Quantified Invariants via Syntax-Guided Synthesis

Grigory Fedyukovich[1], Sumanth Prabhu[2], Kumar Madhukar[2], and Aarti Gupta[1]

[1] Princeton University, Princeton, USA {grigoryf, aartig}@cs.princeton.edu
[2] TCS Research, Pune, India {sumanth.prabhu, kumar.madhukar}@tcs.com

Abstract. Programs with arrays are ubiquitous. Automated reasoning about arrays necessitates discovering properties about ranges of elements at certain program points. Such properties are formally specified by universally quantified formulas, which are difficult to find, and difficult to prove inductive. In this paper, we propose an algorithm based on an enumerative search that discovers quantified invariants in stages. First, by exploiting the program syntax, it identifies ranges of elements accessed in each loop. Second, it identifies potentially useful facts about individual elements and generalizes them to hypotheses about entire ranges. Finally, by applying recent advances of SMT solving, the algorithm filters out wrong hypotheses. The combination of properties is often enough to prove that the program meets a safety specification. The algorithm has been implemented in a solver for Constrained Horn Clauses, FreqHorn, and extended to deal with multiple (possibly nested) loops. We show that FreqHorn advances state-of-the-art on a wide range of public array-handling programs.

1 Introduction

Formally verifying programs against safety specifications is difficult. This problem worsens in the presence of data structures like lists, arrays, and maps, which are ubiquitous in real-world applications. For instance, proving an array-handling program safe often requires discovering an inductive invariant that is universally quantified over ranges of array elements. Such invariants help to prove the unreachability of error states independently of the size of the array. However, the majority of invariant synthesis approaches are limited to quantifier-free numerical invariants. The approach presented in this paper advances the knowledge by an effective technique to discover quantified invariants over arrays and linear integer arithmetic.

Syntax-guided techniques [3] have recently been applied to synthesize quantifier-free numerical invariants [16,15,17,34] in the approach called FreqHorn. In a nutshell, FreqHorn collects various statistics from the syntactical patterns occurring in the program's source code and uses them to construct a set of formal grammars that specify a search space for invariants. It is often sufficient to perform an *enumerative search* over the formulas produced from these grammars

and identify a set of suitable inductive invariants among them using an off-the-shelf solver for Satisfiability Modulo Theories (SMT). The presence of arrays complicates this reasoning in a few respects: it is hard to find suitable candidates and difficult to prove them inductive.

In this paper, we present a novel technique that extends the approach of enumerative search in general, and its instantiation in FreqHorn in particular, to reason about quantifiers. It discovers invariants over arrays in multiple stages. First, by exploiting the program syntax, it identifies ranges of elements accessed in each loop. Second, it identifies potentially useful facts about individual elements and generalizes them to hypotheses about entire ranges. The SMT-based validation of candidates, which are quantified formulas, is often inexpensive as they are constructed using the same syntactic patterns that appear in the source code. Furthermore, for supporting certain corner cases, our approach allows specifying additional rules that help in generalizing learned properties. The combination of properties proven inductive by an SMT solver is often enough to prove that the program meets a safety specification.

We show that FreqHorn advances state-of-the-art on a selection of array-handling programs from SVCOMP[3] and literature. For instance, it can prove completely automatically that an array is monotone after applying a sorting algorithm. Furthermore, FreqHorn is able to discover quantifier-free invariants over integer variables in the program, use them as inductive relatives while checking inductiveness of quantified candidates over arrays; and vice versa.

While a detailed discussion of the related work comes later in the paper (Sect. 6), it is noteworthy that being syntax-guided crucially helps us overcome several limitations of other techniques to verify array-handling programs [11,35,2,9]. Most of them avoid inferring quantified invariants explicitly and thus do not produce checkable proofs. As a result, tools are fragile and in practice often output false positives (see Sect. 5 for concrete results). By comparison, our approach never produces false positives, and its results can be validated by existing SMT solvers.

The core contributions made through this work are:

- a novel syntax-guided approach to generate universally quantified invariants for programs manipulating arrays;
- an algorithm and its fully automated implementation; and
- a thorough experimental evaluation comparing our technique with state-of-the-art in verification of array-handling programs.

The rest of the paper is structured as follows. In Sect. 2, we give background and notation and illustrate our approach on an example. Our main contributions are then presented in Sect. 3 (main algorithm) and Sect. 4 (important design choices). In Sect. 5, we show the evaluation and comparison with state-of-the-art. Finally, the related work and conclusion complete the paper in Sect. 6 and 7, respectively.

[3] Software Verification Competition, `http://sv-comp.sosy-lab.org/`.

2 Background

The Satisfiability Modulo Theories (SMT) task is to decide whether there is an assignment m of values to variables in a first-order logic formula φ that makes it true. We write $\varphi \implies \psi$, if every satisfying assignment to φ is also a satisfying assignment to some formula ψ. By *Expr* we denote the space of all possible quantifier-free formulas in our background theory and by *Vars* a range of possible variables.

2.1 Programs as constrained Horn clauses

To guarantee expected behaviors, programs require proofs, such as inductive invariants, ranking functions, or recurrence sets. It is becoming increasingly popular to consider a verification task as a *proof synthesis* task which is formulated as a system of SMT formulas involving unknown predicates, also known as *constrained Horn clauses* (CHC). The synthesis goal is to discover a suitable interpretation of all unknown predicates that make all CHCs true. CHCs offer the advantages of flexibility and modularity in designing verifiers for various systems and languages. CHCs can be constructed in a way that captures the operational semantics of a language in question, and an off-the-shelf CHC solver can be used for solving the resulting formulas.

Definition 1. *A* linear constrained Horn clause *(CHC) over a set of uninterpreted relation symbols $\mathcal{R}$ is a formula in first-order logic that has the form of one of three implications (called respectively a* fact*, an* inductive clause*, and a* query*):*

$$\begin{aligned}
&\varphi(\vec{x_1}) \implies \boldsymbol{inv}_1(\vec{x_1}) \\
&\boldsymbol{inv}_1(\vec{x_1}) \wedge \varphi(\vec{x_1}, \vec{x_2}) \implies \boldsymbol{inv}_2(\vec{x_2}) \\
&\boldsymbol{inv}_1(\vec{x_1}) \wedge \varphi(\vec{x_1}) \implies \bot
\end{aligned}$$

where $\boldsymbol{inv}_1, \boldsymbol{inv}_2 \in \mathcal{R}$ are uninterpreted symbols, $\vec{x_1}, \vec{x_2}$ are vectors of variables, and φ, called a body*, is a fully interpreted formula (i.e., φ does not have applications of $\boldsymbol{inv}_1$ or $\boldsymbol{inv}_2$).*

For a CHC C, by $src(C)$ we denote an application of $\boldsymbol{inv} \in \mathcal{R}$ in the premise of C (if C is a fact, we write $src(C) \stackrel{\text{def}}{=} \top$). Similarly, by $dst(C)$ we denote an application of $\boldsymbol{inv} \in \mathcal{R}$ in the conclusion of C (if C is a query, we write $dst(C) \stackrel{\text{def}}{=} \bot$). We define functions *rel* and *args*, such that for each $\boldsymbol{inv}(\vec{x})$, $rel(\boldsymbol{inv}(\vec{x})) \stackrel{\text{def}}{=} \boldsymbol{inv}$ and $args(\boldsymbol{inv}(\vec{x})) \stackrel{\text{def}}{=} \vec{x}$. For a CHC C, by $body(C)$ we denote the body (i.e., φ) of C.

