Property Checking Array Programs Using Loop Shrinking

Shrawan Kumar[1(✉)], Amitabha Sanyal[2], R. Venkatesh[1], and Punit Shah[1]

[1] Tata Consultancy Services Ltd., Pune, India
{shrawan.kumar,r.venky,shah.punit}@tcs.com
[2] Department of CSE, IIT Bombay, Mumbai 400076, India
as@cse.iitb.ac.in

Abstract. Most verification tools find it difficult to prove properties of programs containing loops that process arrays of large or unknown size. These methods either fail to abstract the array at the right granularity and are therefore limited in precision or scalability, or they attempt to synthesize an appropriate invariant that is quantified over the elements of the array, a task known to be difficult. In this paper, we present a different approach based on a notion called *loop shrinkability*, in which an array processing loop is transformed to a loop of much smaller bound that processes only a few non-deterministically chosen elements of the array. The result is a finite state program with a drastically reduced state space that can be analyzed by bounded model checkers. We show that the proposed transformation is an over-approximation, i.e. if the transformed program is correct, so is the original. In addition, when applicable, the method is impervious to the size or existence of the bound of the array. As an assessment of usefulness, we tested a tool based on our method on the *ArraysReach* category of SV-COMP 2017 benchmarks. After excluding programs with feature not handled by our tool, we could successfully verify 87 of the 93 remaining programs.

1 Introduction

An array processing loop is a common occurrence in programs, and an assurance of reliability often requires the program developer to prove properties that are quantified over the elements of the array being processed. This is, in general, difficult because such programs have huge, at times infinite state space. So while static analysis techniques like array smashing and partitioning [4,5,11,14,16,17] fail due to abstractions that are too coarse, attempts with bounded model checkers or theorem provers that are equipped with array theories [3,8,9,15,18,22,23] tend to fail for lack of scalability or their inability to synthesize the right quantified invariants.

In certain situations, the decidability of property checking of finite state programs can be used to prove properties of infinite state space programs. Consider a program P and a property ψ that can be transformed to an abstract finite state program P' and a property ψ', such that if the property ψ' holds in P'

```
1  #define N 7
2  main()
3  {
4   int i, m;
5   int a[N]={8,4,6,2,11,2,2};
6   m = a[0];
7   i=0;
8   while(i < N)
9   {
10   if(m >= a[i]-1)
11     m = a[i];
12   i++;
13  }
14  assert ∀j∈[0..N−1].(m≤a[j]);
15 }
```

(a) Concrete program

```
1  #define N 7
2  main() {
3   int i, m, a[N]={8,4,6,2,11,2,2};
4   unsigned li, it[2];
5   m = a[0]; i=0;
6   it[0]=nondet(); it[1]=nondet();
7   assume(1 <= it[0] && it[0]<it[1]);
8   for (li=0; li < 2 ; li++) {
9    i = it[li] - 1;
10   if (!(i < N)) break;
11   if(m >= a[i]-1) m = a[i];
12   i++;
13  }
14  assume(li==2);
15  assert ∀t∈it.(m≤a[t−1]);
16 }
```

(b) Abstract program

Fig. 1. Loop shrinking abstraction illustration

then the property ψ holds in P. Then P' can be analyzed for ψ' to show that ψ holds in P. In this paper we present such a transformation for programs which process arrays in loops. The property ψ is usually a $\forall$ or a $\exists$ property over elements of the array, but can also be a property over scalar variables modified in the loop. The transformation replaces the loop that manipulates an array of possibly large or even unknown size with a smaller loop that operates only on a few non-deterministically chosen elements of the array.

As an example, consider the program in Fig. 1(a). The loop in the program purportedly computes in a variable `m`, the minimum element, denoted *min*, of an array `a`. However, due to a programmer error at line 10 (`a[i]-1` instead of `a[i]`), the program actually computes the last value in the longest subsequence `a`$[i_1]$, `a`$[i_2], \ldots,$ `a`$[i_p]$ of the array, such that `a`$[i_1] = min$, and for any two consecutive elements `a`$[i_k]$ and `a`$[i_{k+1}]$ of the subsequence, `a`$[i_{k+1}] \leq$ `a`$[i_k] + 1$.[1] Notice that for ease of exposition, we have used a $\forall$ to express universal quantification; in reality, a loop will be used instead. The property holds for the example because the longest subsequence of the array with the stated properties is $\{2, 2, 2\}$, and the last element happens to be the same as *min*. However, the assertion will fail if, for example, the last two elements of the array are changed to 3 and 5, so that the longest subsequence is now $\{2, 3\}$.

Abstraction based verifiers as well as bounded model checkers fail to verify this program when the array size is increased to 1000. For example, CBMC 5.8 [8] reports "out of memory", when run with an unwinding count of 20. Abstraction based verifiers like SATABS 3.2 [9] and CPAchecker 1.6 [3] keep on iterating in their abstraction refinement cycle in search of an appropriate loop invariant, until they run out of memory. Therefore, it is worthwhile to look for an abstraction

[1] There is a unique such subsequence for any array.

of the property checking problem for array processing loops that can be verified by a bounded model checker (BMC).

Observe that in this program, the assertion will hold if and only if, after the last index containing the minimum value min, no other index in `a` contains the value $min + 1$. This can be conservatively checked by examining for each pair of array indices, say k and $k + j$, $j > 0$, whether $\texttt{a}[k + j] = \texttt{a}[k] + 1$. The computation is effected by selecting a pair of indices non-deterministically and executing in sequence the loop body with the loop index `i` first instantiated to k and then to $k + j$. The resulting value of `m` can then be checked for the condition $\texttt{m} \leq \texttt{a}[k] \wedge \texttt{m} \leq \texttt{a}[k + j]$. As we shall see later, it is helpful to think in terms of iteration numbers instead of array indices; the correspondence between the two for the present example is that the value at index i of the array is accessed at iteration number $i + 1$.

In other words, we compute `m` for every pair of iterations of the loop, and check if `m` satisfies the property for the chosen iterations. For example, the value of `m` computed for the iterations numbered 2 and 3 of the loop is 4, and the property restricted to these two iterations, $\texttt{m} \leq \texttt{a}[1] \wedge \texttt{m} \leq \texttt{a}[2]$, is satisfied. On the other hand, if we change the last two elements to `3` and `5` then the property fails for the original program. However, we can now find a pair of iterations, namely 4 and 6, such that value of `m` calculated on the basis of just these two iterations will be 3, and it will not satisfy the corresponding property $\texttt{m} \leq \texttt{a}[3] \wedge \texttt{m} \leq \texttt{a}[5]$, since $\texttt{a}[3]$ is 2. In summary, if executing the loop for every sequence of two iterations $[i_1, i_2]$, $i_2 > i_1$, establishes the property restricted to these iterations, then the property will also hold for the entire loop. Read contrapositively, if the given program does not satisfy the assertion, then there must be a sequence of two iterations for which the property will not hold. This is true irrespective of the size or the contents of the array in the program. Loops which exhibit this feature for iteration sequences of length k (k is 2 in this example) will be called *shrinkable loops with a shrink-factor* k.

