Putting the Squeeze on Array Programs: Loop Verification via Inductive Rank Reduction

Oren Ish-Shalom[1], Shachar Itzhaky[2], Noam Rinetzky[1], and Sharon Shoham[1]

[1] Tel Aviv University, Israel
[2] Technion, Israel

Abstract. Automatic verification of array manipulating programs is a challenging problem because it often amounts to the inference of inductive quantified loop invariants which, in some cases, may not even be first-order expressible. In this paper, we suggest a novel verification technique that is based on induction on user-defined *rank* of program states as an alternative to loop-invariants. Our technique, dubbed *inductive rank reduction*, works in two steps. Firstly, we simplify the verification problem and prove that the program is correct when the input state contains an input array of length ℓ_{B} or less, using the length of the array as the rank of the state. Secondly, we employ a *squeezing function* Υ which converts a program state σ with an array of length $\ell > \ell_{\mathsf{B}}$ to a state $\Upsilon(\sigma)$ containing an array of length $\ell - 1$ or less. We prove that when Υ satisfies certain natural conditions then if the program violates its specification on σ then it does so also on $\Upsilon(\sigma)$. The correctness of the program on inputs with arrays of arbitrary lengths follows by induction.
We make our technique automatic for array programs whose length of execution is proportional to the length of the input arrays by (i) performing the first step using symbolic execution, (ii) verifying the conditions required of Υ using Z3, and (iii) providing a heuristic procedure for synthesizing Υ. We implemented our technique and applied it successfully to several interesting array-manipulating programs, including a bidirectional summation program whose loop invariant cannot be expressed in first-order logic while its specification is quantifier-free.

1 Introduction

Automatic verification of array manipulating programs is a challenging problem because it often amounts to the inference of inductive quantified loop invariants. These invariants are frequently quite hard to come up with, even for seemingly simple and innocuous program, both automatically and manually. The purpose of this paper is to suggest an alternative kind of correctness witness, which is often simpler than inductive invariants and hence more amenable to automated search.

Loop invariants, the basis of traditional verification approaches, offer an induction scheme based on the time axis, *i.e.*, on the number of loop iterations. We suggest an alternative approach in which induction is carried out on the space

axis, i.e. on a (user-defined notion of the) *rank* (e.g., size) of the program state. This is particularly useful in the setting of infinite-state systems, where the size of the state may be unbounded. In this induction scheme, establishing the induction step relies on a *squeezing function* $\Upsilon : \Sigma \to \Sigma$ (read Υ as *squeeze*) that maps program states to lower-ranked program states (up to a given minima). Roughly speaking, the squeezing function should satisfy the following conditions, described here intuitively and formalized in Definition 3:

- **Initial anchor.** Υ maps initial states to initial states.
- **Simulation inducing.** Υ induces a certain form of simulation between the program states and their squeezed counterparts.
- **Fault preservation.** Υ maps unsafe states to unsafe states.

Our main theorem (Theorem 1) shows that if these conditions are satisfied then P is correct, provided it is correct on its *base*, i.e., on the states with minimal rank. The crux of the proof is that as a consequence of the aforementioned conditions, if P violates its specification on a state σ then it also violates it on $\Upsilon(\sigma)$. Hence, if P satisfies the specification on the base states, by induction it satisfies it on any state.

The function Υ itself can be given by the user or, as we show in Section 4, automatically obtained for a class of array programs which iterate over their input arrays looking for a particular element (e.g., `strchr`) or aggregating their elements (e.g., `max`). In our experiments, we utilized automatically synthesized squeezing functions to verify natural specifications of several interesting array-manipulating programs, some of which are beyond the capabilities of existing automatic techniques. Arguably, the key benefit of the our approach is that the squeezing functions are often rather simple, and thus finding them and establishing that they satisfy the required properties is an easier task than the inference of loop invariants. For example, in the next section we show a program whose loop invariant cannot be expressed in first order logic but can be proven correct using a squeezing function which is first-order expressible, in fact, the reasoning about the automatically synthesized squeezing function is quantifier free.

The last point to discuss is the verification of the program on states in the base of Υ. Here, we apply standard verification techniques but to a simpler problem: we need to establish correctness only on the base, a rather small subset of the entire state space. For example, for the programs in our experiments it is possible to utilize symbolic execution to verify the correctness of the programs on all arrays of length three or less. This approach is effective because on the programs in our benchmarks, the bound on the length of the input arrays also determines a bound on the length of the execution. As this aspect of our technique is rather standard we do not discuss it any further.

Outline. The rest of the paper is structured as follows: We first give an informal overview of our approach (Section 2) which is followed by a formal definition of our technique and a proof of its soundness (Section 3). We continue with a description of our heuristic procedure for synthesizing squeezing functions (Section 4) and a discussion about our implementation and experimental results

(Section 5). We then review closely related work (Section 6) and conclude (Section 7).

2 Overview

In this section, we give a high-level view of our technique.

Running example. Program `sum_bidi`, shown in Figure 1, computes the sum of the input array `a` in two ways: One computation accumulates elements from left to right, and the other — from right to left (assuming that indexes grow to the right). Ignoring its dubious usefulness, `sum_bidi` possesses an intricate property: the variables `l` and `r` are both computed to be the sum of the input array `a`. A natural property one expect to hold when the program terminates is that $l = r$.

The challenge. To verify the aforementioned postcondition when the length of the array is not known and *unbounded*, a loop invariant is often employed. It is important to remember, that a loop invariant must hold on all intermediate loop states — every time execution hits the loop header. For this reason, the loop invariant needed in this case is more involved than the mere assertion $l = r$ that follows the loop. The right side of Figure 1 shows a possible loop invariant for this scenario. Intuitively, the invariant says that `l` and `r` differ by the sum of the elements that they have not yet, respectively, accumulated. Notice that the invariant's formulation relies on a function $\mathrm{sum}(\cdot)$ for arrays (and array slices), the definition of which is also included in the figure. This definition is recursive; indeed, any definition of sum will require some form of recursion or loop due to the unbounded sizes of arrays in program memory. This kind of "logical escalation" (from quantifier-free $l = r$ to a fixed-point logic) makes such verification tasks challenging, since modern solvers are not particularly effective in the presence of quantifiers and recursive definitions.

Moreover, a system attempting to automate discovery of such loop invariants is prone to serious scalability issues since it has to discover the definition of $\mathrm{sum}(\cdot)$ along the way. The subject program `sum_bidi` effectively computes a sum, so this auxiliary definition is at the same scale of complexity as the program itself.

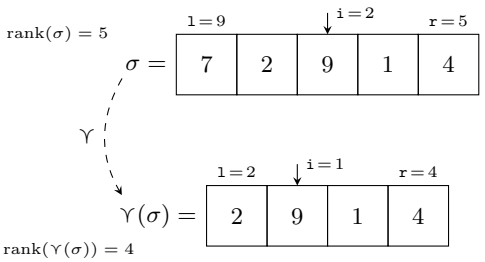
Our approach. We suggest to leverage the semantics already present in the subject program for a more compact proof of safety. Instead of having to summarize partial executions of the program via a loop invariant, we show that the program is correct for all arrays of size $0...r$ for some *base rank* r (the size of the array serves as the rank of the program state), and further show how to derive the correctness of the program for arrays of size $n > r$, from its correctness for arrays of size $n - 1$. To achieve the latter, we rely on a function that "squeezes" states in which the array length is n to states in which the array length is $n - 1$, as we illustrate next.

```
void sum_bidi(int a[], int n) {
    int l = 0, r = 0;
    for (int i = 0; i < n; i++) {
        l += a[i];
        r += a[n - i - 1];
    }
    assert(l == r);
}
```

$$I \mathrel{\widehat{=}} \big(l + \mathrm{sum}(a[i:n]) = r + \mathrm{sum}(a[0:n-i])\big)$$

$$\mathrm{sum}(a[j:k]) \mathrel{\widehat{=}} \textbf{if } j < k \textbf{ then } a[j] + sum(a[j+1:k]) \textbf{ else } 0$$

Fig. 1. A bidirectional sum example and a loop invariant for it.




```
ϒ : {
        if (i > 0) {
            remove(a,0);
            i--;
            l -= a[0];
            r -= a[n - i];
        } else {
            remove(a,0);
        }
    }
```

Fig. 2. A bidirectional sum example and its squeezing function.