Example 1. Fig. 1 gives a program in the C programming language that handles two integer arrays, `A` and `B`, both of an unknown size `N`. The `A` array has unknown content, and the program first identifies a value `m` which is smaller or equal to all elements of `A` (it might be either a minimal element among the content of `A` or 0).

```
int N = nondetInt();
int *A = nondetArray(N);
int m = 0;
for (int i = N - 1; i ≥ 0; i--) { if (m > A[i]) m = A[i]; }
int *B = malloc(N*sizeof(int));
for (int i = 0; i < N; i++) { B[N - i - 1] = A[i] - m; }
int s = 0;
for (int i = 0; i < N; i++) { s = s + B[i]; }
assert(s ≥ 0);
```

Fig. 1: Example program: source code in C.

(A) $i' = N' - 1 \wedge m' = 0 \implies \boldsymbol{inv}_1(A', i', m', N')$

(B) $\boldsymbol{inv}_1(A, i, m, N) \wedge i \geq 0 \wedge m' = ite(m > A[i], A[i], m) \wedge i' = i - 1 \implies \boldsymbol{inv}_1(A, i', m', N)$

(C) $\boldsymbol{inv}_1(A, i, m, N) \wedge i < 0 \wedge i' = 0 \implies \boldsymbol{inv}_2(A, B, i', m, N)$

(D) $\boldsymbol{inv}_2(A, B, i, m, N) \wedge i < N \wedge B' = store(B, N - i - 1, A[i] - m]) \wedge i' = i + 1 \implies \boldsymbol{inv}_2(A, B', i', m, N)$

(E) $\boldsymbol{inv}_2(A, B, i, m, N) \wedge i \geq N \wedge i' = 0 \wedge s' = 0 \implies \boldsymbol{inv}_3(A, B, i', m, s', N)$

(F) $\boldsymbol{inv}_3(A, B, i, m, s, N) \wedge i < N \wedge s' = s + B[i] \wedge i' = i + 1 \implies \boldsymbol{inv}_3(A, B, i', m, s', N)$

(G) $\boldsymbol{inv}_3(A, B, i, m, s, N) \wedge i \geq N \wedge s < 0 \implies \bot$

Fig. 2: Example program: CHC encoding.

Then, the program populates `B` by values of `A` with `m` subtracted. Interestingly, the order of elements `A` and `B` is not preserved, e.g., `A[0] - m` gets written to `B[N - 1]`, and so on. Finally, the program computes the sum `s` of all elements in `B` and requires us to prove that `s` is never negative.

Fig. 2 gives a CHC encoding of the program. The system has three uninterpreted predicates, $\boldsymbol{inv}_1$, $\boldsymbol{inv}_2$, and $\boldsymbol{inv}_3$ corresponding to invariants at heads of the three loops. The primed variables correspond to modified variables. Rules **B**, **D**, and **F** encode the loop bodies, and the remaining rules encode the fragments of code before, after, or between the loops. In particular, rule **G** ensures that after the third loop has terminated, a program state with a negative value of s is unreachable. Before we describe how our technique solves this CHC system (see Sect. 2.2), we briefly introduce the notion of satisfiability of CHCs.

Definition 2. *Given a set of uninterpreted relation symbols $\mathcal{R}$ and a set S of CHCs over $\mathcal{R}$, we say that S is satisfiable if there exists an interpretation that assigns to each n-ary symbol $\boldsymbol{inv} \in \mathcal{R}$ a relation over n-tuples and makes all implications in S valid.*

In the paper, we assume that a relation assigned by an interpretation is represented by a formula ψ over at most n free variables.

We call a CHC C inductive when $rel(src(C)) = rel(dst(C)) = \boldsymbol{inv}$ for some $\boldsymbol{inv}$. While accessing an array in a loop, we assume the existence of an integer counter variable. More formally:

Definition 3. *Let C be an inductive CHC, $\vec{x} = args(src(C))$, and $\vec{x}' = args(dst(C))$. We say that C is* array-handling *if there exist numbers c and a, such that* 1) $1 \leq c \leq |\vec{x}|$ *and* $1 \leq a \leq |\vec{x}|$; 2) $\vec{x}[c]$ *(and consequently, its "primed copy" $\vec{x}'[c]$) has type integer,* 3) *either of these implications holds:*

$$body(C) \implies \vec{x}[c] < \vec{x}'[c] \tag{1}$$

$$body(C) \implies \vec{x}[c] > \vec{x}'[c] \tag{2}$$

4) $\vec{x}[a]$ *(and consequently $\vec{x}'[a]$) has type array, and* 5) *there is an* access function *f that identifies a relationship between an access to $\vec{x}[a]$ in $body(C)$ and $\vec{x}[c]$.*

2.2 Illustrating Example

The CHC system in Fig. 2 has a solution, indicating that the program meets its specification. In particular:

$$\begin{aligned}
\boldsymbol{inv}_1 \mapsto & \forall j \,.\, i < j < N \implies m \leq A[j] \\
\boldsymbol{inv}_2 \mapsto & \forall j \,.\, 0 \leq j < N \implies m \leq A[j] \wedge \\
& \quad \forall j \,.\, 0 \leq j < i \implies B[N-j-1] = A[j] - m \\
\boldsymbol{inv}_3 \mapsto & \forall j \,.\, 0 \leq j < N \implies m \leq A[j] \wedge \\
& \quad \forall j \,.\, 0 \leq j < N \implies B[N-j-1] = A[j] - m \wedge \\
& \quad s \geq 0
\end{aligned}$$

The interpretation of $\boldsymbol{inv}_1$ means that as the first loop progresses (i.e, all elements $A[N-1]$, $A[N-2]$, …, $A[i+1]$ are sequentially considered), the value of m is always smaller than all the considered elements. Thus, we refer to the interpretation of $\boldsymbol{inv}_1$ as a *progress lemma*. When the first loop has terminated, clearly, this property holds for all elements from $A[0]$ to $A[N-1]$. Because A leaks through the second loop without any changes, the interpretation of $\boldsymbol{inv}_1$ gets finalized (thus, it becomes a *finalized lemma*) and added to an interpretation of $\boldsymbol{inv}_2$.