We create a second program, shown in Fig. 1(b), that over-approximates the behaviour of the original with respect to the property being checked. The `while` loop is substituted with a loop that executes the non-deterministically chosen iteration sequence stored in the two-element array `it`. The `while` loop in the original program, schematically denoted as `while` (C) B, is replaced by a `for` loop that is equivalent to the unrolled program fragment `i=it[0]-1;if(`C`){`B`;i=it[1]-1;if(`C`)`B`}`. We call this `for` loop (or its unrolled equivalent) the *residual loop* for the iteration sequence `it`. The `break` statement ensures that the chosen iteration numbers do not result in an out-of-bounds access of the array, and the `assume` statement ensures that exactly two iterations are chosen. Similarly, the given property is also substituted by a *residual property* quantified over array indexes corresponding to the same chosen iteration sequence. CBMC is able to verify the property on this transformed program, as the original loop, even with a changed bound of 1000, is now reduced to only two iterations. We call this method *property checking by loop shrinking.* Needless to say, the method can only be applied to a program if its shrinkability and shrink-factor are known. We develop a method to determine both using a BMC.

Thus the central idea, demonstrated in the rest of the paper, is that over-approximation using shrinkability is an effective technique to verify properties of programs that iterate over arrays of large or unknown size. Specifically, our contributions are:

1. We introduce and formalize a concept called *shrinkability* for loops that process arrays. We show formally that a shrinkable loop with shrink-factor k can be over-approximated by a loop that executes only k non-deterministically chosen iterations.
2. We provide an algorithm to find the shrink-factor k for which the loop is shrinkable.
3. We describe an implementation of the proposed abstraction.
4. We report experimental results showing the effectiveness of the technique on SV-COMP 2017 [2] benchmarks in the *ArraysReach* category.

2 Background

We shall present our ideas in the context of imperative programs that consist of assignment statements, conditional statements, while loops, and function calls. We assume that conditional expressions have no side effects. We restrict ourselves to goto-less programs with single-entry single-exit loops. This makes for an easier formal treatment of our method without losing expressibility.

Let Var be the set of variables in a program P and Val be the set of possible values which the variables in Var can take. A *program state* is a valuation of the variables in Var that is consistent with their declared types. It is represented by a map $\sigma : \mathit{Var} \rightarrow \mathit{Val}$. $\sigma(v)$ denotes the value of v in the program state σ.

Property checking will be expressed in a formalism called a *Hoare triple* and denoted as $\{\varphi\}P\{\psi\}$. Here φ and ψ are first order formulas representing sets of states, and P is a program. A Hoare triple is said to be *valid* if and only if starting from an initial state satisfying φ, the execution of P terminates in a final state that satisfies ψ. In this paper we shall only consider programs that are deterministic and guaranteed to terminate. A fact that we shall make use of is that in the special case when φ represents a single program state σ. Since our programs are deterministic, ψ also will be a unique single state. Therefore, the invalidity of $\{\sigma\}P\{\psi\}$ is equivalent to the validity of $\{\sigma\}P\{\neg\psi\}$.

An *iteration sequence* is a strictly ascending sequence of numbers, representing iteration counts. Iterations of a loop are counted starting from 1. The notation $i\!:\!T$ will represent a sequence whose first element is i and the sequence comprising the rest of the elements is T. Given sequences U and T, we shall use $U \sqsubset T$ to mean that U is a strict subsequence of T. Further, we shall write $\mathcal{P}_k(T)$ to denote the set of all k-sized subsequences of a sequence T. For example, if $T = [1, 2, 5]$ then $\mathcal{P}_2(T) = \{[1, 2], [2, 5], [1, 5]\}$.

Loop acceleration [19] is a commonly used technique for finding loop invariants. It captures the effect of a loop through closed-form expressions that give the value of variables at the beginning of an iteration in terms of the initial state and the iteration count. Variables whose values can be expressed in this manner are called *accelerable*. For example, in the program of Fig. 1, the value of the variable `i` in the beginning of an iteration j is expressible as j-1. We assume that we have available tools [12] to identify accelerable variables and their corresponding accelerating expressions. While our approach does not require us to identify all accelerable variables, the precision of the result does depend on the identification of as many accelerable variables as possible.

Our technique makes good use of bounded model checkers (BMCs). Industrial strength BMCs exist [8] and are widely used to detect property violations in safety critical software. Given a program P and a property ψ, a BMC searches for a counterexample to ψ in executions of P whose length is bounded by some integer n. If it finds a counterexample to ψ within the bound, then it reports the program as being unsafe. However, if it does not find a counter example within the given bound, then the program cannot be regarded as either being safe or unsafe. BMCs are, therefore, very effective in finding bugs but not in proving properties.

3 Programs and Properties of Interest

We focus on programs that process arrays in loops that we assume always terminate. The property to be checked is encoded in a fragment of code that follows the array processing loop. If the property is expressed as a loop, we denote it in our discussion as a universally or existentially quantified formula over the elements of the array. As an illustration, the property checked in the motivating example is $\forall j. 0 \leq j < N \implies m \leq a[j]$. Similarly, the program `min2` of Fig. 3 checks for the property $\exists j. 0 \leq j < S \wedge min = a[j]$. In particular, we consider program fragments R ; Q ; ψ, in which R is a simple loop possibly manipulating arrays, Q is a loop free (possibly empty) sequence of statements and ψ is the property to be checked. We call R ; Q as an *array processing loop*. In addition, we assume R has an upper bound on number of iterations which can be computed through static analysis [10]. The property ψ is assumed to have at most one quantifier. We assume that the array-processing loop and the loop which checks the property have the same number of iterations. Finally, since the quantified variable ranges over a finite domain (iteration counts of a finite loop), it is useful to think of ψ as a set of quantifier-free formulas, connected by conjunction in the case of $\forall$ and disjunction in the case of $\exists$.

```
i=1;
if (i < N) {
  if (m >= a[i]-1) m = a[i];
  i++;
} else
    goto loop_exit;
i=3;
if (i < N) {
  if (m >= a[i]-1) m = a[i];
  i++;
} else goto loop_exit;
loop_exit: ;
```

