Extending VIAP to Handle Array Programs

Pritom Rajkhowa and Fangzhen Lin

August 3, 2018

Extending VIAP to Handle Array Programs

Pritom Rajkhowa and Fangzhen Lin

Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong,
{prajkhowa,flin}@cse.ust.hk

Abstract. In this paper, we extend our previously described fully automated program verification system called VIAP primarily for verifying the safety properties of programs with integer assignments to programs with arrays. VIAP is based on a recent translation of programs to first-order logic proposed by Lin [1] and directly calls the SMT solver Z3. It relies more on reasoning with recurrences instead of loop invariants. In this paper, we extend it to programs with arrays. Our extension is not restricted to single dimensional arrays but general and works for multidimensional and nested arrays as well. In the most recent *SV-COMP* 2018 competition, VIAP with array extension came in second in the ReachSafety-Arrays sub-category, behind *VeriAbs*.

Keywords: Automatic Program Verification, Array, Structure, Multidimensional, Nested, First-Order Logic, Mathematical Induction, Recurrences, SMT, Arithmetic

1 Introduction

Arrays are widely used data structures in imperative languages. Automatic verification of programs with arrays is considered to be a difficult the task as it requires effective reasoning about loops and nested loops in case of multidimensional arrays. We have earlier reported a system called VIAP [2] that can prove non-trivial properties about programs with loops without using loop invariants. In this paper, we extend VIAP to arrays. In particular, we show how our system can handle multidimensional arrays. While there have been a few systems that can prove some non-trivial properties about one-dimensional arrays automatically, we are not aware of any that can do so for multidimensional arrays. Systems like Dafny [3],VeriFast [4] and Why [5] can indeed prove non-trivial properties about programs with multidimensional arrays, but they require user-provided loop invariant(s). Program verification is in general an undecidable problem, so there cannot be a fully automated system that works in all cases. Still, it is worthwhile to see how much one can do with fully automatic systems, hence the interest competitions like SV-COMP for fully automated systems.

In the following, we first describe how our system works. We then discuss some related work and finally make some concluding remarks.

2 Translation

Our translator consider programs in the following language:

```
E ::= array(E,...,E) |
      operator(E,...,E)
B ::= E = E |
      boolean-op(B,...,B)
P ::= array(E,...,E) = E |
      if B then P else P |
      P; P |
      while B do P
```

where the tokens `E, B, P` stand for integer expressions, Boolean expressions, and programs respectively. The token `array` stands for program variables, and the tokens `operator` and `boolean-op` stand for built-in integer functions and Boolean functions, respectively. Notice that for `array`, if its arity is 0, then it stands for an integer program variable. Otherwise, it is an array variable. Notice also that while the notation `array[i][j]` is commonly used in programming languages to refer to an array element, we use the notation `array(i,j)` here which is more common mathematics and logic.

Our system actually accepts C-like programs which are converted to these programs by a preprocessor. In particular, goto-statements are removed using the algorithm proposed in [6].

Given a program P, and a language $\boldsymbol{X}$, our system generates a set of first-order axioms denoted by $\Pi_P^{\boldsymbol{X}}$ that captures the changes of P on $\boldsymbol{X}$. Here by a language we mean a set of functions and predicate symbols, and for $\Pi_P^{\boldsymbol{X}}$ to be correct, $\boldsymbol{X}$ needs to include all program variables in P as well as any functions and predicates that can be changed by P.

The set $\Pi_P^{\boldsymbol{X}}$ of axioms are generated inductively on the structure of P. The algorithm is described in details in [1] and an implementation is [2]. This paper extends it to handle arrays. The inductive cases are given in table provided in the supplementary information depicted in [1]. There are two primitive cases, one for integer assignment and one for array element assignment. Before we describe them, we first describe our representation of arrays.

We consider arrays as first-order objects that can be parameters of functions, predicates, and can be quantified over. In first-order logic, this means that we have sorts for arrays, and one sort for each dimension. In the following, we denote by int the integer sort, and $array_k$ the k-dimensional array sort, where $k \geq 1$.

