Extending VIAP to Handle Array Programs

Pritom Rajkhowa and Fangzhen Lin

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

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, Multi-dimensional, 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 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 considers programs in the following language:

\[
E ::= \text{array}(E, \ldots, E) | \\
\text{operator}(E, \ldots, E)
\]

\[
B ::= E = E | \\
\text{boolean-op}(B, \ldots, B)
\]

\[
P ::= \text{array}(E, \ldots, E) = E | \\
\text{if } B \text{ then } P \text{ else } P | \\
P; P | \\
\text{while } B \text{ do } P
\]

where the tokens \(E, B, P\) stand for integer expressions, Boolean expressions, and programs, respectively. The token \text{array} stands for program variables, and the tokens \text{operator} and \text{boolean-op} stand for built-in integer functions and Boolean functions, respectively. Notice that for \text{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 \text{array}[i][j] is commonly used in programming languages to refer to an array element, we use the notation \text{array}(i,j) here which is more common in 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 \cite{6}.

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

The set \(\Pi_{X}^{P}\) of axioms are generated inductively on the structure of \(P\). The algorithm is described in details in \cite{1} and an implementation is \cite{2}. This paper extends it to handle arrays. The inductive cases are given in table provided in the supplementary information depicted in \cite{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 \(\text{int}\) the integer sort, and \(\text{array}_k\), 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 \(d\text{array}\) of the arity:

\[
d\text{array} : \text{array}_k \times \text{int}^k \rightarrow \text{int},
\]

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

\[1\] https://goo.gl/2ZBGUx
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 \( E = a(1, 2) + b(1) \), where \( a \) is a two-dimensional array and \( b \) a one-dimensional array, then \( E \), the first-order term that corresponds to \( E \), is \( d2array(a, 1, 2) + darray(b, 1) \).

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

**Definition 21** If \( P = V = E \), and \( V \in X \), then \( \Pi^X_p \) is the set of the following axioms:

\[
\forall x. X1(x) = X(x), \text{ for each } X \in X \text{ that is different from } V, \\
V1 = \hat{E}
\]

where for each \( X \in X \), 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 \( X \) is \( \{I, a, b, darray, darray\} \) (\( a \) and \( b \) are for the two array variables in the assignment, respectively), then \( \Pi^X_{P_1} \) is the set of following axioms:

\[
I1 = d2array(a, 1, 2) + darray(b, 1), \\
a1 = a, \\
b1 = b, \\
\forall x, i, darray1(x, i) = darray(x, i), \\
\forall x, i, j, d2array1(x, i, j) = d2array(x, i, j).
\]

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

\[
\neg \text{in-bound}(1, b) \rightarrow \text{arrayError}, \\
\neg \text{in-bound}(1, 2, a) \rightarrow \text{arrayError},
\]

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

**Definition 22** If \( P = V(e_1, e_2, \ldots, e_k) = \hat{E} \), then \( \Pi^X_p \) is the set of the following axioms:

\[
\forall x. X1(x) = X(x), \text{ for each } X \in X \text{ which is different from } darray, \\
darray1(x, i_1, \ldots, i_k) = \\
\text{ite}(x = V \land i_i = \hat{e}_i \land \cdots \land i_k = \hat{e}_k, \hat{E}, darray(x, i_3, \ldots, i_k)),
\]
where $\text{ite}(c, e, e')$ is the conditional expression: if $c$ then $e$ else $e'$.

For example, if $P_2$ is $b(1) = a(1, 2) + b(1)$, and $X$ is $\{I, a, b, d\text{array}_1, d\text{array}_2\}$, then $\Pi^X_{P_2}$ is the set of following axioms:

$I_1 = I,$
$a_1 = a,$
$b_1 = b,$
$\forall x, i. d\text{array}_1(x, i) =$
$\text{ite}(x = b \land i = 1, d\text{array}(a, 1, 2) + d\text{array}(b, 1), d\text{array}(x, i)),$
$\forall x, i, j. d\text{array}_1(x, i, j) = d\text{array}(x, i, j)).$

Notice that $b_1 = b$ 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 $a_1 = b$.

