidate) $[\models \psi(z) \Rightarrow (\varphi_0(z) \wedge \neg\alpha(z))]$
$Q := Q \cup \{(\psi, 0)\}$

(DecideMust) $\left[\begin{array}{l}(\psi, n) \in Q \\ \mathcal{M} \models \varphi_{n+1}(x) \wedge \mathcal{U}(y) \wedge \tau(x,y,z) \wedge \psi(z)\end{array}\right]$
let $\vartheta(x) = \textsc{Mbp}(\exists yz.\varphi_{n+1}(x) \wedge \mathcal{U}(y) \wedge \tau(x,y,z) \wedge \psi(z),\ \mathcal{M})$ in
$Q := Q \cup \{(\vartheta, n+1)\}$

(DecideMay) $\left[\begin{array}{l}(\psi, n) \in Q \\ \mathcal{M} \models \varphi_{n+1}(x) \wedge \varphi_{n+1}(y) \wedge \tau(x,y,z) \wedge \psi(z)\end{array}\right]$
let $\vartheta(y) = \textsc{Mbp}(\exists xz.\varphi_{n+1}(x) \wedge \varphi_{n+1}(y) \wedge \tau(x,y,z) \wedge \psi(z),\ \mathcal{M})$ in
$Q := Q \cup \{(\vartheta, n+1)\}$

(Conflict) $\left[\begin{array}{l}(\psi, n) \in Q \\ \models \varphi_{n+1}(x) \wedge \varphi_{n+1}(y) \wedge \tau(x,y,z) \Rightarrow \neg\psi(z)\end{array}\right]$
let $\vartheta(z) = \textsc{Itp}(\varphi_{n+1}(x) \wedge \varphi_{n+1}(y) \wedge \tau(x,y,z),\ \neg\psi(z))$ in
$\varphi_i(z) := \varphi_i(z) \wedge \vartheta(z)$ for $i \geq n$

(Leaf) $[(\psi, n) \in Q, \models \varphi_{n+1}(x) \wedge \varphi_{n+1}(y) \wedge \tau(x,y,z) \Rightarrow \neg\psi(z)]$
$Q := Q \cup \{(\psi, n+1)\}$

(Successor) $\left[\begin{array}{l}(\psi, n) \in Q \\ \mathcal{M} \models \mathcal{U}(x) \wedge \mathcal{U}(y) \wedge \tau(x,y,z) \wedge \psi(z)\end{array}\right]$
let $\gamma(z) = \textsc{Mbp}(\exists xy.\mathcal{U}(x) \wedge \mathcal{U}(y) \wedge \tau(x,y,z) \wedge \psi(z),\ \mathcal{M})$ in
$\mathcal{U} := \mathcal{U} \vee \gamma$

(Induction) $\left[\begin{array}{l}\varphi_n = (\cdots \wedge (\psi \vee \psi') \wedge \cdots) \\ \models (\varphi_n(x) \wedge \psi(x)) \wedge (\varphi_n(y) \wedge \psi(x)) \wedge \tau(x,y,z) \Rightarrow \psi(z)\end{array}\right]$
$\varphi_i := \varphi_i \wedge \psi$ for $i = n-1, n, n+1, \ldots, N$

(Unfold) $[\models \varphi_\epsilon(z) \Rightarrow \alpha(z)]$
$\varphi_{n+1} := \varphi_n$ for $n = N, N-1, \ldots, 0$
$\varphi_0 := \top, N := N+1$

Fig. 1. Transtion rules for Spacer [Komuravelli et al. 2015, 2014, 2016]. The rules (Safe) and (Unsafe) in the original system, which perform output, are not regarded as transition rules in our formalism.

A state is a tuple $(N, \Phi, Q, \mathcal{U})$. We explain the meaning of each component.

- N is the depth of the current approximation.
- $\Phi = (\varphi_n)_{0 \leq n \leq N}$ is a trace with the linear structure. The trace must be *monotone*, *i.e.* $\varphi_{n+1}(z) \Rightarrow \varphi_n(z)$ for every n.
- Q is a collection of pairs (ψ, i) of a formula ψ and $0 \leq i \leq N$. A pair (ψ, i) is called a *query* and expresses a request to strengthen φ_i so that $\models \varphi_i(z) \Rightarrow \neg\psi(z)$.
- $\mathcal{U} \subseteq \mathcal{S}$ is an under-approximation of the minimum solution of $\{\iota(x) \Longrightarrow P(x), P(x) \wedge P(y) \wedge \tau(x,y,z) \Longrightarrow P(z)\}$. Hence, if γ intersects with $\neg\alpha$, the CHC system is unsatisfiable.

What is new about Spacer compared to its predecessors such as [Bradley 2011; Een et al. 2011] and GPDR [Hoder and Bjørner 2012] is that it also manages under-approximations.

The initial state is $(0, \Phi_0, \emptyset, \emptyset)$ where Φ_0 consists only of the root $\varphi_0 = \top$. The procedure will terminate with SAT (resp. UNSAT) when the transition system reaches a state where $\models \varphi_n(z) \Rightarrow \varphi_{n+1}(z)$ for some n (resp. where $\mathcal{U} \wedge \neg\alpha \neq \emptyset$).

The transition rules are shown in fig. 1. Each rule can be invoked only if all conditions in $[\cdots]$ are satisfied. The rule **(Unfold)** corresponds to Line 4 of Algorithm 2 and other rules compute a refinement. The rules **(Candidate)**, **(DecideMay)**, **(DecideMust)** and **(Leaf)** are for query generation. **(Candidate)** requires that $\neg\varphi_0(c)$ for a bad state c. **(DecideMay)** and **(DecideMust)** generate a query for level $n+1$ from a query for level n: **(DecideMay)** generates a sufficient condition and **(DecideMust)** generates a necessarily condition to resolve the query for level n. Note that **(DecideMust)** utilizes the under-approximation $\mathcal{U}$ to generate a query that is a necessary condition for (ψ, n). The rule **(Leaf)** propagates a query that has been successfully resolved to the adjacent level. The rule **(Conflict)** resolves a query by strengthening φ_n. If a query (ψ, n) is found to be unresolvable, **(Successor)** produces an under-approximation, which witnesses that the query is unresolvable. The rule **(Induction)** strengthens the trace by conjoining a subformula in the trace. This rule is heuristics to improve the efficiency.

Variation and refutational completeness. The transition system in fig. 1 comes from Komuravelli et al. [2015] and differs from the original procedure [Komuravelli et al. 2014, 2016] in some details.[8] Here, we summarize the variants and the status of their refutational completeness.

Spacer is parameterized by two procedures: an interpolating theorem prover and a model-based projection. Refutational completeness depends on the choices of background theory and these procedures,[9] and Komuravelli et al. [2016] claimed refutational completeness independent of the choice of these procedures. So, we focus on refutational completeness independent of the choice of subprocedures. As mentioned in Section 1, Tsukada and Unno [2022, Section 6.6] gave a counterexample (for details, see Appendix C in the arXiv version [Tsukada and Unno 2021]).

Theorem 9 (Tsukada and Unno [2022]). *The procedure described in Komuravelli et al. [2016] diverges on a CHC system over linear integer arithmetic, for a specific choice of an interpolating theorem prover and a model-based projection. So, the refutational completeness independent of subprocedures fails.*

There is a further subtlety. The transition system in fig. 1 coming from Komuravelli et al. [2015] differs from the original procedure [Komuravelli et al. 2014, 2016]: (1) the original procedure manages the under-approximation $\mathcal{U}$ by level, so $\mathcal{U} = (\mathcal{U}_i)_{0 \le i \le N}$; and (2) the argument formulas for Mbp are different. For example, in **(Successor)** in the original procedure, the argument formula is $\mathcal{U}_n(x) \wedge \mathcal{U}_n(y) \wedge \tau(x, y, z)$, which does not involve the query ψ. Despite these differences, the counterexample in Theorem 9 also applies to Komuravelli et al. [2015]. Rather, these changes made by Komuravelli et al. [2015] introduce two additional sources of incompleteness (cf. Section 5).

The implementation of Spacer[10] is based on the description in Komuravelli et al. [2015] but has some differences. The biggest difference is that the transition system in fig. 1 does not specify the order of rules to apply, whereas the implementation adopts a specific order. The counterexample in Theorem 9 also applies to the implementation since the order of rule applications in the implementation coincides with that is adopted in Tsukada and Unno [2021, Appendix C].