Additionally, the interpretation of $\boldsymbol{inv}_2$ gets a relational fact about pairs of elements $A[0]$ and $B[N-1]$, $A[1]$ and $B[N-2]$, …, $A[i-1]$ and $B[N-i-2]$, which again appears as a progress lemma and then gets finalized in an interpretation of $\boldsymbol{inv}_3$. With these two quantified invariants about all elements of A, and relation about pairs of elements of A and B, it is possible to derive the remaining lemma in the interpretation of $\boldsymbol{inv}_3$, namely, $s \geq 0$; which concludes the proof.

3 Invariants via Enumerative Search

In this work, we aim at discovering a solution for a CHC system S over a set of uninterpreted symbols $\mathcal{R}$ enumeratively, i.e., by guessing a candidate formula for each $\boldsymbol{inv} \in \mathcal{R}$, substituting it for all CHCs $C \in S$ and checking their validity.

3.1 Quantifier-free invariants

We build on top of an algorithm, called FreqHorn, recently proposed in [17]. Its key insight is an automatic construction of a set of formal grammars $G(\boldsymbol{inv})$ for each $\boldsymbol{inv} \in \mathcal{R}$ based on either source code, program behaviors, or both. Importantly, these grammars are *conjunction-free*: they cannot be used to produce a conjunction of clauses and can give rise to only a finite number of formulas, potentially related to invariants (otherwise, the approach does not guarantee strong convergence). Since invariants are often represented by a conjunction of lemmas, FreqHorn attempts to sample (i.e., recursively apply production rules) each lemma from a grammar in separation, until a combination of them is sufficient for the inductiveness and safety, or a search space is exhausted. FreqHorn relies on an SMT solver to filter out unsuccessfully sampled lemmas.

The construction of formal grammars is biased by the syntax of CHC encoding. First, FreqHorn collects a set of *Seeds* by converting the body of each CHC to a Conjunctive Normal Form, extracting, and normalizing each conjunct. Then, the set of seeds could be optionally replenished by a set of *behavioral seeds* and *bounded proofs*. They are constructed respectively from the concrete values of variables obtained from actual program runs, and Craig interpolants from unsatisfiable finite unrollings of the CHC systems. Finally, the production rules are created in a way to enable producing seeds and also their *mutants* (i.e., syntactically similar formulas to seeds). In general, no specific restriction on a grammar-construction method is imposed; so in practice, the grammars are allowed to be more (or less) general to enable a broader (or more focused) search space for invariants.

3.2 Quantified candidates from quantifier-free grammars

The main obstacle for applying the enumerative search to generate array invariants is that the grammars do not allow quantifiers. Because grammars are constructed automatically from syntactic patterns which appear in the original programs, in the presence of arrays, we can expect expressions involving only particular elements of arrays (such as ones accessed via a loop counter). However, since each loop repeats certain operations over a *range* of array elements, we have to *generalize* the extracted expressions about individual elements to expressions about entire ranges.

Let a set of variables associated with a relation symbol $\boldsymbol{inv}$ be $\mathit{Vars}(\boldsymbol{inv}) \stackrel{\text{def}}{=} \mathit{IntVars}(\boldsymbol{inv}) \cup \mathit{ArrVars}(\boldsymbol{inv})$, where $\mathit{IntVars}(\boldsymbol{inv})$ and $\mathit{ArrVars}(\boldsymbol{inv})$ are disjoint and contain integer variables and array variables, respectively. A candidate quantified invariant over arrays consists of three parts:

- a set of quantified integer variables $\mathit{QVars}(\boldsymbol{inv})$, which are introduced by our algorithm and do not appear in $\mathit{Vars}(\boldsymbol{inv})$;
- a *range* formula over $\mathit{QVars}(\boldsymbol{inv}) \cup \mathit{IntVars}(\boldsymbol{inv})$; and
- a quantifier-free *cell property* over $\mathit{QVars}(\boldsymbol{inv}) \cup \mathit{Vars}(\boldsymbol{inv})$.

Algorithm 1: PREPARE$(S, \mathcal{R})$

Input: CHCs S over $\mathcal{R}$
Output: Formal grammars $G(\boldsymbol{inv})$, quantified variables $QVars(\boldsymbol{inv})$ and $progressRange(\boldsymbol{inv})$ for each $\boldsymbol{inv} \in \mathcal{R}$

1 **for each** $\boldsymbol{inv} \in \mathcal{R}$ **do**
2 $Seeds \leftarrow$ SYNTSEEDS$(\boldsymbol{inv}) \cup$ BEHAVSEEDS$(\boldsymbol{inv})$;
3 $cnt \leftarrow$ GETCOUNTERS$(S, \boldsymbol{inv}, ArrVars(\boldsymbol{inv}))$;
4 **if** $\varnothing \neq cnt$ **then**
5 $QVars(\boldsymbol{inv}) \leftarrow$ COPY(cnt);
6 $progressRange(\boldsymbol{inv}) \leftarrow$ GETRANGE(cnt);
7 $G(\boldsymbol{inv}) \leftarrow$ REPLACE(GETGRAMMAR$(Seeds), cnt, QVars(\boldsymbol{inv}))$;

Algorithm 2: SOLVEARRAYCHCS$(S, \mathcal{R})$

Input: CHCs S over $\mathcal{R}$
Output: $res \in \{\text{SAT}, \text{UNKNOWN}\}$, $Lemmas : \mathcal{R} \to 2^{Expr}$