Continuing with the example `sum_bidi` described above, we use the function $\Upsilon : \Sigma \to \Sigma$, defined as a code block on the right side of Figure 2, to "squeeze" program states. In this case, the state consists of the variables $\langle$`a`, `n`, `i`, `l`, `r`$\rangle$, and it is squeezed by removing the first element of `a` and adjusting the indices and sums accordingly. The base rank here is $r = 0$, since any non-empty array can be squeezed in this manner. The bottom part of Figure 3 shows the effect of applying Υ to each of the states in the execution trace of `sum_bidi` on the example input `[7,2,9,1,4]`. The first property that is demonstrated by the diagram is the "initial anchor" property, stating that initial states are "squeezed" into initial states. As is obvious from the diagram, the execution on the squeezed array `[2,9,1,4]` is accordingly shorter, so Υ cannot be injective — in this case, $\Upsilon(\sigma_0) = \Upsilon(\sigma_1) = \sigma'_0$. Still, the sequence $\sigma'_0 \to \sigma'_1 \to \sigma'_2 \to \sigma'_3 \to \sigma'_4$ constitutes a valid trace of `sum_bidi`. This is the second property required of Υ, which we refer to as *simulation inducing* and define it formally in the next section.

Now, draw attention to *fault preservation*, the third property required of Υ: whenever a state σ falsifies the safety property φ, denoted $\sigma \not\models \varphi$, it is also the case that $\Upsilon(\sigma)$ falsifies the safety property, i.e. $\Upsilon(\sigma) \not\models \varphi$. In our example, the safety property can be formalized as $\varphi \mathrel{\widehat{=}} (i = n \to l = r)$. The reasoning establishing fault preservation is not immediate but still quite simple: if $\sigma \not\models \varphi$, it means that $i = n$ but $l \neq r$ (at σ). In that case, $a[n - i] = a[0]$; so $l' =$

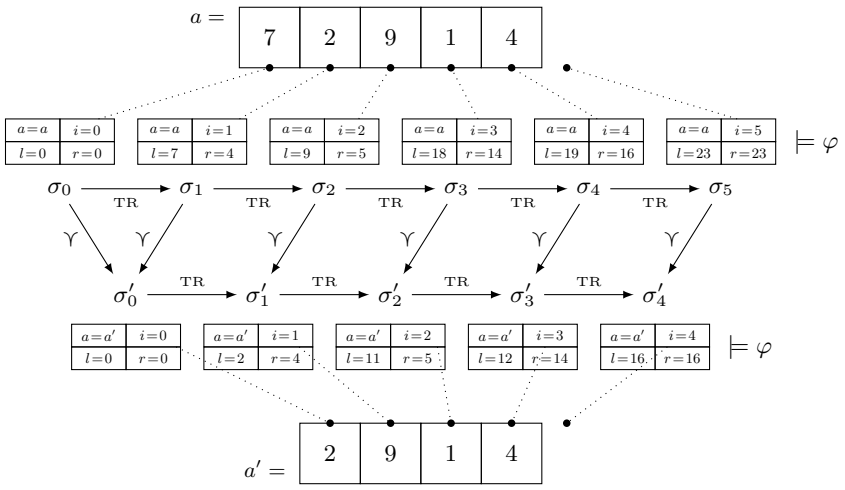


Fig. 3. Example trace of `sum_bidi`, and the corresponding shrunken image.

$l - a[0] \neq r - a[n-i] = r'$, where l', r' are the values of `l` and `r`, respectively, at state $\Upsilon(\sigma)$. Since i and n are both decremented[3] we get $\Upsilon(\sigma) \not\models \varphi$.

In this manner, from *the assumption that* $\Upsilon(\sigma_j)$*, for* $j = 0..5$*, induces a safe trace*, we conclude that σ_j is safe as well. This lends the notion of constructing a proof by induction on the size of the initial state σ_0, provided that Υ cannot "squeeze forever" and that we can verify all the minimal cases more easily, *e.g.* with bounded verification. This is definitely true for `sum_bidi`, since the minimal case would be an empty array, in which the loop is never entered. In some situations the minima contains states with small but not empty arrays. In general, if one can verify that the program is correct when started with a minimal initial state, thus establishing the base case of the induction, our technique would lift this proof to hold for unbounded initial states. In particular, if the length of the program's execution trace can be bounded based on the size of the initial state then bounded model checking and symbolic execution can be lifted to obtain unbounded correctness guarantee.

It is worth mentioning at this point that Υ is in no sense "aware" that it is, in fact, reasoning about sums. It only has to handle scalar operations, in this case subtraction (as the counterpart of addition that occurs in `sum_bidi`; the same will be true for any other commutative, invertible operation.) The folding semantics arises spontaneously from the induction over the size of the array.

Recap. We suggest a novel verification technique that is based on induction on the size of the input states as an alternative to loop-invariants. The technique is based on utilizing a squeezing function which converts high-ranked states

[3] Notice that we assume a positive size ($n > 0$), otherwise the array cannot be squeezed in the first place.

into low-ranked ones, and then applying a standard verification technique to establish the correctness of the program on the minimally-ranked states. In a manner analogous to that which is carried out with "normal" verification using loop invariants, the squeezer has to uphold the three properties described in Section 1, namely *initial anchor*, *simulation inducing*, and *fault preservation*. (See Section 3 for a formal definition.) These properties ensure that the mapping induces a valid reduction between the safety of any trace and that of its squeezed counterpart.

Why bother. The attentive readers may ask themselves, given that both loop invariants and squeezers incur some proof obligations for them to be employed for verification, what benefit may come of favoring the latter over the former. While the verification condition scheme proposed here is not inherently simpler (and arguably less so) than its Floyd-Hoare counterpart, we would like to point out that the *squeezer itself*, at least in the case of `sum_bidi`, *is indeed simpler* than the loop invariant that was needed to verify the same specification. It is simpler in a sense that it resides in a *weaker logical fragment*: while the invariant relies on having a definition of (partial) sums, itself a recursive definition, the squeezer Υ can be axiomatized in a quantified-free formula using a theory of strings (sequences) [8] and linear arithmetic. In Section 4 we take advantage of the simplicity if the squeezing function, and show that it is feasible to *generate it automatically* using a simple enumerative synthesis procedure.

On top of that, it is quite immediate to see that the induction scheme outlined above is still sound even if the properties of Υ (initial anchor, simulation, and fault preservation) only hold for *reachable* states. Obviously, the set of reachable states cannot be expressed directly — otherwise we would have just used its axiomatization together with the desired safety property, making any use of induction superfluous. Even so, if we can acquire any known property of reachable states, *e.g.* through a preliminary phase of abstract interpretation [16], then this property can be added as an assumption, simplifying Υ itself. A keen reader may have noticed that the specification of `sum_bidi` has been written down as $\varphi \mathrel{\widehat{=}} (i = n \rightarrow l = r)$, while a completely honest translation of the assertion would in fact produce a slightly stronger form, $\varphi' \mathrel{\widehat{=}} (i \geq n \rightarrow l = r)$. This was done for presentation purposes; in an actual scenario the "proper" specification φ' is used, and a premise $0 \leq i \leq n$ is assumed. Such range properties are prevalent in programs with arrays and indexes, and can be discovered easily using static analysis, e.g., using the Octagon domain [36].

This final point is encouraging because it gives rise to a hybrid approach, where a *partial* loop invariant is used as a baseline — verified via standard techniques — and is then *stengthened* to the desired safety property via squeezer-based verification. Or, the order could be reversed. There can even be alternating strengthening phases each using a different method. These extended scenarios are potentialities only and are matter for future work.

3 Verification by Induction over State Size

In this section we formalize our approach for verifying programs that operate over states (inputs) with an unbounded size. The approach mimics induction over the state size. The base case of the induction is discharged by verifying the program for executions over "small" low-ranked states (to be formalized later). For the induction step, we need to deduce correctness of executions over "larger" higher-ranked states from the correctness of executions over "smaller" states. This is facilitated by the use of a *simulation-inducing squeezing function* Υ. Intuitively, the function transforms a state σ into a corresponding "smaller" state $\Upsilon(\sigma)$ such that executions starting from the latter simulate executions starting from the former. The simulation ensures that correctness of the executions starting from the smaller state, $\Upsilon(\sigma)$, implies correctness of the executions starting from the larger one, σ.