These incompleteness issues have been overlooked even by experts, and our inductive approach clarifies this issue so that non-specialists can easily understand it.

4 INDUCTIVE APPROACH TO REFINEMENT

This section develops a refinement procedure defined by induction. The inductive structure that we exploit is the tree structure of an approximation of the input CHC system.

[8]We choose Komuravelli et al. [2015] because it seems to serve as a basis for the implementation and subsequent studies such as Krishnan et al. [2020]. Furthermore, two of the three authors of the original paper [Komuravelli et al. 2014, 2016] are also authors of Komuravelli et al. [2015]. These facts justify our choice.

[9]For example, it is refutationally complete if quantifier elimination is used as model-based projection.

[10]See https://github.com/Z3Prover/z3. The code related to Spacer is in the directory `src/muz/spacer`.

This section starts by clarifying the problem that we should solve by induction. Just as proof by induction often requires proving a stronger proposition than the one you wish to show, solving a problem by induction sometimes requires solving a more general problem. We propose a generalization of the refinement problem suitable for induction.

The generalized problem has a naïve inductive solver using quantifier elimination (or equivalently, manipulation of quantified formulas). We will then modify this solver. The idea is to replace the quantifier elimination procedure with Algorithm 1, which iteratively applies the model-based projection procedure Mbp, and then to make calls to Mbp as lazy as possible, while keeping the inductive structure of the solver unchanged. The idea of making quantifier elimination lazy can be found in the original Spacer paper [Komuravelli et al. 2016], but this section differs in that we do not change the inductive structure nor properties of the procedure.

4.1 Generalizing the Refinement Problem

This subsection explains a difficulty in inductively solving the non-linear CHC refinement problem and proposes a generalization appropriate for an inductive solver.

For the subclass of CHC systems known as *linear CHC systems*, one can easily provide an inductive refinement procedure. A CHC $P_1(x_1) \wedge \cdots \wedge P_n(x_n) \wedge \tau(\vec{x}, y) \Rightarrow Q(y)$ is *linear* if $n \leq 1$. For example, consider a linear CHC system

$$\left\{\ \iota(x) \Longrightarrow P(x), \qquad P(x) \wedge \tau(x,y) \Longrightarrow P(y), \qquad P(x) \Longrightarrow \alpha(x)\ \right\}$$

and a trace $\Phi = (\varphi_k, \ldots, \varphi_1, \varphi_0)$ (so $\varphi_{i+1}(x) \wedge \tau(x,y) \Rightarrow \varphi_i(x)$ for $0 \leq i < k$, $\iota(x) \Rightarrow \varphi_i(x)$ for $0 \leq i \leq k$, and $\varphi_i(x) \Rightarrow \alpha(x)$ for $0 < i \leq k$). The refinement problem asks to strengthen the trace Φ to satisfy $\varphi_0(x) \Rightarrow \alpha_0(x)$ for a given property α_0. It is easy to solve the refinement problem by induction: we first refine the subtrace $\Phi_1 = (\varphi_k, \ldots, \varphi_1)$ against the assertion $\alpha_1(x) := \neg(\exists y. \tau(x,y) \wedge \neg\alpha_0(y))$, yielding $\Phi'_1 = (\varphi'_k, \ldots, \varphi'_1)$, and then find φ'_0 that satisfies $(\iota(x) \vee (\varphi'_1(x) \wedge \tau(x,y))) \Rightarrow \varphi'_0(y)$ and $\varphi'_0(x) \Rightarrow \alpha_0(x)$ using an interpolating theorem prover. The assertion α_1 is canonical in the sense that the refinement problem (Φ, α_0) has a solution if and only if (Φ_1, α_1) has a refinement.

The difficulty of non-linear CHC solving is that there is no canonical choice of the assertion.

Example 10. Consider the CHC system

$$\left\{\ (x = 3) \Longrightarrow P(x), \quad P(x) \wedge P(y) \wedge (z = |x - y|) \Longrightarrow P(z), \quad P(z) \Rightarrow (z \leq 5)\ \right\}$$

and a trace $\varphi_L(x) = \varphi_R(x) = (x \leq 5)$ and $\varphi_\epsilon = \top$ for its approximation of depth 1. An inductive refinement procedure IndRefine needs to strengthen φ_L and/or φ_R so that

$$\varphi_L(x) \wedge \varphi_R(y) \wedge (z = |x - y|) \Longrightarrow (z \leq 5)$$

by invoking recursive calls IndRefine(φ_L, α_L) and/or IndRefine(φ_R, α_R) for appropriate α_L, α_R. So the procedure should appropriately choose a direction $d \in \{L, R\}$ and an assertion α_d, but there is no canonical choice. A candidate is the pair (L, α_L) where

$$\alpha_L \quad := \quad \neg\exists yz. \varphi_R(y) \wedge (z = |x - y|) \wedge (z > 5),$$

which is sufficiently strong in the sense that if IndRefine(φ_L, α_L) succeeded, we would finish the whole refinement procedure. However α_L is too strong so IndRefine(φ_L, α_L) fails. A similar approach for $d = R$ fails for the same reason. □

The above example shows that a natural choice of an assertion for a recursive call may be unnecessarily strong. This observation motivates us to generalize the refinement problem so that the procedure performs best, even if no solution exists.

Definition 11. Let Φ be a trace and $\neg\beta$ be an assertion. The *counterexample* γ for the refinement problem $(\Phi, \neg\beta)$ is defined by $\gamma := \min\{\gamma' \mid \textsc{Refine}(\Phi, (\neg\beta) \vee \gamma') \neq \texttt{None}\}$. The *generalized refinement problem* asks, given a trace Φ and an assertion $\neg\beta$, to find a pair (Φ', γ) such that γ is the counterexample of $(\Phi, \neg\beta)$ and $\Phi' \in \textsc{Refine}(\Phi, \neg\beta \vee \gamma)$. □

For implementational reasons, we often slightly weaken the requirement for γ. A predicate γ is a counterexample in the weak sense if (1) $\textsc{Refine}(\Phi, \neg\beta) \neq \texttt{None}$ and $\gamma = \bot$ or (2) $\textsc{Refine}(\Phi, \neg\beta) = \texttt{None}$ and $\gamma \wedge \beta$ is a counterexample in the proper sense.

4.2 Naïve Procedure

Algorithm 3 presents a naïve procedure for the generalized refinement problem.

- **Line 2**: The procedure immediately returns without making any change to the input Φ if the requirement is trivially satisfied.
- **Lines 4–6**: If the initial state $\iota(z)$ intersects with $\neg\alpha(z)$, the requirement is trivially unachievable. The requirement α must be weakened at least to $\alpha(z) \vee \gamma(z)$ where $\gamma(z) = (\iota(z) \wedge \neg\alpha(z))$. The following part checks if α should be further weakened.
- **Line 7**: The condition checks if one can refine the input trace Φ without changing the subtraces Φ_L and Φ_R. If it is not the case, *i.e.* one needs to change at least one of Φ_L and Φ_R, lines 8–15 are executed.
- **Lines 8 and 9**: We first try to refine the trace without changing the left subtrace (i.e. changing only the subtrace Φ_R). The condition required for the refinement subtrace Φ'_R is that $\varphi_L(x) \wedge \varphi'_R(y) \wedge \tau(x, y, z)$ does not intersects with $\neg\alpha(z)$. Line 8 lets $\neg\psi_R$ be the weakest predicate that satisfies this requirement. Line 9 checks if Φ_R can be refined so that $\varphi'_R(y) \models \neg\psi_R(y)$.
- **Lines 10–12**: If $\gamma_R \neq \bot$, we need to change Φ_L. The refinement procedure for Φ_L is similar to that of Φ_R (lines 8 and 9).
- **Lines 13–15**: If $\gamma_L \neq \bot$, there is no refinement of Φ satisfying α. The counterexample found at this step comes from γ_L and γ_R.
- **Lines 16 and 17**: Now α is sufficiently weakened and φ_L and φ_R are sufficiently strengthened. We calculate an appropriate φ_ϵ.