1 $\langle G, QVars, progressRange\rangle \leftarrow$ PREPARE$(S, \mathcal{R})$;
2 **for each** $\boldsymbol{inv} \in \mathcal{R}$ **do** $Lemmas(\boldsymbol{inv}) \leftarrow \varnothing$;
3 **while** $\exists C \in S \,.\, \Big(\bigwedge_{\ell \in Lemmas(rel(src(C)))} \ell(args(src(C))) \wedge body(C) \implies \bot \Big)$ **do**
4 **if** $\forall \boldsymbol{inv} \in \mathcal{R} \,.\,$ ALLBLOCKED$(G(\boldsymbol{inv}))$ **then return** $\langle$UNKNOWN$, \varnothing\rangle$;
5 $\boldsymbol{inv} \leftarrow$ PICKLOOP$(\mathcal{R})$;
6 **if** $QVars(\boldsymbol{inv}) = \varnothing$ **then** $Cand(\boldsymbol{inv}) \leftarrow$ SAMPLE$(G(\boldsymbol{inv}))$;
7 **else** $Cand(\boldsymbol{inv}) \leftarrow \forall QVars(\boldsymbol{inv}) \,.$
 $QVars(\boldsymbol{inv}) \in progressRange(\boldsymbol{inv}) \implies$ SAMPLE$(G(\boldsymbol{inv}))$;
8 $ExtCand \leftarrow$ EXTEND$(S, \{\boldsymbol{inv}\}, Cand, Lemmas)$;
9 **if** $\forall \boldsymbol{inv}' \in \mathcal{R} \,.\, ExtCand(\boldsymbol{inv}') = \top$ **then** $G(\boldsymbol{inv}) \leftarrow$ BLOCK$(G, Cand, \boldsymbol{inv})$;
10 **else**
11 **for each** $\boldsymbol{inv}' \in \mathcal{R}$ **do**
12 $Lemmas(\boldsymbol{inv}') \leftarrow Lemmas(\boldsymbol{inv}') \cup \{ExtCand(\boldsymbol{inv}')\}$;
13 $G(\boldsymbol{inv}') \leftarrow$ BLOCK$(G, ExtCand, \boldsymbol{inv}')$;
14 **return** $\langle$SAT$, Lemmas\rangle$;

A naive idea for getting a range formula and a cell property is to sample them separately, and then to bind them together using some $QVars(\boldsymbol{inv})$. But it would result in a large search space. Alg. 1 gives a more tailored procedure on the matter. The central role in this process is taken by an analysis of the loop counters which are used to access array elements (line 3). This analysis is performed once for each loop before the main verification process, and thus its results are reused in all iterations of the verification process.

Our algorithm identifies $QVars(\boldsymbol{inv})$ by creating a fresh variable for each counter, including counters of nested loops (line 5). It then generates range formulas based on the results of the analysis (line 6) such that: 1) the range formula itself is an inductive invariant for $\boldsymbol{inv}$, and 2) the range formula is expressed over the initial values of counters of $\boldsymbol{inv}$ and the counters themselves.

Algorithm 3: WEAKEN(S', $\mathcal{R}'$, *Cand*, *Lemmas*)

Input: CHCs S' over $\mathcal{R}'$, candidates $Cand(\boldsymbol{inv})$; learned $Lemmas(\boldsymbol{inv})$ for each $\boldsymbol{inv} \in \mathcal{R}'$
Output: weakened *Cand*

1 $toRecheck \leftarrow \bot$;
2 **for all** $C \in S'$ **do**
3 **if** $\bigwedge_{\ell \in Lemmas(rel(src(C)))} \ell(args(src(C))) \wedge Cand(rel(src(C)))(args(src(C))) \wedge body(C) \not\Rightarrow Cand(rel(dst(C)))(args(dst(C)))$ **then**
4 **if** isFinalizedArrayCand($Cand, rel(dst(C))$) **then**
5 $Cand(rel(dst(C)))) \leftarrow$ getRegressCand($Cand, rel(dst(C))$);
6 **else**
7 $Cand(rel(dst(C))) \leftarrow \top$;
8 $toRecheck \leftarrow \top$;
9 **break**;
10 **if** *toRecheck* **then return** WEAKEN($S', \mathcal{R}'$, *Cand*, *Lemmas*);
11 **else return** *Cand*;

Algorithm 4: EXTEND($S, \mathcal{R}$, *Cand*, *Lemmas*), cf [17].

Input: CHCs S over $\mathcal{R}$; $\mathcal{R}' \subseteq \mathcal{R}$, candidates $Cand(\boldsymbol{inv})$; learned $Lemmas(\boldsymbol{inv})$ for each $\boldsymbol{inv} \in \mathcal{R}'$
Output: extended *Cand*

1 $Cand \leftarrow$ WEAKEN($S', \mathcal{R}'$, *Cand*, *Lemmas*);
2 **for all** $C \in S$ **s.t.** $rel(src(C)) \in \mathcal{R}'$ **do**
3 $Cand(rel(dst(C))) \leftarrow$ PROPAGATE(C, *Cand*);
4 $Cand \leftarrow$ EXTEND($S, \mathcal{R}' \cup \{rel(dst(C))\}$, *Cand*, *Lemmas*);
5 **return** *Cand*;

Finally, only a cell property is going to be produced from the grammar $G(\boldsymbol{inv})$, constructed from the seeds (recall Sect. 3.1), in which all counters are replaced by the corresponding variables from $QVars(\boldsymbol{inv})$ (line 7). Thus, the only part of the candidate formula where the counter can appear is the range formula.

Once grammars, *QVars*, and ranges are detected, our approach proceeds to sample candidates and to check them with an SMT solver. The general flow of this algorithm is illustrated in Alg. 2. For each $\boldsymbol{inv} \in \mathcal{R}$, it initiates a set $Lemmas(\boldsymbol{inv})$ (line 2). Then it iteratively guesses lemmas until a combination of them is inductive and safe, or a search space is exhausted (lines 3-4).

Compared to the baseline approach from [17], our new algorithm fixes a shape for the candidates for arrays. At the same time, it permits to sample quantifier-free candidates (line 6): they could be either formulas over counters or any other variables in the loop, or even formulas over isolated array elements (if, e.g., accessed by a constant). Then (line 8), Alg. 2 propagates candidates through all available implications in CHCs using quantifier elimination and identifies lemmas among the candidates. This step is similar to the baseline approach from [17], but

for completeness of presentation, we provide the pseudocode in Alg. 4 and Alg. 3. The only differences are 1) in the implementation of the candidate propagation for array candidates and 2) in the weakening of failed candidates (both in Alg. 3, to be discussed in Sect. 4.3 and Sect. 4.4, respectively).

Both successful and unsuccessful candidates are "blocked" from their grammars to avoid re-sampling them in the next iterations. This fact together with the property of grammars being conjunction-free gives the main hint for proving the following theorem.

Theorem 1. *Alg. 2 always makes a finite number of iterations, and if it returns with* SAT *then the CHC system is satisfiable.*

Next section discusses a particular instantiation of important subroutines that make our invariant synthesizer effective in practice.

4 Design Choices

Our main contribution is a completely automated algorithm for finding quantified invariants for array-handling loops. In this section, we first show how by exploiting the program syntax we can identify ranges of elements accessed in each loop (Sect. 4.1). Second, we present an intuitive justification to why our candidates can often be proved as lemmas by an off-the-shelf SMT solver (Sect. 4.2). Finally, we extend our algorithm to handle more complicated cases of multiple loops (Sect. 4.3 – Sect. 4.4), and benchmarks of the tiling [9] technique, which are adapted from the industrial code of battery controllers (Sect. 4.5).