To denote the value of an array at some indices, for each $k \geq 1$, we introduce a special function named $d\text{k}array$ of the arity:

$$d\text{k}array: \; array_k \times int^k \to int,$$

as we consider only integer valued arrays. Thus $d1array(a, i)$ denotes the value of a one-dimensional array a at index i, i.e. $a[i]$ under a conventional notation,

[1] `https://goo.gl/2ZBGUr`

and $d2array(b,i,j)$ stands for $b[i][j]$ for two-dimensional array b. We can also introduce a function to denote the size of an array. However, we do not consider it here as the programs that we deal with in this paper does not involve operations about array sizes and we assume that all array references are legal.

When we translate a program to first-order axioms, we need to convert expressions in the program to terms in first-order logic. This is straightforward, given how we have decided to represent arrays. For example, if E is $a(1,2)+b(1)$, where a is a two-dimensional array and b a one-dimensional array, then $\hat{E}$, the first-order term that corresponds to E, is $d2array(a,1,2)+d1array(b,1)$.

We are now ready to describe how we generate axioms for assignments, First, for integer variable assignments:

Definition 21 *If P is* `V = E`*, and $V \in \boldsymbol{X}$, then $\Pi_P^{\boldsymbol{X}}$ is the set of the following axioms:*

$$\forall \boldsymbol{x}.X1(\boldsymbol{x}) = X(\boldsymbol{x}), \textit{ for each } X \in \boldsymbol{X} \textit{ that is different from } V,$$
$$V1 = \hat{E}$$

where for each $X \in \boldsymbol{X}$, we introduce a new symbol $X1$ with the same arity standing for the value of X after the assignment, and $\hat{E}$ is the translation of the expression `E` *into its corresponding term in logic as described above.*

For example, if P_1 is

```
I = a(1,2)+b(1)
```

and $\boldsymbol{X}$ is $\{I, a, b, d1array, d2array\}$ (a and b are for the two array variables in the assignment, respectively), then $\Pi_{P_1}^{\boldsymbol{X}}$ is the set of following axioms:

$$I1 = d2array(a,1,2) + d1array(b,1),$$
$$a1 = a,$$
$$b1 = b,$$
$$\forall x, i.d1array1(x,i) = d1array(x,i),$$
$$\forall x, i, j.d2array1(x,i,j) = d2array(x,i,j).$$

Again we remark that we assume all array accesses are legal. Otherwise, we would need axioms like the following to catch array errors:

$$\neg \textit{in-bound}(1,b) \rightarrow arrayError,$$
$$\neg \textit{in-bound}((1,2),a) \rightarrow arrayError,$$

where $\textit{in-bound}(\boldsymbol{i}, array)$ means that the index $\boldsymbol{i}$ is within the bound of $array$, and can be defined using array sizes.

Definition 22 *If P is* `V(e1,e2,...,ek) = E`*, then $\Pi_P^{\boldsymbol{X}}$ is the set of the following axioms:*

$$\forall \boldsymbol{x}.X1(\boldsymbol{x}) = X(\boldsymbol{x}), \textit{ for each } X \in \boldsymbol{X} \textit{ which is different from } dkarray,$$
$$dkarray1(x, i_1, ..., i_k) =$$
$$ite(x = V \wedge i_i = \hat{e_1} \wedge \cdots \wedge i_k = \hat{e_k}, \hat{E}, dkarray(x, i_1, ..., i_k)),$$

where $ite(c, e, e')$ is the conditional expression: if c then e else e'.

For example, if P_2 is `b(1)=a(1,2)+b(1)`, and $\boldsymbol{X}$ is $\{I, a, b, d1array, d2array\}$, then $\Pi_{P_2}^{\boldsymbol{X}}$ is the set of following axioms:

$$\begin{aligned}
&I1 = I,\\
&a1 = a,\\
&b1 = b,\\
&\forall x, i.d1array1(x, i) =\\
&\qquad ite(x = b \wedge i = 1, d2array(a, 1, 2) + d1array(b, 1), d1array(x, i)),\\
&\forall x, i, j.d2array1(x, i, j) = d2array(x, i, j).
\end{aligned}$$

Notice that $b1 = b$ means that while the value of b at index 1 has changed, the array itself as an *object* has not changed. If we have array assignments like `a=b` for array variables `a` and `b`, they will generate axioms like $a1 = b$.

