Synthesizing History and Prophecy Variables for Symbolic Model Checking

Cole Vick and Kenneth L. McMillan

UT Austin

Abstract. Introduction of history and prophecy variables can allow a proof to be expressed in a weaker logic or a more localized form. This fact has been used, for example, to allow purely propositional, quantifier-free, invariant generators to produce proofs for parameterized systems requiring universal quantification in the inductive invariant. However, automatic synthesis of history and prophecy remains an open problem. We introduce counterexample-guided heuristics for this purpose based on property-driven refutation of counterexamples and Craig interpolation. The approach is evaluated on a set of benchmarks based on array manipulating programs with multiple loops.

1 Introduction

The addition of auxiliary variables is a common tactic in program verification. These can be *history variables* that record some information about past program state, or *prophecy variables* that predict some aspect of future program state. In some cases auxiliary variables may be necessary for (relative) completeness of a proof system. For example, in the Owicki/Gries system, history variables are necessary [27], while prophecy is needed to prove program refinement using refinement maps [1]. In other cases, auxiliary variables are used to *simplify* a proof, allowing it to be constructed in a less expressive language. For example, history and prophecy variables are used in [25] in a scheme that reduces the proofs of parameterized protocols to a propositional invariant generation problem. More subtly, we can think of automated compositional proof using grammatical inference [5] as inference of a history variable (the state of an automaton) that reduces the inductive invariant to a conjunction of local invariants.

Eliminating quantifiers with auxiliary variables A particularly important application of auxiliary variables is to eliminate the need for quantifiers in an inductive invariant (or to eliminate quantifier alternation). This can allow us to apply a model checker to the invariant generation problem, even if the model checker cannot handle quantifiers. The key question is how to introduce auxiliary variables in an *automated* way. This might provide an important advantage, as the direct synthesis of quantified inductive invariants has proved challenging, even for problems of modest size.

(a) Original program

```
for (int $i = 0$; $i < N$; $i$++) {
    $a[i] = i$;
}
for (int $i = 0$; $i < N$; $i$++) {
    $x = f(i)$;
    $b[i] = a[x]$;
}
assert $\forall\ 0 \leq j < N.\ b[j] \geq 0$;
```

(b) Instrumented program

```
for (int $i = 0$; $i < N$; $i$++) {
    $a[i] = i$;
}
for (int $i = 0$; $i < N$; $i$++) {
    $x = f(i)$;
    $b[i] = a[x]$;
    $\eta_2 = x\ ?\ i = \pi_1\ :\ \eta_2$;
}
assert $0 \leq \pi_1 < N \wedge \pi_2 = \eta_2 \rightarrow b[\pi_1] \geq 0$;
```

Fig. 1: Array scattering program, original and instrumented. The function $f(i)$ returns a non-deterministically chosen index of a.

As an example, consider the program of Fig. 1(a), a fragment in a notional C-like language. In this program, the array a is first filled with non-negative values. Then the array b is filled with values from a chosen by an unknown function f. We then assert that all elements of b are non-negative. In the proof of this program, we need universal quantifiers over array indices. For example, a suitable invariant between the loops would be $\forall\ 0 \leq j < n.\ a[j] \geq 0$. We can't express the invariant in the array theory without quantifiers, because a quantifier-free invariant can only reference a bounded number of elements of the arrays, while the property depends on all N elements and N is arbitrarily large.

On the other hand, by adding auxiliary variables we can reduce the dependence of the property to only a single element of a and b. Fig. 1(b) shows the program instrumented in this way. In this program π_1 and π_2 are prophecy variables and η_2 is a history variable. Variable π_1 predicts an index of b for which the assertion fails. It replaces the quantified variable j in the assertion (using a process called Herbrandization). Prophecy variable π_2 predicts the index of a that is assigned to this element of b. The history variable η_2 records the index of this element of a so the prophecy can be validated when the assertion fails. A suitable quantifier-free inductive invariant of the second loop in this program is $0 \leq i \leq n$ and $0 \leq \pi_2 < n \rightarrow a[\pi_2] \geq 0$ and $\pi_2 = \eta_2 \wedge \pi_1 < i \rightarrow b[\pi_1] \geq 0$. In effect, the prophecy variables have replaced the quantifiers, allowing us to write the invariant in a way that refers to only one element each of the arrays a and b.

History variables from counterexamples Apart from [5], little attention has been paid to the problem of *automated* generation of auxiliary variables. Recently, Mann, *et al.*, have considered automated generation of auxiliary variables for array-manipulating programs such as Fig. 1(a) [20]. This method allows the programs to be proved by an invariant generator that does not handle quantifiers, and in fact does not even implement the theory of arrays.

The method works by abstracting away the array theory, effectively replacing the array operations by uninterpreted functions. It then uses a CEGAR approach to refine the abstraction by adding ground instances of the array theory axioms to the program's transition relation. A false counterexample is one that violates an array axiom instance. If this violation spans only a single transition in the counterexample, the counterexample is easily eliminated by adding a single instance of the axiom to the transition relation. The trick is to handle axiom instances that span multiple transitions. In this case, we replace symbols in the violated axiom with prophecy variables until it spans only a single transition. As an example, this instance of an array theory axiom results in prophecy variable π_2:

$$i^2 = i^0 \rightarrow \mathtt{select}(\mathtt{store}(a^0, i^0, i^0), i^2) = i^0.$$

Here the superscripts represent states in a counterexample where each loop is executed once, i.e. $N = 1$. That is, if in the second loop (at state 2) we read the same index of a that was written in the first loop (at state 0) then we obtain the value written. To make this instance span a single transition, we replace i^2 by a prophecy variable π_2 that predicts its value. Thus we obtain:

$$\pi_2 = i^0 \rightarrow \mathtt{select}(\mathtt{store}(a^0, i^0, i^0), \pi_2) = i^0.$$

The method then synthesizes a history variable η_2 that captures the value of i^2. We then implement the prophecy variable π_2 by conditioning the assertion on $\pi_2 = \eta_2$.

Unfortunately, the method of [20] does not synthesize the *conditional* history variable shown in Fig. 1(b). It generates instead a variable that stores the value of i unconditionally, effectively delaying it by one iteration. This eliminates counterexamples with $N = 1$, resulting in a new counterexample with $N = 2$, giving a new history variable with a delay of two steps, and so on to infinity. In this paper, we introduce a method of synthesizing a history variable that stores a value *conditionally*, as shown in the figure. This allows the method to converge in cases of unbounded loops. Our method is based on searching for a *property-directed refutation* of the counterexample, using the array axioms. From this proof, we extract both the relevant axiom instance and the capture condition of the history variable ($i = \pi_1$ in the Fig. 1(b)).

Capture conditions from invariants Although this approach is effective in some instances, we encounter many problems for which appropriate capture condition involves reasoning beyond the array theory. Consider the program in Fig. 2(a) that sums up an array of non-negative integers, asserting that the sum is non-negative. The second loop maintains an invariant that the sum j is non-negative. To prove this program with a quantifier-free invariant, we need to capture the loop index x at a point when the invariant goes from true to false, which implies that $a[i]$ is negative. One suitable capture condition is shown in the instrumented program of Fig. 2(b). The history variable captures i at the last moment when the invariant is true. We can prove the instrumented program using the following inductive invariant for the second loop: $0 \leq i \leq N$ and $a[\pi_1] >= 0$ and $\pi_1 = \eta_1 \wedge \pi_1 < i \rightarrow j \geq 0$.