Transition systems and safety properties. To formalize our technique, we first define the semantics of programs using *transition systems*. The definition is standard.

Definition 1 (Transition Systems). *A* transition system $TS = (\Sigma, \mathit{Init}, \mathit{Tr}, P)$ *is a quadruple comprised of a* universe *(a set of states)* Σ*, a set of* initial states $\mathit{Init} \subseteq \Sigma$*, a* transition relation $\mathit{Tr} \subseteq \Sigma \times \Sigma$*, and a set of* good states $P \subseteq \Sigma$*.*

A *trace* of TS is a (finite or infinite) sequence of states $\tau = \sigma_0, \sigma_1, \ldots$ such that for every $0 \leq i < |\tau|$, $(\sigma_i, \sigma_{i+1}) \in \mathit{Tr}$. In the following, we write Tr^k, for $k \geq 0$ to denote k self compositions of Tr, where $\mathit{Tr}^0 = \mathit{Id}$ denotes the identity relation. That is, $(\sigma, \sigma') \in \mathit{Tr}^k$ if and only if σ' is reachable from σ by a trace of length k (where the length of a trace is defined to be the number of transitions along the trace).

A transition system $TS = (\Sigma, \mathit{Init}, \mathit{Tr}, P)$ is *safe* if all its *reachable states* are good (or "safe"), where the set of reachable states is defined, as usual, to be the set of all states that reside on traces that start from the initial states. A *counterexample trace* is a trace that starts from an initial state and includes a "bad" state, i.e., a state that is not in P. The transition system is safe if and only if it has no counterexample traces.

Simulation inducing squeezing function. To present our technique, we start by formalizing the notion of a simulation-inducing squeezing function (*squeezer* for short).

Definition 2 (Squeezing function). *Let X be a set and $\preceq$ a well-founded partial order over X. Let $B \supseteq \min(X)$ be a* base *for X, where $\min(X)$ is the set of all the minimal elements of X w.r.t. $\preceq$, and let $\rho : \Sigma \to X$ be a* rank *on the program states. A function $\Upsilon : \Sigma \to \Sigma$ is a* squeezing function*, or* squeezer *for short,* with base B *if for every state $\sigma \in \Sigma$ such that $\rho(\sigma) \in X \setminus B$, it holds that $\rho(\Upsilon(\sigma)) \prec \rho(\sigma)$.*

That is, Υ must strictly decrease the rank of any state unless its rank is in the base, B. We refer to states whose size is in B as *base states*, and denote them $\Sigma_B = \{\sigma \in \Sigma \mid \rho(\sigma) \in B\}$. We denote by $\Sigma_{\overline{B}} = \Sigma \setminus \Sigma_B$ the remaining states. Since $\preceq$ is well-founded and all the minimal elements of X w.r.t. $\preceq$ must be in B (additional elements may be included as well), any maximal strictly decreasing sequence of elements from X will reach B (i.e., will include at least one element from B). Hence, the requirement of a squeezer ensures that any state will be transformed into a base state by a *finite* number of Υ applications.

Example 1. In our examples, we use $(\mathbb{N}, \leq)$ as a well-founded set, and define the base as an interval $[0, k]$ for some (small) $k \geq 0$. While it suffices to define $B = \min(\mathbb{N}) = \{0\}$, it is sometimes beneficial to extend the base to an interval since it excludes additional states from the squeezing requirement of Υ (see Section 5). For array-manipulating programs, the rank used is often (but not necessarily) the size of the underlying array, in which case, the "squeezing" requirement is that whenever the array size is greater than k, the squeezer must remove at least one element from the array. For example, for `sum_bidi` (Figure 2), we consider $k = 0$, i.e., the base consists of arrays of size 0, and, indeed, whenever the array size is greater than 0, it is decremented by Υ. For arrays of size 0, Υ behaves as the identity function (this case is omitted from the figure). In addition, whenever the state contains more than one array, we will use the sum of lengts of all arrays as a rank.

Definition 3 (Simulation-inducing squeezer). *Given a transition system $TS = (\Sigma, Init, Tr, P)$, a squeezer $\Upsilon : \Sigma \to \Sigma$ is* simulation-inducing *if the following three conditions hold for every $\sigma \in \Sigma$:*

- **Initial anchor:** *if $\sigma \in Init$ then $\Upsilon(\sigma) \in Init$ as well.*
- **Simulation inducing:** *there exist $n_\sigma \geq 1$ and $m_\sigma \geq 0$ such that if $(\sigma, \sigma') \in Tr^{n_\sigma}$ then $(\Upsilon(\sigma), \Upsilon(\sigma')) \in Tr^{m_\sigma}$, i.e., if σ reaches σ' in n_σ steps, then the same holds for their Υ-images, except that the number of steps may be different.*
- **Fault preservation:** *if $\sigma \notin P$ then $\Upsilon(\sigma) \notin P$ as well.*

The definition implies that $\{(\sigma, \Upsilon(\sigma)) \mid \sigma \in \Sigma\}$ is a form of a "skipping" simulation relation, where steps taken both from the simulated state, σ, and from the simulating state, $\Upsilon(\sigma)$, may skip over some states. This allows the simulated and the simulating execution to proceed in a different pace, but still remain synchronized. In fact, to ensure that we obtain a "skipping" simulation, it suffices to consider a weaker simulation inducing requirement where the parameter m_σ that determines the number of steps in the simulating trace depends not only on σ but also on σ' and may be different for each σ'. Note that for deterministic programs (as we use in our experiments) these requirements are equivalent. Another possible, yet stronger, relaxation is to weaken the requirement that $(\Upsilon(\sigma), \Upsilon(\sigma')) \in Tr^{m_\sigma}$ into $(\Upsilon(\sigma), \Upsilon(\sigma')) \in Tr^{i}$ for some $0 \leq i \leq m_\sigma$.

Example 2. To illustrate the simulation inducing requirement, recall the program `sum_bidi` from Example 1. For the base states ($n = 0$), Υ behaves as the

identity function. Hence, for such states the skipping parameters n_σ and m_σ are both 1 (letting each step be simulated by itself). For non-base states, n_σ, the "skipping" parameter of σ, is still 1, while m_σ, the "skipping" parameter of $\Upsilon(\sigma)$, is 0 if σ is an initial state, and 1 otherwise. This accounts for the fact that Υ truncates the head of the array; hence, the first step in an execution is skipped in the corresponding "squeezed" execution, while the rest of the steps are synchronized in both executions (see Figure 3 for an illustration).

Intuitively, one may conjecture that given a loop that iterates over an array, it will essentially perform fewer iterations when run on $\Upsilon(\sigma)$ than it does on σ, always resulting in $m_\sigma \leq n_\sigma$. The following example shows that this is not necessarily the case.

```
bool is_sorted(int a[], int n) {
  for (int i = 1; i < n; i++)
    if (a[i] < a[i-1])
      return false;
  return true;
}
```

```
ϒ: if (a[n-3] <= a[n-2] &&
       a[n-2] <= a[n-1])
     remove(a,n-1);
   else remove(a,n-4);
```

Fig. 4. Another program with Υ demonstrating a scenario where $n_\sigma < m_\sigma$.

Example 3. The program `is_sorted` (Figure 4) checks whether the input array elements are ascending by comparing all consecutive pairs. Our squeezer (for $n > 3$) checks whether the last three elements form an ascending sequence; if so, removes the last element, otherwise it removes the forth element from the right. Consider the input a=1,0,2,3,1 and the squeezed a'=1,2,3,1. `is_sorted(a)` terminates after one iteration, but `is_sorted(a')` after three iterations. Let $\sigma = \big[a, i \mapsto 1\big]$. The simulation inducing requirement can only be satisfied with $n_\sigma = 1$ and $m_\sigma = 3$. Since $Tr^{n_\sigma}(\sigma) = \big[a, ret = \text{false}\big]$, no smaller value of m_σ can satisfy the requirement that $Tr^{m_\sigma}\big(\Upsilon(\sigma)\big) = \Upsilon\big(Tr^{n_\sigma}(\sigma)\big)$.

