Abstract. Proving properties of systems frequently requires the user to provide hand-written invariants and pre- and post-conditions. A significant body of work exists attempting to automate the generation of loop invariants in code, but the state of the art cannot yet tackle the combination of quantifiers and potentially unbounded data structures. We present SynRG, a synthesis algorithm based on restricting the synthesis problem to generate candidate solutions with quantification over a finite domain, and then generalizing these candidate solutions to the unrestricted domain of the original specification. We give an exemplar of our method that generates invariants with quantifiers over array indices. Our algorithm is, in principle, able to synthesize predicates with arbitrarily many levels of alternating quantification. We report experiments that require invariants with one alternation that are already out of reach of all existing solvers.

1 Introduction

Symbolic methods for proving properties of programs use inductive arguments to reason about execution traces of potentially unbounded length. Furthermore, many programs or software systems have a potentially unbounded state space, which might arise from unbounded data structures in memory or an unbounded number of threads or an unbounded numbers of machines on a network. Correctness proofs for non-trivial properties of such systems require quantification over the state-holding elements, e.g., the elements of a potentially unbounded array, i.e., we require invariants of the form

$$\forall i \in \{0, \ldots, n\}. I(i)$$

where $n$ denotes the number of components and where $I(i)$ is a property of the state of the component with index $i$. The community has invested a large amount of effort into this problem, and identified a broad variety of special cases in which reachability properties of parametric systems are decidable. For the unrestricted case, the community has devised numerous heuristic methods for guessing and possibly refining the predicate $I$ \cite{19, 20, 29}.

This paper addresses the general case in which we wish to synthesize an expression with quantifier alternation that satisfies some logical specification. For instance, consider the task of synthesizing a safety invariant for the loop
shown in Figure 1. The two arrays \( A \) and \( B \) are initially equal. The elements of \( A \) are swapped nondeterministically. We use \( * \) to indicate a non-deterministic choice. The assertion in this snippet checks that for every element of \( A \) there exists an equal element in \( B \). The most natural way to formalize this property is to use a formula with one quantifier alternation. The given property is inductive as-is, and it is easy for state-of-the-art SMT solvers to show validity of the corresponding verification conditions.

```c
int A[], int B[], int c;
assume: \( \forall i . A[i] == B[i] \)
assume: \( c > 0 \)
while(*)
x'='; y'=';
A'[x] = A[y]
A'[y] = A[x]
swap (A[x], A[y])
\( \forall i A'[i] = A[i] + c \wedge B'[i] = B[i] + c \)
assert: \( \forall x \exists y . A[x] == B[y] \)
```

**Fig. 1.** An example of a safety invariant that is naturally specified using one quantifier alternation

There are many verification problems like the one above, in many forms and shapes. The issue is, of course, that while it is easy to verify that the assertion above holds, it is difficult for humans to write, and thus, there is a need for algorithmic methods that identify these invariants automatically. Nevertheless, to the best of our knowledge, no existing synthesis engine is able to generate the inductive invariant for the example above. When formulated as a synthesis problem, CVC4 [3] version 1.8 returns “unknown”, and when formulated as a Constrained Horn Clause problem Z3 [12,18] (version 4.8.7) also returns “unknown”.

Furthermore, for the most general case described above, where we synthesize an invariant with quantifier alternation, we allow quantifiers in the specification as well as in the invariant. Use-cases that benefit from specifications with quantification are nested loops, the initialization of arrays, or asserting properties about arrays. Consider the example given in Figure 2. A reachability problem such as this one requires reasoning with alternating quantification: a candidate inductive invariant for this loop must satisfy the base case of the induction proof, i.e., that \( \forall x (init(x) \implies inv(x)) \), where \( x \) denotes the set of all possible inputs to the program, \( init(x) \) asserts that the initial conditions of the program hold, and \( inv(x) \) is the inductive invariant we wish to synthesize. This is equivalent to

\[
\forall x \ (\neg init(x) \lor inv(x)).
\]
For Figure 2, the initial conditions are \((c < 0) \land \forall i A[i] \geq 0\), and thus we obtain the following base case of the induction proof:

\[
\forall A, c (c < 0 \lor (\exists i A[i]) < 0) \lor \text{inv}(A, c).
\]

Note the quantifier alternation in \(\forall A, c \exists i\). Nested quantifiers are not supported by recent works in synthesis of quantified invariants [14, 15], and are explicitly prohibited in the Constrained-Horn Clause [13] and in the SyGuS competition [2] formats. Consequently, the simple inductive invariant for this example cannot be synthesized by the leading solvers in these competition: CVC4 [28] version 1.8 [pre-release]; the Z3 [18] Horn solver version 4.8.7; FreqHorn [14] or Quic3 [15].

In this paper we present SYN RG: Synthesis via Restriction and Generalization. The algorithm can be viewed as a form of CounterExample Guided Inductive Synthesis (CEGIS) [31], a well-known paradigm for solving program synthesis problems of the form \(\exists P \forall x \sigma(P, x)\), where \(\sigma\) and \(P\) are the specification and the program to be synthesized. Current state-of-the-art techniques can only handle quantifier-free \(\sigma\) and \(P\). The original CEGIS algorithm performs synthesis in two stages: first it synthesizes a candidate solution that works for a simpler version of the specification (i.e., it works for a subset of the possible inputs) and then verifies whether that candidate solution works for the full specification (i.e., for all the possible inputs). Using this principle, we perform synthesis by restricting a specification that contains quantifiers over infinite domains into a simpler quantifier-free specification on a restricted domain. We synthesize candidate solutions for this simpler specification, and then attempt to generalize them to satisfy the full specification. SYN RG integrates well with existing syntax-guided synthesis solvers: by reducing complex synthesis specifications with nested quantification over infinite domains to quantifier-free specifications over restricted domains, we take advantage of state-of-the-art synthesis solvers for quantifier-free theories.