4.1 Discovery of progress lemmas

We start with the simplest scenario of a single loop handling just one array. Let S be a system of CHCs over a set of uninterpreted relation symbols $\mathcal{R}$. Let $\boldsymbol{inv} \in \mathcal{R}$ correspond to a loop, in which arrays are accessed using some counter variable i (counters are automatically identified by posing and solving queries of forms (1) and (2)).

Recall that we do not necessarily require the array elements to be accessed directly by i, and we allow an access function f to identify relationships between i and an index of the accessed element. However, we assume that the counter is unique in the loop because it is the case in most of the practical applications. In principle, our algorithm can be extended to loops handling several independent counters (although it is rare in practice), with the help of additionally discovered lemmas that describe relationships among counters. We leave a discussion about this to future work.

Definition 4. *A* range *of* $\boldsymbol{inv}$ *and a counter* i *is a formula over* $IntVars(\boldsymbol{inv})$ *and a free variable* v *having form* $L < v \wedge v < U$*, such that either of formulas* $L < i$ *or* $i < U$ *is a lemma for* $\boldsymbol{inv}$*. A* progress lemma *is either a formula* $L < v \wedge v < i$ *(if* $L < i$ *is a lemma), or a formula* $i < v \wedge v < U$ *(if* $i < U$ *is a lemma).*

Both ranges and progress ranges can be identified statically. Let C_1 and C_2 be two CHCs, such that $\boldsymbol{inv} = \mathit{rel}(\mathit{dst}(C_1)) = \mathit{rel}(\mathit{src}(C_2)) = \mathit{rel}(\mathit{dst}(C_2))$ and $\boldsymbol{inv} \neq \mathit{rel}(\mathit{src}(C_1))$. It is common in practice that $\mathit{body}(C_1)$ identifies a symbolic bound b on the initial value of i: it could be either a lower bound (if i increments in $\mathit{body}(C_2)$) or an upper bound (if i decrements). In this case, a progress range of $\boldsymbol{inv}$ is simply computed as a lemma for $\boldsymbol{inv}$ over i and b. A range of $\boldsymbol{inv}$ can often be constructed as a conjunction of the progress range with the negation of the termination condition of $\mathit{body}(C_2)$.[4]

Example 2. For the CHC-encoding of the program is shown in Fig. 2, the ranges of $\boldsymbol{inv}_1, \boldsymbol{inv}_2$ and $\boldsymbol{inv}_3$ are all equal to $-1 < v < N$. The progress range of $\boldsymbol{inv}_1$ is $i < v < N$, and the progress ranges of $\boldsymbol{inv}_2$ and $\boldsymbol{inv}_3$ are $-1 < v < i$.

We call candidates, that use progress ranges in their left sides, *progress candidates*:

$$\forall \vec{q}\,.\, \mathit{progressRange}(\boldsymbol{inv})(\vec{q}) \implies \mathit{cand}$$

where $\vec{q} = \mathit{QVars}(\boldsymbol{inv})$ and *cand* is a quantifier-free formula over $\mathit{QVars}(\boldsymbol{inv}) \cup \mathit{IntVars}(\boldsymbol{inv})$. As can be seen from Alg. 1, all sampled candidates are progress candidates. However, during the next steps of the algorithm (i.e., propagation and weakening) we will use other kind of candidates (namely, *regress* and *finalized*, see Sect. 4.3 and Sect. 4.4 respectively).

If a progress candidate is proven inductive, we call it a *progress lemma*.

4.2 SMT-based inductiveness checking

We rely on recent advances of SMT solving to identify successful candidates, a conjunction of which is directly used to prove the desired safety specification. In general, solving quantified formulas for validity is a hard task, however, in certain cases, the initiation and inductiveness queries can be simplified and reduced to a sequence of (sometimes even quantifier-free) formulas over integer arithmetic. We illustrate such proving strategy, inspired by the *tiling* approach [9], on the following example.

Example 3. Recall the CHC system from Fig. 2. Consider a progress candidate $\forall j\,.\, i < j < N \implies m \leq A[j]$ for $\boldsymbol{inv}_1$. Checking its initiation (i.e., for CHC $\mathbf{A}$) requires deciding validity of the following quantified formula:

$$i' = N' - 1 \wedge m' = 0 \implies \Big(\forall j\,.\, i' < j < N' \implies m' \leq A'[j]\Big) \tag{3}$$

The range formula $i' < j < N'$ simplifies to $N' - 1 < j < N'$, which is always false, makes formula (3) always valid.

[4] Thus, we explicitly require guards of loops to have the forms of an inequality, which is the most common array access pattern.

Checking the inductiveness of the candidate (i.e., for CHC $\mathbf{B}$) boils down to solving a more complicated formula:

$$\Big(\forall j\,.\,i < j < N \implies m \le A[j]\Big) \wedge \\ i \ge 0 \wedge m' = ite(m > A[i], A[i], m) \wedge i' = i - 1 \implies \\ \Big(\forall j\,.\,i' < j < N \implies m' \le A[j]\Big) \qquad (4)$$

Although quantifiers are present on both sides of (4), proving its validity is not hard. Indeed, the query is reducible to two implications:

$$\Big(\forall j\,.\,i < j < N \implies m \le A[j]\Big) \wedge m' = ite(m > A[i], A[i], m) \implies m' \le A[i]$$

$$\Big(\forall j\,.\,i < j < N \implies m \le A[j]\Big) \wedge \\ m' = ite(m > A[i], A[i], m) \implies \Big(\forall j\,.\,i < j < N \implies m' \le A[j]\Big)$$

The former does not require any information about $A[i+1], \ldots, A[N-1]$, so the entire quantified conjunction is ignored, and $A[i]$ could be replaced by a fresh integer variable. The latter is trickier: it requires to prove that if all elements in a range are greater or equal than m, then they are also greater or equal to $ite(m > A[i], A[i], m)$. This again is reduced to a quantifier-free formula over integer arithmetic:

$$m \le A[j] \wedge m' = ite(m > A[i], A[i], m) \implies m' \le A[j]$$

Thus, because formulas (3) and (4) are valid, the progress candidate is proved a progress lemma.

In general, we cannot always conduct proofs that easily. Often, the prerequisite for success is the commonality of an access function f in the candidate and the body of the CHC. Fortunately, our algorithm ensures that all access functions used in the candidates are borrowed directly from bodies of CHCs. Thus, in many cases, FreqHorn is able to check large amounts of candidates quickly.