Checking if a squeezer is simulation-inducing. The initial anchor and fault preservation requirements are simple to check. To facilitate checking the simulation inducing requirement, we do not allow arbitrarily large numbers n_σ, m_σ but, rather, determine a bound N on the value of n_σ and a bound M on the value of m_σ. This makes the simulation inducing requirement stronger than required for soundness, but avoids the need to reason about pairs of states that are reachable by traces of unbounded lengths (n_σ and m_σ).

Using simulation-inducing squeezer for safety verification. Roughly, the existence of a simulation-inducing squeezer ensures that any counterexample to safety, i.e., an execution starting from an initial state and ending in a *bad* state (a state that falsifies the safety property), can be "squeezed" into a counterexample

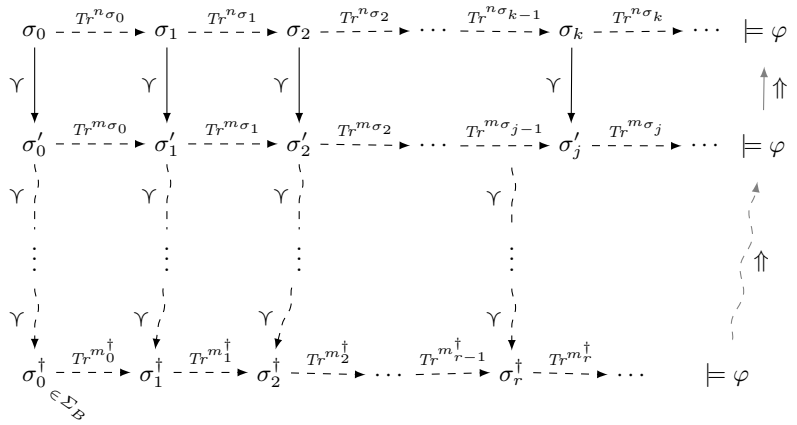


Fig. 5. Soundness proof sketch; an arbitrary trace can be reduced to a low-ranked trace by countable applications of Υ. Since ranks form a well-founded set, a base element is encountered after finitely many such reductions. Arrows with vertical ellipses indicate alternating applications of Υ and Tr^*, except for initial states where Definition 3(1) ensures straight applications of Υ alone.

that starts from a "smaller" initial state. In this sense, the squeezer establishes the induction step for proving safety by induction over the state rank. To ensure the correctness of this argument, we need to require that a "bad" state may not be "skipped" by the simulation induced by the squeezer.

Formally, this is captured by the following definition.

Definition 4. *A transition system* $TS = (\Sigma, \mathit{Init}, \mathit{Tr}, P)$ *is* recidivist *if no "bad" state is a dead-end, i.e.,* $\sigma \notin P \implies \exists \sigma'. (\sigma, \sigma') \in \mathit{Tr}$*, and that transitions leaving "bad" states lead to "bad" states, i.e.,* $\sigma \notin P \wedge (\sigma, \sigma') \in \mathit{Tr} \implies \sigma' \notin P$.

Recidivism can be obtained by removing any outgoing transition of a bad state and adding a self loop instead. Importantly, this transformation does not affect the safety of the underlying program. In our examples, terminal states of the program are treated as self loops, thus ensuring recidivism.

Lemma 1. *Let* $\Upsilon : \Sigma \to \Sigma$ *be a simulation-inducing squeezer for a recidivist transition system* $TS = (\Sigma, \mathit{Init}, \mathit{Tr}, P)$*. For every* $\sigma_0 \in \Sigma$*, if there exists a counterexample that starts from* σ_0*, then there also exists a counterexample that starts from* $\Upsilon(\sigma_0)$.

The proof is constructive: given a counterexample trace from σ_0, we use the simulation-inducing parameters n_σ of the states σ along the trace to divide it into segments such that the first and last state of each segment are the ones used as synchronization points for the simulation and the inner ones are the ones "skipped" over. We then match each segment (σ, σ') with the corresponding trace of length m_σ from $\Upsilon(\sigma)$ to $\Upsilon(\sigma')$, whose existence is guaranteed by the simulation

inducing requirement. The concatenation of these traces forms a counterexample trace from $\Upsilon(\sigma_0)$. Formally:

Proof. Let $\tau = \sigma_0, \sigma_1, \ldots, \sigma_n$ be a counterexample trace starting from an initial state $\sigma_0 \in \mathit{Init}$. If the counterexample is of length 0, then $\Upsilon(\sigma_0)$ is also a counterexample of length 0 (by the initial anchor and fault preservation requirements). Consider a counterexample of length $n > 0$. We show how to construct a corresponding counterexample from $\Upsilon(\sigma_0)$. We first split the indices $0, \ldots, n$ into (overlapping) intervals $I_0, \ldots, I_k$, where $I_0 = 0, \ldots, n_\sigma$, and for every $i \geq 1$, if the last index in I_{i-1} is j for $j < n$, then $I_i = j, \ldots, j + n_{\sigma_j}$. If $j + n_{\sigma_j} \geq n$, then $k := i$. Since TS is recidivist, we may assume, without loss of generality, that $j + n_{\sigma_j} = n$ (otherwise, because TS is recidivist and $\sigma_n \notin P$, we can exploit one of the transitions leaving σ_n, which necessarily exists and leads to a bad state, to extend the counterexample trace as needed.) We denote by $\mathit{first}(I_i)$, respectively $\mathit{last}(I_i)$, the smallest, respectively largest, index in I_i. By the definition of the intervals, for every $0 \leq i \leq k$, we have that $\mathit{last}(I_i) = \mathit{first}(I_i) + n_{\sigma_{\mathit{first}(I_i)}}$. Hence, the simulation inducing requirement for $\sigma_{\mathit{first}(I_i)}$ ensures that there exists a trace of $m_{\sigma_{\mathit{first}(I_i)}}$ steps from $\Upsilon(\sigma_{\mathit{first}(I_i)})$ to $\Upsilon(\sigma_{\mathit{last}(I_i)})$. Since $\sigma_{\mathit{first}(I_0)} = \sigma_0$ and for every $0 < i \leq k$, $\sigma_{\mathit{first}(I_i)} = \sigma_{\mathit{last}(I_{i-1})}$, we can glue these traces together to obtain a trace from $\Upsilon(\sigma_0)$ to $\Upsilon(\sigma_{\mathit{last}(I_i)})$. Finally, it remains to show that $\Upsilon(\sigma_{\mathit{last}(I_k)}) \notin P$. This follows from the fault preservation requirement, since $\mathit{last}(I_k) = n$, hence $\sigma_{\mathit{last}(I_k)} = \sigma_n \notin P$. □

Ultimately, the existence of a simulation-inducing squeezer implies that a counterexample can be "squeezed" to one that starts from a base initial state. Hence, to establish that the transition system is safe, it suffices to check that it is safe when the initial states are restricted to the base states, i.e., to $\mathit{Init} \cap \Sigma_B$.

Theorem 1 (Soundness). *Let $\Upsilon : \Sigma \to \Sigma$ be a simulation-inducing squeezer with base B for a recidivist transition system $TS = (\Sigma, \mathit{Init}, \mathit{Tr}, P)$. If $TS_B = (\Sigma, \mathit{Init} \cap \Sigma_B, \mathit{Tr}, P)$ is safe then TS is safe.*

Proof. Suppose for the sake of contradiction, that $\{\sigma_i\}_{i=0}^{d}$ is a counterexample trace with minimal rank for σ_0 (such a state with a minimal rank exists since $\preceq$ is well-founded). Since TS_B is safe, it must be that $\sigma_0 \in \Sigma_{\overline{B}}$ (since $\sigma_0 \in \mathit{Init}$, while safety of TS_B ensures that no counterexample trace can start from $\mathit{Init} \cap \Sigma_B$). By Lemma 1, we have that $\Upsilon(\sigma_0)$ also has an outgoing counterexample trace. However, since $\sigma_0 \in \Sigma_{\overline{B}}$, we get that $\rho(\Upsilon(\sigma_0)) \prec \rho(\sigma_0)$, in contradiction to the minimality of σ_0. □

In all of our examples, the transitions of TS do not increase the rank of the state. In such cases, we can also restrict the state space of TS_B (and accordingly Tr) to the base states in Σ_B. Furthermore, in these examples, the size of the state (array) also determines the length of the executions up to a terminal state. Hence, bounded model checking suffices to determine (unbounded) safety of TS_B, and together with Υ, also of TS.