(a) Original program

```
for (int $i = 0$; $i < N$; $i$++) {
    $a[i] = i$;
}
$j = 0$;
for (int $i = 0$; $i < N$; $i$++) {
    $j = j + a[i]$;
}
assert $j \geq 0$;
```

(b) Instrumented program

```
for (int $i = 0$; $i < N$; $i$++) {
    $a[i] = i$;
}
$j = 0$;
for (int $i = 0$; $i < N$; $i$++) {
    $j = j + a[i]$;
    $\eta_1 = i \ ?\ j \geq 0 : \eta_1$;
}
assert $\pi_1 = \eta_1 \rightarrow j \geq 0$;
```

Fig. 2: Array summing program, original and instrumented.

To discover this capture condition, we need a way to synthesize relevant invariants of the program (though not necessarily an inductive invariant). We will present a way to do this using sequence interpolants [14].

Contributions The primary contributions of this paper are *(1)* a method of inferring *conditional history variables* from counterexamples, based on *property-directed refutations* of abstract counterexamples, *(2)* a method of inferring capture conditions from sequence interpolants and *(3)* a benchmark evaluation, showing that this approach is substantially more effective than the unconditional approach of [20] and that it out-performs state-of-the-art CHC solvers that produce quantified invariants on small-scale benchmarks.

Limitations We abstract only the array theory. Our implementation handles only prenex-universal assertions. We do not consider recursive programs. We do not consider the question of scalability to large programs.

2 Related work

We can divide the related work into two categories. Methods in the first category differ from the current approach in that they construct and verify a quantified inductive invariant [28, 10, 15, 19, 16, 11]. Some of these restrict the verification conditions to decidable fragments. These include the Invisible Invariants method [28] which relies on a small model theorem, and UPDR [15] which uses the decidable EPR fragment. Other methods (*e.g,* [19]) rely on incomplete heuristic quantifier instantiation.

In the second category, we have methods that, like the current method, transform the problem in some way, allowing verification without the use of quantifiers and reusing an existing invariant generator or CHC solver. The most common approach is to transform the problem into a *non-linear* CHC satisfiability problem [2, 26, 12]. These methods differ from the present method in two ways: they require a non-linear CHC solver and they are pre-processing techniques (eager abstraction) while our method is a CEGAR (lazy abstraction) approach.

Many works use manually-introduced auxiliary variables to eliminate the need for quantifiers [22, 21, 23, 3, 25]. We are aware of only one approach, however, that automates this process, that of Mann, *et al.* [20], which is the starting point for this work. That approach is strongly limited by the restriction to unconditional history variables, a limitation that we address here.

3 Preliminaries

Logic Let $FO_=(\mathbb{S}, \mathbb{V})$ be standard sorted first-order logic with equality, where $\mathbb{S}$ is a collection of first-order sorts and $\mathbb{V}$ is a vocabulary of sorted non-logical symbols. We assume a special sort $\mathbb{B} \in \mathbb{S}$ that is the sort of propositions. Each symbol $f{:}S \in \mathbb{V}$ has an associated sort S of the form $D_1 \times \cdots \times D_n \to R$, where $D_i, R \in \mathbb{S}$ and $n \geq 0$ is the *arity* of the symbol. If $n = 0$, we say $f{:}S$ is a *constant*, and if $R = \mathbb{B}$ it is a *relation*. We write $\text{vocab}(t)$ for the set of non-logical symbols occurring in term t.

Given a set of sorts $\mathbb{S}$, a *universe* U maps each sort in $\mathbb{S}$ to a non-empty set (with $U(\mathbb{B}) = \{\mathbf{tt}, \mathbf{ff}\}$). An *interpretation* of a vocabulary $\Sigma \subseteq \mathbb{V}$ over universe U maps each symbol $f{:}D_1 \times \cdots \times D_n \to R$ in Σ to a function in $U(D_1) \times \cdots \times U(D_n) \to U(R)$. A Σ-structure is a pair $\mathcal{M} = (U, \mathcal{I})$ where U is a universe and $\mathcal{I}$ is an interpretation of Σ over U. The structure is a *model* of a proposition ϕ in $FO_=(\mathbb{S}, \mathbb{V})$ if ϕ evaluates to $\mathbf{tt}$ under $\mathcal{I}$ according to the standard semantics of first-order logic. In this case, we write $\mathcal{M} \models \phi$. Given an interpretation $\mathcal{J}$ with domain disjoint from $\mathcal{I}$, we write $\mathcal{M}, \mathcal{J}$ to abbreviate the structure $(U, \mathcal{I} \cup \mathcal{J})$.

Vocabularies We divide $\mathbb{V}$ into several disjoint classes. The *background symbols* $\mathbb{V}_B$ are used to represent the signature of a background theory such as linear integer arithmetic or the theory of arrays. A *background theory* (theory in the sequel) is a collection of formulas over the background symbols $\mathbb{V}_B$. The *state symbols* $\mathbb{V}_S$ are used to represent the state of a system. A *state formula* is a formula over $\mathbb{V}_B \cup \mathbb{V}_S$. The primed symbols $\mathbb{V}'_S$ contain, for each state symbol s, a distinct symbol s'. For any term t in the logic, we denote by t' the result of replacing every state symbol occurring in t with the corresponding primed symbol s'. A *transition formula* is a formula over $\mathbb{V}_B \cup \mathbb{V}_S \cup \mathbb{V}'_S$. We write $\text{unprime}(t)$ for the result of replacing every primed symbol s' in t with s. We also distinguish a sequence of disjoint vocabularies $\mathbb{V}^i_S$, for $i = 0, 1, \ldots$, such that $\mathbb{V}^i_S$ contains a distinct symbol denoted s^i for every state symbol s. For term t, we write t^i for the result of replacing every state symbol s with s^i and s' with x^{i+1} in t. We write $\text{unindex}_i(t)$ for the result of replacing s^i with s and s^{i+1} with s' in t, for every state symbol s. For any $\Sigma \subseteq \mathbb{V}$, we write Σ_B for $\Sigma \cap \mathbb{V}_B$, Σ_S for $\Sigma \cap \mathbb{V}_S$, Σ'_S for $\Sigma \cap \mathbb{V}'_S$ and Σ^i_S for $\Sigma \cap \mathbb{V}^i_S$.

Transition systems A Σ-trace, for vocabulary Σ, is a pair $\langle n, \mathcal{M} \rangle$ where $n \geq 0$ and $\mathcal{M}$ is a structure over $\Sigma_B \cup \Sigma^0 \cup \cdots \cup \Sigma^n$. Given a Σ-trace $\tau = \langle n, \mathcal{M} \rangle$ and a vocabulary $\hat{\Sigma} \subseteq \Sigma$, the *projection* of τ onto $\hat{\Sigma}$, denoted $\tau \downarrow \hat{\Sigma}$ is the $\hat{\Sigma}$-trace $\tau = \langle n, \mathcal{M} \downarrow \hat{\Sigma} \rangle$. We say τ is an *extension* of $\hat{\tau}$.