4.3 Strategy of lemma propagation

In this subsection, we identify a useful strategy for propagation of quantified lemmas through adjacent CHCs in the given system, inspired by [17]. Let some $\boldsymbol{inv}_1 \in \mathcal{R}$ have the following lemma:

$$\forall \vec{q}\,.\,\rho(\vec{q}) \implies \ell$$

where $\vec{q} = QVars(\boldsymbol{inv}_1)$, formula ρ over $\vec{q} \cup IntVars(\boldsymbol{inv}_1)$ is either a range or a progress range, and ℓ is over $\vec{q} \cup Vars(\boldsymbol{inv}_1)$. Let then a CHC C be such that $rel(src(C)) = \boldsymbol{inv}_1$ and $rel(dst(C)) = \boldsymbol{inv}_2$, and its body be $\varphi(\vec{x}_1, \vec{x}_2)$.

Definition 5. Forward propagation *of lemma* $\forall \vec{q}\,.\,\rho(\vec{q}) \implies \ell$ *through* C *gives a formula of the following form:*

$$\forall \vec{q}\,.\,(\exists \vec{x}_1\,.\,\rho(\vec{q})(\vec{x}_1) \wedge \varphi(\vec{x}_1, \vec{x}_2)) \implies (\exists \vec{x}_1(\vec{x}_1, \vec{q})\,.\,\ell \wedge \varphi(\vec{x}_1, \vec{x}_2))$$

Example 4. Recall the example from Fig. 2 and the following lemma for $\boldsymbol{inv}_1$:

$$\forall j\,.\,i < j < N \implies m \leq A[j]$$

The body of $\mathbf{C}$ is $i < 0 \wedge i' = 0$, thus the forward propagation gives the following formula:

$$\forall j\,.\,(\exists i\,.\,i < j < N \wedge i < 0 \wedge i' = 0) \implies (\exists i\,.\,m \leq A[j] \wedge i < 0 \wedge i' = 0)$$

Applying quantifier elimination to both sides of the implication, we get the following formula:

$$\forall j\,.\,0 \leq j < N \implies m \leq A[j]$$

Note that this formula is not going to be immediately learned as a lemma, but instead should be checked by the solver for inductiveness. Intuitively, such a candidate represents some facts about array elements that were accessed during a loop that has terminated. If after the propagation it appeared that the candidate uses the entire range then we refer to such candidate to as a *finalized* candidate.

4.4 Weakening strategy

Whenever a finalized candidate cannot be proven inductive, we often do not want to withdraw it completely. Instead, our algorithm runs *weakening* and proposes *regress candidates*. The main idea is to calculate a range of elements which have not been touched by the loop yet. This is an inverse of the procedure outlined in Sect. 4.1.

Definition 6. *Given* $\boldsymbol{inv} \in \mathcal{R}$*, its* $\mathit{Range}(\boldsymbol{inv})$ *and* $\mathit{progressRange}(\boldsymbol{inv})$ *formulas, we call a* regress range *a formula of the following kind:*

$$\mathit{regressRange}(\boldsymbol{inv}) \stackrel{\text{def}}{=} \mathit{Range}(\boldsymbol{inv}) \wedge \neg \mathit{progressRange}(\boldsymbol{inv})$$

We call candidates that use regress ranges in their left sides as *regress candidates*. Clearly, a regress candidate is weaker than the corresponding finalized candidate. Thus, from the failure to prove inductiveness of the finalized candidate it does not follow that the regress candidate is not inductive; and it makes sense to try proving it in the next iteration.

4.5 Learning from sub-ranges

In complicated scenarios of loops with multiple iterators, multiple array variables or multiple access functions, the iterative process of lemma discovery, might end

```
int N = nondetInt();
int *A = nondetArray(2*N);
int val1 = 1, val2 = 3, m = nondetInt();
for (int i = 1; i ≤ N; i++) {
  if (m < val2) A[2*i-2] = val2; else A[2*i-2] = 0;
  if (m < val1) A[2*i-1] = val1; else A[2*i-1] = 0; }
for (int i = 0; i < 2*N; i++) assert(A[i]==0 || A[i] ≤ m);
```

Fig. 3: Learning from sub-ranges.

up in a large number of quantified formulas and get lost while checking a candidate for inductiveness (recall Sect. 4.2). To overcome current limitations in existing SMT solvers, it appeared to be useful to help the solver while generalizing learned lemmas. In particular, a property could be learned for two subranges of an array, and then combined in the following way:

Lemma 1. *Let for some* $\boldsymbol{inv} \in \mathcal{R}$ *two lemmas be of the following kind:*

$$\forall \vec{q}\,.\,\rho_1(\vec{q}) \implies \ell \qquad\qquad \forall \vec{q}\,.\,\rho_2(\vec{q}) \implies \ell$$

Then, the following is also a lemma for $\boldsymbol{inv}$*:*

$$\forall \vec{q}\,.\,\rho_1(\vec{q}) \vee \rho_2(\vec{q}) \implies \ell$$

Example 5. Fig. 3 shows a program from the tiling benchmark suite [9]. If lemmas $\forall j\,.\,0 < j < N \implies A[2*j-1] = 0 \vee A[2*j-1] \leq m$ and $\forall j\,.\,0 < j < N \implies A[2*j-2] = 0 \vee A[2*j-2] \leq m$ are discovered, then formula $\forall j\,.\,0 \leq j < 2*N-1 \implies A[j] = 0 \vee A[j] \leq m$ is also a lemma.

5 Evaluation

We have implemented our algorithm on top of the FreqHorn[5] tool. It takes a system of CHCs with arrays as input and performs an enumerative search as presented in Sect. 4. The tool uses Z3 [12] to solve SMT queries.

We have evaluated FreqHorn on 137 satisfiable CHC-translations of publicly available C programs (whose assertions are safe) taken from the SVCOMP ReachSafety Array subcategory and literature. These programs include variations of standard array copying, initializing, maximum, minimum, sorting, and tiling benchmarks. Among these 137 benchmarks, 79 have a single loop, and 58 have multiple loops, including 7 that have nested loops. These programs are encoded using the theories of Arrays, Linear (LIA) and Non-linear Integer Arithmetic (NIA). Our experiments have been performed on an Ubuntu 18.04 machine running at 2.5 GHz and having 16 GB memory, with a timeout of 100 seconds for

[5] The source code and benchmarks are available at `https://github.com/grigoryfedyukovich/aeval/tree/rnd`.