To evaluate and compare our approach experimentally we present an instance of the algorithm that solves formulas with quantifiers over the indices of arrays. We evaluate this implementation on a set of benchmarks containing both hand-crafted examples and benchmarks adapted from the software verification competition [5]. We observe that the algorithm outperforms the state-of-the-art solvers; we hypothesize that this is owed to the fact the array theory enjoys the small model property [8], i.e., the validity of quantifier-free array formulas can be determined in a finite domain by computing a set of indices that is often surprisingly small.

We conjecture that some variant of this algorithmic framework would be extensible to any theory over infinite domains if they have the small model property. That is, for any formula \(f\) over an infinite domain, where a formula \(f_b\) can be constructed over some finite domain and \(f_b\) is satisfiable iff \(f\) is satisfiable.

**Contributions:** The key contribution of this paper is SYN RG: a general program synthesis algorithm which can synthesize expressions containing quantifiers, which satisfy specifications containing quantifiers. As far as we know, this is the
first solver general enough to automatically synthesize expressions containing alternating quantifiers.

```
int A[];
int c = \text{\texttt{ast}};
assume: \quad c > 0
assume: \forall i . A[i] \geq 0

while(*)
\quad \forall i A'[i] = A[i] + c
assert: \quad \neg \exists i A[i] < 0
```

**Fig. 2.** Simple running example

**Motivation:** Our work is motivated by the desire to automatically synthesize invariants for proofs such as the one presented by Subramanyan et al. [32], which reason about unbounded memory. This task is beyond the reach of state-of-the-art synthesis tools due to the quantification over infinite domains and the sheer size of the formal proofs. We believe our algorithm is a significant step towards being able so synthesize such invariants by tackling the quantification, if not yet the scalability.

![SynRG: algorithm for synthesis of programs with quantifiers and arrays](image)

**Fig. 3.** SynRG: algorithm for synthesis of programs with quantifiers and arrays
2 Background

2.1 Program synthesis

Program synthesis can be framed as the following existential second-order logic formula:

$$\exists P. \forall x. \sigma(x, P)$$

where $P$ ranges over functions (where a function is represented by the program computing it) and $x$ ranges over ground terms. We interpret the ground terms over some domain, and we use $I$ to denote the set of all possible inputs to the function, and so in the formula above $x \in I$. In our presentation of the algorithm we allow $P$ and $\sigma$ to contain linear integer arithmetic, arrays and quantification over array indices. We restrict the array indices to terms in Presburger arithmetic.

CEGIS \cite{31} is an algorithmic framework often used to tackle the program synthesis problem described above. It consists of two phases; the synthesis phase and the verification phase: Given the specification of the desired program, $\sigma$, the inductive synthesis procedure produces a candidate program $P$ that satisfies the specification $\sigma(x, P)$ for all $x$ in a set of inputs $I_G$ (which is a subset of $I$). The candidate program $P$ is passed to the verification phase, which checks whether $P$ is a full solution, i.e., it satisfies the specification $\sigma(x, P)$ for all $x$. If so, we have successfully synthesized a full solution and the algorithm terminates. Otherwise, the verifier passes a counterexample, i.e., the satisfying assignment to $x$, which is added to the set of inputs $I_G$ passed to the synthesizer, and the loop repeats.

2.2 Safety invariants

The synthesis of safety invariants can be formulated as a general program synthesis problem described above. It consists of two phases; the synthesis phase and the verification phase: Given the specification of the desired program, $\sigma$, the inductive synthesis procedure produces a candidate program $P$ that satisfies the specification $\sigma(x, P)$ for all $x$ in a set of inputs $I_G$ (which is a subset of $I$). The candidate program $P$ is passed to the verification phase, which checks whether $P$ is a full solution, i.e., it satisfies the specification $\sigma(x, P)$ for all $x$. If so, we have successfully synthesized a full solution and the algorithm terminates. Otherwise, the verifier passes a counterexample, i.e., the satisfying assignment to $x$, which is added to the set of inputs $I_G$ passed to the synthesizer, and the loop repeats.

3 Algorithm overview

The classic CEGIS algorithm breaks down a hard problem into an easier problem for synthesis, and then attempts to generalize a solution to the easier problem to
the full problem. That is, the synthesis phase attempts to synthesize a solution that works only for a subset of inputs, i.e., it solves $\exists P \forall x \in \mathcal{I} \sigma(P, x)$. The verification phase then checks whether a candidate solution satisfies the specification for all possible inputs.

We apply a similar approach to CEGIS: We take a specification $\sigma$, which contains quantification over the infinite domain of arrays. We restrict the domain of arrays in $\sigma$, generating a restricted-domain specification $\sigma^b$, which considers only $b$ elements of each array. This specification then only contains quantification over finite domains, which can be removed with exhaustive quantifier instantiation. We synthesize a solution to $\sigma^b$ with an existing state-of-the-art Syntax-Guided Synthesis solver. We use $P^b$ to denote a solution to $\exists P \forall x \sigma^b(P, x)$. We then attempt to generalize $P^b$, to give a candidate solution to the full specification, which we denote $P^*$, and verify whether $P^*$ is a solution to $\sigma$, i.e., $\exists x \neg \sigma(P^*, x)$. If we fail to find a solution, we can either return to the synthesis phase and look for another solution to the same $\sigma^b$, or increase the bound $b$ used for generating $\sigma^b$. An overview of this general algorithmic framework is illustrated in Figure 3.