We now give two simple examples of how the inductive cases work described in the tables[2] provided as supplementary material mentioned previously. See [1] for more details.

Consider P_3 which is the sequence of first P_1 then P_2:

```
I = a(1,2)+b(1);
b(1)=a(1,2)+b(1)
```

The axiom set $\Pi_{P_3}^{\boldsymbol{X}}$ is generated from $\Pi_{P_1}^{\boldsymbol{X}}$ and $\Pi_{P_2}^{\boldsymbol{X}}$ by introducing some new symbols to connect the output of P_1 with the input of P_2:

$$\begin{aligned}
&I2 = d2array(a, 1, 2) + d1array(b, 1),\\
&a2 = a,\\
&b2 = b,\\
&\forall x, i.d1array2(x, i) = d1array(x, i),\\
&\forall x, i, j.d2array2(x, i, j) = d2array(x, i, j),\\
&I1 = I2,\\
&a1 = a2,\\
&b1 = b2,\\
&\forall x, i.d1array1(x, i) =\\
&\qquad ite(x = b2 \wedge i = 1, d2array2(a2, 1, 2) + d1array2(b2, 1), d1array2(x, i)),\\
&\forall x, i, j.d2array1(x, i, j) = d2array2(x, i, j),
\end{aligned}$$

where $I2, a2, b2, d1array2, d2array2$ are new symbols to connect P_1's output with P_2's input. If we do not care about the intermediate values, these temporary

[2] `https://goo.gl/2ZBGUr`

symbols can often be eliminated. For this program, eliminating them yields the following set of axioms:

$$
\begin{aligned}
& I1 = d2array(a,1,2) + d1array(b,1), \\
& a1 = a, \\
& b1 = b, \\
& \forall x, i.d1array1(x,i) = \\
& \qquad ite(x = b \wedge i = 1, d2array(a,1,2) + d1array(b,1), d1array(x,i)), \\
& \forall x, i, j.d2array1(x,i,j) = d2array(x,i,j).
\end{aligned}
$$

The most important feature of the approach in [1] is in the translation of loops to a set of first-order axioms. The main idea is to introduce an explicit counter for loop iterations and an explicit natural number constant to denote the number of iterations the loop executes before exiting. It is best to illustrate by a simple example. Consider the following program P_4:

```
while I < M {
  I = I+1;
}
```

Let $\boldsymbol{X} = \{I, M\}$. To compute $\Pi_{P_4}^{\boldsymbol{X}}$, we need to generate first the axioms for the body of the loop, which in this case is straightforward:

$$
\begin{aligned}
& I1 = I + 1, \\
& M1 = M
\end{aligned}
$$

Once the axioms for the body of the loop are computed, they are turned into inductive definitions by adding a new counter argument to all functions and predicates that may be changed by the program. For our simple example, we get

$$\forall n.I(n+1) = I(n) + 1, \tag{1}$$

$$\forall n.M(n+1) = M(n), \tag{2}$$

where the quantification is over all natural numbers. We then add the initial case, and introduce a new natural number constant N to denote the terminating index:

$$
\begin{aligned}
& I(0) = I \wedge M(0) = M, \\
& I1 = I(N) \wedge M1 = M(N), \\
& \neg(I(N) < M(N)), \\
& \forall n.n < N \rightarrow I(n) < M(n).
\end{aligned}
$$

One advantage of making counters explicit and quantifiable is that we can then either compute closed-form solutions to recurrences like (1) or reason about them using mathematical induction. This is unlike proof strategies like k-induction where the counters are hard-wired into the variables. Again, for more details about this approach, see [1] which has discussions about related work as well as proofs of the correctness under operational semantics.

3 VIAP

We have implemented the translation to make it work with programs with a C-like the syntax used SymPy to simplify algebraic expressions and compute the closed-form solutions to simple recurrences, and finally verified assertions using Z3. The resulting system, called VIAP, is fully automated. We reported in an earlier paper [2] how it works on integer assignments. We have now extended it to handle arrays. We have described how the translation is extended to handle array element assignments in the previous section. In this section, we describe some implementation details.