Fig. 2. Residual loop for iteration sequence [2,4] for the program in Fig. 1

3.1 Residual Loop and Residual Property

Consider a program P consisting of an array processing loop $L \equiv \texttt{while}(C)\{B\}Q$ followed by code that checks the property ψ. Let $T = [j_1, j_2, ..., j_n]$ be an arbitrary iteration sequence of the loop. We define the *residual loop* for the iteration sequence T, denoted as L_T, as the statements $\{S_{j_1}; S_{j_2}; \ldots S_{j_n}; \texttt{loop\_exit:}\text{Q}\}$, where each S_{j_r} is $\{A_{j_r}$; $\texttt{if}(\text{C})\{B\}$; $\texttt{else goto loop\_exit};\}$. Here A_{j_r} is the sequence of statements assigning to each accelerable variable the corresponding expression defining its value at the beginning of iteration j_r. Obviously, for $T = j : T'$ with T' being nonempty, $L_T = S_j; L_{T'}$. As an illustration, the code fragment in Fig. 2 is the residual loop for the iteration sequence [2,4] for the program in Fig. 1(a). If the loop iterates for a maximum of N times, then $[1, 2, ..., N]$ will be called the *complete iteration sequence* of the loop. It is obvious that, the residual loop $L_{[1,2,...,N]}$ represents an unrolling of L and the two are semantically equivalent. Similarly, for the iteration sequence $T = [j_1, j_2, ..., j_m]$ and the property ψ, we define the residual property ψ_T as a conjunction or disjunction of a set of clauses $\{\psi_{j_1}, \psi_{j_2}, \ldots, \psi_{j_m}\}$.

Let us represent the set of initial states at the beginning of the loop L as φ. Then the set of states at the beginning of an iteration numbered i would be given by $sp(S_1; S_2; ...S_{i-1}, \varphi)$, the strongest post-condition of $S_1; S_2; ...S_{i-1}$ wrt φ. However, we sometimes have to estimate these set of states in the context of an arbitrary iteration sequence T that contains iteration i and in which the sequence of iterations preceding i is not exactly known. Therefore, instead of the earlier exact calculation, we over-approximate the set of states at the beginning of iteration i, denoted φ_i, through the recurrences $\varphi_1 = \varphi$, and $\varphi_i = sp(S_{i-1}, \varphi_{i-1}) \cup \varphi_{i-1}$. The additional term φ_{i-1} in the union accounts for the possibility that the iteration $i-1$ may not precede i in T, and therefore the set of states at the beginning of i should also include the states at the beginning of $i-1$.

```
int i, min, a[S];
min = a[0]; i=1;
while (i< S) {
  if (a[i] < min) min = a[i];
  i++;
}
assert ∃j∈[0..S−1].(a[j]==min);
```

(a) Program min2

```
int i, m, a[S];
m = a[0]; i=0;
while(i < S){
  if(m >= a[i]-1) m = a[i];
  i++;
}
assert ∀j∈[0..S−1].(m≤a[j]+d);
```

(b) Program lmin

Fig. 3. Examples showing property loops

4 Shrinkability of Loops

We now characterize the conditions under which the behaviour of an array-processing loop L with respect to a property ψ can be over-approximated by a residual loop L_U with respect to the corresponding residual property ψ_U, where the iteration sequence U consists of fewer (non-deterministically chosen) iterations than the iterations in the original program, i.e. $U \sqsubset [1, 2, \ldots, N]$.

Definition 1. *(Shrinkable loops) Consider a program consisting of a loop L and a property ψ to be checked. Let T represent the complete sequence of iterations of the loop. The loop is said to be shrinkable with respect to ψ and with a shrink-factor k, $0 < k < |T|$, if and only if, starting from any state $\sigma \in \varphi$, the loop L satisfies ψ whenever the residual loops L_U of each k-length subsequence U of T satisfy the corresponding residual property ψ_U. Formally:*

$$\forall \sigma \in \varphi : ((\forall U \in \mathcal{P}_k(T) : \{\sigma\} L_U \{\psi_U\}) \implies \{\sigma\} L \{\psi\}) \quad (1)$$

It will often be useful to read the formal description above in a contrapositive manner, i.e. starting from a state in φ, if the loop L fails to satisfy ψ, then the failure is also witnessed by a k-length sequence U whose residual loop L_U also fails to satisfy the corresponding residual property ψ_U. Note that executions of both L and L_U begin with the same state in φ.

A shrinkable loop with a shrink-factor k will be called *k-shrinkable.* If we know that a loop is k-shrinkable, we can construct an abstract program that non-deterministically chooses an iteration sequence of size k, runs the residual loop and then checks the corresponding residual property. If the residual property holds, then shrinkability guarantees the correctness of the original program. However, a counter-example in the abstract program does not necessarily imply a violation of the property in the original program, except in situations described below.

In the absence of loop-carried dependences [1], the values assigned to variables that are not accelerable, in any iteration are independent of the values assigned in any other iteration. In addition, consider the case when the array elements accessed in ith iteration of the array-processing loop are also asserted in ψ_i. In this situation, if the original program P violates the property ψ, in particular the clause, say ψ_i, then the program consisting of the residual loop $L_{[\,i\,]}$, constructed on the basis of the only iteration i, will also violate the residual clause $\psi_{[\,i\,]}$. Thus a loop without loop-carried dependences is 1-shrinkable. More significantly, if the property being tested for such programs is universal, the converse is also true, i.e. if the residual loop corresponding to a sequence consisting of a single iteration violates its residual property, then the original program will also not satisfy its specified property.

Note that according to Definition 1, if a program P satisfies its property ψ, then the loop constituting the program is k-shrinkable for any shrink-factor $k > 0$. Similarly, a loop with a bound of m iterations is trivially m-shrinkable. Obviously, if the shrink-factor is small, then the abstract program with a smaller length iteration sequence loads the verifier to a lesser extent and thus offers greater chances of verifier returning with an answer. Therefore, we are interested in finding shrink-factors that are much smaller than the loop bound.

However, finding out whether a loop is shrinkable is difficult as we illustrate through an example. Consider the two programs `min2` and `lmin` in Fig. 3 which are similar in structure and in the nature of what they compute. The program `min2` computes the minimum of the array and is correct with respect to the asserted property. Thus the loop in the program is k-shrinkable for all k from 1 to `S`. The second program `lmin` is similar to our motivating example with a property that asserts that the final value of `m` does not exceed any array element by more than a value d. The reader can verify that this property does not hold for $d < \texttt{S} - 1$.[2] It turns out that the loop in `lmin` is shrinkable with a shrink-factor $k = d + 2$. This illustrates the difficulty of analytically finding whether a given loop is shrinkable, and based on the development in rest of this section, we shall suggest an empirical method in Sect. 5.