Verification phase: The verification phase, using a standard SMT-solver, guarantees the soundness of our approach. Key to the CEGIS algorithm is deciding what to do if there is a counterexample. The default in our implementation is to relax the restriction on the array size, i.e., the arrays in $\sigma^b$ increase in size, but are still smaller than the arrays in $\sigma$.

Synthesis phase: The synthesis phase solves the formula $\exists P^b \forall x \sigma^b(P^b, x)$. This formula contains only theories permitted in SyGuS-IF [2], the universal input format for SyGuS solvers, and we are able to apply standard synthesis-guided synthesis algorithms. The synthesis problem is not, in general, decidable, and the syntax-guided synthesis algorithms that complete this task are not complete [10], and consequently SYNRG is also not complete.

Restriction and Generalization: In the following sections, we discuss the fragments of array logic for which such a restricted-domain specification is guaranteed to exist. We then give details of a specific instance of this algorithmic framework, where the restricted-domain specification considers explicitly $b$ elements of every array, and the generalization approach is based on applying a series of syntactic rules.

4 Exploiting the small model property

The algorithm we have presented relies on the existence of a restricted-domain program $P^b$ that can be generalized to the unrestricted domain. In this section we prove that such a restricted-domain program is guaranteed to exist for a restricted fragment of array theory.

However, as illustrated by the dashed line in Figure 3, it may be interesting to use the counterexamples to block specific restricted-domain candidate solutions, and repeat the synthesis step with the same restricted-domain specification.
4.1 Fragment of array theory

Consider the case where the specification $\sigma$ and the solution to be synthesized $P$ are restricted in such a way that the verification query can be written as a boolean combination of array properties [8].

**Definition 1.** Array property: an array theory formula is called an array property [8,21] iff it is of the form

$$\forall i_1 \ldots \forall i_k \in T_I, \phi_I(i_1 \ldots i_k) \implies \phi_V(i_1 \ldots i_k),$$

where $T_I$ is the index theory, $i_1, \ldots, i_k$ are array index variables, $\phi_I$ is the index guard and $\phi_V$ is the value constraint, and the formula also satisfies the following conditions:

- The index guard $\phi_I$ must follow the grammar
  
  $is\ guard ::= is\ guard \land is\ guard \mid is\ guard \lor is\ guard \mid is\ term \mid is\ term = is\ term$

  $is\ term ::= i_1 \mid \ldots \mid i_k \mid term$

  $is\ term ::= integer\ constant \mid integer\ constant \ast index\ identifier \mid term + term$

  The index identifier used in term must not be one of the universally quantified variables.

- The index variables $i_1, \ldots, i_k$ can only be used in array read expressions.

If this is the case, then we know that the verification query is solvable [8] and, crucially, that there is a finite set of index terms such that instantiating universally quantified index variables from only this set is sufficient for completeness. This set of index terms, $\mathcal{R}$ is made up of all expressions used as an array index term, or inside index guards, in $\phi$ that are not quantified variables.

The example shown in Figure 4 shows an invariant synthesis problem with one possible target solution. The verification query for checking the inductive step of this target solution is:

$$\forall x(x < i \implies a[x] = 0)$$

$$\land a'[i] = 0 \land \forall j \neq i.a'[j] = a[j]$$

$$\land \exists x.(x \leq i \land a'[x] \neq 0).$$

We instantiate the existential quantifier with a fresh variable $z$:

$$\forall x(x < i \implies a[x] = 0)$$

$$\land a'[i] = 0 \land \forall j \neq i.a'[j] = a[j]$$

$$\land \exists x.(x \leq i \land a'[x] \neq 0).$$

The set of index terms $\mathcal{R}$ is \{i, z\}. If we replace the universal quantifier $\forall i P(i)$ with a conjunction $\bigwedge_{i \in \mathcal{R}} P(i)$, we get the following quantifier-free formula:

$$(z < i \implies a[z] = 0)$$

$$\land (a'[i] = 0) \land (z \neq i \implies a'[z] = a[z])$$

$$\land (z \leq i \land a'[z] \neq 0).$$
Thus, it is sufficient to verify the candidate $P$ by only considering two elements of the array, $a[z]$ and $a[i]$, provided we consider the cases where $z < i$, $z = i$, and $z > i$. There is a restricted-domain candidate program $P^b$ and a restricted-domain specification $\sigma^b$ such that $\exists x - \sigma^b(P^b, x)$ is equisatisfiable with the original verification formula. In this case the restricted-domain specification for the inductive step would be $(P^b(a) \land a'[i] = 0 \land (z \neq i \implies a'[z] = a[z])) \implies P^b(a')$ and the restricted-domain program $P^b(a)$ is $z < i \implies a[z] = 0$. Given that we only need to reason about array indices $a[z]$ and $a[i]$, if we find a solution that works for an arrays of size 2, we will have a solution that we can generalize to the infinite case by a procedure detailed in Section 6 that is based on reversing the steps we used to obtain the restricted-domain specification.