Proposition 12. *Algorithm 3 always terminates and solves the generalized refinement problem.* □

4.3 Handling Quantification by Model-Based Projection

Algorithm 3 is simple but it uses quantified formulas. The quantifiers can be removed by using quantifier elimination, but a quantifier elimination procedure is computationally expensive.

Algorithm 4 is a refinement procedure using model-based projection for reasoning about quantified formulas. It is basically obtained by replacing the quantified formulas φ_R, φ_L and γ in lines 8, 11 and 14 in Algorithm 3 with the quantifier-elimination based on model-based projection (Algorithm 1). Intuitively lines 8–9 (resp. lines 11–12 and lines 14–15) perform quantifier elimination of $\varphi_L(x) \wedge \tau(x, y, z) \wedge \neg\alpha(z)$ (resp. $\gamma_R(y) \wedge \tau(x, y, z) \wedge \neg\alpha(z)$ and $\gamma_L(x) \wedge \gamma_R(y) \wedge \tau(x, y, z)$).

It is worth noting an important twist in Algorithm 4 for its termination. Line 7 records the value of φ_L at that point, and model-based projection uses the recorded values. This twist makes the formulas for model-based projection loop invariants, and the loop invariance plays a crucial role in the termination proof. Note that the image finiteness of model-based projection is applied only to the case where the argument formula is unchanged.

Algorithm 4 enjoys nice properties that the naïve algorithm (Algorithm 3) has. The proof is essentially the same as that for Algorithm 3 except for the above-mentioned twist.

Theorem 13. *Algorithm 4 always terminates and solves the generalized refinement problem.*

Algorithm 3 Naïve refinement procedure

```
function NAÏVE($\Phi, \alpha$)
    if dom($\Phi$) $= \emptyset$ or $\varphi_\epsilon(x) \models \alpha(x)$ then return $(\Phi, \bot)$
    $\gamma := \bot$
    if $\models \exists z.\iota(z) \wedge \neg\alpha(z)$ then
        $\gamma(z) := \iota(z) \wedge \neg\alpha(z)$
        $\alpha(z) := \alpha(z) \vee \gamma(z)$
    if $\models \exists x.\exists y.\exists z.\varphi_L(x) \wedge \varphi_R(y) \wedge \tau(x,y,z) \wedge \neg\alpha(z)$ then
        $\psi_R(y) := \exists x.\exists z.\varphi_L(x) \wedge \tau(x,y,z) \wedge \neg\alpha(z)$
        $(\Phi_R, \gamma_R) :=$ NAÏVE($\Phi_R, \neg\psi_R$)
        if $\gamma_R \neq \bot$ then
            $\psi_L(x) := \exists y.\exists z.\gamma_R(y) \wedge \tau(x,y,z) \wedge \neg\alpha(z)$
            $(\Phi_L, \gamma_L) :=$ NAÏVE($\Phi_L, \neg\psi_L$)
            if $\gamma_L \neq \bot$ then
                $\gamma(z) := \gamma(z) \vee \big(\exists x.\exists y.\gamma_L(x) \wedge \gamma_R(y) \wedge \tau(x,y,z) \wedge \neg\alpha(z)\big)$
                $\alpha(z) := \alpha(z) \vee \gamma(z)$
    $\varphi_\epsilon(z) :=$ ITP$\big(\iota(z) \vee \big(\varphi_L(x) \wedge \varphi_R(y) \wedge \tau(x,y,z)\big),\ \varphi_\epsilon(z) \wedge \alpha(z)\big)\big)$
    return $(\Phi, \gamma)$
```

Algorithm 4 Naïve procedure with model-based projection

```
function NAÏVEMBP($\Phi, \alpha$)
    if dom($\Phi$) $= \emptyset$ or $\varphi_\epsilon(x) \models \alpha(x)$ then return $(\Phi, \bot)$
    $\Gamma(z) := \bot$
    if $\models \exists z.\iota(z) \wedge \neg\alpha(z)$ then
        $\gamma(z) := \iota(z) \wedge \neg\alpha(z)$
        $\Gamma(z) := \gamma(z)$
    const $\varphi_{0,L} := \varphi_L$
    while $\exists \mathcal{M}_R.\ \mathcal{M}_R \models \varphi_L(x) \wedge \varphi_R(y) \wedge \tau(x,y,z) \wedge \neg(\alpha(z) \vee \Gamma(z))$ do
        $\psi_R(y) :=$ MBP($\exists xz.\varphi_{0,L}(x) \wedge \tau(x,y,z) \wedge \neg\alpha(z), \mathcal{M}_R$)
        $(\Phi_R, \gamma_R) :=$ NAÏVEMBP($\Phi_R, \neg\psi_R$)
        while $\exists \mathcal{M}_L.\ \mathcal{M}_L \models \varphi_L(x) \wedge \gamma_R(y) \wedge \tau(x,y,z) \wedge \neg(\alpha(z) \vee \Gamma(z))$ do
            $\psi_L(x) :=$ MBP($\exists yz.\gamma_R(y) \wedge \tau(x,y,z) \wedge \neg\alpha(z), \mathcal{M}_L$)
            $(\Phi_L, \gamma_L) :=$ NAÏVEMBP($\Phi_L, \neg\psi_L$)
            while $\exists \mathcal{M}.\ \mathcal{M} \models \gamma_L(x) \wedge \gamma_R(y) \wedge \tau(x,y,z) \wedge \neg(\alpha(z) \vee \Gamma(z))$ do
                $\gamma(z) :=$ MBP($\exists xy.\gamma_L(x) \wedge \gamma_R(y) \wedge \tau(x,y,z), \mathcal{M}$)
                $\Gamma(z) := \Gamma(z) \vee \gamma(z)$
    $\varphi_\epsilon(z) :=$ ITP$\big(\iota(z) \vee (\varphi_L(x) \wedge \varphi_R(y) \wedge \tau(x,y,z)),\ \varphi_\epsilon(z) \wedge (\alpha(z) \vee \Gamma(z))\big)$
    return $(\Phi, \Gamma)$
```

PROOF SKETCH. We prove the termination; the soundness is relatively easy to see. The point is that the arguments for MBP in lines 9, 12, and 15 are invariants of the loops starting from lines 8, 11, and 14, respectively. Hence, ψ_R, ψ_L, and γ can only take on a finite variety of values. So, it suffices to show that the same value does not occur twice, which follows from the progress property. □

Algorithm 5 is a further lazy version of Algorithm 4. Recall that the counterexample Γ generated by Algorithm 4 is the disjunction $\Gamma = \gamma_1 \vee \cdots \vee \gamma_n$ of formulas given by MBP in line 15 (where n is the

Algorithm 5 A Spacer-like refinement procedure

```
1:  function IndSpacer($\Phi, \alpha$)
2:      if dom($\Phi$) $= \emptyset$ or $\varphi_\epsilon(x) \models \alpha(x)$ then return $(\Phi, \bot)$
3:      $\Gamma(z) := \bot$; $\Gamma_R(y) := \bot$
4:      if $\models \exists z.\iota(z) \land \neg\alpha(z)$ then
5:          $\gamma(z) := \iota(z) \land \neg\alpha(z)$
6:          return $(\Phi, \gamma)$
7:      const $\varphi_{L,0} := \varphi_L$
8:      while $\exists \mathcal{M}_R.\ \mathcal{M}_R \models \varphi_L(x) \land \varphi_R(y) \land \tau(x,y,z) \land \neg(\alpha(z) \lor \Gamma(z))$ do
9:          $\psi_R(y) := \text{Mbp}(\exists xz.\varphi_{L,0}(x) \land \tau(x,y,z) \land \neg\alpha(z), \mathcal{M}_R)$
10:         $(\Phi_R, \gamma_R) := \text{IndSpacer}(\Phi_R, (\neg\psi_R) \lor \Gamma_R)$
11:         $\Gamma_R(y) := \Gamma_R(y) \lor \gamma_R(y)$
12:         while $\exists \mathcal{M}_L.\ \mathcal{M}_L \models \varphi_L(x) \land \gamma_R(y) \land \tau(x,y,z) \land \neg(\alpha(z) \lor \Gamma(z))$ do
13:             $\psi_L(x) := \text{Mbp}(\exists yz.\gamma_R(y) \land \tau(x,y,z) \land \neg\alpha(z), \mathcal{M}_L)$
14:             $(\Phi_L, \gamma_L) := \text{IndSpacer}(\Phi_L, \neg\psi_L)$
15:             while $\mathcal{M} \models \gamma_L(x) \land \gamma_R(y) \land \tau(x,y,z) \land \neg(\alpha(z) \lor \Gamma(z))$ do
16:                 $\gamma(z) := \text{Mbp}(\exists xy.\gamma_L(x) \land \gamma_R(y) \land \tau(x,y,z), \mathcal{M})$
17:                 return $(\Phi, \gamma)$
18:     $\varphi_\epsilon(z) := \text{Itp}\big(\iota(z) \lor (\varphi_L(x) \land \varphi_R(y) \land \tau(x,y,z)),\ \varphi_\epsilon(z) \land (\alpha(z) \lor \Gamma(z))\big)$
19:     return $(\Phi, \bot)$
```