A *transition system* is a pair $M = \langle \Sigma, I, T \rangle$ where $\Sigma \subset \mathbb{V}$ is a vocabulary, I is a state formula over Σ and T is a transition formula over Σ. For theory $\mathcal{T}$, a $\mathcal{T}$*-trace* of M is a Σ-trace $\langle n, \mathcal{M} \rangle$ such that:

- $\mathcal{M} \models \mathcal{T}$ and
- $\mathcal{M} \models I^0$, and
- for $0 \leq i < N$, $\mathcal{M} \models T^i$.

A *safety problem* (problem in the sequel) $\Pi = \langle \Sigma, I, T, \phi \rangle$ is a system $M = \langle \Sigma, I, T \rangle$ equipped with a *safety condition* ϕ over Σ_S. A $\mathcal{T}$-counterexample to Π is a $\mathcal{T}$-trace $\langle n, \mathcal{M} \rangle$ of M such that $\mathcal{M} \not\models \phi^n$. A problem is $\mathcal{T}$*-valid* if it has no $\mathcal{T}$-counterexamples.

The *bounded model checking unfolding* $\text{BMC}(\Pi, n)$ is the sequence of formulas $I^0, T^0, \ldots, T^{n-1}, \neg\phi^n$. We note that the $\mathcal{T}$-counterexamples of Π are exactly the $\mathcal{T}$-models of $\text{BMC}(\Pi)$.

In the sequel, we assume without loss of generality that the safety conditions of all problems are prenex-existential. This can be achieved by Herbrandization, the dual process of Skolemization. In particular, this replaces leading universal quantifiers with fresh background constants (as π_1 replaces $\forall j$ in Fig. 1).

4 Theory abstraction and refinement

As in [20], our procedure begins by abstracting away the array theory, effectively treating the array operators as uninterpreted functions. In the sequel, we fix a background theory $\mathcal{T}$, an abstract theory $\mathcal{T}_A$ and a refinement theory $\mathcal{T}_R$ such that $\mathcal{T} = \mathcal{T}_A \cup \mathcal{T}_R$. In practice, $\mathcal{T}_A$ is EUFLIA (uninterpreted functions with equality and linear integer arithmetic) while $\mathcal{T}_R$ is the array theory. The following theorem states that model checking with $\mathcal{T}_A$ is sound:

Theorem 1. *If problem Π is $\mathcal{T}_A$-valid, then Π is $\mathcal{T}$-valid.*

Proof sketch A $\mathcal{T}$-counterexample of Π is also a $\mathcal{T}_A$-counterexample of Π since $\mathcal{T}$ implies $\mathcal{T}_A$. $\square$

4.1 Refinement

After theory abstraction we may obtain false counterexamples. A refinement preserves the set of concrete counterexamples, and hence is sound. However, it may eliminate abstract counterexamples. We employ three classes of refinements: theory refinements, prophecy refinements and history refinements.

Formally, a *refinement* is a partial function $\mathcal{R}$ from problems to problems such that, if problem Π is in $\text{dom}(\mathcal{R})$, then for every $\mathcal{T}$-counterexample τ of Π, there exists a $\mathcal{T}$-counterexample $\hat{\tau}$ of $\mathcal{R}(\Pi)$ that is an extension of τ.

The *composition* of two partial functions $f \circ g$ is the partial function such that $(f \circ g)(x) = f(g(x))$ when $x \in \text{dom}(g)$ and $g(x) \in \text{dom}(f)$. We say $f \sqsubseteq g$

if for all $x \in \mathrm{dom}(f)$ we have $x \in \mathrm{dom}(g)$ and $g(x) = f(x)$. We write $\mathbf{1}$ for the identify function over any domain.

Lemma 1. *Refinements are closed under composition.*

A refinement $\mathcal{R}$ is said to *kill* $\mathcal{T}_A$-counterexample τ of problem Π if no extension of τ is a $\mathcal{T}_A$-counterexample of $\mathcal{R}(\Pi)$.

Theory refinement A theory refinement adds some validity of $\mathcal{T}$ to the initial condition or the transition relation. As these formulas are equivalent to $\mathbf{tt}$ modulo $\mathcal{T}$, this leaves the concrete traces unchanged.

Formally, a *theory refinement* is a problem transformer we denote $\mathcal{R} = \textsc{TheoryRef}(\psi_I, \psi_T)$ where ψ_I is a ground state formula and ψ_T is a ground transition formula such that $\mathcal{T} \models \psi_I, \psi_T$. If problem $\Pi = \langle \Sigma, I, T, \phi \rangle$ is such that $\mathrm{vocab}(\psi_I, \psi_T) \subseteq \Sigma$, then $\mathcal{R}(\Pi) = \langle \Sigma, I \wedge \psi_I, T \wedge \psi_T, \phi \rangle$.

Theorem 2. $\textsc{TheoryRef}(\phi_I, \phi_T)$ *is a refinement.*

Prophecy refinement A prophecy refinement introduces a fresh background constant that predicts the value of some expression at the end of a trace. Since it is a background symbol, it is invariant over time.

A *prophecy refinement* is a problem transformer we denote $\mathcal{R} = \textsc{ProphRef}(x, t)$ where x is a background constant and t is a state constant. If $\Pi = \langle \Sigma, I, T, \phi \rangle$ is a problem such that $x \notin \Sigma$, then $\mathcal{R}(\Pi) = \langle \Sigma \cup \{x\}, I, T, x = t \rightarrow \phi \rangle$.

Theorem 3. $\textsc{ProphRef}(x, t)$ *is a refinement.*

Proof sketch Let τ be $\mathcal{T}$-counterexample of Π. Extend τ such that $x = t^n$. This is a $\mathcal{T}$-counterexample of $\mathcal{R}(\Pi)$. $\square$

History Refinement A *history refinement* is a problem transformer we denote $\textsc{HistRef}(x, \psi, t)$ where x is a state constant, t is a state term and ψ is a transition formula. If $\Pi = \langle \Sigma, I, T, \phi \rangle$ is a problem such that $x \notin \Sigma$ and $\mathrm{vocab}(t, \psi) \subseteq \Sigma$, then $\textsc{HistRef}(x, \psi, t)(\Pi) = \langle \Sigma \cup \{x\}, I, T \wedge x' = \mathrm{ite}(\psi, t, x), \phi \rangle$.

N.B. Our class of refinements differs from that of [20] in that the history variable x stores the value of t conditionally, while in the prior work t is delayed unconditionally.