However, without knowing the solution $P$ in advance, we cannot determine the size of the set $I$ needed for the verification query. Consequently, we use a heuristic approach in this work where we begin with two array elements and increase the number of elements if we are unable to find a solution. The exact process we use is detailed in Section 5. We note that the restricted-domain synthesis query itself does not fall within a decidable fragment [10], even for this array fragment, and so, perhaps unsurprisingly, the unrestricted-domain synthesis problem is in general undecidable.

```c
int A[];
int i = 0;

while(i<50)
    A'[i]=0; i'=i+1;

invariant:
    \forall x, (x < i) \implies A[x] = 0

assert: x < 50 \implies A[x] = 0
```

Fig. 4. Example safety invariant expressed in array property fragment

4.2 Beyond the array property fragment

The array property fragment in Definition 1 is restrictive, but expressive enough for many benchmarks we adapted from the SV-Comp [5]. As an exemplar of a synthesis problem that falls outside this fragment, consider that we have synthesized an invariant that states that array $A$ contains at least one element that is not in array $B$. That is $\exists i \forall j A[i] \neq B[j]$. The verification of this invariant will include the subformula $\exists A, B \forall i \exists j A[i] = B[j]$. This formula falls outside the decidable fragment described above, and formulas of the form $\exists x \forall i \exists j$, where $x$ is some array and $i$ and $j$ are indices, are in general undecidable. This can be shown by reduction to the termination of integer loops [8]. Consequently we are
unable to show that a finite model exists. However, our experimental evaluation shows that the approach we present in this paper is a heuristic that can be used to some problems that fall outside of the decidable fragment.

5 Restriction of $\sigma$ to $\sigma^b$

We aim to generate a modified specification $\sigma^b$ that considers a finite set of $b$ index terms for each array. We do this by bounding the length of the arrays in the specification to length $b$, by replacing any predicate $e$ in $\sigma$ that reasons about an array index $i$, with an implication $(0 \leq i < b) \implies e$. The restricted-domain specification is guaranteed to always be weaker than the original specification, i.e., it permits more solutions than the original specification, and the sequence of restrictions we build as we increase $b$ is monotonic.

The algorithm for bounding arrays, shown in Algorithm 1, applies these rules recursively on the syntax tree of each constraint. This method of considering only the first $b$ elements of the arrays may require us to consider a larger specification than strictly necessary. For instance, suppose we have a specification that reasons only about array element $x[99]$. Our algorithm will not work until we have increased $b$ to 100, and we then have to solve a synthesis query that considers all elements $x[0]..x[99]$. Future work will explore more sophisticated heuristics for generating this specification.

Algorithm 1: Pseudocode for generating $\sigma^b$

| Data: $\sigma$, bound b |
|--------------------------|
| Result: $\sigma^b$       |
| 1 idx: list of array indices; |
| 2 Qidx: list of quantified array indices; |
| 3 $\sigma^b \leftarrow \emptyset$; |
| 4 idx$\leftarrow \emptyset$; |
| 5 Qidx$\leftarrow \emptyset$; |
| 6 for constraint $c \in \sigma$ do |
| 7 $c^b \leftarrow \text{boundQuantification}(c, b, idx, Qidx)$; |
| 8 $c^b \leftarrow \{(idx < b) \implies c^b\}$; |
| 9 idx$\leftarrow \emptyset$; |
| 10 $\sigma^b \leftarrow c^b$; |
| 11 return $\sigma^b$ |

Remove quantification: Once we have obtained this restricted-domain specification for $\sigma$, all quantification is now over finite domains. We can hence use exhaustive quantifier instantiation to remove all quantifiers over array indices and replace universal quantifiers with conjunctions and existential quantifiers with disjunctions. This exhaustive quantifier instantiation is possible only because we have bound the size of the arrays to make the data types finite.
Algorithm 2: boundQuantification: Algorithm for bounding an expression

Data: expression $e$, bound $b$, idx, Qidx
Result: finite-domain expression $e^b$, updated idx and Qidx

1 if $e$ is $\forall$ or $e$ is $\exists$ then
2 $\text{idx} \leftarrow \text{Qidx}$;
3 $\text{Qidx} \leftarrow \emptyset$;
4 for operand $o \in e$.operands do
5 $o \leftarrow \text{boundQuantification}(o, b, \text{idx}, \text{Qidx})$;
6 if ($e$ is $\forall$ or $e$ is $\exists$) $\land$ Qidx $\neq \emptyset$ then
7 $e$.Predicate $\leftarrow \{(\text{Qidx} < B) \implies e$.Predicate$\}$;
8 $\text{Qidx} \leftarrow \emptyset$;
9 else if $e$ is an array element then
10 $\text{Qidx} \leftarrow \text{getIndex}(e)$;
11 return $e$;

Running example: Consider the example presented in Figure 2. The constraint asserting that the invariant holds at the initial conditions would be:

$$(c > 0) \land (\forall i . A[i] \geq 0) \implies \text{inv} (c, A) \land$$

$$\text{inv} (c, A) \land (\forall i . A'[i] = A[i] + c) \implies \text{inv} (c, A') \land$$

$$\text{inv} (c, A) \implies \neg \exists i . A[i] < 0.$$

If we apply these steps to our running example, bounding the arrays to size two, we get the following constraints:

$$(c > 0) \land (\bigwedge_{0 \leq i < 2} A[i] \geq 0) \implies \text{inv} (c, A) \land$$

$$\text{inv} (c, A) \land (\bigwedge_{0 \leq i < 2} A'[i] = A[i] + c) \implies \text{inv} (c, A') \land$$

$$\text{inv} (c, A) \implies \neg \bigvee_{0 \leq i < 2} A[i] < 0.$$

(1)

6 Generalization

Assuming the synthesis block has found a solution $P^b$, we now attempt to generalize this solution to obtain a candidate $P^*$ that may satisfy the full specification, by introducing quantifiers in the place of conjunctions or disjunctions.