We now give two simple examples of how the inductive cases work described in the tables provided as supplementary material mentioned previously. See \cite{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 $\Pi^X_{P_3}$ is generated from $\Pi^X_{P_1}$ and $\Pi^X_{P_2}$ by introducing some new symbols to connect the output of $P_1$ with the input of $P_2$:

$I_2 = d\text{array}(a, 1, 2) + d\text{array}(b, 1),$
$a_2 = a,$
$b_2 = b,$
$\forall x, i. d\text{array}_2(x, i) = d\text{array}(x, i),$
$\forall x, i, j. d\text{array}_2(x, i, j) = d\text{array}(x, i, j),$
$I_1 = I_2,$
$a_1 = a_2,$
$b_1 = b_2,$
$\forall x, i. d\text{array}_1(x, i) =$
$\text{ite}(x = b_2 \land i = 1, d\text{array}_2(a_2, 1, 2) + d\text{array}_2(b_2, 1), d\text{array}_2(x, i)),$
$\forall x, i, j. d\text{array}_1(x, i, j) = d\text{array}_2(x, i, j),$

where $I_2, a_2, b_2, d\text{array}_2, d\text{array}_2$ 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 symbols can often be removed. For example, if $P_2$ is $b(1) = a(1, 2) + b(1)$, then $\Pi^X_{P_2}$ is the set of following axioms:

$I_1 = I,$
$a_1 = a,$
$b_1 = b,$
$\forall x, i. d\text{array}_1(x, i) =$
$\text{ite}(x = b \land i = 1, d\text{array}(a, 1, 2) + d\text{array}(b, 1), d\text{array}(x, i)),$
$\forall x, i, j. d\text{array}_1(x, i, j) = d\text{array}(x, i, j).$

2 https://goo.gl/2ZBGUl
symbols can often be eliminated. For this program, eliminating them yields the following set of axioms:

\[
I_1 = d2array(a, 1, 2) + d1array(b, 1), \\
a_1 = a, \\
b_1 = b, \\
\forall x, i. d1array1(x, i) = \\
\text{ite}(x = b \land i = 1, d2array(a, 1, 2) + d1array(b, 1), d1array(x, i)), \\
\forall x, i, j. d2array1(x, i, j) = d2array(x, i, j).
\]

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 \( P_4 \):

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

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

\[
I_1 = I + 1, \\
M_1 = M
\]

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

\[
\forall n. I(n + 1) = I(n) + 1, \quad (1) \\
\forall n. M(n + 1) = M(n), \quad (2)
\]

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:

\[
I(0) = I \land M(0) = M, \\
I_1 = I(N) \land M_1 = M(N), \\
\neg(I(N) < M(N)), \\
\forall n. n < N \rightarrow I(n) < M(n).
\]

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 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:

\[
\forall x_1, x_2, ..., x_{k+1}, n. d\text{array}(x_1, x_2, ..., x_{k+1}, n + 1) = \\
\text{ite}(x_1 = A \land x_2 = E_2 \land ... \land x_{k+1} = E_{h+1}, E, d\text{array}(x_1, x_2, ..., x_{k+1}, n))
\]

(3)

where

- $A$ is a $k$-dimensional array.
- $d\text{array}$ 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:

\[
\forall x_1, x_2 \ldots x_{k+1}. \text{darray}_i(A, x_2 \ldots x_{k+1}, n + 1) = \\
\text{ite}(x_2 = E_2 \land \ldots \land x_{k+1} = E_{h+1}, E, \text{darray}_i(A, x_2 \ldots x_{k+1}, n))
\] (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:

\[
\forall n. \text{darray}_i(A, E_2 \ldots E_{k+1}, n + 1) = E
\] (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]:

```
1. int COUNT, MIN, i=1;
2. int volArray[COUNT];
3. if (COUNT %4 != 0) return;
4. while(i <= COUNT/4) {
5.     if (5 >= MIN) { volArray[i*4-4]=5; }
6.         else { volArray[i*4-4]=0; }
7.     if (7 >= MIN) { volArray[i*4-3]=7; }
8.         else { volArray[i*4-3]=0; }
9.     if (3 >= MIN) { volArray[i*4-2]=3; }
10.        else { volArray[i*4-2]=0; }
11.    if (1 >= MIN) { volArray[i*4-1]=1; }
12.       else { volArray[i*4-1]=0; }
13. assert (volArray[i]>=MIN || volArray[i]==0);
14. 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. $i1 = \text{ite}(((COUNT\%4) == 0), (N1 + 1), 1)$
6. $\forall x_1, x_2. \text{darray}_1(x_1, x_2) =$

\[
\text{ite}( (COUNT\%4) == 0, \text{darray}_13(x_1, x_2, N1), \text{darray}(x_1, x_2))
\]
∀ COUNT VIAP successfully proved the assertion irrespective of the value of COUNT. Systems that can prove this assertion regardless of the value of COUNT are Booster [14] and Vaphor [15]. Fail to prove the assertion even for a small value COUNT = 10000 is non-deterministic or bigger. Assertions for arrays with small values of COUNT are non-deterministic or bigger in the other hand, tools like CBMC [5] and SMACK+Corral [9] which prove this assertion for arrays with small values 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.

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

1. ∀ n₁, 0 ≤ n₁ < COUNT →
   \[ darray₁₃(volArray, (n₁ + 1) \cdot 4 - 1, n₁ + 1) = ite(1 \geq MIN, 1, 0) \]
2. ∀ n₁, 0 ≤ n₁ < COUNT →
   \[ darray₁₃(volArray, (n₁ + 1) \cdot 4 - 2, n₁ + 1) = ite(3 \geq MIN, 1, 0) \]
3. ∀ n₁, 0 ≤ n₁ < COUNT →
   \[ darray₁₃(volArray, (n₁ + 1) \cdot 4 - 3, n₁ + 1) = ite(7 \geq MIN, 1, 0) \]
4. ∀ n₁, 0 ≤ n₁ < COUNT →
   \[ darray₁₃(volArray, (n₁ + 1) \cdot 4 - 4, n₁ + 1) = ite(5 \geq MIN, 1, 0) \]

For the the following assertion to prove:

\[ darray₁₃(volArray, n₁ + 0, n₁) \geq 2 \vee darray₁₃(volArray, n₁ + 0, n₁) = 0 \]

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 in COUNT = 10000 and this has been also reported by [8]. Other tools like UAumizer [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:

```c
1. int i,j,A[P][Q],B[P][Q],C[P][Q];
2. i=0;j=0;
3. while(i < P){
4.   j=0;
5.   while(j < Q){
7.     assert(C[i][j] == A[i][j]+B[i][j])
8.   j=j+1;}
9. i=i+1;}
```

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

1. $P1 = P$
2. $Q1 = Q$
3. $A1 = A$
4. $B1 = B$
5. $C1 = C$
6. $i1 = (N2 + 0)$
7. $j1 = j5(N2)$
8. $\forall x1,x2,x3,d2array1(x1,x2,x3) = d2array5(x1,x2,x3,N2)$
9. $\forall x1,x2,x3,n1,n2.d2array2(x1,x2,x3,(n1 + 1),n2) =
   \text{ite}(x1 = C \land x2 = n1 \land x3 = n2, \newline
   d2array2(A,n1 + 0,n2 + 0,n1,n2) + d2array2(B,n1 + 0,n2 + 0,n1,n2), \newline
   d2array2(x1,x2,x3,n1,n2))$
10. $\forall x1,x2,x3,n2.d2array2(x1,x2,x3,0,n2) = d2array5(x1,x2,x3,n2)$
11. $\forall n2.(N1(n2) \geq Q)$
12. $\forall n1,n2.(n1 < N1(n2)) \rightarrow (n1 < Q)$
13. $\forall n2,j5((n2 + 1)) = (N1(n2) + 0)$
14. $\forall x1,x2,x3,n2.d2array5(x1,x2,x3,(n2 +1)) = d2array2(x1,x2,x3,N1(n2),n2)$
15. $j5(0) = 0$
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], UAutorimer [10], ESBMC [12], Ceagle [13]. The major limitation of UAnotimer 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