Theorem 4. $\textsc{HistRef}(x, \psi, t)$ *is a refinement.*

Proof sketch Let τ be $\mathcal{T}$-counterexample of Π. Extend τ with a fresh variable x such that $x^{i+1} = \mathrm{ite}(\psi, t, x)$ for $i = 1 \ldots n$. This is a $\mathcal{T}$-counterexample of $\mathcal{R}(\Pi)$. $\square$
We use history refinements to store the value of term t at some given time j until the end of the trace, so that $x^n = t^j$. The storage time i is the last time such that the capture condition ψ holds. With respect to a trace $\tau = \langle n, \mathcal{M} \rangle$, we say that refinement $\textsc{HistRef}(\eta, \psi, x)$ *captures* x^j if $\mathcal{M} \models \phi^j$ and for all $j < i < n$, $\mathcal{M} \not\models \phi^i$.

```
**Function** CEGAR($\varPi$)
    **Input** problem $\varPi$
    **Returns** true if $\varPi$ is $\mathcal{T}$-valid

    **if** $\varPi$ is $\mathcal{T}_A$-valid:
        **return** true
    **let** $\tau$ be a $\mathcal{T}_A$-counterexample to $\varPi$
    **if** $\tau$ is a $\mathcal{T}$-counterexample to $\varPi$:
        **return** false
    **choose** $\mathcal{R}$ **in** REFINEMENTS($\varPi$,$\tau$)
    **return** CEGAR($\mathcal{R}(\varPi)$)
```

Fig. 3: Counterexample-guided refinement

Lemma 2. *If τ is a trace of problem $\varPi$ and refinement $\mathcal{R} = \textsc{HistRef}(\eta, \psi, x)$ captures x^j in τ, then for every extension of $\hat{\tau}$ of τ in $\mathcal{R}(\varPi)$, $\hat{\tau} \models \eta^n = x^j$.*

Proof sketch Let τ be $\mathcal{T}$-counterexample of $\varPi$. Extend τ with a fresh variable x such that $x^{i+1} = \text{ite}(\psi, t, x)$ for $i = 1 \ldots n$. This is a $\mathcal{T}$-counterexample of $\mathcal{R}(\varPi)$. □

5 Counterexample-guided refinement

The algorithms we present are non-deterministic, with non-deterministic choice indicated by the "choose" keyword. Choice could be implemented by backtracking, but in practice we use heuristics that are detailed in Sec. 8.

We introduce refinements lazily with counterexample-guided abstraction refinement (CEGAR). The general refinement loop is shown in Fig. 3. If model checking determines that the property is true modulo the abstract theory, we return true. Else, we obtain an abstract counterexample. If this is also a concrete counterexample, we return false. Else, we call REFINEMENTS, which generates a sequence of refinements that kill the abstract counterexample. We choose a refinement among these (or abort if none are found). Then we apply CEGAR to the refined problem.

Theorem 5 (Soundness). *Assume* REFINEMENTS*($\varPi$,τ) is a set of refinements for all problems $\varPi$ and counterexamples τ to $\varPi$. If* CEGAR*($\varPi$) terminates, the result is true iff $\varPi$ is valid.*

Proof sketch Refinements preserve concrete counterexamples. Thus, if we return true, by Thm. 1 there are none. Moreover, if we return false, τ is a $\mathcal{T}$-counterexample of the original $\varPi$. □

If the refinements returned by REFINEMENTS always kill τ, and if it always returns a refinement, then we say that CEGAR makes *refinement progress* in the sense that each refinement eliminates at least one abstract counterexample. This is a heuristically useful property, but it does not guarantee termination.

5.1 Refinement with local axiom instances

Fig. 4 shows our general algorithm for refinement. It is similar to [20], except in that the history variables are conditional. We are given a ground instance ψ of an axiom in $\mathcal{T}_R$ that is false in abstract counterexample τ and a set Γ of potential history variable conditions. If the counterexample is of length $n = 0$, we kill it by conjoining $\text{unindex}_0(\psi)$ to the initial condition. If the vocabulary of the axiom instance ψ spans a single transition from time i to time $i+1$, we kill the counterexample by conjoining $\text{unindex}_i(\psi)$ to the transition relation. Otherwise, we *localize* ψ to a single transition from time i to time $i+1$ by replacing each symbol x^j for $j \neq i, i+1$ by a prophecy variable that captures its value in the trace. We do this by introducing a history variable η that captures x^j, using a Boolean combination of the predicates in Γ as the condition. We then introduce a prophecy variable π that predicts η at time n.

We now argue correctness of this procedure. Say that formula ϕ is *initial* if $\text{vocab}(\phi) \subseteq \mathbb{V}_B \cup \mathbb{V}_S^0$ and *i-local* for $i \geq 0$ if $\text{vocab}(\phi) \subseteq \mathbb{V}_B \cup \mathbb{V}_S^i \cup \mathbb{V}_S^{i+1}$.

Lemma 3. *If τ is a $\mathcal{T}_A$-counterexample for problem Π and ψ is an initial formula such that $\mathcal{T} \models \psi$ and $\tau \not\models \psi$, then* $\textsc{TheoryRef}(\textit{unindex}_0(\psi), \boldsymbol{tt})$ *kills τ.*

Proof sketch Since $\tau \not\models \text{unindex}_0(\psi)^0$ it follows that no extension of τ can be a $\mathcal{T}_A$-trace of the refinement. $\square$

Lemma 4. *If τ is a $\mathcal{T}_A$-counterexample for problem Π and ψ is an i-local formula such that $\mathcal{T} \models \psi$ and $\tau \not\models \psi$, then* $\textsc{TheoryRef}(\boldsymbol{tt}, \textit{unindex}_i(\psi))$ *kills τ.*

Proof sketch Since $\tau \not\models \text{unindex}_i(\psi)^i$ it follows that no extension of τ can be a $\mathcal{T}_A$-trace of the refinement. $\square$

Theorem 6. *If τ is a $\mathcal{T}_A$-counterexample to problem Π, and $\mathcal{T} \models \psi$ and $\tau \not\models \psi$ then every refinement in* $\textsc{AxiomRefine}(\tau, \psi, \Gamma)$ *kills τ.*

Proof sketch If $n = 0$ we apply Lem. 3. Else, construct a trace $\hat{\tau}$ from τ by extending $\hat{\tau}$ at each iteration of the loop such that $\eta^k = x^j$ for all $i < k \leq n$ and $\pi = x^j$ for all $0 \leq k \leq n$. Invariantly, $\hat{\tau} \not\models \psi$, and by Theorems 3 and 4, $\hat{\tau}$ is a $\mathcal{T}_A$-counterexample to $\mathcal{R}(\Pi)$. By Lem. 4, the returned refinement kills $\hat{\tau}$ and hence τ.

6 Proof-based prophecy heuristic

To use the above algorithm to eliminate a counterexample, we must find a suitable axiom instance violation and predicates from which to construct the capture condition. Heuristically, we wish to find ground instances of $\mathcal{T}_R$ axioms that are violated by the abstract counterexample and are *causally related* to the property failure. That is, we seek axiom violations that do not depend on accidental aspects of the counterexample that are unrelated to the property. In this way, we hope to produce refinements that kill a large space of counterexamples.

```
Function AxiomRefine($\tau = \langle n, \mathcal{M} \rangle$,$\psi$,$\Gamma$)
    Input counterexample $\tau$, axiom instance $\psi$ false in $\mathcal{M}$,
        condition set $\Gamma$
    Yields sequence of refinements that kill $\tau$

    if $n = 0$:
        yield TheoryRef($\mathrm{unindex}_0(\psi), \mathbf{tt}$)
    else:
        $\mathcal{R} \leftarrow \mathbf{1}$
        choose $0 \leq i < n$
        for each $x^j$ in $\psi$ such that $j \neq i, i+1$:
            choose Boolean combination $\rho$ over $\Gamma$ such that:
                (1) $\rho$ is $j$-local and
                (2) $\mathrm{unindex}_j(\rho)$ captures $x^j$ in $\tau$
            let $\pi, \eta$ be fresh constants in $\mathbb{V}_B, \mathbb{V}_S$
            $\psi \leftarrow \psi[\pi/x^j]$
            $\mathcal{R} \leftarrow$ ProphRef($\pi, \eta$) $\circ$ HistRef($\eta, \mathrm{unindex}_j(\rho), x$) $\circ \mathcal{R}$
        return TheoryRef($\mathbf{tt}, \mathrm{unindex}_i(\psi)$) $\circ \mathcal{R}$
```

Fig. 4: Refinement with an axiom instance

We achieve this by constructing a property-driven refutation of the counterexample. We start with a *goal term* whose value in the counterexample we wish to contradict. We then use trigger-based quantifier instantiation, as introduced in the Simplify theorem prover [8] to match the goal term against a *trigger pattern* in one of the axioms. This gives us a ground axiom instance in which the trigger term is equal to the goal term. If this axiom instance is false in the counterexample, we use it to generate a refinement. Otherwise, we extract from the axiom instance a new goal term and try to contradict the value of this term. We also extract relevant predicates to use in history variable conditions from the axiom instances in the refutation.