Remark 1. As evident from the proof of Theorem 1, it suffices to require that Υ decreases the rank of the *initial* non-base states, and not of all the non-base states.

4 Synthesizing Squeezing Functions

So far we have assumed that the squeezer Υ is readily available, in much the same way that loop invariants are available — typically, as user annotations — in standard unbounded loop verification. As demonstrated by the examples in Sections 2 and 3, Υ is specific to a given program and safety property. Thus, it might be tedious to provide a different squeezer every time we wish to check a different safety property. In this section we show how to lighten the burden on the user by automating the process of obtaining squeezing functions for a class of typical programs that loop over arrays.

The solution for the *squeezer-inference* problem we take in this paper is to utilize a rather standard enumerative synthesis technique of multi-phase generate-and-test: We take advantage of the relative simplicity of Υ and provide a synthesis loop where we generate grammatically-correct squeezing functions and test whether they induce simulation.

4.1 Generate

First we note that while Υ is applied to arbitrary states in Definition 3, it is only required to reduce the rank of non-base states $\sigma \in B$. For states $\sigma \in B$ it is trivial to satisfy all the requirements by defining $\Upsilon(\sigma) = \sigma$. In the sequel, we therefore only consider squeezing functions whose restriction to B is the identity, and synthesize code for squeezing non-base states.

A central insight is that squeezing functions Υ for different programs still have some structure in common: for programs with arrays, squeezing amounts to removing an element from the array, and adjusting the index variables accordingly. Some more detailed treatment may be needed for general purpose variables, such as the accumulators `l` and `r` of `sum_bidi` (recall Figure 1), but the resulting expressions are still small.

We have found that, for the set of programs used in our experiments, Υ can be characterised by the grammar in Figure 6. The grammar allows for functions comprised of a single if statement, where in each branch an array is squeezed using the `remove` function, and several integer variables are set. Conditions are generating by composing array elements, local variables and a fixed set of constants based on the given program, with standard comparison operators and boolean connectives. The semantics of `remove(arr, position)` are such that a single element is removed from the array at the specified position, and all index variables are adjusted by decrementing them if they are larger from the index of the element being removed. This behavior is hard-coded and is specific to array-based loops. Our experience has shown that a single conditional statement is indeed sufficient to cover many different cases (see Section 5).

$$
\begin{array}{lll}
\text{body} & ::= & \texttt{if (} \text{cond} \texttt{)} \\
& & \quad \texttt{remove(} \text{ arr, } \text{expr}_{index} \texttt{)} \ \left[\text{var}_{int} \texttt{ = } \text{expr}_{int}\right]^* \\
& & \texttt{else} \\
& & \quad \texttt{remove(} \text{ arr, } \text{expr}_{index} \texttt{)} \ \left[\text{var}_{int} \texttt{ = } \text{expr}_{int}\right]^* \\
\text{cond} & ::= & \text{elem}_\tau \diamond (\text{elem}_\tau \mid \text{const}_\tau) \\
& \mid & \text{var}_{index} \diamond (\text{var}_{index} \mid \text{const}_{index}) \qquad\qquad \diamond ::= \ \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} \\
& \mid & \text{cond } \texttt{\&\&} \text{ cond} \ \mid \ \text{cond } \texttt{||} \text{ cond} \\
\text{expr}_{index} & ::= & \text{const}_{index} \mid \text{var}_{index} \mid \texttt{len(arr) -} \ (\text{const}_{index} \mid \text{var}_{index}) \\
\text{expr}_{int} & ::= & \text{var}_{int} \ (\texttt{+} \mid \texttt{-}) \ \text{elem}_{int} \\
\text{elem}_\tau & ::= & \text{arr } \texttt{[} \text{expr}_{index} \texttt{]} \\
\text{const}_{index} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \\
\text{const}_\tau & ::= & \texttt{0} \mid \text{other constants occurring in the program} \\
\multicolumn{3}{l}{\text{arr, var}_{index}\text{, var}_{int}\text{, var}_{char} \text{ — identifiers occurring in the program}}
\end{array}
$$

Fig. 6. Program space for syntax-guided synthesis of Υ. Expressions are split into three categories: *index*, *int*, and *char* as described in Section 4. $\tau \in \{int, char\}$.

To bound the search space, expressions and conditions have bounded sizes (in terms of AST height) imposed by the generator and the user selects the set of basic predicates from which the condition of the `if` statement is constructed. The resulting space, however, is still often too large to be explored efficiently. To reduce it, some type-directed pruning is carried out so that only valid functions are passed to the checker. Moreover, our synthesis procedure distinguishes between variables that are used as indices to the array (var_{index}) and regular integer variables (var_{int}), and does not mix between them. We further assume that we can determine, from analyzing the program's source code, which index variable is used with which array(s). So when generating expressions of the form arr`[` i `]` *etc.*, only relevant index variables are used. Also, we note that generated squeezers preserve in bounds access by construction.

4.2 Test

The *test* step checks whether a candidate squeezer that is generated by the synthesizer satisfies the requirements of Definition 3. For the simulation-inducing requirement, we restrict $n_\sigma = 1..2$ and $m_\sigma = 0..1$. The step is divided into three phases. In the first phase, candidates are checked against a bank of concrete program states (both reachable and unreachable). In the second phase, candidates are verified for a bounded array size, but with no restrictions on the values of the elements. Those that pass bounded verification enter the third phase where full, unbounded verification is performed.

The second and third phases of the test step require the use of an SMT solver. The second phase is useful since incorrect candidates may cause the solver to diverge when queried for arbitrary array sizes. Limiting the array size to a small number (we used 6) enables to rule out these candidates in under a second. To

simplify the satisfiability checks, we found it beneficial to decompose the verification task. To do so, we take advantage of the structure of the squeezer, and split each satisfiability query (that corresponds to one of the requirements in Definition 3) into two queries, where in each query we make a different assumption regarding the branch the squeezer function takes. We note in this context that the capabilities of the underlying solver direct (or limit in some sense) the expressive power of the squeezer. In this aspect, it is also worth mentioning that sequence theory support for element removal helped to define squeezers format.

For the simulation inducing check, we further exploit the property that for the kind of programs and squeezers we consider, the transitions of the program usually do not change the truth value of the condition of the if statement in the definition of the squeezer. Namely, if σ makes a transition to σ' then either both of them satisfy the condition or both of them falsify it; either way, their definition of Υ follows the same branch. This form of preservation can be checked automatically using additional queries. When it holds, we can consider the same branch of the squeezer program in both the pre- and post-states, thus simplifying the query for checking simulation. Similarly, we can opportunistically split the transition relation of the program into branches (e.g., one that executes an iteration of the loop and one that exits the loop). In most cases, the same branch that was taken for σ is also the one that needs to be taken from $\Upsilon(\sigma)$ to establish simulation. This leads to another simplification of the queries, which is sound (i.e., never concludes that the simulation-inducing requirement holds when it does not), but potentially incomplete. We can therefore use it as a "cheaper" check and resort to the full check if it fails.

4.3 Filtering out unreachable states

For soundness, a squeezer needs to satisfy Definition 3 only on the reachable states. As we do not have a description of this set, for otherwise the verification task would be essentially voided, we need to ensure that the requirements of simulation-inducement on a safe over-approximation of this set. A simple over-approximation would be the set of all states. However, this over-approximation might be too coarse, indeed we noticed in our experiments that in some cases, unreachable states have caused phases 1, 2 and 3 to produce false negatives, *i.e.*, disqualify squeezers which can be used safely to verify the program. Therefore we used an over-approximation of reachable states using

1. Bound constraints on the index variables: the index is expected to be within bounds of the traversed array. This property can be easily verified using other verifiers or by applying our verifier in stages, first proving this property and then proving the actual specification of the verified procedure under the assumption that the property hold.
2. 2-step bounded reachability: We found out that for our examples, looking only at states that are reachable from another state in at most two steps is a general enough inclusion criterion. Note that we do not require 2-step reachability from an initial state, but rather from *any* state, hence this set over-approximates the set of reachable states.