4.1 Identifying Shrinkable Loops

While Definition 1 lays down the consequences of a loop being shrinkable, it does not provide a convenient method to decide whether a loop is shrinkable and find the shrink-factor. To get around this problem, we first extend the notion of shrinkability from loops to arbitrary iteration sequences. We then identify the conditions under which the shrinkability of smaller iteration sequences (that are checked explicitly) would imply the shrinkability of larger iteration sequences and eventually of the entire loop.

Definition 2. *(Shrinkable iteration sequence) Consider a program consisting of a loop L and a property ψ to be checked. Let T be an iteration sequence, and let j be the first iteration in T. The sequence T is k-shrinkable with respect to ψ, $0 < k < |T|$, if and only if, starting from every state $\sigma \in \varphi_j$, the residual loop L_T satisfies the residual property ψ_T whenever the residual loops L_U of each k-length subsequences U of T satisfy the corresponding residual property ψ_U. Formally:*

$$\forall \sigma \in \varphi_j : ((\forall U \in \mathcal{P}_k(T) : \{\sigma\} L_U \{\psi_U\}) \implies \{\sigma\} L_T \{\psi_T\}) \qquad (2)$$

The only difference between the notion of shrinkability of a loop and an iteration sequence is the starting state σ, which, in this case, is from the set φ_j. Recall that φ_j is an over-approximation of the set of states at the beginning of iteration j in the residual loop of any iteration sequence that contains j. As in the case of loops, by k-shrinkable sequence we shall mean a shrinkable sequence with shrink-factor k. It is obvious that, if the sequence consisting of all iterations of a loop is k-shrinkable then the loop itself is k-shrinkable.

As an illustration of an iteration sequence that is not shrinkable, consider the program `lmin` in Fig. 3(b) with $d = 0$. Consider the array `a` with its initial two elements as $\{0, 1\}$ and the iteration sequence $T = [1, 2]$. The residual loop of T computes $\texttt{m} = 1$ for which the residual property ψ_T does not hold ($\texttt{m} > \texttt{a[0]}$). However, the residual loop for every 1-length sequence satisfies its residual property, and thus T is not 1-shrinkable. Also notice that when $d = 0$, the program is the same as the motivating example in Fig. 1 except for array initialisation. Thus, from the observations in Sect. 1, every iteration sequence of length 3 is 2-shrinkable.

[2] Observe that, starting with the second element of the array, if the value of each element exceeds the value of the previous element by 1, then `m` will exceed the first element by $\texttt{S} - 1$.

4.2 Conditions Guaranteeing Shrinkability of Loops

We are interested in a method which guarantees that a loop is shrinkable by examining iteration sequences up to a given length. More specifically, we are interested in a pair of numbers n and a k, such that the k-shrinkability of all sequences of length between $k+1$ and n would imply the k-shrinkability of any sequence longer than n—in particular, the complete sequence of iterations comprising the loop. If we can identify the conditions under which we can find such a pair, then our strategy would be to establish the k-shrinkability of sequences up to n empirically, and the k-shrinkability of all iteration sequences with lengths greater than n will follow.

Since empirical verification of k-shrinkability for all subsequences of length between $k+1$ and n would be costly, we shall consider the case where $n = k+1$, i.e. we shall empirically find a k such that all $k+1$ length iteration sequences are k-shrinkable. The identified conditions will then ensure the k-shrinkability of sequences larger than $k+1$. Notice that the generalization from $k+1$ to larger sequences does not happen unconditionally. As an example, consider the program `lmin` in Fig. 3(b). For `d=2`, all the iteration sequences of size 3 are 2-shrinkable but not all sequences of size 4 are 2-shrinkable.

To derive the required conditions, let us examine what it takes to ensure the k-shrinkability of a sequence of length $k+2$, given the k-shrinkability of all sequences of length $k+1$. For simplicity of exposition, we shall limit ourselves to conjunctive properties. The treatment for disjunctive properties is very similar, and we shall merely touch upon it later in this section.

Consider an iteration sequence T of size $k+2$, represented as $j : T'$. Obviously, T' being of size $k+1$, is k-shrinkable. Taking a contrapositive view of the condition for shrinkability, assume that starting from σ, the residual property ψ_T is violated for the program L_T i.e. $\{\sigma\}L_T\{\neg\psi_T\}$ is true. Given that all sequences of length $k+1$ are k-shrinkable, it suffices to find a subsequence $T'' \sqsubset T$ of length $k+1$ such that $\{\sigma\}L_{T''}\{\neg\psi_{T''}\}$ is true. k-shrinkability will then ensure that there is a k-length subsequence $U \sqsubset T'' \sqsubset T$ such that $\{\sigma\}L_U\{\neg\psi_U\}$. Let the state after the iteration j in the sequence be σ'. Clearly $\{\sigma'\}L_{T'}\{\neg\psi_{[j]} \vee \neg\psi_{T'}\}$ is true.

1. Consider the case when $\psi_{T'}$ is violated. Since T' is k-shrinkable, it is possible to find a k-length subsequence U within T' such that starting from σ', ψ_U would be violated after L_U. Now consider the iteration sequence $T'' = j : U$. Clearly, starting from σ, $\psi_{T''}$ would be violated after executing $L_{T''}$, and thus the $k+1$-length sequence that we want is T''.
2. Now suppose that ψ_T is violated only because the clause $\psi_{[j]}$ is violated. There are two subcases to be considered. In the first, assume that the violation of $\psi_{[j]}$ also shows up in the state after iteration $k+1$. In this case the T'' that we want is the $(k+1)$-length prefix of T.
3. The interesting case is when the violation of ψ_T is solely because of $\psi_{[j]}$, and this violation of $\psi_{[j]}$ does not show up in the state after iteration $k+1$. In this case, the definition of shrinkability, in its current form, does not enable

us to produce the required sequence T''. To remedy this, notice that for the subsequence T', there is an iteration in the past, namely j, whose clause $\psi_{[\,j\,]}$ has been violated. If we revise the definition of k-shrinkability of iteration sequences (Definition 2) to ensure that this violation also shows up at the end of some k-length subsequence U' of T', then we are done. The required $k+1$-length subsequence T'' in this case would be $j : U'$ for which $\{\sigma\}L_{T''}\{\neg\psi_{T''}\}$ would be satisfied.