A *trigger-form* axiom is of the form $\forall V.\ \psi \rightarrow t_t = t_g$ where V is a set of variables, quantifier-free formula ψ is the *precondition*, term t_t is the *trigger* and term t_g is the *goal*. Each variable in V must occur in the trigger t_t exactly once. In the sequel, we assume the refinement theory $\mathcal{T}_R$ is a set of trigger-form axioms.

Here are the array theory axioms in trigger form:

$$\forall A, X, Y, V.\ (X = Y) \rightarrow \mathtt{select}(\mathtt{store}(A, X, V), Y) = V \tag{1}$$

$$\forall A, X, Y, V.\ (X \neq Y) \rightarrow \mathtt{select}(\mathtt{store}(A, X, V), Y) = \mathtt{select}(A, Y) \tag{2}$$

$$\forall X, V.\ \mathbf{tt} \rightarrow \mathtt{select}(\mathtt{constArr}(V), X) = V \tag{3}$$

Example of property-driven counterexample refutation Suppose the following sub-formulas appear in the BMC unfolding:

$$a^1 = \mathtt{store}(a^0, i^0, x^0) \tag{4}$$
$$a^2 = a^1 \tag{5}$$
$$b^3 = \mathtt{store}(b^2, j^2, \mathtt{select}(a^2, k^2)) \tag{6}$$
$$\neg p(\mathtt{select}(b^3, l^3)) \tag{7}$$

The last of these represents the failure of the safety property. Let us take $\mathtt{select}(b^3, l^3)$ as our goal term. By substitution with (6), this term is equal to $\mathtt{select}(\mathtt{store}(b^2, j^2, \mathtt{select}(a^2, k^2)), l^3)$. Thus, modulo equality, we can match it against the trigger term $\mathtt{select}(\mathtt{store}(A, X, V), Y)$ of axiom (1) with the assignment $A = b^2$, $X = j^2$, $V = \mathtt{select}(a^2, k^2)$, $Y = l^3$. This gives us the following axiom instance:

$$(j^2 = l^3) \rightarrow \mathtt{select}(\mathtt{store}(b^2, j^2, \mathtt{select}(a^2, k^2)) = \mathtt{select}(a^2, k^2)$$

Suppose that the precondition $j^2 = l^3$ of this instance is true. In this case we consider the axiom instance to be relevant to the goal. If the instance is false, we use it as a refinement. It is 2-local (referring only to symbols at times 2 and 3) therefore we can add it directly to the transition relation to kill the counterexample. On the other hand, suppose the instance is true. This implies our goal term is equal to $\mathtt{select}(a^2, k^2)$, the goal term of the axiom. We therefore take this as our new goal, attempting to contradict its value in the counterexample. Moreover, we consider the precondition $j^2 = l^3$ as a potential history variable condition. Intuitively, this is the condition under which the new goal term influences the property.

The new goal $\mathtt{select}(a^2, k^2)$ again matches axiom (1) yielding this axiom instance:

$$(i^0 = k^2) \rightarrow \mathtt{select}(\mathtt{store}(a^0, i^0, x^0), k^2) = x^0$$

Again supposing that the precondition is true but the axiom instance is false, we can use this instance to refine. This instance, however, is not local because it contains k^2. We can localize it by capturing the value of k^2 with a history variable η under the condition $j^2 = l^3$ derived from our first inference. This adds $\eta' = \text{ite}(j = l', k, \eta)$ to the transition relation and rewrites the safety property to $\pi = \eta \rightarrow p(\mathtt{select}(b, l))$. Substituting the non-local term k^2 by π we add the following axiom instance to the transition relation, killing the counterexample:

$$(i = \pi) \rightarrow \mathtt{select}(\mathtt{store}(a, i, x), \pi) = x$$

E-graphs We now describe our algorithm for refinement based on property-driven counterexample refutations.

An E-graph [8] is a structure that maps a set of terms to equality classes. If we are given a model of a formula, the corresponding E-graph is a partial interpretation of the symbols over the universe that is sufficient to evaluate all

```
**Generator** TriggerMatches($\tau$, $\mathcal{M}$, $t_m$)
    **Input** trace $\tau$, E-graph $\mathcal{M}$, term $t_m$
    **Yields** sequence of trigger matches for $t_m$

    **for** $\forall V.\psi \to t_t = t_g$ **in** $\mathcal{T}_R$:
        **for** $\sigma$ **in** EMatch($\mathcal{M}$, $t_t$, $t_m$):
            **if** $\tau \models \psi[\sigma]$:
                **yield** $(\psi \to t_t = t_g)[\sigma]$
```

Fig. 5: Trigger matching algorithm

the sub-terms of the formula. In our case, the formula is a BMC unfolding and the model is a counterexample to the safety property. As in Simplify, we match triggers to goal terms by substituting the free variables with terms occurring in the E-graph. Heuristically, by using existing terms in the BMC formula we hope to obtain axiom instances that are generally useful and not specific to one counterexample.

A *partial Σ-interpretation* over a universe U maps each symbol $f^{D_1 \times \cdots \times D_n \to R}$ in Σ to a *partial* function $U(D_1) \times \cdots \times U(D_n) \to U(R)$. For partial Σ-interpretations $\mathcal{I}, \hat{\mathcal{I}}$, we say $\mathcal{I} \sqsubseteq \hat{\mathcal{I}}$ if $f[\mathcal{I}] \sqsubseteq f[\hat{\mathcal{I}}]$ for all $f \in \Sigma$. An *E-graph* over symbols Σ is a pair $\mathcal{M} = (U, \mathcal{I})$ where U is a universe and $\mathcal{I}$ is a partial Σ-interpretation over U. For E-graphs $\mathcal{M} = \langle U, \mathcal{I} \rangle$ and $\hat{\mathcal{M}} = \langle U, \hat{\mathcal{I}} \rangle$ over Σ, we say $\mathcal{M} \sqsubseteq \hat{\mathcal{M}}$ if $\mathcal{I} \sqsubseteq \hat{\mathcal{I}}$.

We assume a special value $\perp$ not present in any universe. The interpretation $t[\mathcal{M}]$ of a term t in an E-graph $\mathcal{M} = \langle U, \mathcal{I} \rangle$ is defined as follows:

- $x[\mathcal{M}] = \mathcal{I}(x)$ if x is a constant in $\mathrm{dom}(\mathcal{I})$ else $\perp$,
- $f(t_1, \ldots, t_n)[\mathcal{I}] = \mathcal{I}(f)(t_1[\mathcal{I}], \ldots, t_n[\mathcal{I}])$ if $f \in \mathrm{dom}(\mathcal{I})$ and $(t_1[\mathcal{I}], \ldots, t_n[\mathcal{I}]) \in \mathrm{dom}(\mathcal{I}(f))$ else $\perp$.