We have already mentioned that temporary variables introduced during the translation process can often be eliminated, and that SymPy can be used to simplify algebraic expressions and compute closed-form solutions to simple recurrences. All of these have already been implemented for basic integer assignments and described in our earlier paper [2], therefore we do not repeat them here. For arrays, an important module that we added is for instantiation.

Our main objective is translating a program to first-order logic axioms with arithmetic. This translation provides the relationship between the input and output values of the program variables. The relationship between the input and output values of the program variables is independent of what one may want to prove about the program. SMT solver tools like Z3 is just an off shelf tool, so we never considered using the built-in array function there.

3.1 Instantiation

Instantiation is one of the most important phases of the pre-processing of axioms before the resulting set of formulas is passed on an SMT-solver according to some proof strategies. The objective is to help an SMT solver like Z3 to reason with quantifiers. Whenever an array element assignment occurs inside a loop, our system will generate an axiom like the following:

$$\begin{aligned}\forall x_1, x_2 ... x_{k+1}, n. d\mathrm{k}array_i(x_1, x_2 ... x_{k+1}, n+1) = \\ ite(x_1 = A \wedge x_2 = E_2 \wedge ... \wedge x_{k+1} = E_{h+1}, E, \\ d\mathrm{k}array_i(x_1, x_2 ... x_{k+1}, n))\end{aligned} \tag{3}$$

where

- A is a k-dimensional array.
- $d\mathrm{k}array_i$ is a temporary function introduced by translator.
- x_1 is an array name variable introduced by translator, and is universally quantified over arrays of k dimension.
- x_2,....,x_{k+1} are natural number variables representing array indices, and are universally quantified over natural numbers.
- n is the loop counter variable universally quantified over natural numbers.
- $E, E_2, ..., E_{k+1}$ are expressions.

For an axiom like (3), our system performs two types of instantiations:

– **Instantiating Arrays**: this substitutes each occurrence of variable x_1 in the axiom (3) by the array constant A, and generates the following axiom:

$$\begin{aligned}&\forall x_1, x_2 ... x_{k+1}.d\mathrm{k}array_i(A, x_2 ... x_{k+1}, n+1) = \\ &\qquad ite(x_2 = E_2 \wedge ... \wedge x_{k+1} = E_{h+1}, E, d\mathrm{k}array_i(A, x_2 ... x_{k+1}, n))\end{aligned} \tag{4}$$

– **Instantiating Array Indices**: This substitutes each occurrence of variable x_i, $2 \leq i \leq k$, in the axiom (4) by E_i, and generates the following axiom:

$$\forall n.d\mathrm{k}array_i(A, E_2 ... E_{k+1}, n+1) = E \tag{5}$$

Example 1. This example shows the effect of instantiation on a complete example. Consider the following Battery Controller program from the SV-COMP benchmark [7,8]:

```
int COUNT , MIN ,i=1 ;
int volArray[COUNT];
if( COUNT %4 != 0) return ;
while(i <= COUNT /4) {
     if (5 >= MIN ){ volArray [i*4-4]=5; }
         else { volArray [i*4-4]=0; }
     if (7 >= MIN ){ volArray [i*4-3]=7; }
         else { volArray [i*4-3]=0; }
    if (3 >= MIN ){ volArray [i*4-2]=3; }
         else { volArray [i*4-2]=0; }
     if (1 >= MIN ){ volArray [i*4-1]=1; }
         else  { volArray [i*4-1]=0; }
     assert ( volArray[i]>=MIN ||volArray[i]==0);
     i=i+1; }
```

Our system generates the following set of axioms after the recurrences from the loop are solved by SymPy:

1. $COUNT1 = COUNT$
2. $j1 = j$
3. $volArray1 = volArray$
4. $MIN1 = MIN$
5. $i1 = ite(((COUNT\%4) == 0), (N_1 + 1), 1)$
6.

$$\begin{aligned}&\forall x_1, x_2.d1array1(x_1, x_2) = \\ &\qquad ite((COUNT\%4) == 0, d1array_{13}(x_1, x_2, N_1), d1array(x_1, x_2))\end{aligned}$$

7.