number of executions of Line 15 and γ_i is the result of Mbp in the i-th iteration). Algorithms 4 and 5 behave similarly until the first disjunct γ_1 of the counterexample is found, but then Algorithm 5 returns γ_1 without calculating the remaining part $\gamma_2 \lor \cdots \lor \gamma_n$ of the counterexample $\Gamma = \gamma_1 \lor \gamma_2 \lor \cdots \lor \gamma_n$. Basically Algorithm 5 is obtained by replacing $\Gamma(z) := \Gamma(z) \lor \gamma(z)$ with **return** γ.[11]

Because of this change, Algorithm 5 is no longer a procedure for the generalized refinement problem. It just returns a piece of the counterexample, so one needs to invoke IndSpacer many times to obtain the whole counterexample as in the following program:

```
$\Gamma(z) := \bot$
$(\Phi, \gamma) := \text{IndSpacer}(\Phi, \alpha)$
while $\gamma \neq \bot$ do
    $\Gamma(z) := \Gamma(z) \lor \gamma(z)$                                        $(*)$
    $(\Phi, \gamma) := \text{IndSpacer}(\Phi, \alpha \lor \Gamma)$
return $(\Phi, \Gamma)$.
```

The variable Γ_R in Algorithm 5 originates from Γ in this program.

Theorem 14. *The procedure $(*)$ using Algorithm 5 always terminates and solves the generalized refinement problem.*

Proof sketch. We prove the termination; the soundness is relatively easy to show.

The most important property of IndSpacer is that the number of values possibly returned as γ is finite. This claim can be easily proved by induction. The return value γ is generated in line 16, which invokes Mbp, so the variety of γ is finite for a fixed γ_L and γ_R. The induction hypothesis shows that γ_L and γ_R range over a finite set. Hence, the variety of γ is finite.

In the program $(*)$, the returned γ is accumulated in Γ and IndSpacer$(\Phi, \alpha \lor \Gamma)$ will never return γ such that $\gamma \Rightarrow \Gamma$. So, the while loop in $(*)$ terminates, provided that each iteration terminates.

[11]The variable Γ will never be updated in Algorithm 5 and thus can be removed, but it is left for comparison with Algorithm 4.

The while loops in Algorithm 5 terminate for the same reason as Algorithm 4, namely that the arguments of Mbp are loop invariants. So, the progress ensures the termination. □

Remark 15. In fact, Algorithms 4 and 5 enjoy termination even without line 7 (and replace $\varphi_{L,0}$ with φ_L), but the termination proof is fairly sublte. For example, for the termination of the variants without line 7, it is necessary to store φ_L and φ_R separately, and it is not the case in Spacer implementations (as the traces are managed by levels). In such cases, $\varphi_{L,0}$ is essential for termination.

More subtly, $\varphi_{L,0}$ need not be a constant for the whole procedure. One can freely update $\varphi_{L,0}$ to the current value of φ_L during the execution of the body of the middle loop (Lines 12-16 and 13–17 in Algorithms 4 and 5, respectively) without losing refutational completeness. This claim can be proved by providing a termination measure reduced by entry into the body of the middle loop. The construction of a termination measure is, however, crucially relies on details of Algorithms 4 and 5.

5 COMPARING OUR PROCEDURES WITH SPACER

This section compares Algorithm 5 with procedures in Komuravelli et al. [2014, 2016] and Komuravelli et al. [2015], as well as GPDR [Hoder and Bjørner 2012]. Section 5.1 discusses the similarily; Section 5.2 discusses the dissimilarity mainly focusing on the refutational completeness. Section 5.3 deals with other differences and ways to fill the gap.

5.1 Similarity

The simplest way to formally show the similarity is to provide a transition system corresponding to Algorithm 5. Algorithm 5 is a first-order program and thus executable by a stack machine. The transition rules for the induced stack machine resemble the rules in fig. 1.

Unfortunately, we cannot discuss this in detail here due to space constraints, so we explain the correspondence between the rules in fig. 1 and the procedure in Algorithm 5 at an intuitive level. **(DecideMust)** is performed in lines 12–14 (recall that γ_R corresponds to $\mathcal{U}$). **(DecideMay)** is performed in lines 8–10. **(Conflict)** corresponds to lines 18–19. **(Successor)** is lines 15–17. **(Candidate)** and **(Unfold)** can be found in the top level function (Algorithm 2). **(Candidate)** is the function call IndSpacer(Φ, α) in line 5, and **(Unfold)** is line 4. **(Leaf)** and **(Induction)** have no corresponding part in Algorithm 5. These rules shall be discussed below.

Remark 16. Algorithm 5 with a special choice of Mbp yields a procedure like GPDR [Hoder and Bjørner 2012]. The choice is given by

$$\textsc{Mbp}(\exists\vec{x}.\vartheta(\vec{x},\vec{y}),\ \mathcal{M}) \quad := \quad (y_1 = c_1) \wedge \cdots \wedge (y_n = c_n)$$

where $y_1 \dots y_n = \vec{y}$ and $c_i = \mathcal{M}(y_i)$. In other words, this procedure returns the logical representation of the model $\mathcal{M}$ restricted to $\vec{y}$. This function satisfies most conditions for model-based projection except for the image finiteness.

The naïve procedure (Algorithm 3) is also obtained by choosing Mbp as the quantifier elimination. In this sense, Algorithm 5 unifies Spacer, GPDR, and the naïve algorithm.

5.2 Dissimilarity

There are three major differences between Algorithm 5 and Spacer presented in fig. 1. Each of these differences may cause the divergence of the procedure, breaking the refutational completeness.

The first difference is the arguments of Mbp in lines 9 and 13. The arguments in Algorithm 5 are loop invariants, and this fact plays a crucial role in the termination proof. In the corresponding rules **(DecideMust)**, **(DecideMay)** and **(Successor)** in fig. 1, the arguments are not "invariants". For example, the argument of Mbp in **(DecideMust)** (corresponding to lines 12–14 in Algorithm 5) is $\varphi_{n+1}(x) \land \mathcal{U}(y) \land \tau(x,y,z) \land \psi(z)$, which is $\varphi_L(x) \land \gamma_R(y) \land \tau(x,y,z) \land \neg\alpha(z)$ written in the symbols in Algorithm 5. This is not an invariant of the middle loop because of $\varphi_L(x)$.

The second difference is the argument of Mbp in line 16. This Mbp is called at most once, so the loop invariance does not matter here. However, the finiteness of possible return values γ, which is the key to the termination proof of Algorithm 5 (Theorem 14), essentially relies on the choice of the formula in line 16. The corresponding formula $\mathcal{U}(x) \land \mathcal{U}(y) \land \tau(x,y,z) \land \psi(z)$ in **(Successor)** in fig. 1 contains the query formula ψ, of which the variety cannot be finitely bound.

The third difference is the management of the counterexamples. Whereas Algorithm 5 deals the counterexamples γ_R and γ_L locally, the transition system in fig. 1 manipulates their cumulative union $\mathcal{U}$. This change also breaks the finiteness of γ possibly returned by the procedure, as we shall see below.

Interestingly, the second and third points were changes made in Komuravelli et al. [2015]. The original Spacer [Komuravelli et al. 2014, 2016] is closer to ours, except for the argument of Mbp in line 9: in the original Spacer, the argument of Mbp in (DecideMay)[12] is $\varphi_{n+1}(x) \land \tau(x,y,z) \land \psi(z)$, which involves a non-invariant φ_{n+1} (corresponding to φ_L in Algorithm 5). In our understanding, this difference is the unique source of possible divergence in the original Spacer.