5 Implementation and Experimental Results

We implemented an automatic verifier for array programs based on our approach, and applied it successfully to verify natural properties of a few interesting array-manipulating programs.

Base case. We discharged the base case of the induction (the verification on the base states) using KLEE [12]—a state-of-the-art symbolic execution [13] engine. It took KLEE less than one tenth of a second to verify the correctness of each program in our benchmarks on the states in its base. This part of our verification approach is standard, and we discuss it no further; in the rest of this section we focus on the generation of the squeezing functions.

5.1 Implementation

The generate step and phase 1 of the test step of the squeezer synthesizer were implemented using a standalone C++ application that generates all Υ candidates with an AST of depth three. Each squeezer was tested on a pre-prepared state bank and every time a squeezer passed the tests it was immediately passed on to phase 2. The state bank contained states with arrays of length five or less. For each benchmark, we used up to 24,386 states with randomly selected array contents. The number of states was determined as follows: Suppose the program state is comprised of k variables and an array of size n. We randomly selected p elements that can populate the array: $p = \{'a', 'b', 0\}$ for string manipulating procedures and $p = \{-4, -2, 9, 100, 200\}$ for programs that manipulate integer arrays. We determined the number test states according to the following formula: $d^k \cdot |p|^n / df$, where df is an arbitrary dilution factor used to reduce the number of states from thousands to hundreds. (In our experiments, $df = 17$.)

The second and third phases were implemented using Z3 [17], a state of the art SMT solver. We chose to use the theory of sequences, since its API allows for a straightforward definition of the operation remove(arr,i) (see Figure 6). In practice, the sequence solver proved to be overall more effective than a corresponding encoding using the more mature array solver. In that aspect, it is worth mentioning that verifying fault preservation on its own *is* faster with the theory of arrays. We conjecture that this is because the specification has quantifiers while the other requirements can be verified using quantifier-free reasoning.

The transition relation was manually encoded in SMT-LIB2 format. However, it should be straightforward to automate this step.

5.2 Experimental Evaluation

We evaluated our technique by verifying a few array-manipulating programs against their expected specifications. The experiments were executed on a laptop with Intel i7-8565 CPU (4 cores) with 16GB of RAM running Ubuntu 18.04.

Program	Predicates
`strnchr(c)`	`s[0]==c`, `s[0]!=c`, `s[0]==0`, `s[0]!=0`
`strncmp`	`s1[0]==s2[0]`, `s1[0]!=s2[0]`, `s1[0]==0`, `s2[0]==0`, `s1[0]!=0`, `s2[0]!=0`
`max_ind`	`s[0]<=s[1]`, `s[0]<=s[2]`, `s[0]<=s[n-2]`, `s[0]<=s[n-1]`, `s[1]<=s[2]`, `s[1]<=s[n-2]`, `s[1]<=s[n-1]`, `s[2]<=s[n-1]`, `s[n-2]<=s[n-1]`
`min_ind`	`s[0]>=s[1]`, `s[0]>=s[2]`, `s[0]>=s[n-2]`, `s[0]>=s[n-1]`, `s[1]>=s[2]`, `s[1]>=s[n-2]`, `s[1]>=s[n-1]`, `s[2]>=s[n-1]`, `s[n-2]>=s[n-1]`
`sum_bidi`	`i==0`, `l==0`, `r==0`, `s[0]==0`, `s[n-i]==0`
`is_sorted`	`s[n-3]<=s[n-2]`, `s[n-3]<=s[n-1]`, `s[n-2]<=s[n-1]`
`long_pref`	`s[0]<=s[1]`, `s[0]<=s[2]`, `s[1]<=s[2]`, `s[0]>s[1]`, `s[0]>s[2]`, `s[1]>s[2]`

Table 1. User-supplied atom predicates used to synthesize Υ.

			Phase 1			Phase 2		Phase 3	Total Time	QUIC3
Program	**B**	**# Cand**	**|Bank|**	**Test**	**Time**	**BMC**	**Time**	**Time**	**G&T+KLEE**	**Time**
`strnchr`	2	80	356	29	0.004	1	0.12	0.16	0.28 + 0.07	0.32
`strncmp`	2	980	76	196	0.02	1	7.2	154.48	161.70 + 0.05	0.19
`max_ind`	2	8000	368	10	0.18	2	1.86	4.44	4.73 + 0.05	0.11
`min_ind`	2	8000	257	9	0.26	2	2.1	16.86	17.21 + 0.05	0.09
`sum_bidi`	2	6328125	4602	1200	2.18	1	0.57	0.61	3.36 + 0.05	t.o.
`is_sorted`	4	900	25736	764	4.37	1	0.59	0.67	5.63 + 0.06	0.15
`long_pref`	3	6480	24386	4696	22.93	1	1.25	0.89	25.07 + 0.05	t.o.

Table 2. Experimental results (end-to-end). Time in seconds. G&T is a shorthand for Generate&Test

Benchmarks. We ran our experiments on seven array-manipulating programs: `strnchr` looks for the first appearance of a given character in the first n characters of a string buffer. `strncmp` compares whether two strings are identical up to their first n characters or the first zero character. `max_ind` (resp. `min_ind`) looks for the index of the maximal (resp. minimal) element in an integer array. `sum_bidi` is our running example. `is_sorted` checks if the elements of an array are sorted in an increasing order. `long_pref` is looking for the longest prefix of an array comprised of either a monotonically increasing or a monotonically decreasing sequence.

Table 1 lists the user-supplied predicates used when synthesizing each squeezer. These were selected based on understanding what the program does and the operations it uses internally. *E.g.*, for `strncmp` equality comparisons between same-index elements of the two input arrays are used, as well as comparison with constant 0; for `long_pref`, order comparisons between different elements of the same array or used instead.

Results. Table 2 describes the end-to-end running times of our verifier, i.e., the time it took our tool to establish the correctness of each example. In this

	Phase 1				Phase 2			Phase 3		
Program	**|Pos.|**	**Time**	**|Neg.|**	**Time**	**|Pos.|**	**Time**	**Time$_{\|Neg.\|}$**	**|Pos.|**	**Time**	**Time$_{\|Neg.\|}$**
`strnchr`	1	ϵ	9	ϵ	1	0.94	–	1	0.98	–
`strncmp`	3	ϵ	36	ϵ	3	14.29	–	3	154.48	–
`max_ind`	11	ϵ	3	ϵ	2	0.78	1.08	1	31.00	0.41
`min_ind`	11	ϵ	7	ϵ	2	0.91	1.19	1	16.00	0.43
`sum_bidi`	12	ϵ	1	0.05	1	0.56	0.69	1	0.61	–
`is_sorted`	1	ϵ	18	ϵ	1	0.59	–	1	0.67	–
`long_pref`	2	ϵ	74	ϵ	1	1.03	1.22	1	0.89	–

Table 3. Experimental results. Time in seconds. $\epsilon \leq 0.0001$

experiment, every candidate squeezer was tested before the next squeezer was generated. The table shows the time it took the synthesizer to find the first simulation-inducing squeezer plus the time it took to establish the correctness of the programs on the states in the base using KLEE (Total Time). The table also compares our verifier to Quic3 [28], an automatic synthesizer of loop invariants. In general, when both tools where able to prove that the analyzed procedure is correct, Quic3 was somewhat faster, and in the case of `strncmp` much faster. However, on two of our benchmarks Quic3 timed out (1 hour) whereas our tool was able to prove them correct in less than 30 seconds.

Table 2 also provides more detailed statistics regarding the experiments: The rank of the base states (B), the total number of possible candidates based on the supplied predicates and the bound on the depth of the AST (# Cand), and a more detailed view of each phase in the testing step. For phase 1, it reports the number of states in the pre-prepared state bank (|Bank|), the number of squeezers tested until a simulation-inducing one was found (Test), and the total time spent to test these squeezers (Time). For phase 2, it reports the number of candidates which passed phase 1 and survived bounded verification (BMC) and the time spent in this phase (Time). For phase 3, we report how many simulation-inducing squeezers were found the time it took to apply full verification.

In all our experiments except of `max/min_ind` only the simulation-inducing squeezers passed bounded verification. In the latter case, a squeezer passed BMC due to the use of arrays of size at most five where the cells $a[2]$ and $a[n-2]$ are adjacent. Had we increased the array bound to six, these false positives would have been eliminated by the bounded verification.