$$\begin{array}{l}
\forall x_1, x_2, n_1.d1array_{13}(x_1, x_2, (n_1+1)) = ite(1 \geq MIN, \\
\quad ite(x_1 = volArray \wedge x_2 = (n_1+1)*4-1, 1, d1array_{13}(volArray, x_2, n_1)), \\
\quad ite(x_1 = volArray \wedge x_2 = (n_1+1)*4-1, 0, \\
\qquad ite(3 \geq MIN, \\
\qquad\; ite(x_1 = volArray \wedge x_2 = (n_1+1)*4-2, 1, d1array_{13}(volArray, x_2, n_1)) \\
\qquad\; ite(x_1 = volArray \wedge x_2 = (n_1+1)*4-2, 0, \\
\qquad\quad ite(7 \geq MIN, \\
\qquad\qquad ite(x_1 = volArray \wedge x_2 = (n_1+1)*4-3, 1, d1array_{13}(volArray, x_2, n_1)), \\
\qquad\qquad ite(x_1 = volArray \wedge x_2 = (n_1+1)*4-3, 0, \\
\qquad\qquad\; ite(5 \geq MIN, \\
\qquad\qquad\quad ite(x_1 = volArray \wedge x_2 = (n_1+1)*4-4, 1, \\
\qquad\qquad\qquad\qquad d1array_{13}(volArray, x_2, n_1)), \\
\qquad\qquad\quad ite(x_1 = volArray \wedge x_2 = (n_1+1)*4-4, 0, \\
\qquad\qquad\qquad\qquad d1array_{13}(volArray, x_2, n_1)))))))))
\end{array}$$

8. $\forall x_1, x_2.d1array_{13}(x_1, x_2, 0) = d1array(x_1, x_2)$
9. $(N_1 + 1) > (COUNT/4)$
10. $\forall n_1.(n_1 < N_1) \rightarrow (n_1 + 1) \leq (COUNT/4)$

where $(COUNT\%4) == 0$ is copied directly from the conditional `COUNT%4 !=0` in the program and is converted to $(COUNT\%4) = 0$ in Z3.

The instantiation module will then generate the following new axioms from the one in 7:

1. $\forall n_1.0 \leq n_1 < COUNT \rightarrow$
   $d1array_{13}(volArray, (n_1+1)*4-1, n_1+1) = ite(1 \geq MIN, 1, 0)$
2. $\forall n_1.0 \leq n_1 < COUNT \rightarrow$
   $d1array_{13}(volArray, (n_1+1)*4-2), n_1+1) = ite(3 \geq MIN, 1, 0)$
3. $\forall n_1.0 \leq n_1 < COUNT \rightarrow$
   $d1array_{13}(volArray, (n_1+1)*4-3, n_1+1) = ite(7 \geq MIN, 1, 0)$
4. $\forall n_1.0 \leq n_1 < COUNT \rightarrow$
   $d1array_{13}(volArray, (n_1+1)*4-4, n_1+1) = ite(5 \geq MIN, 1, 0)$

For the the following assertion to prove:

$$d1array_{13}(volArray, n_1+0, N_1) \geq 2 \vee d1array_{13}(volArray, n_1+0, N_1) = 0$$

VIAP successfully proved the assertion irrespective of the value of `COUNT`. On the other hand, tools like CBMC [5] and SMACK+Corral [9] which prove this assertion for arrays with small values of `COUNT=100` fail when the COUNT value is non-deterministic or bigger(`COUNT=10000`) and this has been also reported by [8]. Other tools like UAutomizer [10], Seahorn [11], ESBMC [12], Ceagle [13], Booster [14], and Vaphor [15] fail to prove the assertion even for a small value of `COUNT`. To our knowledge, Vaphor [15] and VeriAbs [16] are the only other systems that can prove this assertion regardless of the value of `COUNT`.

3.2 Proof strategies

Currently, VIAP tries to prove the given assertion by first trying it directly with Z3. If this direct proof fails, it tries a simple induction scheme which works as follows: if N is a natural number constant in the assertion $\beta(N)$, it is replaced by a new natural number variable n and proves the universal assertion $\forall n\beta(n)$ using an induction on n. There is much room for improvement here, especially in the heuristics for doing the induction. This is an active future work for us.