We implement a syntax-based quantifier introduction procedure based on identifying conjunctions or disjunctions of predicates that use array indices. We describe the steps for universal quantifiers, and note that existential quantifiers can be introduced by treating disjunctions in the same way. In order to introduce a universal quantifier, in place of an expression:
Algorithm 3: removeQuantifiers: This pseudocode is simplified to handle only quantifiers that bind only to a single variable

```plaintext
Data: expression e, bound b
Result: quantifier-free expression

1 for operand o ∈ e.operands do
  removeQuantifiers(o, b);
3 if e is ∀ or e is ∃ then
  4 v ← e.Binding;
  5 P ← e.Predicate;
  6 if e is ∀ then
    7 e_{qf} ← conjunction;
  8 else
    9 e_{qf} ← disjunction;
10 for 0 ≤ i < b do
  11 P_i ← replaceVarWithConstant(P, v, i);
  /* add P_i to the operands of e_{qf}, which is a conjunction or disjunction. */
  12 e_{qf}.operands ← P_i;
13 return e_{qf}
14 return e
```

- the expression must be a conjunction of predicates that reason about array elements;
- replacing an array index in the same location in each predicate with a new variable must result in equisatisfiable predicates;
- and the conjunction must cover all possible indices of the (bounded) array.

These three items are sufficient for generalizing expressions that are part of the array fragment given in Section 4 which disallows Presburger arithmetic on quantified index-variables, and allows quantified variables to only be used in array read operations.

**Definition 2.** Two predicates $\phi_1$ and $\phi_2$ are matching predicates if $\phi_1$ contains an array read operation $A[c]$ and $\phi_2$ contains an array read operation $A[d]$, where $c$ and $d$ are constants, and if we replace both $c$ and $d$ with the same fresh variable $z$, $\phi_1$ and $\phi_2$ are equisatisfiable.

Given a predicate $\phi$ which contains an array read $A[c]$, we use $\phi(z)$ as short-hand for the same predicate with the constant $c$ replaced by a fresh variable. This relationship is transitive, if $\phi_1$ and $\phi_2$ are matching predicates, and $\phi_3$ and $\phi_2$ are matching predicates then $\phi_1$ and $\phi_3$ are matching predicates. It is also commutative.

A conjunction $C$ over predicates $\phi_0, ..., \phi_n$ can be replaced with a universal quantifier over $\phi(z)$ if the constant array indices we replaced in $\phi_0, ..., \phi_n$
to obtain \( \phi(z) \) span the full range of the bounded arrays in the finite-domain specification.

**Definition 3.** A conjunction \( C \) over predicates \( \phi_0, ..., \phi_n \) is equisatisfiable with the expression \( \forall z. \phi_0(z) \), in the finite-domain with bounded arrays, if \( \phi_0, ..., \phi_n \) are matching predicates, and the original constants span the full range of the finite-domain bounded arrays.

A similar statement to definition 3 can be written for disjunctions and existential quantifiers. Using definition 2 and definition 3, we are able to apply a procedure to replace conjunctions and disjunctions iteratively on the syntax-tree of the finite-domain candidate program \( P^b \), starting at the leaf nodes and working upwards, as shown in Algorithm 4.

**Running example:** Consider a possible \( P^b \) for our running example, in a finite-domain with arrays of length 2: \((A[0] \geq 0 \land A[1] \geq 0 \land c \geq 0)\). This is a conjunction of predicates:

\[
\phi_0 = A[0] > 0, \quad \phi_1 = A[1] > 0, \quad \phi_2 = c > 0
\]

If we replace the constant indices in the array read operations in \( \phi_0 \) and \( \phi_1 \), we can see that the two predicates are matching, and the constants span the full range of the finite-domain array. \( \phi_2 \) does not match any other predicate. We thus replace the first conjunction with a universal quantifier, and the expression becomes \((\forall z. A[z] > 0) \land (c > 0)\).

**Beyond the decidable array fragment:** We add two more checks that allow us to handle limited cases outside the decidable array property fragment, specifically we consider the case where Presburger arithmetic is applied to quantified index-variables and where limited cases where quantified index-variables are used outside of array read operations. That is:

- if more than one element of the same array is indexed, we look for constant difference relationships between the array elements indexed in the predicate, and check these relationships are the same across all predicates;
- and if the predicate contains constants of the same type as the array index that are not used for indexing arrays, we look for constant adjustment relationships between the constants and the array indices, and check if these are the same across all predicates.

We extend our definition of matching predicates to allow constant difference relationships between the array elements indexed in the predicate:

**Lemma 1.** Two predicates \( \phi_1 \) and \( \phi_2 \) are matching predicates if \( \phi_1 \) contains array read operations \( A[c_0], ..., A[c_n] \) and \( \phi_2 \) contains array read operations \( A[d_0], ..., A[d_n] \), and if we replace \( c_0, ..., c_n \) and \( d_0, ..., d_n \) with the same set of expressions \( z + e_0, ..., z + e_n \) where \( z \) is the same fresh variable and \( e_0, ..., e_n \) is a set of constants, and \( \phi_1 \) and \( \phi_2 \) are equisatisfiable.
We use \( \phi(z) \) to indicate the expression obtained by replacing multiple array read operations in \( \phi \) with a set of expressions \( z + e_0, ..., z + e_n \). We create a similar rule for constants of the same type as the array indices, that are used outside of array read operations.

**Lemma 2.** Two predicates \( \phi_1 \) and \( \phi_2 \) contain array read operations \( A[c_0], ..., A[c_n] \) and constants \( x_0, ..., x_n \), and \( \phi_2 \) contains array read operations \( A[d_0], ..., A[d_n] \) and constants \( y_0, ..., y_n \). We replace \( c_0, ..., c_n \) and \( d_0, ..., d_n \) with the same set of expressions \( x + e_0, ..., x + e_n \) where \( x \) is the same fresh variable and \( e_0, ..., e_n \) is a set of constants. We replace \( x_0, ..., x_n \) and \( y_0, ..., y_n \) with \( x + f_0, ..., x + f_n \) where \( x \) is the same variable as before, and \( f_0, ..., f_n \) is a set of constants. If \( \phi_1 \) and \( \phi_2 \) are now equisatisfiable, the two predicates are matching predicates.