Given a Σ-structure $\mathcal{M}$ and a set of terms $\mathcal{L}$ over Σ, let EGraph$(\mathcal{M}, \mathcal{L})$ denote the least $\hat{\mathcal{M}} \sqsubseteq \mathcal{M}$ such that $t[\hat{\Sigma}] \neq \perp$ for all terms $t \in \mathcal{L}$.

Matching modulo equality Given an E-graph $\mathcal{M}$, a term t_t with free variables V and a ground term t_m, a *match modulo equality* $t_t \to t_m$ is an assignment σ of ground terms to variables in V such that $\mathcal{M} \models t_t[\sigma] = t_m$.

Given a Σ-structure τ, an E-graph $\mathcal{M}$, a term t_m and a set of trigger-form axioms $\mathcal{T}$, a *trigger match* for t_m is $\psi[\sigma]$ for any $\psi \in \mathcal{T}$ of the form $\forall V.\psi \to t_t = t_g$ such that σ is a match $t_t \to t_m$ modulo equality and $\tau \models \psi[\sigma]$.

The algorithm shown in Fig. 5 generates a stream of trigger matches given τ, $\mathcal{M}$ and t_m. It is described as a generator (as in [8]) in which each "yield" statement appends an element to the stream. We rely on a procedure Ematch that yields a stream of of matches $t_t \to t_m$ in a given E-graph. This is implemented in the same way as in [8].

```
**Generator** VIOLATIONS($\tau$, $\mathcal{M}$, $t_m$, $\Gamma$)
    **Input** trace $\tau$, E-graph $\mathcal{M}$, term $t_m$, conditions $\Gamma$
    **Yields** sequence of violations $\langle \phi, \Gamma \rangle$

    **for** $\psi \to t_t = t_g$ **in** TRIGGERMATCHES($\tau$, $\mathcal{M}$, $t_m$):
        **if** $\tau \models \psi \to t_t = t_g$:
            **for** $v$ **in** VIOLATIONS($\tau$, $\mathcal{M}$, $t_g$, $\Gamma \cup \{\psi\}$):
                **yield** v
        **else**:
            **yield** $\langle \psi \to t_t = t_g, \Gamma \rangle$
```

Fig. 6: Property-guided search for counterexample refutations

```
**Generator** REFINEMENTS($\Pi = \langle \Sigma, I, T, \phi \rangle$, $\tau = \langle n, \mathcal{M} \rangle$)
    **Input** problem $\Pi$, counterexample $\tau$
    **Yields** sequence of refinements that kill $\tau$

    $\mathcal{M} \leftarrow$ EGRAPH($\tau$, terms(BMC($\Pi$)))
    $\Gamma \leftarrow$ MINECONDITIONS($\Pi, \tau$)
    **for** $t_m$ in MINETERMS($\Pi, \tau$)
        **for** $\langle \psi, \Gamma \rangle$ **in** VIOLATIONS($\tau$, $\mathcal{M}$, $t_m$, $\Gamma$):
            **for** $\mathcal{R}$ **in** AXIOMREFINE($\tau$, $\psi$, $\Gamma$):
                **yield** $\mathcal{R}$
```

Fig. 7: Procedure for generating refinements

Fig. 6 shows a procedure that searches for a refutation of a counterexample. A *violation* is a pair consisting of an axiom instance that is false in the counterexample, and a set of preconditions for the instance to be relevant to the goal. The procedure searches for trigger matches against the goal term. For each match found, if the axiom instance is false, it returns a violation. If the axiom instance is true, it recurs on the instance's goal term, adding its precondition to the list of preconditions.

Finally, Fig. 7 shows our procedure for generating refinements from a counterexample. We begin by building the E-graph $\mathcal{M}$ for the terms in the BMC formula. We call a procedure MINECONDITIONS to collect a set of predicates from the problem that are heuristically likely to be useful history variable conditions. Similarly MINETERMS returns a stream of terms that are likely to be causally related to the property, ordered from most to least relevant. See Sec. 8 for details of these functions in our implementation. For each term, we call VIOLATIONS to search for relevant axiom violations, and for each of these, we call AXIOMREFINE to generate refinements.

Theorem 7. *If τ is an abstract $\mathcal{T}_A$-counterexample for problem Π then every refinement in* REFINEMENTS($\tau, \mathcal{T}_R$) *kills τ.*

Proof sketch Every ψ generated by VIOLATIONS is an axiom instance false in τ. Thus by Thm. 6 every generated refinement kills τ. □

7 Capture conditions from interpolants

Consider the program of Fig. 2. In this program, non-negative values from array a are added to values in array b that are initially zero. The safety property requires that all values in b are non-negative. Suppose that as before, we capture the index at which the property is violated with prophecy variable π_1 and the last index of array a that flows to $b[\pi_1]$ with prophecy variable π_2. This refinement can kill a counterexample in which the *last* write to $b[\pi_1]$ causes it to become negative. However, it does not kill a counterexample in which $b[\pi_1]$ is already negative at the last time it is written. In this case, we need a more nuanced notion of causality. That is, the second loop maintains the invariant that $b[\pi_1] \geq 0$. The cause of the property failure is actually the last write that turned the invariant from true to false. A suitable condition for capturing the index x would thus be $x = \pi_1 \wedge b[\pi_1] \geq 0$, or perhaps $b[\pi_1] \geq 0 \wedge \neg(b'[\pi_1] \geq 0)$.

The question is how to guess the invariant that the loop maintains. In simple cases such as this, it is just the safety condition. However it is easy to construct examples where this is not the case (for example, with multiple loops). A common approach to guessing an invariant of a loop is to construct a sequence interpolant for the BMC unfolding.

If $\beta = \beta_0, \ldots, \beta_n$, for $n \geq 2$, is a formula sequence, formula sequence $\mathcal{I} = \mathcal{I}_0, \ldots, \mathcal{I}_{n-1}$ is a *sequence interpolant modulo $\mathcal{T}$* if:

- $\mathcal{T} \models \beta_0 \rightarrow \mathcal{I}_0$, and
- for $i = 1 \ldots n$, $\mathcal{T} \models \beta_i \wedge \mathcal{I}_{i-1} \rightarrow \mathcal{I}_i$, and
- $\mathcal{T} \models \mathcal{I}_{n-1} \wedge \beta_n \rightarrow \mathbf{ff}$, and
- for $0 \leq i < n$, $\text{vocab}(\mathcal{I}_i) \subseteq (\text{vocab}(\beta_0, \ldots, \beta_i) \cap \text{vocab}(\beta_{i+1}, \ldots, \beta_n)) \cup \mathbb{V}_B$.

A sequence interpolant can be constructed from an unsatisfiable sequence of formulas using an interpolating theorem prover [24] such as SMTInterpol [4]. Fig. 8 shows a modified version of MINECONDITIONS that adds predicates derived from an interpolant for the BMC unfolding of the problem. This makes it possible to handle problems such as Fig. 2 in which an array holds aggregate values from another array. Note that we extract from the interpolant only atomic predicates and their negations. In principle, we could use a larger class of Boolean combinations, at the expense of a more expensive search for a refinement.