3.3 Multi-dimensional arrays

Finally, we show an example of a program with multi-dimensional arrays. In fact, with our approach, nothing special needs to be done here. Consider the following program for doing matrix addition:

```
int i,j,A[P][Q],B[P][Q],C[P][Q];
i=0;j=0;
while(i < P){
  j=0;
  while(j < Q){
    C[j][i] = A[i][j]+B[i][j];
    assert(C[i][j] == A[i][j]+B[i][j])
    j=j+1;}
  i=i+1;}
```

For this program, our system genrates the following set of axioms:

1. $P1 = P$
2. $Q1 = Q$
3. $A1 = A$
4. $B1 = B$
5. $C1 = C$
6. $i1 = (N_2 + 0)$
7. $j1 = j_5(N_2)$
8. $\forall x_1, x_2, x_3.d2array1(x_1, x_2, x_3) = d2array5(x_1, x_2, x_3, N_2)$
9.

$$
\begin{aligned}
&\forall x_1, x_2, x_3, n_1, n_2.d2array_2(x_1, x_2, x_3, (n_1 + 1), n2) = \\
&\quad ite(x_1 = C \wedge x_2 = n1 \wedge x_3 = n_2, \\
&\qquad d2array_2(A, n_1 + 0, n_2 + 0, n_1, n_2) + d2array_2(B, n_1 + 0, n_2 + 0, n_1, n_2), \\
&\qquad d2array2(x1, x2, x3, n1, n2))
\end{aligned}
$$

10. $\forall x_1, x_2, x_3, n_2.d2array_2(x_1, x_2, x_3, 0, n_2) = d2array_5(x_1, x_2, x_3, n_2)$
11. $\forall n_2.(N_1(n_2) \geq Q)$
12. $\forall n_1, n_2.(n_1 < N_1(n_2)) \rightarrow (n_1 < Q)$
13. $\forall n_2.j_5((n_2 + 1)) = (N_1(n_2) + 0)$
14. $\forall x_1, x_2, x_3, n_2.d2array_5(x_1, x_2, x_3, (n_2{+}1)) = d2array2(x_1, x_2, x_3, N_1(n_2), n_2)$
15. $j_5(0) = 0$

16. $\forall x_1, x_2, x_3 . d2array_5(x_1, x_2, x_3, 0) = d2array(x_1, x_2, x_3)$
17. $(N_2 \geq P)$
18. $\forall n_2.(n_2 < N_2 \rightarrow (n_2 < P))$

and the following assertion to prove:

$$\forall n_1, n_2 . d2array_5(C, (n_1 + 0), (n_2 + 0), N_2) = \\ d2array_5(A, (n_1 + 0), (n_2 + 0), N_2) + d2array_5(A, (n_1 + 0), (n_2 + 0), N_2).$$

VIAP proved it in 30 seconds using the direct proof strategy. In comparison, given that the program has multi-dimensional arrays and nested loops, state-of-art systems like SMACK+Corral [9],UAutomizer [10], Seahorn [11], ESBMC [12], Ceagle [13], Booster [14], VeriAbs [16] and Vaphor [15] failed to prove it.

Verifiability:VIAP is implemented in python. The source code, benchmarks and the full experiments are available in [17].

4 Related work

Tools like Dafny [3], VeriFast [4] and Why [5] can prove the correctness of a program with multi-dimensional array only if provided with suitable invariants, however, VIAP is a fully automatic prover. The Vaphor tool [15], is a Horn clause base approach which uses the Z3[18] solver in the back-end, and cannot handle array program with non-sequential indices, unlike VIAP. Seahorn [11] is another horn clause based verification framework. Seahorn can only prove 3 out of 88 programs from the Array-Example directory of SV-COMP benchmarks. There is a sizable body of work that considers the verification of C programs including programs with an array such as SMACK+Corral [9], UAutomizer[10], ESBMC [12], Ceagle [13]. The major limitation of UAutomizer is that it can only handle most of the programs with array when the property is not quantified. Ceagle [13] and SMACK+Corral [9] got first and second position in the ReachSafety-Arrays sub-category of ReachSafety category. SMACK+Corral is not very effective when it comes to dealing with multi-dimensional programs. Similarly, the Booster [14] verification tool failed when interpolants for universally quantified array properties (like programs with multidimensional array) became hard to compute.