Table 3 provides average times required to pass all the generated squeezers through the testing pipeline. For phase 1, it reports the number of squeezers which passed (Pos) resp. failed (Neg) testing against the randomly generated states and the average time it took to test the squeezers in each category (Time). The table reports the statistics pertaining to phase 2 and 3 in a similar manner, except that it omits the number of squeezers which failed the phase as this number can be read off the number of squeezers which reached this phase.

Table 4 shows the automatically generated squeezers. We obtained a single simulation-inducing squeezer in all of our tests except for `strncmp` where three

Program	Squeezer
strchr(c)	`if ( s[0] == c || s[0]==0 ) remove(s,1) else remove(s,0)`

strncmp	(1) `if (s1[0] == s2[0] && s1[0] != 0) remove(s1,0); remove(s2,0)`
	`else remove(s1,1); remove(s2,1)`
	(2) `if (s1[0] == s2[0] && s2[0] != 0) remove(s1,0); remove(s2,0)`
	`else remove(s1,1); remove(s2,1)`
	(3) `if (s1[0] != s2[0]) || (s1[0] == 0 && s2[0] == 0))`
	`remove(s1,1); remove(s2,1)`
	`else remove(s1,0); remove(s2,0)`

max_ind	`if (s[n-2] <= s[n-1]) remove(s,n-2) else remove(s,n-1)`
min_ind	`if (s[n-2] >= s[n-1]) remove(s,n-2) else remove(s,n-1)`
sum_bidi	`if (i == 0) remove(s,0); l := l; r := r` `else remove(s,0); l := l - s[0]; r := r - s[n-i]`
is_sorted	`if (s[n-3]<=s[n-2] && s[n-2]<=s[n-1]) remove(s,n-1)` `else remove(s,n-4)`

long_pref	`if ((s[0] <= s[1] && s[1]<= s[2]) || (s[0] > s[1] && s[1] > s[2]))`
	`remove(s,0)`
	`else remove(s,n-1)`

Table 4. Syntesized squeezers. n is the size of the input array

squeezers were synthesized. The three differ only syntactically by the condition of the `if` statements. However, semantically, the three conditions are equivalent. Thus, improving the symmetry-detection optimizations to include equivalence up-to-de morgan rules would have filtered out two of the three squeezers.

6 Related Work

Automatic verification of infinite-state systems, *i.e.*, systems where the size of an individual state is unbounded such as numerical programs (where data is considered unbounded), array manipulating programs (where both the length of the array and the data it contains may be unbounded), programs with dynamic memory allocation (with unbounded number of dynamically-allocatable memory objects), and parameterized systems (where, in most cases, there is an unbounded number of instances of finite subsystems) is a long standing challenge in the realm of formal methods.

Well structured transition systems. Well structured transition systems (WSTS) [1, 2, 22] are a class of infinite-state transition systems for which safety verification is decidable, with a backward reachability analysis being a decision procedure. In these transitions systems, the set of states is accompanied by a well-quasi order that induces a simulation relation: a state is simulated by those that are "larger" than it. As a result, the set of backward-reachable states is upward closed. The simulation-inducing well-quasi order used in WSTS resembles our

condition of a simulation-inducing squeezer. However, there are several fundamental differences: (i) The order underlying our technique is required to be well-founded, which is a strictly weaker requirement than that of a well-quasi order; (ii) The simulation-inducing requirement requires each state to be simulated by its squeezed version, which has a *lower* rank rather than greater; further, a state need not be simulated by *every* state with a lower rank; accordingly, the set of backward-reachable states need not be upward (nor downward) closed. (iii) Our procedure is not based on backward (or any other form of) reachability analysis.

Reductions. Cutoff-based techniques, e.g., [18], reduce model checking of unbounded parameterized systems to model checking for systems of size (up to) a small predetermined cutoff size. Verification based on dynamic cut-offs [3, 31] also considers parameterized systems but employs a verification procedure which can dynamically detect cut-off points beyond which the search of the state space need not continue. Invisible invariants [40, 50] are used to verify unbounded parameterized systems in a bounded way. The idea is to use the standard deductive invariance rule for proving invariance properties but consider only bounded systems for discharging the verification conditions, while ensuring that they hold for the unbounded system. The approach provides (i) a heuristic to generate a candidate inductive invariant for the proof rule, and (ii) a method to validate the premises of the proof rule once a candidate is generated [50].

Similar reductions were applied to array programs–a particular form of parameterized systems but with unbounded data–as we consider in this work. For example, in [33], *shrinkable* loops are identified as loops that traverse large or unbounded arrays but may be soundly replaced by a bounded number of non-deterministically chosen iterations; and in [37], abstraction is used to replace reasoning about unbounded arrays and quantified properties by reasoning about a bounded number of array cells.

A fundamental difference between our approach and these works is that we do not reduce the problem to a bounded verification problem. Instead, we generate verification conditions which amount to a proof by induction on the size of the system. In fact, from the perspective of deductive verification, our work can be seen as introducing a new induction scheme.

Loop invariant inference. Arguably, inference of loop invariants is the ubiquitous approach for automatic verification of infinite-size systems. Recent research efforts in the area have concentrated around inference of quantified invariants, in particular, the search for universal loop invariants is a central issue.

Classical predicate abstraction [7, 25] has been adapted to quantified invariants by extending predicates with *skolem* (fresh) variables [23, 34]. This is sufficient for discovering complex loop invariants of array manipulating programs similar to the simpler programs used in our experiments.

A research avenue that has received ongoing popularity is the use of constrained Horn clauses (CHCs) to model properties of transition systems which have been used for inference of universally quantified invariants [9, 27, 38] by

limiting the quantifier nesting in the loop invariant being sought. In [20], universally quantified solutions (inductive invariants) to CHCs are inferred via syntax-guided synthesis.

Another active research area is Model-Checking Modulo Theories (MCMT) [24] which extends model checking to array manipulating programs and has been used for verifying heap manipulating programs and parameterized systems (e.g., [15]) using quantifier elimination techniques. For example, in Safari [4] (and later Booster [5]), the theory of arrays [11] is used to construct a quantifier-free proof of bounded safety which is generalized by universally quantifying out some terms.

IC3 [10] extends predicate abstraction into a framework in which the predicate discovery is directed by the verification goal and heuristics are used to generalize proofs of bounded depth execution to inductive invariants. UPDR [32] and Quic3 [28] extend IC3 to quantified invariants. UPDR focuses on programs specified using the Effectively PRopositional (EPR) fragment of *uninterpreted* first order logic (e.g., without arithmetic) for which quantified satisfiability is decidable. As such, UPDR does not deal with quantifier instantiation. Quic3 uses model based projection and generalizations based on bounded exploration.

Like these techniques we also use heuristics to overcome the unavoidable undecidability barrier. In our case, this amounts to the selection of the squeezing function. In contrast to all the aforementioned approaches, our technique does not rely on the inference of loop invariant but rather proves programs correct by induction on the size (rank) of their states.

We note that we do not position our technique as a replacement to automatic inference of loop invariants but rather as a complementary approach. Indeed, while some tricky properties can be easily verified by our approach, e.g., the postcondition of `sum_bidi`, a property which we believe no other automatic technique can deduce, other properties which are simple to establish using loop invariants, e.g., that variable `i` is always in the range $0..n-1$, are surprisingly challenging for our technique to establish.

Recurrences. Other approaches represent the behavior of loops in array-programs via recurrences defined over an explicit loop counter, and use these recurrences to directly verify post-conditions with universal quantification over the array indices. In [41] this is done by customized instantiation schemes and explicit induction when necessary. In [14], verification is done by identifying a relation between loop iterations (characterized by the loop counter) and the array indices that are affected by them, and verifying that the post-condition holds for these indices. Similarly to our approach, these works do not rely on loop invariants, but they do not allow to verify global properties over the arrays, such as the postcondition of `sum_bidi`.

Program synthesis. The inference we use for Υ is indeed a form of program synthesis, as was alluded to in Section 2 by representing Υ via pseudo-code. In particular, *syntax-guided synthesis* (SyGuS) [6] is the domain of program

synthesis where the target program is derived from a programming language according to its syntax rules. [19, 30, 47, 48] all fall within this scope.