5.3 Optimizations

This subsection discusses how to fill the gaps between our algorithms and actual implementations by modifying our algorithms. We refer to the implementational tricks discussed in this section *optimizations* in the hope that such tricks will improve performance. Some optimizations are implemented and empirically evaluated in Section 7.

Let us first discuss a criterion for ensuring the termination property for a modification of Algorithm 5. The points of the termination proof are (1) the arguments of model-based projection are loop invariants, (2) the same countermodel $\mathcal{M}$ does not appear twice in Line 8 (resp. in Line 12, in Line 15). Point (1) is achieved by saving the necessary values to local variables (as in Line 7), so any change to Φ does not affect this point, but a change to γ_L and/or γ_R would need care. Point (2) is not affected if an optimization only strengthens the trace, and optimizations below satisfy this criterion.

We discuss five optimizations. The first four do not break the termination, but the last one needs care since it is about γ_L and γ_R.

Predicate Sharing. We use a trace Φ with $\mathrm{dom}(\Phi) = \{L,R\}^{\leq k}$, which has a tree structure. Many existing procedures use a trace $(\varphi_i)_{0\leq i\leq k}$ of the linear structure, that means, from our viewpoint, they maintain additional constraints $\{\, \varphi_w = \varphi_{w'} \mid w, w' \in \{L,R\}^{\leq n}, |w| = |w'| \,\}$.

This difference can be bridged by turning traces into trees of *references* to logical formulas rather than trees of logical formulas. The tree $(\ell_w)_{w\in\{L,R\}^{\leq k}}$ of references to formulas is constructed so that the nodes ℓ_w and $\ell_{w'}$ point to the same reference cell whenever $|w| = |w'|$. Then Line 18 in Algorithm 5 updates, in effect, all the formulas at the same level simultaneously.

Monotone Trace. Another difference in the structure of trace is *monotonicity*. That means, many existing solvers maintain $\varphi_{i+1} \Rightarrow \varphi_i$. This gap can be filled by changing Line 18 in Algorithm 5 to conjoin the interpolant to every formula φ_w in the trace Φ.

Induction Rule. Roughly speaking, the induction rule checks whether a property ψ established at a level (*i.e.* φ_w contains ψ as a subformula) can be promoted to the adjacent level (and promotes the property if possible). This rule can be used at any time in the definition of the transition system, and it is difficult to incorporate the behavior of such a flexible rule into Algorithm 5. However,

[12]The rule is called (Query) in Komuravelli et al. [2016].

as Hoder and Bjørner [2012] noted in their description of the induction rule, there are typically two situations where this rule can be used effectively, namely, (1) when a refinement process of a recursive call finishes (*i.e.* at the end of each iteration of the outer and middle loops in Algorithm 5), and (2) when the trace is expanded at Line 4 in Algorithm 2. It is not difficult to perform the induction rule applied only for these moments: Simply analyze φ_L and/or φ_R at that moment, and conjoin to φ_ϵ the properties that also hold at the root ϵ. The analysis of φ_L and φ_R may take time, so our implementation records candidate properties for the induction rule in the refinement process and returns the list of candidates in addition to the frame and the counterexample.

Query Reuse. A version of Spacer given in Komuravelli et al. [2015] poses a query that has been successfully resolved to the adjacent level. This behavior can be simulated by inserting

$$(\Phi, \_) := \textsc{IndSpacer}(\Phi, \psi_L)$$

at the end of the middle loop in Algorithm 5 and similar code at the end of the outer loop.

Counterexample Sharing. Algorithm 5 passes only the counterexamples relevant to each query, but many procedures often remember only the cumulative union of the counterexamples found. To simulate this behavior, one can use a global variable, which stores the cumulative union of the counterexamples, and replace γ's in Algorithm 5 with the dereference of the global variable.

As we mentioned in Section 3.3, the original Spacer [Komuravelli et al. 2014, 2016] managed the counterexamples by levels, whereas fig. 1 and Komuravelli et al. [2015] unifies the counterexamples of all the levels. The former change is harmless, but the latter is not.

To see the point, let us recall the proof of the termination of Algorithm 5 (Theorem 14). The key is the fact that γ returned by Algorithm 5 has only a finite variety, and this fact is proved by induction. In the former style of counterexample sharing, the variety of the cumulated values $\mathcal{U}_i$ is finite (by induction on $N - i$). However, since the cumulated value $\mathcal{U}$ is not indexed by levels, the inductive argument to prove the finiteness of possible values for $\mathcal{U}$ fails. This is the second additional source of incompleteness introduced by the changes made in [Komuravelli et al. 2015].

6 A TERMINATING PROCEDURE USING COROUTINES

Although Algorithm 5 enjoys the termination property, it essentially relies on the finiteness of the variety of possible return values γ. The finiteness of return values is not a procedure-local property (i.e. it requires an analysis of scenarios where the procedure is invoked more than once), so it is hard to maintain. Indeed, we have just seen that the counterexample sharing across levels (as is done in Komuravelli et al. [2015]) breaks this property and cannot be incorporated into Algorithm 5.

This section presents another approach to termination guarantee. The termination guarantee of Algorithm 5 is hard because this procedure discards the remaining computation of the counterexample when it generates the first piece γ of the counterexample. A desirable procedure should return the control to the caller as soon as a piece γ of the counterexample is found but, at the same time, allow us to resume the continuation if necessary. These apparently incompatible requirements are achieved using a rich control mechanism known as *coroutine*.

Algorithm 6 shows the proposed algorithm. The structure is close to Algorithm 5 but it uses **yield** instead of **return**. Similar to **return**, the **yield** construct suspends the procedure and returns the control to the caller. But **yield** allows the caller to resume the continuation if necessary.

Let us first explain the behavior of the procedure. The procedure call $\textsc{McrCoroutine}(\Phi, \alpha)$ is immediately returned. Its value *cor* can be used to enumerate counterexamples. To get the next counterexample, call *cor*.**next**(α'), where α' is the latest assertion, which may be weaker than the original assertion α. This call returns a counterexample for α' if it exists. If there is no more counterexample, this call raises an exception `StopIteration`(Φ'), where Φ' is the refinement.

Algorithm 6 A Spacer-like procedure with early return using coroutine

```
1:  function $\textsc{McrCoroutine}(\Phi, \alpha)$
2:    if $\mathrm{dom}(\Phi) = \emptyset$ or $\varphi_\epsilon(x) \models \alpha(x)$ then return $\Phi$
3:    if $\models \exists z.\iota(z) \land \neg\alpha(z)$ then
4:      $\gamma(z) := \iota(z) \land \neg\alpha(z)$
5:      $\alpha :=$ yield $\gamma$
6:    const $\varphi_{0,L} := \varphi_L$; $\alpha_0 := \alpha$
7:    while $\exists \mathcal{M}_R.\ \mathcal{M}_R \models \varphi_L(x) \land \varphi_R(y) \land \tau(x,y,z) \land \neg\alpha(z)$ do
8:      $\psi_R(y) := \textsc{Mbp}(\exists xz.\varphi_{0,L}(x) \land \tau(x,y,z) \land \neg\alpha_0(z), \mathcal{M}_R)$
9:      $cor_R := \textsc{McrCoroutine}(\Phi_R, \neg\psi_R)$
10:     try loop
11:       $\gamma_R := cor_R$.next$(\neg\psi_R)$
12:       $\alpha_1 := \alpha$
13:       while $\exists \mathcal{M}_L.\ \mathcal{M}_L \models \varphi_L(x) \land \gamma_R(y) \land \tau(x,y,z) \land \neg\alpha(z)$ do
14:         $\psi_L(x) := \textsc{Mbp}(\exists yz.\gamma_R(y) \land \tau(x,y,z) \land \neg\alpha_1(z), \mathcal{M}_L)$
15:         $cor_L := \textsc{McrCoroutine}(\Phi_L, \neg\psi_L)$
16:         try loop
17:           $\gamma_L := cor_L$.next$(\neg\psi_L)$
18:           while $\exists \mathcal{M}.\ \mathcal{M} \models \gamma_L(x) \land \gamma_R(y) \land \tau(x,y,z) \land \neg\alpha(z)$ do
19:             $\gamma(z) := \textsc{Mbp}(\exists xy.\gamma_L(x) \land \gamma_R(y) \land \tau(x,y,z), \mathcal{M})$
20:             $\alpha :=$ yeild $\gamma$
21:           $\psi_L(x) := \psi_L(x) \land \neg\textsc{Itp}(\gamma_L(x),\ (\gamma_R(y) \land \tau(x,y,z)) \Rightarrow \alpha(z))$
22:         with StopIteration$(\Phi'_L) \to \Phi_L := \Phi'_L$
23:       $\psi_R(y) := \psi_R(y) \land \neg\textsc{Itp}(\gamma_R(y),\ (\varphi_L(x) \land \tau(x,y,z)) \Rightarrow \alpha(z))$
24:     with StopIteration$(\Phi'_R) \to \Phi_R := \Phi'_R$
25:   $\varphi_\epsilon(z) := \textsc{Itp}\big(\iota(z) \lor (\varphi_L(x) \land \varphi_R(y) \land \tau(x,y,z)),\ \varphi_\epsilon(z) \land \alpha(z)\big)$
26:   return $\Phi$
```