8 Evaluation

In our evaluation, we consider looping programs using arrays that require quantifiers in the inductive invariant to prove safety properties. We address two research questions: *(1)* Does our approach to synthesizing conditional history variables effectively allow a quantifier-free model checker to verify the programs, and *(2)*

```
**Function** MineConditionsItp($\Pi = \langle \Sigma, I, T, \phi \rangle$, $\tau = \langle n, \mathcal{M} \rangle$)
    **Input** problem $\Pi$, counterexample $\tau$ to $\Pi$
    **Yields** set of conditions

    $\Gamma \leftarrow$ MineConditions($\Pi, \tau$)
    **if** $\mathcal{T} \models \neg$BMC($\Pi$):
        **let** $\mathcal{I}$ be a sequence interpolant for BMC($\Pi$) modulo $\mathcal{T}$
        **for** $\psi$ in $\mathcal{I}$:
            **for** atomic $p$ occurring in $\psi$:
                $\Gamma \leftarrow \Gamma \cup \{p, \neg p\}$
    **return** $\Gamma$
```

Fig. 8: Capture conditions from interpolants

are conditional history variables more effective than unconditional history variables. We use two baselines: the algorithm of [20] and an ablation that uses our property-driven refutation method to find axiom violations, but uses unconditional history variables.

8.1 Implementations

We implemented our algorithm in the Python programming language. We will call this implementation CondHist. We use the tool IC3ia [6] for model checking in algorithm CEGAR. As in [20], we chose this tool because it supports uninterpreted function symbols, which we need for the abstract theory. For counterexample generation, we use [7] to solve the BMC unfolding at the depth of the IC3ia counterexample. This is needed to obtain the values of the uninterpreted function symbols. For satisfiability and interpolation in algorithm MineConditionsItp we use SMTInterpol [4].

Input and preprocessing We take input in the form of linear CHC solving problems in the SMTLIB2 format using the Z3 fixpoint convention or the standard HORN logic. These problems can have multiple uninterpreted predicates corresponding to program control locations. We translate this form to a simple transition system by adding an integer control location symbol *pc*. We also Herbrandize the safety condition by replacing leading universals with background symbols. As in [20], we replace large numeric constants with free symbols, which helps to prevent IC3ia from diverging. If we obtain a false counterexample, we run again without this abstraction. Additionally, we replace any reference to a primed variable in a transition with an equivalent unprimed expression if possible. These preprocessing tactics are held constant across the ablation experiments.

Heuristics There are a few nondeterministic choices in the algorithm. Here, we detail the heuristics we use to make each of these choices. These heuristics are primitive, and we expect that they can be significantly improved.

In CEGAR we choose the first refinement obtained. In algorithm AXIOM-REFINE we chose the time i in which to localize the axiom instance as the least time occurring in the instance. The Boolean combination ρ is the conjunction of all of the predicates in Γ that are j-local and true in the counterexample. We try this first without using interpolant-derived predicates and then add them if this fails to achieve capture of x^j. We could perhaps improve this by greedily removing predicates as long as capture is maintained.

In MINECONDITIONS we produce a set of predicates of the form $pc = i$ where pc is a special symbol representing the program control location and i is an integer. We also produce p and $\neg p$ for every program branch condition p. We found this control flow information to be useful in constructing capture conditions. In MINETERMS we list all of the terms in the BMC unfolding in reverse time order. In this way, we prioritize refutations of counterexamples that are causally connected to the safety property.

Baseline implementations The implementation of the algorithm of [20] by its authors is called PROHIC3. Unfortunately, we found that this tool produced incorrect results (see the Appendix for details). Because of this, we implemented their algorithm as described in [20], using the same underlying tools and preprocessing as CONDHIST. We will call this implementation UNCONDHIST1. It differs from CONDHIST in several ways. First, it captures a value x^j at time n using a sequence of $n - j$ unconditional history variables each delaying the value by one step. Second, it produces axiom violations by enumerating and evaluating all of the axioms instances that can be constructed using index terms, `select` terms and `constArr` terms in the BMC formula. This process continues until all BMC counterexamples at depth n are eliminated, at which point the UNSAT core is used to select a sufficient set of instances. Additionally, for each BMC counterexample, we select just the i-local violations if there are any, to avoid generating unnecessary prophecy. We implemented only one version of the algorithm, the so-called "strong abstraction", which abstracts only the array theory and not the "weak abstraction" which also abstracts the equality axioms for equality between arrays. This version gives a more direct comparison to CONDHIST.

We also implemented a version of our algorithm UNCONDHIST2 that is identical to CONDHIST except that generates unconditional history variables in the same way as UNCONDHIST1.

8.2 Experiments

For evaluation, we use as a benchmark a set of 193 problems from the distribution of the FreqHorn tool [9]. These represent properties of small array manipulating programs that typically have from one to three loops. Single loop benchmarks are grouped in the Single category, totaling 118 benchmarks, while all other benchmarks are grouped in the Multi category, totaling 75 benchmarks. All require

Tool	Single Solved	Single Timeouts	Multi Solved	Multi Timeouts
CondHist	95 | 0	23	64 |28	11
Quic3	81 | 0	37	36 | 0	39
GSpacer	59 | 0	59	30 | 0	45
UnCondHist1	71 | 39	47	37 | 16	38
UnCondHist2	93 | 0	25	29 | 1	47

Table 1: Comparison of tools on benchmark problems.

quantifiers in the inductive invariant. This tests the ability of the algorithm to avoid divergence by introducing suitable history and prophecy variables, though it does not address the question of scalability.

On this benchmark set, we compare against the performance of the algorithm of [20]. This tests the hypothesis that conditional history variables are more effective in preventing divergence than unconditional history variables. We should note, however, that even in cases where the generated history and prophecy variables *allow* an unquantified inductive invariant, the underlying model checker (IC3ia) may still diverge. Improvements in the model checker could improve the performance of all of the algorithms that we compare.

We applied the tools on a AWS EC2 instance, with 8GB of physical memory, using a timeout of 120 seconds and no limit on memory usage.

Results Tab. 1 shows the benchmark results. For each tool we show the number of solved problems and the number of timeouts. In both "Solved" columns the number to the left represents the total number of benchmarks solved and the number to the right, the number of those solutions which required prophecy. We observe that the new algorithm solves a substantially larger number of problems than either baseline. This supports a positive answer to research question 2. We noted that, of the 39 problems solved by CondHist1 using history variables, all 39 were solved by CondHist2 with *no* history variables. A possible explanation of this is that history variables in the method of [20] are serving only to compensate for poor choices of axiom violations.

The table also shows results for two existing tools that generate quantified invariants without auxiliary variables. These are Quic3 [13] and GSpacer [18], two versions of Spacer [17] [17] extended with universally quantified predicates. These tools have both performed well recently in the arrays category of CHC-COMP [29], the CHC solving competition. We omitted FreqHorn from the comparison, since the benchmark set is supplied by the FreqHorn developers and thus is likely to be biased toward problems that FreqHorn can solve. We observe that CondHist outperforms state-of-the-art CHC solvers, supporting a positive answer to research question 1.

9 Conclusion and Future Work

We introduced an approach to the introduction of auxiliary variables that can allow array-manipulating programs to be verified using quantifier-free invariants. This makes it possible to apply existing model checkers and CHC solvers that cannot generate quantified invariants. The two key aspects of this approach are conditional history variables and property-directed refutations of counterexamples.