We use \( \phi(z) \) to indicate the expression obtained by replacing multiple array read operations in \( \phi \) with a set of expressions \( z + e_0, ..., z + e_n \) and multiple constants with \( z + f_0, ..., z + f_n \). A conjunction \( C \) that reasons about predicates \( \phi_0, ..., \phi_n \) can be replaced with the expression \( \forall z., \phi_0(z) \) if \( \phi_0, ..., \phi_n \) are matching predicates and the constants that we replaced with \( z + 0 \) span the full range of the restricted domain. That is, definition 3 still applies. Consider the example \( (A[0] < A[1]) \land (A[1] < A[2]) \), in the finite-domain of arrays of length 2:

\[
\phi_0 = (A[0] < A[1]), \quad \phi_1 = (A[1] < A[2]),
\]

If we replace the array reads with \( A[z + 0], A[z + 1] \), then, by definition 1, the two predicates are matching, and the constants that we replaced by \( z + 0 \) span the full range of the restricted domain. We can thus replace the conjunction with the expression \( \forall z' A[z] < A[z + 1] \).

**Nested quantifiers:** Although nested quantifiers are outside of the decidable array fragment, transforming a finite-domain candidate solution \( P^k \) to a solution with nested quantifiers requires no further transformation rules. Algorithm 4 applies the transformations recursively and, given an expression as input, begins by calling itself on all of the operands of that expression, and by doing so is able to introduce nested quantifiers.

**Example:** Consider the expressions \( (A[0] = B[0] \lor A[0] = B[1]) \land (A[1] = B[0] \lor A[1] = B[1]) \). The syntax tree for this expression is shown in Figure 5. The key comparison the algorithm makes are:

1. Compare the disjunction operands:
   \( (A[0] = B[0]) \) and \( (A[0] = B[1]) \).
   Replace with:
   \( \exists z_1 A[0] = A[z_1] \).

2. Compare the disjunction operands:
   \( (A[1] = B[0]) \) and \( (A[1] = B[1]) \).
   Replace with:
   \( \exists z_2 A[1] = B[z_2] \).
3. Compare the conjunction operands:
\[ \exists z_1 A[1] = B[z_1] \text{ and } \exists z_2 A[0] = B[z_2] \]
Replace with:
\[ \forall z_3 \exists z_1 A[z_3] = B[z_1] \]

6.1 Extensions:
The generalization phase is incomplete. There is scope for syntactic generalization of further expressions outside of the decidable array fragment. For instance, Array indices as expressions outside of Presburger arithmetic. Future work will also explore framing the generalization procedure as a synthesis problem, which may allow us to use information from multiple finite-domain candidate solutions obtained with different bounds.

7 Evaluation
We implement SynRG using CVC4 version 1.8 as the synthesis phase. We use Z3 version 4.8.7 for verification. The communication between the transformation phases and the synthesis phase is done in SyGuS-IF, allowing any existing SyGuS solver to be substituted into the synthesis phase. Furthermore, the verification phase produces standard SMT-lib, allowing any existing SMT solver to be used as a back-end. We evaluate our algorithm on 13 benchmarks adapted from the Software Verification Competition [5] array examples, as well as 15 example problems crafted to test the capabilities of our algorithm. 4 of the benchmarks from SV-comp and 7 of our crafted examples fall outside of the decidable array fragment. The Syntax-Guided Synthesis Competition currently does not use any benchmarks that contain quantifiers, and so our algorithm would have no need to restrict the specification to a finite-domain, and would simply pass the unmodified problem to the synthesis phase shown in Figure 3.

We run Quic3 [15] and the Z3 [18] Horn Solver, both contained within Z3 version 4.8.7, on the examples. Quic3 and the Z3 Horn clause solver officially do not support quantification in specifications, but were able to solve a subset of the benchmarks. SynRG is, however, able to solve significantly more of these
Algorithm 4: generalize: algorithm for reintroducing quantifiers

Data: finite-domain expression $e$

Result: An unrestricted-domain expression

1 for operand $o \in e.\text{operands}$ do
2   generalize($o$);

/* set of matching operands */
3 $Ops \leftarrow \emptyset$;

/* set of sets of matching operands */
4 $Sets \leftarrow \emptyset$;

5 if $e$ is $\land$ or $\lor$ then
6   $Ops \leftarrow e.\text{operand}[0]$;
7   $Sets \leftarrow Ops$;
8   $N \leftarrow e.\text{operands}.\text{size}()$;
9   for $1 \leq i < N$ do
10      placed $\leftarrow$ false
11      for set $\in Sets$ do
12         if compareExpr(set, $e.\text{operand}[i]$) then
13            set $\leftarrow E.\text{operand}[i]$;
14            placed $\leftarrow$ true;
15      if !placed then
16         newSet $\leftarrow e.\text{operand}[i]$ $Sets \leftarrow$ newSet
17
18      result $\leftarrow$ true
19      for set $\in Sets$ do
20         /* Replace array indices with local variables */
21         P $\leftarrow$ replaceIndicesWithVars(set, $v_{loc}$);
22         if $e$ is $\land$ then
23            quantifiedExpr $\leftarrow \forall v_{loc} \ P$
24         else if $e$ is $\lor$ then
25            quantifiedExpr $\leftarrow \exists v_{loc} \ P$
26         result $\leftarrow$ result $\land$ quantifiedExpr
27   result $\leftarrow$ result $\lor$ quantifiedExpr
28 return result;
examples than any of the existing tools, including several examples which require
the synthesis of alternating quantifiers, and is able to solve several benchmarks
for which other solvers return “unknown”. SynRG solves 6 of the benchmarks
adapted from SV-comp and 10 crafted examples, all in under 10 s. This speed is
because a single iteration of our algorithm in general runs in less than a second,
and it typically takes between 1 and 4 iterations to find a model size large enough
for the solution generation by CVC4 to be generalizable to the full unrestricted
domain.