We call the modification introduced above as *past-preservation.* The revised definition of shrinkability that includes past-preservation is presented below.

Definition 3. *(Shrinkable iteration sequence, revised) Consider a program consisting of a loop L and a property ψ to be checked. Let T be an iteration sequence, and let j be the first iteration in T. In addition, let i stand for any iteration before j. The sequence T is k-shrinkable with respect to a property ψ, $0 < k < |T|$, if and only if, starting from every state $\sigma \in \varphi_j$ the residual loop L_T satisfies $\psi_T \wedge \psi_{[\,i\,]}$ whenever the residual loops L_U of each k-length subsequences U of T satisfy the corresponding property $\psi_U \wedge \psi_{[\,i\,]}$. In other words:*

$$\forall\sigma \in \varphi_j, \forall 0 \le i < j : ((\forall U \in \mathcal{P}_k(T) : \{\sigma\}L_U\{\psi_U \wedge \psi_{[\,i\,]}\}) \implies \{\sigma\}L_T\{\psi_T \wedge \psi_{[\,i\,]}\}) \qquad (3)$$

A contrapositive reading of the revised condition for shrinkability of T says that if an execution of L_T with initial state σ results in a violation of its residual property ψ_T or the clause $\psi_{[\,i\,]}$ corresponding to a past iteration i, then there exists a k-length subsequence U of T such that execution of L_U with the same initial state also violates ψ_U or $\psi_{[\,i\,]}$. Henceforth we will consider this to be the definition of shrinkability of iteration sequences.

As a technical point, notice that we include 0 as a possible value of a past iteration. Otherwise, any sequence that starts with iteration 1, would have an empty set of past iterations and the condition of k-shrinkability would be vacuously true for the sequence. We therefore include 0 as a past-iteration and define $\psi_{[\,0\,]}$ to be *true.* A pleasing consequence of this is when the iteration sequence consists of all the iterations of a loop, the revised definition that includes past-preservation also coincides with the definition of shrinkability of loops (Definition 1).

Consider the example `lmin` in Fig. 3(b) with `S` $=5$ and $d = 1$. Not all sequences of length two are 1-shrinkable by the revised definition. To see this, consider the case of an array `a` as $\{2, 1, 2, 3, 4\}$. Let T be [4,5] and take past iteration i as 1. Let `m` be 2 in a state σ. Then $\psi_{[\,1\,]} =$ `m` $\le$ `a[0]` $+ 1 \equiv$ `m` ≤ 3. Clearly, starting from state σ, for the residual loops of size 1 subsequences U, i.e. [4] and [5], the resulting `m` will be 3 and 2 respectively and $\psi_{[\,1\,]} \wedge \psi_U$ is satisfied. But starting from the same state σ, the residual loop L_T, will produce `m` $= 4$, and therefore $\psi_{[\,1\,]} \wedge \psi_T$ is not satisfied. On the other hand, it is easy to see that, for the same d, all the sequences of size 4 are 3-shrinkable.

We now formally prove the result that we have been working towards: For a loop to be k-shrinkable, it is enough if every iteration sequence of size $k+1$ is k-shrinkable. Our method of determining shrinkable and the shrink-factor will make use of this important result.

Theorem 1. *An array processing loop is k-shrinkable with respect to a property ψ, if every iteration sequence of size $k+1$ is k-shrinkable with respect to ψ.*

Proof. To show that the loop is k-shrinkable, it is enough to show that the complete iteration sequence of the loop is k-shrinkable according to Definition 3. However, we shall show a stronger condition that all sequences of size greater than k are k-shrinkable. The proof is by induction on the length n of an iteration sequence T of the loop. For the base case $n = k+1$, the k-shrinkability of T is a given in the statement of the theorem. Now let n be greater than $k+1$ and assume as the induction hypothesis that every sequence of length less than n is k-shrinkable. Let $T = j : T'$. As usual, we take a contrapositive view of the shrinkability condition and assume that for some past iteration i of T, starting from a state $\sigma \in \varphi_j$, the property $\psi_{[\,i\,]} \wedge \psi_T$ fails after executing L_T i.e. $\{\sigma\}L_T\{\neg\psi_{[\,i\,]} \vee \neg\psi_T\}$ is true. We show that there exists a k-sized subsequence $U \sqsubset T$ such that $\{\sigma\}L_U\{\neg\psi_{[\,i\,]} \vee \neg\psi_U\}$ is true.

Since $L_T = S_j; L_{T'}$ and $\psi_T = \psi_{[j]} \wedge \psi_{T'}$, we have $\{\sigma\}S_j; L_{T'}\{\neg\psi_{[\,i\,]} \vee \neg\psi_{[j]} \vee \neg\psi_{T'}\}$. Assume that starting with σ, the state reached after executing S_j, the loop body for the iteration j, is σ_1, i.e. $\{\sigma\}S_j\{\sigma_1\}$. We then have $\{\sigma_1\}L_{T'}\{\neg\psi_{[\,i\,]} \vee \neg\psi_{T'}\} \bigvee \{\sigma_1\}L_{T'}\{\neg\psi_{[j]} \vee \neg\psi_{T'}\}$. We show the existence of the desired U by assuming that the first disjunct is true. Since i and j are both past iterations for T', the proof in the case in which only the second disjunct is true is similar. Assume that the first iteration of T' is j'. Obviously $\sigma_1 \in \varphi_{j'}$. Since T' is k-shrinkable, we must have a k-sized subsequence $U' \sqsubset T'$ such that $\{\sigma_1\}L_{U'}\{\neg\psi_{[i]} \vee \neg\psi_{U'}\}$ is true. It follows that $\{\sigma\}S_j; L_{U'}\{\neg\psi_{[i]} \vee \neg\psi_{U'}\}$ and therefore $\{\sigma\}S_j; L_{U'}\{\neg\psi_{[i]} \vee \neg\psi_{[j]} \vee \neg\psi_{U'}\}$ are also true. Let T'' be $j : U'$. Obviously, $T'' \sqsubset T$. Since the size of T'' is $k+1$, T'' is k-shrinkable by the induction hypothesis and therefore there exists a k-sized subsequence $U \sqsubset T'' \sqsubset T$ such that $\{\sigma\}L_U\{\neg\psi_{[i]} \vee \neg\psi_U\}$ holds. ■

For a disjunctive property ψ, the definition (3) of sequence shrinkability, changes as follows:

$$\forall \sigma \in \varphi_j, \forall 0 \le i < j : ((\exists U \in \mathcal{P}_k(T) : \{\sigma\}L_U\{\psi_U \vee \psi_{[\,i\,]}\}) \implies \{\sigma\}L_T\{\psi_T \vee \psi_{[\,i\,]}\}) \quad (4)$$

Theorem 1 applies to disjunctive properties as well, and the proof is similar.