We explain the implementation (Algorithm 6). When the refinement procedure finds a partial counterexample (lines 18 and 19), it is passed to the **yield** construct (line 20). The **yield** construct suspends the rest of the computation of the counterexample, and the suspension is reported to the caller, with the information of the found partial counterexample γ (as the return value to the **next** call). The suspended procedure will be resumed with the information of a weakened requirement (passed as the argument to **next**), when the caller's calculation of the weakening will be finished. This additional information is obtained as the "return value" of **yeild**, and the procedure updates α to the weakened one (line 19). This weakening is propagated to the left and right subtraces by the recalculations (lines 21 and 23). (In line 22, the strengthening of φ'_L is also propagated.) When the refinement is completed, the procedure returns the refinement (line 26). The execution of **return** Φ raises the exception `StopIteration(Φ)`, which could be caught by the caller.

Although the control flow of Algorithm 6 is quite complicated, it is still defined by induction on the structure of the trace. This inductive structure helps us to reason about the algorithm.

Theorem 17. *The following procedure (Algorithm 6 with a wrapper) terminates and solves the generalized refinement problem.*

```
$\gamma := \bot$; $cor := \textsc{McrCoroutine}(\Phi, \alpha)$
try loop
```

```
        $\gamma := \gamma \vee cor.\mathbf{next}(\alpha \vee \gamma)$
    with StopIteration($\Phi'$) $\to$ return $(\Phi', \gamma)$.
```

Algorithm 6 can accomodate the cross-level counterexample sharing. Its termination proof only relies on the loop-invariance of arguments of Mbp, so it suffices to keep γ_L and γ_R invariant of the loop starting from line 18 by storing these values before the loop as in line 6.

7 IMPLEMENTATION AND EXPERIMENTS

We implemented variants of Spacer as a new tool called MuCyc[13] based on the inductive formalization of the paper and conducted a comparative evaluation on them as well as the state-of-the-art CHC solvers Spacer, Golem, and Eldarica that earned top scores in CHC-COMP'23.[14] We also evaluate the practical significance of the tricks to retain the refutational completeness (RC) discussed in the previous sections.

7.1 Implementation

MuCyc is implemented in the functional programming language OCaml 5 and supports, as background theories, Booleans and linear integer and real arithmetic. We adopted Z3 (version 4.12.6) [de Moura and Bjørner 2008] as the backend SMT solver for satisfiability checking and Craig interpolation. We implemented and adopted an original MBP procedure in OCaml, despite being inefficient, because Z3 does not seem to provide APIs for this feature.

It is worth mentioning that, instead of separately implementing different refinement strategies, we used the language feature known as *algebraic effects and handlers*, recently introduced to OCaml 5 [Sivaramakrishnan et al. 2021], so that we can modularly implement refinement strategies based on various control structures including **return**, **yield**, and more.[15] We also tried to keep other parts of the implementation as *pure functional* as possible, by avoiding destructive updates and non-determinism (except for the one exhibited by Z3). We believe that the inductive, modular, and functional implementation makes it easy to understand, reason about, and extend MuCyc.

7.2 Experiments

We now report on the comparative evaluation with various configurations of MuCyc and existing CHC solvers Spacer [Komuravelli et al. 2016], Golem,[16] and Eldarica [Hojjat and Rümmer 2018]. In the experiments, we used a version of Spacer that is bundled with Z3 (version 4.12.6)[17] and the configuration of Golem used in the LIA-nonlin category of CHC-COMP'23 (where a verification engine that implements a Spacer-like algorithm is adopted) and the configuration of Eldarica used in CHC-COMP'22. For MuCyc, we write **Ret(***b, cex***)** and **Yld(***b, cex***)** to denote its configurations based on **return** and **yield**, respectively. The boolean parameter b of **Ret** is either **T** or **F** and represents whether the counterexample accumulation (see Line 11 in Algorithm 5) is enabled. The boolean parameter b of **Yld** represents whether the query weakening via Craig interpolation (see Lines 21 and 23 in Algorithm 6) is enabled. The parameter *cex* represents the counterexample enumeration method used. MuCyc supports **QE**, **MBP(***n***)**, and **Model**. The experiment results using **QE** are omitted in the paper because it significantly degraded the performance. The parameter

[13] Available from https://github.com/hiroshi-unno/coar.

[14] https://chc-comp.github.io/

[15] Our modular implementation allows us to switch **yield** and **return** even during verification depending on, for example, the level of the current trace and the number of queries occurred so far at the level, though an evaluation of the advanced capability is left as a future work.

[16] https://verify.inf.usi.ch/golem

[17] The version is different from GSpacer [Krishnan et al. 2020] that further extends Spacer with various optimization techniques. It is an interesting future direction to incorporate them to MuCyc and perform an experimental comparison.

Table 1. Experimental Results

configuration	sat	unsat	configuration	sat	unsat	configuration	sat	unsat
Ret(F, Model)	605	486	**Yld(F, Model)**	588	479	**Ind(Ret(F, MBP(0)))**	909	519
Ret(T, Model)	604	484	**Yld(T, Model)**	599	481	**Cex(Ret(F, MBP(0)))**	788	524
Ret(F, MBP(0))	**795**	521	**Yld(F, MBP(0))**	729	521	**Que(Ret(F, MBP(0)))**	762	506
Ret(T, MBP(0))	770	525	**Yld(T, MBP(0))**	772	525	**Mon(Ret(F, MBP(0)))**	796	518
Ret(F, MBP(1))	787	519	**Yld(F, MBP(1))**	723	519	**Ind(Yld(T, MBP(1)))**	**915**	526
Ret(T, MBP(1))	770	**526**	**Yld(T, MBP(1))**	**786**	525	**Cex(Yld(T, MBP(1)))**	773	**529**
Ret(F, MBP(2))	676	495	**Yld(F, MBP(2))**	729	520	**Que(Yld(T, MBP(1)))**	669	280
Ret(T, MBP(2))	753	525	**Yld(T, MBP(2))**	775	**526**	**Mon(Yld(T, MBP(1)))**	778	518

$n \in \{0, 1, 2\}$ of **MBP(n)** represents whether the saved frame and query are used (see Lines 7 and 9 in Algorithm 5 and Lines 6, 8, 12, and 14 in Algorithm 6) for RC ($n = 1, 2$) or not ($n = 0$). The difference between $n = 1$ and $n = 2$ is that the former uses more recent information while RC holds by appropriately updating $\varphi_{L,0}$ (see the second paragraph of Remark 15).

MuCyc also supports optimizations discussed in Section 5.3: induction **Ind(*config*)**, counterexample sharing **Cex(*config*)**, query reuse **Que(*config*)**, and monotone trace **Mon(*config*)**. MuCyc also implements predicate sharing but does not support disabling it, and hence no corresponding notation. For example, the configuration of MuCyc closest to Spacer is described as **Ind(Cex(F, Que(Mon(Ret(F, MBP(0)))))).**[18] The configuration closest to GPDR [Hoder and Bjørner 2012] is **Ind(Cex(F, Mon(Ret(F, Model))))**. Besides **Yld** and **Ret**, MuCyc supports a **Solve** configuration that employs an existing method for solving non-linear CHCs [Unno and Kobayashi 2009]. This method aligns with Algorithm 2, but with the Refine step replaced by a recursion-free CHC solver that disregards the current trace Φ. Specifically, **Solve** iteratively expands the given CHCs, solves the result by invoking Spacer as a recursion-free CHC solver, and checks whether the solution is inductive. We use this as a baseline in the experiments.