Benchmarks that we are able to solve, that the other solvers cannot, are
typically benchmarks that reason about the full length of arrays, such as the ones
shown in Figure 1 and Figure 2, or where the solution reasons about sections
of arrays but with a variable as a moving boundary between the sections, for
instance, an invariant for a loop that initializes an array might be \( \forall x, (x < i) \implies A[x] = 0 \). The benchmarks we are unable to solve are unsolvable for one
of two reasons:

**Linear relationships between array indices:** Firstly, we were unable to solve
some examples due to constant difference relationships existing between the array
indices that demanded a larger \( \sigma^b \) than we could solve. For example, an invariant
which requires an array element at index \( i \) to be equal to an array element at
index \( i + 10 \) requires a finite-domain specification that allows arrays at least 11
elements long. We could address this challenge by refining the restriction phase
of our algorithm.

**Dependency on CVC4:** Second, our algorithm depends on the abilities of CVC4,
or another synthesis solver, to solve the specification \( \sigma^b \). For some benchmarks,
where the verification query would fall outside the decidable fragment identi-
fied, CVC4 was unable to solve the smallest model we were able to generate
within the 600 s timeout. Since the actual solutions for these small models are
short (typically 3 – 4 operations, reasoning about 2 – 4 array elements), we be-
lieve that a valuable direction for future work would be exploring enumerative
techniques tailored to these types of problems where the search space for an enu-
merative engine is small. These are shown as unsolved by SynRG in the results
table. However, in order to validate our generalization procedure, we also ran
experiments where mocked the expected result from CVC4 and showed that our
generalization process is capable of producing the correct result.

8 Related work

There are many approaches that synthesize invariants containing quantifiers over
array indices, however, none of them allow for quantification in the specification.
Quic3 [15] is an adaptation of IC3 [7] to synthesize quantified invariants, evalu-
ated on array manipulation programs from SV-COMP. Larraz et al. [23] present
an SMT-based array invariant generation approach, which is limited to univer-
sally quantified loop invariants over arrays and scalar variables. FreqHorn [14]
Table 1. Examples solved by each solver. We ran the experiments with a 60 s timeout. We differentiate between unsolved benchmarks that time-out and unsolved benchmarks where the solver returns unknown. SynRG does not implement any way of returning unknown.

| Solver      | Z3-Horn solver Quic3 SynRG |
|-------------|----------------------------|
| **Benchmarks in array property fragment** |                       |
| No. solved | 3/17 6/17 10/17           |
| Number unknown | 1/17 4/17 n/a          |
| Number time-out | 13/17 7/17 7/17      |
| Average solving time | <0.1s <0.1s 0.3s  |
| **Benchmarks not in array property fragment** |                       |
| No. solved | 0/11 0/11 6/11          |
| Number unknown | 1/11 1/11 n/a        |
| Number time-out | 10/11 10/11 5/11     |
| Average solving time | n/a n/a 0.5s |

uses syntax-guided synthesis to synthesize quantified invariants: they identify potentially useful facts about elements accessed in a loop and attempt to generalize these to hypothesis about the entire range of the variables. This is the approach most similar to our work, however the way they identify the range of elements is specific to a loop invariant synthesis problem. Our approach relies on a more general program synthesis phase to identify useful elements and so is not restricted to loop invariant synthesis. FreqHorn also does not permit quantifiers in the specification. It may be possible to integrate more of these approaches in the synthesis phase of Figure 3 after a transformation has been applied to remove quantifiers in the specification.

I4 [24] is an algorithm that uses a similar insight based on finding invariants for small instances of protocols using model-checking, and generalizing them to larger numbers of nodes. Since the approach is based on model-checking, it is limited to invariant generation, whereas our approach can handle more general synthesis cases. They also handle only universal quantifiers over nodes of the distributed protocol, and not quantifier alternations or existential quantifiers.

CVC4 [3] and LoopInvGen [27] both perform well in the syntax-guided synthesis competition, but neither can handle quantifiers in synthesis.

Our algorithm is in part inspired by verification approaches which use the principle of abstracting a verification problem by considering short versions of bit-vectors and arrays [9,30]. Khasidashvili et al. [17] verify equivalences of memory by translation into first-order logic, and note that for some specific designs this falls into a decidable fragment. Verification procedures such as CEGAR [11] iteratively refine an abstraction, and we iteratively refine $\sigma^b$. A key difference is that CEGAR relies on refining the abstraction until it the abstraction is precise enough that a counterexample is valid on the original program. We only refine $\sigma^b$ until it is precise enough that a satisfying assignment $P^b$ can be generalized.
to be a valid solution $P$ for the original specification. The restricted specification $\sigma^b$ is never precise enough that $P^b$ is a valid solution for $\sigma$.

Our motivating examples are based on synthesizing invariants for arrays. However, there are methods for verifying array programs without using loop invariants: Abstraction of the array to a fixed number of elements is used to reduce array modifying loops with unknown bounds to loops with a known, small bound \cite{16,22,25}. An imprecise approach involves abstracting the array so that all array elements appear in a single memory location \cite{4}. Under-approximating loops in programs by acceleration \cite{6,20} may also remove the need for invariants but since it approximates the loops the result is not guaranteed to be correct.