5 Concluding remarks and future work

In this paper, we describe an approach to prove the correctness of imperative programs with arrays in a system we implemented in an earlier work, called VIAP. VIAP is continuously evolving. In the future, we will work on incorporating proofs of the following in VIAP - (1) programs with more advanced data structures like linked lists, binary trees. (2) program termination (3) and object-oriented programs in languages like Java.

Acknowledgment: We would like to thank Jianmin Ji, Peisen YAO, Anand Inasu Chittilappilly and Prashant Saikia for useful discussions. We are grateful to the developers of Z3 and SymPy for making their systems available for open use. All errors remain ours. This work was supported in part by the HKUST grant IEG16EG01.

References

1. F. Lin, “A formalization of programs in first-order logic with a discrete linear order,” *Artificial Intelligence*, vol. 235, pp. 1 – 25, 2016.
2. P. Rajkhowa and F. Lin, “Viap - automated system for verifying integer assignment programs with loops,” in *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2016, Timisoara, Romania, September 21-24, 2017*. Available at `https://github.com/VerifierIntegerAssignment/sv-comp/blob/master/viap-automated-system.pdf`.
3. K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *Proceedings of the 16th LPAR*, LPAR’10, (Berlin, Heidelberg), pp. 348–370, Springer-Verlag, 2010.
4. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, “Verifast: A powerful, sound, predictable, fast verifier for c and java,” NFM’11, (Berlin, Heidelberg), pp. 41–55, Springer-Verlag, 2011.
5. J.-C. Filliâtre and A. Paskevich, *Why3 — Where Programs Meet Provers*, pp. 125–128. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.
6. A. M. Erosa and L. J. Hendren, “Taming control flow: A structured approach to eliminating goto statements,” in *Proceedings of the IEEE Computer Society ICCLs, Toulouse, France* (H. E. Bal, ed.), pp. 229–240, 1994.
7. Program Committee / Jury, *SV-COMP:Benchmark Verification Tasks*, 2018.
8. S. Chakraborty, A. Gupta, and D. Unadkat, *Verifying Array Manipulating Programs by Tiling*, pp. 428–449. Cham: Springer International Publishing, 2017.
9. M. Carter, S. He, J. Whitaker, Z. Rakamarić, and M. Emmi, “Smack software verification toolchain,” ICSE ’16, (New York, NY, USA), pp. 589–592, ACM, 2016.
10. M. Chechik and J. Raskin, eds., *TACAS 2016,, April 2-8, 2016, Proceedings*, vol. 9636 of *LNCS*, Springer, 2016.
11. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, *The SeaHorn Verification Framework*, pp. 343–361. Cham: Springer International Publishing, 2015.
12. L. Cordeiro, J. Morse, D. Nicole, and B. Fischer, *Context-Bounded Model Checking with ESBMC 1.17*, pp. 534–537. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.
13. D. Wang, C. Zhang, G. Chen, M. Gu, and J. Sun, “C code verification based on the extended labeled transition system model,” in *Proceedings of the MoDELS 2016 Demo and Poster Sessions co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Saint-Malo, France, October 2-7, 2016.*, pp. 48–55, 2016.
14. F. Alberti, S. Ghilardi, and N. Sharygina, “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.
15. D. Monniaux and L. Gonnord, *Cell Morphing: From Array Programs to Array-Free Horn Clauses*, pp. 361–382. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016.
16. B. Chimdyalwar, P. Darke, A. Chauhan, P. Shah, S. Kumar, and R. Venkatesh, “Veriabs: Verification by abstraction competition contribution,” in *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*, (New York, NY, USA), pp. 404–408, Springer-Verlag New York, Inc., 2017.

17. Pritom Rajkhowa and Fangzhen Lin, *VIAP tool and experiments*, 2018. Available at `https://github.com/VerifierIntegerAssignment/VIAP_ARRAY`.
18. L. De Moura and N. Bjørner, "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*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.