Experimentally, we observed that the prior approach using unconditional history variables (*i.e.,* fixed delays) is not effective for verifying programs with arrays when prophecy is actually required. This is not surprising, since most loops in such programs are unbounded. This means that for any finite delay, we can construct a long enough loop execution that the finite-delay history variable becomes irrelevant. In our observation, with more relevant choices of axiom violations, unconditional history variables were almost never helpful. On the other hand, the conditional history variable approach can effectively use prophecy to solve problems like the examples in Sec. 1 that the unconditional approach cannot solve.

There are several limitations of the current approach to be addressed in future work. The method applies to theories that can be axiomatized in what we called "trigger form". This applies to the array theory but other theories will likely require some generalization of the method. Also, for properties with quantifier alternation, we will have to instantiate quantifiers in the property. The current method does not handle this. Moreover, it is easy to construct problems for which history and prophecy are insufficient to eliminate quantifiers from the invariant. For these cases, auxiliary variables may still be helpful, but we cannot make do with a quantifier-free invariant generator and the current approach will not work with a quantified invariant generator. Despite this, it is interesting that the method outperforms theoretically more capable quantified invariant generators on small programs. More engineering work is needed to test the scalability of the approach on problems of realistic size.

Finally, of course, conditionally storing the value of an existing term in the program is common in manual proofs, but still represents a small class of possible history variables. Synthesizing history variables from a richer template may present difficult heuristic challenges.

From a broader perspective, the synthesis of new terms in a proof is a key strategy (perhaps *the* key strategy) in decomposing proofs into simpler lemmas and is known to reduce proof complexity for propositional logic. However, it has proven very challenging to automate. Despite limitations in the form of history variables, we consider the general approach of [20] to be significant and promising. We believe that history variable synthesis in general is an important topic for future research.

References

1. MartÃn Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
2. Nikolaj S. Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified horn clauses. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 105–125. Springer, 2013.
3. Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, pages 382–398, 2004.
4. Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating SMT solver. In Alastair F. Donaldson and David Parker, editors, *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254. Springer, 2012.
5. Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 331–346, 2003.
6. Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta, and Sergio Mover. Infinite-State Liveness-to-Safety via Implicit Abstraction and Well-Founded Relations. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 271–291, Cham, 2016. Springer International Publishing.
7. Leonardo de Moura and Nikolaj BjÃžrner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 337–340, Berlin, Heidelberg, 2008. Springer.
8. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
9. Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified Invariants via Syntax-Guided Synthesis. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, volume 11561, pages 259–277. Springer International Publishing, Cham, 2019. Series Title: Lecture Notes in Computer Science.
10. Silvio Ghilardi and Silvio Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6(4), 2010.
11. Aman Goel and Karem A. Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez, editors, *NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings*, volume 12673 of *Lecture Notes in Computer Science*, pages 131–150. Springer, 2021.

12. Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. Smt-based verification of parameterized systems. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 338–348. ACM, 2016.
13. Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Quantifiers on demand. *CoRR*, abs/2106.00664, 2021.
14. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 232–244. ACM, 2004.
15. Aleksandr Karbyshev, Nikolaj S. Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *J. ACM*, 64(1):7:1–7:33, 2017.
16. Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 703–717, New York, NY, USA, 2020. Association for Computing Machinery.
17. Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs, 2014.
18. H.G.V. Krishnan and Arie Gurfinkel. CHC-COMP 2020 Submission, 2020.
19. S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, pages 267–281, 2004.
20. Makai Mann, Ahmed Irfan, Alberto Griggio, Oded Padon, and Clark Barrett. Counterexample-Guided Prophecy for Model Checking Modulo the Theory of Arrays. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 12651, pages 113–132. Springer International Publishing, Cham, 2021. Series Title: Lecture Notes in Computer Science.
21. K. L. Mcmillan. Circular compositional reasoning about liveness. In *Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME '99), volume 1703 of Lecture Notes in Computer Science*, pages 342–345. Springer-Verlag, 1999.
22. K. L. Mcmillan. Verification of infinite state systems by compositional model checking. In *in CHARME*, pages 219–233. Springer, 1999.
23. Kenneth L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In Tiziana Margaria and Thomas F. Melham, editors, *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK, September 4-7, 2001, Proceedings*, volume 2144 of *Lecture Notes in Computer Science*, pages 179–195. Springer, 2001.
24. Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.
25. Kenneth L. McMillan. Eager Abstraction for Symbolic Model Checking. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 191–208, Cham, 2018. Springer International Publishing.
26. David Monniaux and Laure Gonnord. Cell morphing: From array programs to array-free horn clauses. In Xavier Rival, editor, *Static Analysis - 23rd International*

Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings, volume 9837 of *Lecture Notes in Computer Science*, pages 361–382. Springer, 2016.

27. Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
28. Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, pages 82–97, 2001.
29. Philipp Rümmer. Competition report: CHC-COMP-20. *Electronic Proceedings in Theoretical Computer Science*, 320:197–219, aug 2020.

$$I \doteq a = \texttt{constArr}(0) \wedge b = \texttt{constArr}(1)$$
$$T \doteq b' = \texttt{store}(b, y, \texttt{select}(a, x)) \wedge a' = \texttt{store}(a, x, 1)$$
$$\phi \doteq \texttt{select}(b, Z) \leq 1$$

Fig. 9: Model checking problem

A Prophic3 implementation issues

In testing the tool Prophic3, we obtained a result we believe to be incorrect. In particular, the tool produced a positive result in a case when the algorithm described in [20] should not have terminated successfully. Consider the problem in Fig. 9. In the safety condition ϕ, the symbol Z is a background symbol that results from Herbrandizing the formula $\forall z.\ \texttt{select}(b, z) \leq 1$. The transition relation effectively copies a value from array a at a nondeterministic index x to array b at a non-deterministic address y. The array a is modified only so it is not invariantly a constant array. The result is that b always contains a mix of zeros and ones.

To solve this problem, we must prophecy the value of x at the last time that $y = Z$. This can't be done with unconditional history variables, since they can only store information about a bounded number of steps k before the property failure. Thus, for any set of unconditional history variables, we can create a counterexample long enough the that history becomes irrelevant by appending steps in which $y \neq Z$. Nonetheless the tool Prophic3 seemed to solve this problem using only one history variable that delayed the value of x by one step. Since this would appear to be insufficient, we tried tried to check the proof produced by Prophic3 by taking all of its refinements (axiom instances, history and prophecy variables) and applying them to the original program. We checked this instrumented system with IC3ia (the same model checker used by Prophic3) and IC3ia found the property to be false. It gave the expected counterexample with steps having $y \neq Z$ at the end.

For this reason we decided to implement the algorithm of [20] and attempt to replicate the results in that paper. Those results included a subset of the FreqHorn benchmarks that we use (they use only those that have a single program loop). Our implementation UncondHist1 successfully proved 56 of these shared benchmarks, while Prophic3 proved 67. Moreover, all properties proved by UncondHist1 can be proved without auxiliary variables. Because of the issue described above, we suspect that this difference may be accounted for by an incorrect construction by Prophic3 of the refined system passed to IC3ia, though we cannot verify this as the instrumented models are not in a human-readable format and we do not have tools that can verify correctness of the refinements.