5 Determining Shrinkability and Property Checking

We now show how Theorem 1 can be used to empirically determine whether a given loop is shrinkable and also find the corresponding shrink-factor. Starting with 1, we repeatedly construct the program shown in Fig. 4 for successive values of `k`, the candidate shrink-factor, and feed it to a bounded model checker for verification. If the program is verified to be correct for some value of `k`, then Theorem 1 guarantees that the loop in the given program is `k`-shrinkable. The constructed program depends on `k`, the loop L and the property to be verified, ψ. The process stops when we either find a `k` for which the loop is shrinkable (success), or we reach a predefined limit l that is dependent on the available time and computing resources (failure). As we shall see in Sect. 6, the shrink-factors for shrinkable loops are usually small. This is a favourable situation, since programs with a smaller shrink-factors are relatively easier to verify than programs with larger shrink-factors.

5.1 Checking Shrinkability of an Iteration Sequence

Recall that according to Theorem 1, a loop is k-shrinkable, if every iteration sequence of length $k+1$ is k shrinkable. In addition, with our assumption that the loop has a statically computable upper bound of number of iterations, the number of such iteration sequences will be finite. Given a candidate `k`, the procedure `check_loop` in Fig. 4 non-deterministically chooses an iteration sequence T of length `k+1`, and attempts to verify that T is `k`-shrinkable. This is done in the procedure `check_iter_seq`, which encodes the criterion for sequence shrinkability, as given by Definition 3. The construction shown applies to conjunctive properties; disjunctive properties can be handled in a similar manner.

```
1  check_loop(k)
2  {
3    choose an arbitrary
4      iteration-sequence
5      T of size k+1
6    check_iter_seq(T);
7  }
```

(a) Checking loop shrinkability

```
1  check_iter_seq(T)
2  {
3    j = head(T); i = nondet();
4    assume(0 <= i < j);
5    X_initial= nondet(); c = true;
6    for each k sized U⊏T {
7      X = X_initial; L_U; c=ψ(i)∧ψ_U
8      if (!c) break;
9    }
10   X = X_initial ; L_T; r=ψ(i)∧ψ_T
11   assert(c⟹r);
12 }
```

(b) Checking sequence shrinkability

Fig. 4. Program construction for determining shrinkability. Note that `X` and `X_initial` are vectors of variables, and `nondet()`, accordingly, generates a vector of values.

Assume that the given program consists of an array processing loop L of the form `while(`C`)`$\{B\}; Q$ followed by the assertion assert(ψ). Let `X` denote the vector of variables which *may be* modified (by resolving dereferences, if any, using a safe points-to-analysis) in the loop body B. Recall that the implication in the criterion for shrinkability is required to hold for all states in φ_j, where j is the head of sequence T. The states in φ_j are over-approximated by assigning non-deterministic values to `X` (through `X_initial`). Thus our process of determining shrinkability is conservative and a future extension to this work would be a static analysis to obtain a better approximation of φ_j.

The loop in lines 6–9 checks the antecedent $(\forall U \in \mathcal{P}_k(T).\{\sigma\}L_U\{\psi_U \wedge \psi_{[i]}\})$ in the implication in the shrinkability condition (Definition 3), and stores the result in `c`. This loop executes a maximum of $k+1$ times, the number of subsequences of T of size k. Line 10 checks the consequent $\{\sigma\}L_T\{\psi_T \wedge \psi_{[\,i\,]}\}$ of the same implication, and stores it in `r`. Finally line 11 checks the condition for shrinkability, given by the implication $\texttt{c} \Rightarrow \texttt{r}$ itself. Observe that the residual loop for each subsequence U and the residual loop for the sequence T are all evaluated in the same state denoted by the values of the variables in `X_initial`. It is clear that the program shown in Fig. 4 can be automatically constructed for any given k, L, and ψ.

The fact that shrinkable loops usually have a low shrink-factor has two consequences for the procedure to determine shrinkability: (i) it allows us to keep the number l till which a program is tested for shrinkability at a low value without the fear of missing out many shrinkable programs, and (ii) since the `for` loop in lines 6–9 has a bound of $k+1$, and k is smaller than l, the shrinkability testing procedure is fairly efficient.

5.2 Property Checking for Shrinkable Loops

Once we discover that the loop of a program is k-shrinkable, we construct an abstract program that consists of a program fragment to non-deterministically choose a k-sized iteration sequence T, a residual loop L_T, and a residual property ψ_T. The abstract program is submitted to a BMC for verification. The motivating example of Fig. 1 illustrates the nature of the abstract program, and it is easy to generalize and automate the process of abstraction to arbitrary programs that are within the scope of our method.

Since the quantified property is also encoded as a loop, the residual property can also be constructed as a residual of this loop. Consider a program with a loop L for which the residual has to be constructed with respect to a k-length iteration sequence. Assume that the maximum iteration count of the loop is m. Let $a[e]$ be an arbitrary expression involving an array a of size n. Also assume that the index expression e is accelerable and is of the form $f(i)$, where $i \in [1..m]$ represents a particular loop iteration, and f is the acceleration function. The abstract program non-deterministically chooses a k-length iteration sequence, whose elements are in the range $[1..m]$. The iteration sequence is concretely represented as an array. A loop iterates over all the values of the iteration sequence. The expression $a[e]$ in the loop body is replaced by the corresponding accelerable expression $a[f(i)]$.

To make this clearer, consider the example in Fig. 5(a). Assume that the size `p` of the array is more than $(\texttt{n}+1)/2$. The loop initializes the array element `a[t]` with the value `2*t`. Assume that the loop is `k`-shrinkable for some property. The maximum iteration count m for the loop is $(\texttt{n}+1)/2$. The code in Fig. 5(b), written in a C-like notation, is an abstract description of the residual loop. The call to `init` initializes the array `T` with a non-deterministically chosen `k`-length iteration sequence. The C-style comment indicates the constraints on the chosen iteration sequence `T`. The conditions $\texttt{1} \le \texttt{T[}l\texttt{-1]} < \texttt{T[}l\texttt{]}$ and $\texttt{0} \le \texttt{2*(T[}l\texttt{]-1)} < \texttt{n}$ together ensure that the iteration sequence consists of increasing values in the range $[1\ldots m]$, and the condition $\texttt{T[}l\texttt{]-1} < \texttt{p}$ ensures that the chosen values do not cause an out-of-bounds access of the array. The `for` loop covering lines 7 to 11 iterates over the elements in `T`. Inside the loop body, `i` and `j` are computed through acceleration functions applied to the iteration numbers picked from `T`.