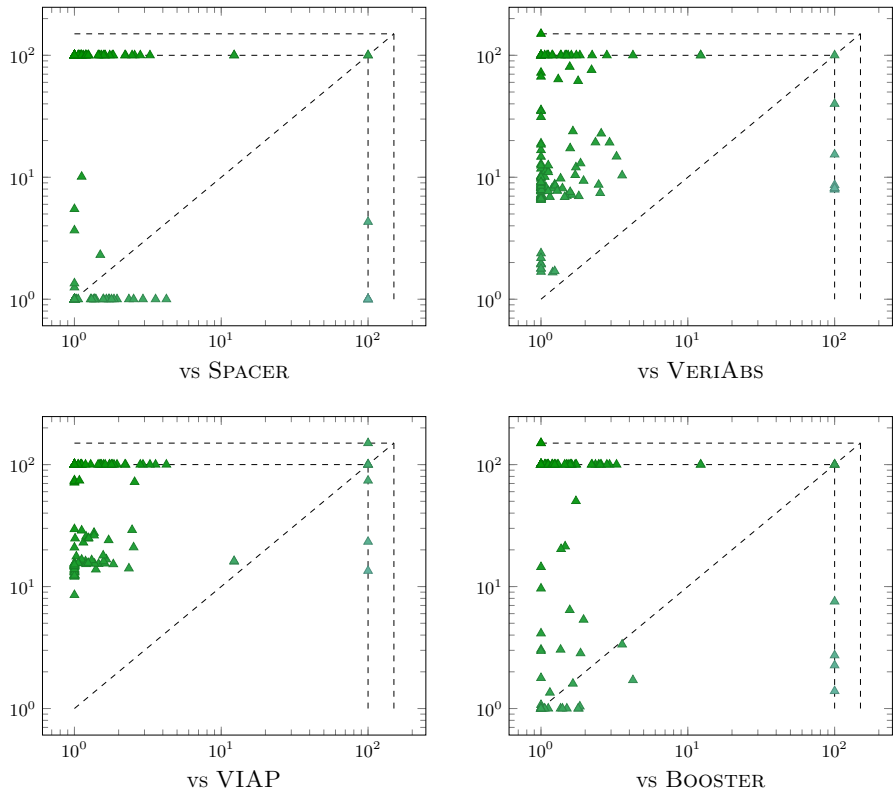

Fig. 4: FreqHorn vs competitors. Each point in a plot represents a pair of the run times (sec × sec) of FreqHorn (x-axis) and a competitor (y-axis). Timeouts are placed on the inner dashed lines; false alarms, unsupported cases, and crashes are on the outer dashed lines.

every benchmark. FreqHorn solved 129 benchmarks within the timeout, of which 73 solved benchmarks had a single loop and 56 had multiple loops.

We have compared our tool with Spacer (Z3 v4.8.3) [26], that implements a recent QUIC3 [22] algorithm, Booster (v0.2) [2], VIAP(v1.0) [35], and VeriAbs (v1.3.10) [11]. The last two tools performed well in the ReachSafety Array subcategory at SVCOMP 2019[6]. Fig. 4 gives a comparison of FreqHorn timings against timings of these tools.[7]

Compare to 129 benchmarks solved by FreqHorn, only 81 were solved by Spacer, 108 – by VeriAbs, 70 – by VIAP, and 48 – by Booster.

FreqHorn solved 54 benchmarks on which Spacer diverged. Our intuition is that Spacer works poorly on programs with non-deterministic assignments and NIA operations, which our tool can handle.

FreqHorn solved 27 benchmarks on which VeriAbs diverged. VeriAbs failed to solve programs with nested loops and when array values were depen-

[6] `https://sv-comp.sosy-lab.org/2019/results/results-verified/`.
[7] The time taken for every benchmark is available at: `http://bit.ly/2VS5Mtf`.

dent on access indices. Furthermore, it decided one of the programs as unsafe, Time-wise, FreqHorn significantly outperformed VeriAbs on all benchmarks. Importantly, the short time taken by FreqHorn includes the time for generating a checkable witness – quantified invariant – an essence that VeriAbs cannot produce by design. On the other side, VeriAbs solved several benchmarks after merging loops. No quantified invariant satisfying the FreqHorn's restrictions exists for these benchmarks before this program transformation.

FreqHorn solved 60 programs on which VIAP diverged. VIAP decided one program as unsafe. There were no programs on which FreqHorn took more time than VIAP. Finally, FreqHorn solved 83 programs on which Booster diverged. And again, Booster decided two programs as unsafe.

6 Related Work

Our algorithm for quantified invariant synthesis extends the prior work on checking satisfiability of CHCs [16,15,17], where solutions do not permit quantifiers. It works in a similar – enumerate-and-check – manner, but there are two crucial changes: 1) introduction of quantifiers, to formulate hypotheses over a subset of array indices, and 2) a generalization mechanism, to derive properties that may hold over the entire range of array indices.

Many existing approaches for verifying programs over arrays are extensions of well-known techniques for programs over scalar variables to quantified invariants. For example, by extending predicates with Skolem variables in predicate abstraction [30], by exploiting the MCMT [19] framework in lazy abstraction with interpolants [1] and its integration with acceleration [2], and, recently, QUIC3 [22], that extends IC3 [8,14] to universally quantified invariants. Apart from the skeletal similarity, however, these approaches rely on orthogonal techniques.

Partitioning of arrays has also been used to infer invariants in many different ways. It refers to splitting an array into symbolic segments, and may be based on syntax [20,23,25] or semantics [10,31]. Invariants may be inferred for each segment separately and generalized for the entire array. The partitioning need not be explicit, as in [13]. However, most of these techniques (except [31,13]) are restricted to contiguous array segments, and work well when different loop iterations write to disjoint array locations or when the segments are non-overlapping. Tiling [9], a property-driven verification technique, overcomes these limitations for a class of programs by inferring array access patterns in loops. But identifying tiles of array accesses is itself a difficult problem, and the approach is currently based on heuristics developed by observing interesting patterns.

There are a number of approaches that verify array programs without inferring quantified invariants explicitly. A straightforward way is to smash all array elements into a single memory location [4], but it is quite imprecise. Every array element might also be considered a separate variable, but it is not possible with unknown array sizes. There are also techniques that abstract an array to a fixed number of elements, e.g. k-distinguished cell abstraction [33,32] and

k-shrinkability [29,24]. Such abstractions usually reduce array modifying loops with unknown bounds to a known, small bound. It may even be possible to get rid of such loops altogether, by accelerating (computing transitive closures of) transition relations involving array updates in that loop [7]. Along similar lines, VIAP [35] resorts to reasoning with recurrences instead of loops. It translates the input program, including loops, to a set of first-order axioms, and checks if they derive the property. But all these techniques do not obtain quantified invariants explicitly, unlike ours. Besides, many of these transformations produce an abstraction of the original program, i.e., they do not preserve safety.