Table 1 summarizes the number of solved SAT and UNSAT instances. We here used a benchmark set consisting of 1,972 CHCs satisfiability problem instances obtained from the benchmark sets of the LIA-lin and LIA-nonlin categories of CHC-COMP from 2018 to 2022: we here implemented and applied a CHC preprocessor to them *a priori* and filtered out easy instances that were solved by just preprocessing. Our intention here is to weaken the effect on the experiment results of different preprocessors implemented in the existing CHC solvers and focus on the strengths and weaknesses of their main algorithms. Our preprocessor repeatedly applies the resolution rule to eliminate redundant predicate variables and apply an existing algorithm for reducing unnecessary arguments of predicate variables [Leuschel and Sørensen 1997]. All the experiments were conducted on StarExec (CentOS Linux release 7.7.1908, Intel(R) Xeon(R) CPU E5-2609 @ 2.40GHz with 27 GiB RAM) with 600 seconds time limit.

7.2.1 Model vs. MBP(0) vs. MBP(1) vs. MBP(2).

By comparing **Ret(F, MBP(0))** (corresponding to Spacer without optimizations) with **Ret(F, Model)** (GPDR without optimizations), we can see that the use of MBP significantly improves the numbers. Recall that the use of the saved query and frame in the Spacer algorithm (indicated by **Ret(F, MBP(2))**) loses the progress property (and causes an infinite loop without refinement) and hence substantially reduces the performance. By contrast, the configuration **Ret(T, MBP(2))** satisfies the progress property again and is even RC. By comparing **Ret(T, MBP(1))** and **Ret(T, MBP(2))** that are both RC, we can see that the use of

[18]Note here that implementation details of the four optimizations in MuCyc can be different from those of Spacer since they are only exposed as non-deterministically applied rules in the papers of Spacer.

more recent information contributes the performance. By comparing **Ret(T, MBP(1))** and **Ret(F, MBP(1))**, we can also see that the counterexample accumulation that is necessary for RC affects negatively to the number of solved SAT instances but positively to that of UNSAT. It is worth noting that the non-RC configuration **Ret(F, MBP(0))** that uses the latest information outperformed the best RC configuration **Ret(T, MBP(1))** with respect to the number of solved SAT instances.

However, in the setting of the **yield**-based strategies, the experimental results showed a different trend, with **Yld(T, MBP(1))**, which utilizes the most recent information while RC holds, outperforming the other configurations.

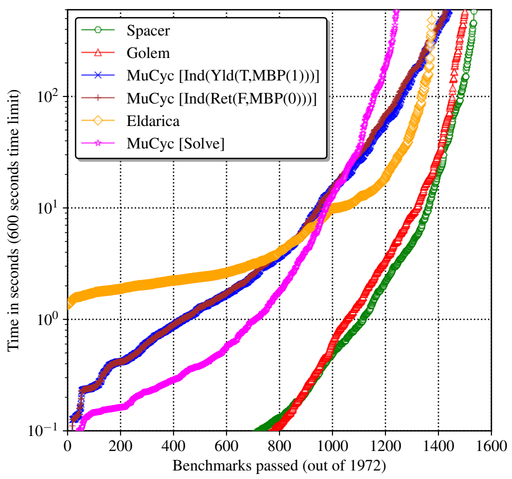
7.2.2 Query weakening. By comparing the two variants **Yld(T, MBP(n))** and **Yld(F, MBP(n))**, which are RC, we can see that enabling query weakening via Craig interpolation improves the performance of **yield**-based strategies.

7.2.3 Optimizations. We now discuss the effect of the optimizations: induction, counterexample sharing, query reuse, and monotone trace. Thanks to the modular and functional implementation of MuCyc, we were able to easily implement the optimizations. However, we found it non-trivial to obtain performance improvements with them, as discussed below. We compared **Ret(F, MBP(0))** and **Yld(T, MBP(1))** with their optimized versions. The induction rule largely contributed to the performance. The sharing of counterexamples also had a positive effect on the number of solved UNSAT instances, though it had a negative effect on that of SAT instances. We believe that the reduction in SAT instances is due to additional overhead, and it seems that reducing this overhead through tuning could make it a useful optimization.

By contrast, query reuse and monotone trace downgraded the performance. In particular, it is surprising that monotone trace reduced the performance since many existing PDR-based verifiers employ monotone trace. A possible reason why query reuse was not effective is that our implementation that follows the explanation in Section 5.3 is different from that of Spacer [Komuravelli et al. 2015] and could result in too many extra queries, the cost of which is not compensated by the gains.

7.2.4 MuCyc vs. State-of-The-Art CHC Solvers. Finally, we compare MuCyc with our RC configuration **Ind(Yld(T, MBP(1)))** with the state-of-the-art CHC solvers as well as the configuration **Ind(Ret(F, MBP(0)))** that is close to Spacer and the baseline **Solve** of MuCyc. The cactus plot shown in Figure 2 plots the number of solved instances (x-axis) against the time taken to solve the instances (y-axis), non-cumulatively, comparing the solvers. The baseline configuration **Solve** solved 738 SAT and 504 UNSAT, Eldarica solved 893 SAT and 482 UNSAT, Golem solved 971 SAT and 529 UNSAT, and Spacer solved 987 SAT and 547 UNSAT instances. The results show that the mature tool Spacer outperforms the others. Despite being new and not yet mature, MuCyc is already as competitive as Eldarica and significantly outperforms the baseline configuration **Solve**. Note also that **Ind(Yld(T, MBP(1)))** slightly outperformed **Ind(Ret(F, MBP(0)))**, despite the observed fact that **Yld(T, MBP(1))** and **Ret(F, MBP(0))** were incomparable in the absence of optimizations. We believe the gap between the performance of MuCyc and Golem, which is also based on the Spacer algorithm, could be partly explained as

Fig. 2. MuCyc vs. existing CHC solvers.



follows: (1) MuCyc does not fully utilize caching and incremental SMT, instead repeatedly calling Z3 with similar, related queries; (2) The implementation language of MuCyc employs garbage collection and is generally not as efficient as C++ used to implement Golem and Spacer. Note that the curves of the MuCyc configurations are more gradual compared to the others. We thus believe that by increasing the time limit or addressing the two aforementioned issues to speed up the process, MuCyc could solve an even greater number of problems.

8 RELATED WORK

PDR has been applied to verification problems that go beyond CHCs such as symbolic model checking of Markov decision processes [Batz et al. 2020], verification of relational properties [Shemer et al. 2019], and verification of distributed protocols [Goel and Sakallah 2021; Karbyshev et al. 2017]. It would be interesting to extend our inductive approach to their PDR algorithms.

There have been proposed extensions of the class of CHCs: existentially quantified CHCs [Beyene et al. 2013], universally quantified CHCs [Bjørner et al. 2013], pCSP that extends CHCs with head-disjunctions [Satake et al. 2020], pfwCSP that extends pCSP with well-foundedness and functional constraints [Unno et al. 2021], and higher-order CHCs [Burn et al. 2018]. Also, note that the class of CHCs can be seen as a fragment of first-order fixpoint logic without greatest fixpoints. Recently, several authors have applied fixpoint logics [Kobayashi et al. 2019, 2018; Unno et al. 2023] as a generalization of the class of CHCs for formal verification. However, to our knowledge, there exists no PDR algorithm proposed for the extended classes. It is interesting to investigate whether our inductive approach can be applied to systematically derive PDR algorithms for them.

As for the CHC solving in the standard setting, Krishnan et al. [2020] and Blicha et al. [2022] proposed ideas to improve the performance of Spacer. A typical challenge with Spacer that these papers address is that to find a counterexample of length n, one has to deal with the n-fold expansion of the input CHC, which tends to be huge. Krishnan et al. [2020] proposed several heuristics, mainly a kind of inspection and manipulation of logical formulas. The idea of Blicha et al. [2022] can be explained in terms of higher-order CHCs [Burn et al. 2018; Kobayashi et al. 2018]. Blicha et al. [2022] translate an input CHC system in the standard sense, say a *first-order* CHC, into a second-order CHC. By making good use of the expressive power of second-order CHC, the number of development steps required to find a counterexample is logarithmically shortened.