9 Conclusions

We have presented an algorithm which can synthesize expressions containing alternating quantifiers, to specifications containing quantification over arrays. The synthesis algorithm works by bounding unrestricted domains in the synthesis specification, synthesizing a solution to this finite-domain specification, and then attempting to generalize the solution to that to the unrestricted domain, thus exploiting the small model property of arrays. We are able to synthesize expressions that elude existing solvers. Furthermore, our algorithm is a framework that exploits the strengths of existing state-of-the-art solvers, and so as the speed and scalability of quantifier-free syntax-guided synthesis improves, so will the performance of our algorithmic framework.

References

1. Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. SyGuS-Comp 2017: Results and analysis. CoRR, abs/1711.11438, 2017.
2. Rajeev Alur, Dana Fisman, Rishabh Singh, and Abhishek Upadhyay. Syntax guided synthesis competition. http://sygus.seas.upenn.edu/SyGuS-COMP2017.html, 2017.
3. Clark W. Barrett, Haniel Barbosa, Martin Brain, Duligur Ibeling, Tim King, Paul Meng, Aina Niemetz, Andres Nötzli, Mathias Preiner, Andrew Reynolds, and Cesare Tinelli. CVC4 at the SMT competition 2018. CoRR, abs/1806.08775, 2018.
4. Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. Foundations and Trends in Programming Languages, 2(2-3):71–190, 2015.
5. Dirk Beyer. Advances in automatic software verification: SV-COMP 2020. In Armin Biere and David Parker, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 347–367, Cham, 2020. Springer International Publishing.
6. Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konecný, and Tomáš Vojnar. Automatic verification of integer array programs. In CAV, volume 5643 of Lecture Notes in Computer Science, pages 157–172. Springer, 2009.
7. Aaron R. Bradley. Sat-based model checking without unrolling. In VMCAI, volume 6538 of Lecture Notes in Computer Science, pages 70–87. Springer, 2011.
8. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What’s decidable about arrays? In VMCAI, volume 3855 of Lecture Notes in Computer Science, pages 427–442. Springer, 2006.
9. Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. An abstraction-based decision procedure for bit-vector arithmetic. STTT, 11(2):95–104, 2009.
10. Benjamin Caulfield, Markus N. Rabe, Sanjit A. Seshia, and Stavros Tripakis. What’s decidable about syntax-guided synthesis? CoRR, abs/1510.08393, 2015.
11. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. J. ACM, 50(5):752–794, 2003.
12. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In TACAS, volume 4963 of Lecture Notes in Computer Science, pages 337–340. Springer, 2008.
13. Grigory Fedyukovich. CHC competition. https://chc-comp.github.io/2018/format.html, 2018.
14. Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In CAV (1), volume 11561 of Lecture Notes in Computer Science, pages 259–277. Springer, 2019.
15. Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Quantifiers on demand. In ATVA, volume 11138 of Lecture Notes in Computer Science, pages 248–266. Springer, 2018.
16. Anushri Jana, Uday P. Khedker, Advaita Datar, R. Venkatesh, and Niyas C. Scaling bounded model checking by transforming programs with arrays. In LOPSTR, volume 10184 of Lecture Notes in Computer Science, pages 275–292. Springer, 2016.
17. Zurab Khasidashvili, Mahmoud Kinanah, and Andrei Voronkov. Verifying equivalence of memories using a first order logic theorem prover. In FMCAD, pages 128–135. IEEE, 2009.
18. Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. Formal Methods in System Design, 48(3):175–205, 2016.
19. Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwang-Hee Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In APLAS, volume 6461 of Lecture Notes in Computer Science, pages 328–343. Springer, 2010.
20. Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. Formal Methods in System Design, 47(1):75–92, 2015.
21. Daniel Kroening and Ofer Strichman. Decision Procedures - An Algorithmic Point of View, Second Edition. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.
22. Shrawan Kumar, Amitabha Sanyal, R. Venkatesh, and Punit Shah. Property checking array programs using loop shrinking. In TACAS (1), volume 10805 of Lecture Notes in Computer Science, pages 213–231. Springer, 2018.
23. Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio. SMT-based array invariant generation. In VMCAI, volume 7737 of Lecture Notes in Computer Science, pages 169–188. Springer, 2013.
24. Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Kareem A. Sakallah. 14: incremental inference of inductive invariants for verification of distributed protocols. In SOSP, pages 370–384. ACM, 2019.
25. David Monniaux and Francesco Alberti. A simple abstraction of arrays and maps by program translation. In SAS, volume 9291 of Lecture Notes in Computer Science, pages 217–234. Springer, 2015.
26. ThanhVu Nguyen, Matthew B. Dwyer, and Willem Visser. SymInfer: inferring program invariants using symbolic states. In ASE, pages 804–814. IEEE Computer Society, 2017.

27. Saswat Padhi, Rahul Sharma, and Todd D. Millstein. Data-driven precondition inference with learned features. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016, pages 42–56, 2016.

28. Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In CAV (2), volume 11562 of Lecture Notes in Computer Science, pages 74–83. Springer, 2019.

29. Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In NeurIPS, pages 7762–7773, 2018.

30. Rohit Sinha, Cynthia Sturton, Petros Maniatis, Sanjit A. Seshia, and David A. Wagner. Verification with small and short worlds. In FMCAD, pages 68–77. IEEE, 2012.

31. Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In ASPLOS, pages 404–415. ACM, 2006.

32. Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A formal foundation for secure remote execution of enclaves. In ACM Conference on Computer and Communications Security, pages 2435–2450. ACM, 2017.