Alternatively, there are approaches that use sufficiently expressive templates to infer quantified invariants over arrays [21,5,27]. However, the templates need to be supplied manually. For instance, [6] uses a template space of quantified invariants and reduces the problem to quantifier-free invariant generation. Thus, universally quantified solutions for unknown predicates in a CHC system may be obtained by extending a generic CHC solver to handle quantified predicates. Learning need not be limited to user-supplied templates; one may do away with the templates entirely and learn only from examples and counterexamples [18]. Alternatively, [36] chooses a template upfront and refurbishes it with constants or coefficients appearing in the program source. Similarly, [28] proposes to infer array invariants without any user guidance or any user-defined templates or predicates. Their method is based on automatic analysis of predicates that update an array and allows one to generate first-order invariants, including those that contain alternations of quantifiers. But it does not work for nested loops. By comparison, our technique supports multiple as well as nested loops, enables candidate propagation between loops and, more importantly, generates the grammar automatically from the syntactical constructions appearing in the program's source.

7 Conclusion

We have presented a new algorithm to synthesize quantified invariants over array variables, systematically accessed in loops. Our algorithm implements an enumerative search that guesses invariants based on syntactic constructions which appear in the code and checks their initiation, inductiveness, and safety with an off-the-shelf SMT solver. Key insights behind our approach are that individual accesses to array elements performed in the loop can be generalized to hypotheses about entire ranges, and the existing SMT solvers can be used to validate these hypotheses efficiently. Our implementation on top of a CHC solver FreqHorn confirmed that such strategy is effective on a variety of practical examples. In a vast majority of cases, our tool outperformed competitors and provided checkable guarantees that prevented from reporting false positives.

Acknowledgements This work was supported in part by NSF Grant 1525936. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect those of the NSF.

References

1. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy abstraction with interpolants for arrays. In *LPAR*, volume 7180 of *LNCS*, pages 46–61. Springer, 2012.
2. F. Alberti, S. Ghilardi, and N. Sharygina. Booster: An acceleration-based verification framework for array programs. In *ATVA*, LNCS, pages 18–23. Springer, 2014.
3. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE, 2013.
4. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends in Programming Languages*, 2(2-3):71–190, 2015.
5. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, volume 4349 of *LNCS*, pages 378–394. Springer, 2007.
6. N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In F. Logozzo and M. Fähndrich, editors, *SAS*, volume 7935 of *LNCS*, pages 105–125. Springer, 2013.
7. M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic verification of integer array programs. In *CAV*, volume 5643 of *LNCS*, pages 157–172. Springer, 2009.
8. A. R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
9. S. Chakraborty, A. Gupta, and D. Unadkat. Verifying array manipulating programs by tiling. In *SAS*, volume 10422 of *LNCS*, pages 428–449. Springer, 2017.
10. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. *POPL*, pages 105–118, 2011.
11. P. Darke, S. Prabhu, B. Chimdyalwar, A. Chauhan, S. Kumar, A. Basakchowdhury, R. Venkatesh, A. Datar, and R. K. Medicherla. VeriAbs: Verification by Abstraction and Test Generation - (Competition Contribution). In *TACAS, Part II*, volume 10806 of *LNCS*, pages 457–462. Springer, 2018.
12. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
13. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, volume 6012 of *LNCS*. Springer, 2010.
14. N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, pages 125–134. IEEE, 2011.
15. G. Fedyukovich and R. Bodík. Accelerating Syntax-Guided Invariant Synthesis. In *TACAS, Part I*, volume 10805 of *LNCS*, pages 251–269. Springer, 2018.
16. G. Fedyukovich, S. Kaufman, and R. Bodík. Sampling Invariants from Frequency Distributions. In *FMCAD*, pages 100–107. IEEE, 2017.
17. G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta. Solving Constrained Horn Clauses Using Syntax and Data. In *FMCAD*, pages 170–178. IEEE, 2018.
18. P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning Universally Quantified Invariants of Linear Data Structures. In *CAV*, volume 8044 of *LNCS*, pages 813–829. Springer, 2013.
19. S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In *IJCAR*, volume 6173, pages 22–29. Springer, 2010.

20. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. *POPL*, pages 338–350, 2005.
21. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246. ACM, 2008.
22. A. Gurfinkel, S. Shoham, and Y. Vizel. Quantifiers on demand. In *ATVA*, volume 11138 of *LNCS*, pages 248–266, 2018.
23. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. *PLDI*, pages 339–348, 2008.
24. A. Jana, U. P. Khedker, A. Datar, R. Venkatesh, and N. C. Scaling bounded model checking by transforming programs with arrays. In *LOPSTR*, volume 10184 of *LNCS*, pages 275–292. Springer, 2016.
25. R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV*, volume 4590 of *LNCS*, pages 193–206. Springer, 2007.
26. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV*, volume 8559 of *LNCS*, pages 17–34, 2014.
27. S. Kong, Y. Jung, C. David, B.-Y. Wang, and K. Yi. Automatically Inferring Quantified Loop Invariants by Algorithmic Learning from Simple Templates. In *APLAS*, volume 6461 of *LNCS*, pages 328–343. Springer, 2010.
28. L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009.
29. S. Kumar, A. Sanyal, R. Venkatesh, and P. Shah. Property checking array programs using loop shrinking. In *TACAS, Part I*, volume 10805 of *LNCS*, pages 213–231. Springer, 2018.
30. S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, volume 2937 of *LNCS*, pages 267–281. Springer, 2004.
31. J. Liu and X. Rival. Abstraction of arrays based on non contiguous partitions. In *VMCAI*, volume 8931 of *LNCS*, pages 282–299. Springer, 2015.
32. D. Monniaux and F. Alberti. A simple abstraction of arrays and maps by program translation. In *SAS*, volume 9291 of *LNCS*, pages 217–234. Springer, 2015.
33. D. Monniaux and L. Gonnord. Cell morphing: From array programs to array-free horn clauses. In *SAS*, volume 9837 of *LNCS*, pages 361–382. Springer, 2016.
34. S. Prabhu, K. Madhukar, and R. Venkatesh. Efficiently learning safety proofs from appearance as well as behaviours. In *SAS*, volume 11002 of *LNCS*, pages 326–343. Springer, 2018.
35. P. Rajkhowa and F. Lin. Extending VIAP to handle array programs. In *VSTTE*, volume 11294 of *LNCS*, pages 38–49. Springer, 2018.
36. R. Sharma and A. Aiken. From Invariant Checking to Invariant Inference Using Randomized Search. In *CAV*, volume 8559 of *LNCS*, pages 88–105. Springer, 2014.