Sketching is a common feature of SyGuS. The term is inspired by Sketch [43], referring to the practice of giving synthesizers a program skeleton with a missing piece or pieces. This uses domain knowledge to reduce the size of the candidate space. It is quite common to use a domain-specific language (DSL) for this purpose [29, 42, 44, 45, 49]. [39] restricts programs by typing rules in addition to mere syntactic rules. [26] develops it further by restricting how operators in the input DSL may be composed. Our synthesis procedure (Section 4) follows the same guidelines: the domain of array-scanning programs dictates the constructed space of squeezer functions, and moreover, inspecting the analyzed program allows for more pruning by (i) matching index variables to array variables and (ii) focusing on operators and literal values occurring in the program. This early pruning is responsible for the feasibility of our synthesis procedure, which apart from that is rather naive and does not facilitate clever optimizations such as equivalence reduction ([21, 39]).

7 Conclusions

At the current state of affairs in automatic software verification of infinite state systems, the scene is dominated by various approaches with a common aim: computing over-approximations of unbounded executions by means of inferring loop invariants. Indeed, *abstract interpretation* [16], *property-directed reachability* [10], unbounded model checking [35], or template-based verification [46] can be seen as different techniques for computing such approximations by finding inductive loop invariants which are tight enough not to intersect with the set of bad behaviors. Experience has shown that these invariants are frequently quite hard to come by, even for seemingly simple and innocuous program, both automatically and manually. The purpose of this paper is to suggest an alternative kind of correctness witness, which may be more amenable to automated search. We successfully applied our novel verification technique to array programs and managed to prove programs and properties which are beyond the ability of existing automatic verifiers. We believe that our approach can be combined with standard techniques to give rise to a new kind of hybrid techniques, where, e.g., a *partial* loop invariant is used as a baseline — verified via standard techniques — and is then *strengthened* to the desired safety property via squeezer-based verification.

Acknowledgement The research leading to these results has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). SS: *what else?*

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: General decidability theorems for infinite-state systems. In: Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996. pp. 313–321 (1996). https://doi.org/10.1109/LICS.1996.561359, https://doi.org/10.1109/LICS.1996.561359
2. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: Algorithmic analysis of programs with well quasi-ordered domains. Inf. Comput. **160**(1-2), 109–127 (2000). https://doi.org/10.1006/inco.1999.2843, https://doi.org/10.1006/inco.1999.2843
3. Abdulla, P.A., Haziza, F., Holík, L.: All for the price of few. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 476–495. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
4. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: SAFARI: smt-based abstraction for arrays with interpolants. In: Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. pp. 679–685 (2012)
5. Alberti, F., Ghilardi, S., Sharygina, N.: Booster: An acceleration-based verification framework for array programs. In: Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings. pp. 18–23 (2014)
6. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. Dependable Software Systems Engineering **40**, 1–25 (2015)
7. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. pp. 268–283 (2001)
8. Bjørner, N., Ganesh, V., Michel, R., Veanes, M.: SMT-LIB sequences and regular expressions. In: 10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012. pp. 77–87 (2012)
9. Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified horn clauses. In: Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. pp. 105–125 (2013)
10. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. pp. 70–87 (2011)
11. Bruttomesso, R., Ghilardi, S., Ranise, S.: Quantifier-free interpolation of a theory of arrays. Logical Methods in Computer Science **8**(2) (2012). https://doi.org/10.2168/LMCS-8(2:4)2012, https://doi.org/10.2168/LMCS-8(2:4)2012
12. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1855741.1855756
13. Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. Commun. ACM **56**(2), 82–90 (Feb 2013).

https://doi.org/10.1145/2408776.2408795, http://doi.acm.org/10.1145/2408776.2408795
14. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs by tiling. In: Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings. pp. 428–449 (2017). https://doi.org/10.1007/978-3-319-66706-5_21, https://doi.org/10.1007/978-3-319-66706-5_21
15. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Invariants for finite instances and beyond. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 61–68 (2013), http://ieeexplore.ieee.org/document/6679392/
16. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 238–252. ACM Press, New York, NY, Los Angeles, California (1977)
17. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
18. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings. pp. 236–254 (2000)
19. Farzan, A., Nicolet, V.: Modular divide-and-conquer parallelization of nested loops. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 610–624. PLDI 2019, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314612, http://doi.acm.org/10.1145/3314221.3314612
20. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. pp. 259–277 (2019). https://doi.org/10.1007/978-3-030-25540-4_14, https://doi.org/10.1007/978-3-030-25540-4_14
21. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: ACM SIGPLAN Notices. vol. 50, pp. 229–239. ACM (2015)
22. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. **256**(1-2), 63–92 (2001). https://doi.org/10.1016/S0304-3975(00)00102-X, https://doi.org/10.1016/S0304-3975(00)00102-X
23. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 191–202 (2002). https://doi.org/10.1145/503272.503291, http://doi.acm.org/10.1145/503272.503291
24. Ghilardi, S., Ranise, S.: MCMT: A model checker modulo theories. In: Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings. pp. 22–29 (2010)
25. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings. pp. 72–83 (1997)

26. Gulwani, S.: Programming by examples (and its applications in data wrangling). In: Esparza, J., Grumberg, O., Sickert, S. (eds.) Verification and Synthesis of Correct and Secure Systems. IOS Press (2016)
27. Gurfinkel, A., Shoham, S., Meshman, Y.: Smt-based verification of parameterized systems. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. pp. 338–348 (2016). https://doi.org/10.1145/2950290.2950330, http://doi.acm.org/10.1145/2950290.2950330
28. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings. pp. 248–266 (2018)
29. Hua, J., Khurshid, S.: Edsketch: Execution-driven sketching for java. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. pp. 162–171. ACM (2017)
30. Itzhaky, S., Singh, R., Solar-Lezama, A., Yessenov, K., Lu, Y., Leiserson, C., Chowdhury, R.: Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 145–164. ACM (2016)
31. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 645–659. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
32. Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., Shoham, S.: Property-directed inference of universal invariants or proving their absence. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 583–602 (2015)
33. Kumar, S., Sanyal, A., Venkatesh, R., Shah, P.: Property checking array programs using loop shrinking. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I. pp. 213–231 (2018). https://doi.org/10.1007/978-3-319-89960-2_12, https://doi.org/10.1007/978-3-319-89960-2_12
34. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In: Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings. pp. 267–281 (2004)
35. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification. pp. 123–136. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
36. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (Mar 2006)
37. Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free horn clauses. In: Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. pp. 361–382 (2016). https://doi.org/10.1007/978-3-662-53413-7_18, https://doi.org/10.1007/978-3-662-53413-7_18
38. Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free horn clauses. In: Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. pp. 361–382 (2016)

39. Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. In: ACM SIGPLAN Notices. vol. 50, pp. 619–630. ACM (2015)
40. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 82–97. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
41. Rajkhowa, P., Lin, F.: Extending VIAP to handle array programs. In: Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers. pp. 38–49 (2018). https://doi.org/10.1007/978-3-030-03592-1_3, https://doi.org/10.1007/978-3-030-03592-1_3
42. Smith, C., Albarghouthi, A.: Mapreduce program synthesis. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 326–340. ACM (2016)
43. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. ACM SIGOPS Operating Systems Review **40**(5), 404–415 (2006)
44. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: ACM Sigplan Notices. vol. 45, pp. 313–326. ACM (2010)
45. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verification and program synthesis. International Journal on Software Tools for Technology Transfer **15**(5-6), 497–518 (2013)
46. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verification and program synthesis. International Journal on Software Tools for Technology Transfer **15**(5), 497–518 (Oct 2013)
47. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M., Alur, R.: Transit: specifying protocols with concolic snippets. ACM SIGPLAN Notices **48**(6), 287–296 (2013)
48. Wang, C., Cheung, A., Bodik, R.: Synthesizing highly expressive sql queries from input-output examples. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 452–466. ACM (2017)
49. Wang, K., Sullivan, A., Marinov, D., Khurshid, S.: Solver-based sketching of alloy models using test valuations. In: International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 121–136. Springer (2018)
50. Zuck, L., McMillan, K.: Invisible Invariants Are Neither, pp. 57–72 (09 2019)