The above-mentioned work studied PDR mainly from a practical perspective. Recently, analysis of PDR from a theoretical perspective was given by Feldman et al. [2019, 2022] and Feldman and Shoham [2022]. They analyzed PDR using logical and/or computational theoretic ideas.

Model-based projection, which is a key technology for Spacer, is also applied to decide quantified first-order logic formulas by Bjorner and Janota [2015] and Farzan and Kincaid [2016].

9 CONCLUSION

We have developed an inductive description of Spacer and discussed the behavior of Spacer variants in the literature based on an intuition from our inductive description. We have implemented and evaluated our procedures. Our experiment has confirmed that some tricks to retain refutational completeness have a practical impact.

An interesting direction for future work is to find a better naïve algorithm. Our procedures are derived from the naïve algorithm (Algorithm 3), a procedure with quantified formulas, by removing the quantifications using model-based projection and making the procedure as lazy as possible. A different choice of the naïve algorithm results in a different procedure. Examples of particular choices include the iterative deepening version, whereas Algorithm 3 is the depth-first version.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their insightful comments. Additionally, we would like to express our appreciation to the anonymous reviewers who provided feedback on the earlier version submitted to POPL. In particular, a technical comment from a POPL reviewer significantly affected the structure of our paper. We thank the StarExec initiative for providing the computational resources necessary for our experiments in this paper. This work was supported by JSPS KAKENHI Grant Numbers JP20H05703, JP22H03564, JP20H04162, and JP19H04084.

REFERENCES

Kevin Batz, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2020. PrIC3: Property Directed Reachability for MDPs. In *Lecture Notes in Computer Science*. Springer International Publishing, 512–538. https://doi.org/10.1007/978-3-030-53291-8_27

Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. 2013. Solving Existentially Quantified Horn Clauses. In *CAV '13 (LNCS)*, Vol. 8044. Springer, 869–882.

Nikolaj Bjorner and Mikolas Janota. 2015. Playing with Quantified Satisfaction. In *LPAR '15 (EPiC Series in Computing)*, Vol. 35. EasyChair, 15–27.

Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In *SAS '13 (LNCS)*, Vol. 7935. Springer, 105–125.

Martin Blicha, Grigory Fedyukovich, Antti E. J. Hyvärinen, and Natasha Sharygina. 2022. Transition Power Abstractions for Deep Counterexample Detection. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science)*, Dana Fisman and Grigore Rosu (Eds.), Vol. 13243. Springer, 524–542. https://doi.org/10.1007/978-3-030-99524-9_29

Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

Toby Cathcart Burn, C.-H. Luke Ong, and Steven J. Ramsay. 2018. Higher-order constrained horn clauses for verification. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 11:1–11:28.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS '08* (Budapest, Hungary, March29 – April 6) *(LNCS)*, Vol. 4963. Springer, 337–340.

Niklas Een, Alan Mishchenko, and Robert Brayton. 2011. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*. 125–134.

Azadeh Farzan and Zachary Kincaid. 2016. Linear Arithmetic Satisfiability via Strategy Improvement. In *IJCAI '16*. AAAI Press, 735–743.

Yotam M. Y. Feldman, Neil Immerman, Mooly Sagiv, and Sharon Shoham. 2019. Complexity and Information in Invariant Inference. 4, POPL, Article 5 (dec 2019), 29 pages.

Yotam M. Y. Feldman, Mooly Sagiv, Sharon Shoham, and James R. Wilcox. 2022. Property-Directed Reachability as Abstract Interpretation in the Monotone Theory. 6, POPL, Article 15 (jan 2022), 31 pages.

Yotam M. Y. Feldman and Sharon Shoham. 2022. Invariant Inference with Provable Complexity from the Monotone Theory. Springer, Cham, 201–226.

Aman Goel and Karem A. Sakallah. 2021. On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In *NFM '21*. 131–150.

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2010. Nested interpolants. *ACM SIGPLAN Notices* 45, 1 (jan 2010), 471–482. https://doi.org/10.1145/1707801.1706353

Kryštof Hoder and Nikolaj Bjørner. 2012. Generalized Property Directed Reachability. In *Theory and Applications of Satisfiability Testing – SAT 2012*. Springer Berlin Heidelberg, 157–171. https://doi.org/10.1007/978-3-642-31612-8_13

Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura. 2011. μZ: An Efficient Engine for Fixed Points with Constraints. In *CAV '11* (Snowbird, UT) *(LNCS)*, Vol. 6806. Springer, 457–462.

Hossein Hojjat and Philipp Rümmer. 2018. The Eldarica Horn Solver. In *FMCAD '18*. IEEE, 1–7.

Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. 2017. Property-Directed Inference of Universal Invariants or Proving Their Absence. *J. ACM* 64, 1, Article 7 (mar 2017), 33 pages.

Naoki Kobayashi, Takeshi Nishikawa, Atsushi Igarashi, and Hiroshi Unno. 2019. Temporal Verification of Programs via First-Order Fixpoint Logic. In *SAS '19*. Springer, 413–436.

Naoki Kobayashi, Takeshi Tsukada, and Keiichi Watanabe. 2018. Higher-Order Program Verification via HFL Model Checking. In *ESOP '18*. Springer, 711–738.

Anvesh Komuravelli, Nikolaj Bjørner, Arie Gurfinkel, and Kenneth L. Mcmillan. 2015. Compositional verification of procedural programs using horn clauses over integers and arrays. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. https://doi.org/10.1109/fmcad.2015.7542257

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *Computer Aided Verification (CAV)*. Springer International Publishing, 17–34. https://doi.org/10.1007/978-3-319-08867-9_2

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design* 48, 3 (jun 2016), 175–205. https://doi.org/10.1007/s10703-016-0249-4

Hari Govind Vediramana Krishnan, Yuting Chen, Sharon Shoham, and Arie Gurfinkel. 2020. Global Guidance for Local Generalization in Model Checking. In *CAV '20 (LNCS)*, Vol. 12225. Springer, 101–125.

Michael Leuschel and Morten Heine Sørensen. 1997. Redundant argument filtering of logic programs. In *LOPSTR '97*. Springer, 83–103.

K. L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification (CAV 2003) (Lecture Notes in Computer Science)*, Vol. 2725. Springer Berlin Heidelberg, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Computer Aided Verification*. Springer Berlin Heidelberg, 123–136. https://doi.org/10.1007/11817963_14

Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid Types. In *PLDI '08*. ACM, 159–169.

Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. 2014. Classifying and Solving Horn Clauses for Verification. In *Verified Software: Theories, Tools, Experiments*. Springer Berlin Heidelberg, 1–21. https://doi.org/10.1007/978-3-642-54108-7_1

Yuki Satake, Hiroshi Unno, and Hinata Yanagi. 2020. Probabilistic Inference for Predicate Constraint Satisfaction. *AAAI '20* 34, 02 (Apr. 2020), 1644–1651.

Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2019. Property Directed Self Composition. In *CAV '19 (LNCS)*, Vol. 11561. Springer, 161–179.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *PLDI '21* (Virtual, Canada) *(PLDI 2021)*. ACM, 206–221.

Takeshi Tsukada and Hiroshi Unno. 2021. Software Model-Checking as Cyclic-Proof Search. https://doi.org/10.48550/ARXIV.2111.05617

Takeshi Tsukada and Hiroshi Unno. 2022. Software model-checking as cyclic-proof search. *Proceedings of the ACM on Programming Languages* 6, POPL (jan 2022), 1–29. https://doi.org/10.1145/3498725

Hiroshi Unno and Naoki Kobayashi. 2009. Dependent Type Inference with Interpolants. In *PPDP '09*. ACM, 277–288.

Hiroshi Unno, Tachio Terauchi, Yu Gu, and Eric Koskinen. 2023. Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification. 7, POPL, Article 72 (jan 2023), 30 pages.

Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-Based Relational Verification. In *CAV '21*. Springer, 742–766.

Received 2023-11-16; accepted 2024-03-31