In practice, the constraints on the values in `T` would be enforced programmatically, and this is shown in Fig. 5(c). Here an increasing sequence of values are chosen, and the constraint that the chosen values are in the range $[1..m]$ is enforced through the the conditional `break`. Similarly, the constraint that the index of `a` does not exceed its bound is enforced through the `assume` at line 8. Finally, `assume(l==k)` ensures that the residual indeed iterates `k` times and does not break out of the loop earlier.

Our method can also be used when the program consists of a cascaded series of simple loops that can be coalesced into one simple loop. To elaborate, let the program be $\{Q_1; R_1; Q_2; R_2; Q_3; R_3\}$, where the Q_is are loop-free statements and the R_is are simple loops of the form `while` (C_i) $\{B_i\}$. Our method can handle such a program, if it can be transformed to a semantically equivalent program Q; `while` (C) $\{B_1; B_2; B_3\}$ for some loop-free statements, Q, and condition C. Even this simple strategy enabled us to verify 50 of the 81 programs with non-nested multiple loops in the SV-COMP 2017 benchmark suite. However, our method, in its present form, cannot handle nested loops.

```
1 int a[p];
2 int i=0,t=0;
3 while(i < n)
4 {
5   a[t] = i;
6   i+=2; t++;
7 }
```

(a) Given program

```
1  int a[p],i=0,t=0,T[k];
2  init(T) ;
3  //∀l∈1..k-1.
4  // 1≤T[l-1]<T[l]
5  // 0≤2*(T[l]-1)<n
6  // T[l]-1<p
7  for j in T {
8    i = (j-1)*2;
9    t = j-1; a[t]=i;
10   i+=2; t++;
11 }
```

(b) Abstract description of residual

```
1  int a[p], T[k], l, i=0, t=0;
2  init(T) ;
3  //∀l∈1..k-1. 1≤T[l-1]<T[l]
4  for (l=0;l<k;l++) {
5    j = T[l]; i = (j-1)*2;
6    t = j-1;
7    if (!(i < n)) break;
8    a[assume(0<=t<p),t] = i;
9    i+=2; t++;
10 }
11 assume(l==k);
```

(c) Residual as generated by the tool

Fig. 5. Example illustrating the residual of a shrinkable loop. Program in (b) is an abstract description of the residual, presented for ease of explanation

6 Implementation and Measurements

The proposed abstraction has been implemented in a tool called *VeriAbs* [7]. Within the scope of our method, i.e. a single loop followed by the property to be checked, the tool supports most C constructs including pointers, structure, arrays, heaps and non-recursive function calls. It uses LABMC [12] to discover index expressions that can be accelerated and CBMC 5.8 as the bounded model checker to determine shrinkability of the loop and to check the residual property on the abstracted program. If a loop is not found shrinkable within a candidate shrink-factor of 5, we report the shrinkability of the loop to be unknown. Given a program with a shrinkable loop, if the verification of the corresponding abstract program succeeds on the residual property, the tool declares the original program to be correct with respect to the given property. On the other hand, if the verification of the abstract program fails and the loop in the program has no loop-carried dependencies, the original program is declared to be incorrect. Otherwise the tool indicates its inability to decide on the correctness of the program.

6.1 Experiments

An early version of the tool *VeriAbs* competed in the SV-COMP 2017 verification competition [2], where it ranked third amongst the 17 participating tools in the *ArraysReach* category. We have re-run the current version on the same benchmark. We ran the experiment on a machine with two i7-4600U cores @2.70 GHz and 8 GB RAM. *ArraysReach* consisted of 135 programs, of which 95 are correct and the remaining 40 incorrect with respect to their properties. Table (a) of Fig. 6 categorizes these programs. 42 of the 135 programs were beyond the scope of *VeriAbs* because they either contained nested loops (12 programs) or contained multiple loops which were not collapsible (30 programs). Out of the remaining 93 programs, 89 programs were 1-shrinkable, 2 were 2-shrinkable, and while our tool could not find the shrinkability of the remaining 2 programs, we manually found those to be non-shrinkable.

Figure 6(b) gives the verification results of the 91 shrinkable programs. All correct programs except one were verified successfully. Moreover, none of the 26 incorrect programs were declared to be correct, demonstrating the soundness of our tool. 23 of these 26 incorrect programs also had no loop carried dependency, and thus the tool could rightly declare these as being incorrect. For the remaining 3 programs our tool remained indecisive. The timing data shows the average time taken in verifying each program. As expected, the bulk of time is taken in determining shrinkability, as the BMC has to verify $O(k^2)$ residual programs to determine that the shrink-factor is k, while property checking of the abstract program involves a loop with just k iterations. Given the limits of the machine configuration, the timings are reasonable.

Programs	True	False	Total
With nested loops	5	7	12
With non-collapsible multiple loops	24	6	30
Shrinkable	65	26	91
Shrinkability unknown	1	1	2
Total	95	40	135

(a) Programs categories

Results on shrinkable programs	#Cases	Average time per program (in seconds)	
		Checking shrinkability	Total
Property declared correct	64	30.60	39.68
Property declared incorrect	23	10.72	19.73
Unable to decide	4	227.26	236.06
Total	91	34.22	43.27

(b) Property verification results

Fig. 6. Experimental results for SV-COMP 2017 ArraysReach benchmarks

An interesting property of *Veriabs* is that, while it is limited by its ability to deal only with shrinkable loops, once a loop is discovered to be shrinkable, the method is impervious to either the existence or the size of loop bounds—increasing the loop bound does not cause an otherwise verifiable program to timeout. Comparison with the two tools that fared better than *VeriAbs* in the competition, namely *ceagle* [24] and *smack* [6], reveals interesting information. We selected four correct programs, one from each of the following categories, array copy, array initialisation, two index copying and finding minimum, of the test suite, and increased the array size considerably (from 100000 to 10000000). While both tools succeeded on the programs with the original array sizes, *smack* started timing out after the increase and *ceagle* either crashed or declared the programs to be incorrect. We surmise that the two tools are based on bounded model checking without any abstraction. In this respect, our tool performs better than these two tools that were placed ahead of ours in the competition.

7 Related Work

The various approaches to handle arrays have their roots in the types of static analyses used for property verification, namely: abstract interpretation, predicate abstraction, bounded model checking and theorem proving.

In abstract interpretation, arrays are handled using *array smashing*, *array expansion* and *array slicing*. In array smashing, all elements of an array are clubbed as a single anonymous element, with writes treated as weak updates. As a result it is imprecise. It cannot be used, for example, to verify the motivating example. In array expansion, array elements are explicated as a collection of scalar variables, and the resulting programs have fewer number of weak updates than array smashing. However, it works well only for small-sized arrays. A mix of smashing and expansion has been used in [4,5] to prove that the program does not perform executions with undefined behaviours such as out-of-bounds array accesses. In array slicing, the idea is to track partitions of arrays based upon some criteria inferred from programs [11,16,17]. Each partition is treated as an independent smashed element. The approach is further refined in [14] to introduce the notion of *fluid updates*, where a write operation may result in a strong update of one partition of the array and weak update of other partitions. In contrast to these approaches, our abstraction is based not only on the program but also on the associated property. By declaring an array-processing loop as k-shrinkable, we guarantee that an erroneous behaviour of the program with respect to the property can indeed be replayed on some k elements of the array.

Methods based on predicate abstraction go through several rounds of counterexample guided abstraction refinement (CEGAR). In each round a suitable invariant is searched based on the counter-example using *Craig interpolants* [21]. Tools like SATABS [9] and CPAchecker [3] are based on this technique. To handle arrays, the approach relies on finding appropriate quantified loop invariants. However generating interpolants for scalar programs is by itself a hard problem. With the inclusion of arrays, which require universally quantified interpolants, the problem becomes even harder [20,22]. Our method, in contrast, does not rely on the ability to find invariants. Instead, we find a bound on the number of loop iterations, and, in turn, the number of array elements that have to be accessed in a run of abstract program.

Theorem proving based methods generate a set of constraints, typically Horn clauses. The clauses relate invariants at various program points and the invariants are predicates over arrays. The constraints are then fed to a solver in order to find a model. However, these methods also face the same difficulty of synthesizing quantified invariants over arrays. A technique, called *k-distinguished cell abstraction*, addresses this problem by abstracting the array to only k elements. A 1-distinguished cell abstraction, for example, abstracts a predicate $P(a)$ involving an array a by $P'(i, a_i)$, where i and a_i are scalars. The relation between the two predicates is that $P'(i, a_i)$ holds whenever $P(a)$ holds and the value of $a[i]$ is a_i. The resulting constraints are easier to solve using a back end solver such as Z3 [13]. This technique and its variants appear in [22,23] and in [15], where the term *skolem constants* is used instead of distinguished cells. We experimented with VAPHOR, a tool based on [23]. By way of comparison, we present two examples, one with a $\exists$ property and the other with a $\forall$ property. The first program computes the minimum of an array and asserts that the minimum is the same as some element in the array. The second program copies all but 1 elements from one large array to another. It then asserts that the copied elements are pairwise equal. While our tool could verify both examples, VAPHOR declared the first program to be incorrect with 1 and 2 distinguished cell abstraction and timed out on the second program.

A method that is properly subsumed by our method is [18]. This uses only one distinguished element called a *witness element*, and transforms a program to a loop free scalar program. This program is model-checked using a BMC. This approach works well on what authors call *full array processing loops* and such loops are a proper subset of our 1-shrinkable loops.

8 Conclusion

We have proposed a fully automatic approach for property checking over array processing programs using loop shrinking. The approach enables us to verify properties over large or even unbounded loops by converting them to loops with a small finite bound. Towards this, we have defined a notion called shrinkability of a loop, and showed that arrays processed by k-shrinkable loops can be abstracted using only k elements. The abstracted program can then be checked using any bounded model checker as back-end. An important contribution of our method is an automated method to find out the required bound k. Although there are approaches that are based on abstracting an array by fewer elements, none of these provide a way to find out the number of elements that are sufficient to reason about the array. Our experiments have shown that the approach is powerful enough to handle a variety of array processing programs. As future work, we want to add a suitable refinement step to address false positives and extend our method to support nested loops and multi-dimensional arrays.

References

1. Allen, R., Kennedy, K.: Automatic translation of fortran programs to vector form. ACM Trans. Program. Lang. Syst. **9**(4), 491–542 (1987)
2. Beyer, D.: SV-COMP 2017–6th International Conference on Software Verification (2017). https://sv-comp.sosy-lab.org
3. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
4. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 85–108. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36377-7_5
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the PLDI 2003, pp. 196–207. ACM, New York (2003)
6. Carter, M., He, S., Whitaker, J., Rakamarić, Z., Emmi, M.: Smack software verification toolchain. In: Proceedings of the 38th International Conference on Software Engineering Companion, ICSE 2016, pp. 589–592. ACM, New York (2016)
7. Chimdyalwar, B., Darke, P., Chauhan, A., Shah, P., Kumar, S., Venkatesh, R.: VeriAbs: verification by abstraction (competition contribution). In: Legay, A., Margaria, T. (eds.) TACAS 2017, Part II. LNCS, vol. 10206, pp. 404–408. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_32
8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
9. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_40
10. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM, New York (1977)
11. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: SIGPLAN Not, vol. 46, no. 1, pp. 105–118, January 2011
12. Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: Proceedings of the DATE 2015, pp. 1407–1412. EDA Consortium, San Jose (2015)
13. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
14. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: beyond strong vs. weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_14
15. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proceedings of the POPL 2002, pp. 191–202. ACM, New York (2002)

16. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: SIGPLAN Not. vol. 40, no. 1, pp. 338–350, January 2005
17. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: SIGPLAN Not, vol. 43, no. 6, pp. 339–348, June 2008
18. Jana, A., Khedker, U.P., Datar, A., Venkatesh, R., Niyas, C.: Scaling bounded model checking by transforming programs with arrays. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 275–292. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_16
19. Jeannet, B., Schrammel, P., Sankaranarayanan, S.: Abstract acceleration of general linear loops. In: Proceedings of POPL 2014, pp. 529–540. ACM, New York (2014)
20. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_23
21. McMillan, K.L.: Applications of Craig interpolants in model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 1–12. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_1
22. Monniaux, D., Alberti, F.: A simple abstraction of arrays and maps by program translation. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 217–234. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9_13
23. Monniaux, D., Gonnord, L.: Cell morphing: from array programs to array-free Horn clauses. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 361–382. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_18
24. Wang, D.: Tool ceagle (2017). http://sts.thss.tsinghua.edu.cn/ceagle