Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2017-IC-008

2017-IC-008

A Framework for Array Invariants Synthesis in Induction-loop Programs

Bin Li, Juan Zhai, Zhenhao Tang, Enyi Tang, Jianhua Zhao

Asia-Pacific Software Engineering Conference 2017

A Framework for Array Invariants Synthesis in Induction-loop Programs

LI Bin, ZHAI Juan, TANG Zhenhao, TANG Enyi, ZHAO Jianhua
State Key Laboratory of Novel Software Technology
Dept. of Computer Sci. and Tech. Nanjing University
Email: {hsslb,zhaijuan,tangzh}@seg.nju.edu.cn, {eytang,zhaojh}@nju.edu.cn

***Abstract*—Abstract interpretation is capable of inferring a wide variety of quantifier-free program invariants. In this paper, we propose a general framework for building universally quantified abstract domains that leverage existing quantifier-free domains in induction-loop programs. This method is sound and converges in finite time. We instantiate this framework using two quantifier-free domains: difference-bound matrices with disequality constraints ($dDBM$) domain and polynomial equations domain. The experiments on a variety of programs using arrays demonstrate the feasibility of the approach.**

I. Introduction

Arrays are an important data structure in all common languages. In many cases, to verify the correctness of programs using arrays an analysis must be able to discover properties of array elements. For example, in scientific programming, a sparse matrix is represented by several arrays, and indirect indexing is used to access matrix elements. In this case, to verify that all array accesses are in bounds, an analysis has to discover upper and lower bounds on the elements stored in the index arrays.

Static reasoning about the behaviors of such programs is a challenging problem. Firstly, array indexing induces complex semantics, in particular the possibility of aliasing. Moreover, an array may represent a large or unbounded number of variables, since the size of the array can be large or even unknown. Invariants with quantifiers are important for verification and static analysis of programs over arrays due to the unbounded nature of array structures. Universally quantified invariants can express relationships among array elements and properties involving arrays and scalar variables.

Universally quantified invariant synthesis for programs manipulating arrays with unbounded data has drawn wide attention in the academic field, see e.g. [8], [10], [11], [13]–[15], [21]. Approaches presented in these papers combine inductive reasoning with predicate abstraction, array partitioning and templates techniques and normally require user guidance in providing necessary templates, assertions or predicates, or only deal with one-dimensional arrays.

In this paper, we present a framework for automatically inferring array invariants without any user guidance and without using a priori defined boolean templates or predicates. Our framework can transform a quantifier-free abstract domain into a quantified domain in induction-loop programs. Since there already is a huge variety of quantifier-free domains available, our framework can leverage them. Moreover, our framework can process multi-dimensional arrays.

Induction-loops are loop statements that manipulate arrays by loop control variables. A loop control variable is a variable that gets increased or decreased by a fixed amount every time. However, it is allowable to increase or decrease such variables multiple times (sometimes even unbounded/unknown times) on every iteration of a loop. Induction-loops are common in programs. Program 1 and 2 show two examples of induction-loop programs, which will be used throughout the paper. Program $find$ (Program 2) is the example that loop control variables i and j gets increased or decreased multiple times on every iteration of the outer loop.

Program 1: init

```
1  unsigned size;
2  int A[size];
3  i:=0;
4  while (i<size) {
5      A[i]:=0;
6      i:=i+1;
7  }
```

Program 2: find: segmentation phase of the QuickSort

```
1   x := A[0] ;
2   i := 1;
3   j := size-1;
4   while (i<=j) {
5       if (A[i] < x) {
6           A[i-1] := A[i];
7           i := i + 1;
8       }
9       else{
10          while (j>=i&&A[j]>=x) {
11              j := j-1;
12          }
13          if (j > i) {
14              A[i-1] := A[j];
15              A[j] := A[i];
16              i := i + 1 ;
17              j := j-1;
18          }
19      }
20  }
21  A[i-1] := x ;
```

Induction-loops often have quantified properties in the following form:

$$\forall x \in [e_1, c, e_2), \psi(x)$$

978-1-5386-3681-7/17 $31.00 © 2017 IEEE
DOI 10.1109/APSEC.2017.8

where $[e_1, c, e_2)$ denotes the set $\{e_1 + n * c \mid n \geq 0 \wedge n < (e_2 - e_1)/c\}$.

For instance, the following property (1) is the important postcondition for "init" program (i.e., all elements of the array are initialized to zero).

$$\forall x \in [0, 1, size), A[x] = 0 \quad (1)$$

The following properties (2)-(3) are the important postconditions for "find" program (i.e., the array is segmented).

$$\forall k \in [0, 1, i-1), A[k] < x \quad (2)$$

$$\forall k \in [i, 1, size), A[k] \geq x \quad (3)$$

So, in spite of severe restrictions both on programs and properties, our framework allows interesting properties to be found about non trivial programs.

Contributions. The contributions of our work include:

- The main contribution of this paper is to propose a general framework for building universally quantified abstract domains that leverage existing quantifier-free domains in induction-loop programs.
- We provide soundness proofs and convergence proofs for the method.
- We implement a tool based on clang [2] and the Z3 SMT solver [9]. The experiments demonstrate the feasibility of our method.

II. An Intuitive Example

We first give a very informal intuition of the method. Let us consider the program $init$ (Program 1).

First, a pre-analysis will identify loop control variables from program variables. A loop control variable is a variable that gets increased or decreased by a fixed amount every time. In program $init$, i is a loop control variable, the initial value is 0, the step is 1. We call the statement $i = 0$ as loop control variable initialization statement, call the statement $i = i + 1$ as loop control variable update statement.

Let us assume that there exists a basic analysis which takes into account simple inequalities and equalities of expressions (e.g., difference bound matrices [29] or octagons [25]). We call these properties generated by the basic analysis as basic properties. Our analysis will generate quantified properties by leveraging such analyses.

The analysis of program $init$ provides:

- At the first iteration:
  - After line 3:
    - basic properties: $(i = 0)$.
    - quantified properties: $(\forall x \in [0, 1, i), false)$.

    $(\forall x \in [0, 1, i), false)$ is a special quantified property. Since $[0, 1, i) = \emptyset$ after statement $i := 0$, thus, $(\forall x \in [0, 1, i), false)$ holds.
  - After line 4:
    - basic properties: $(i = 0), (i < size)$.
    - quantified properties: $(\forall x \in [0, 1, i), false)$.
  - After line 5:
    - basic properties: $(i = 0), (i < size), (A[i] = 0)$.
    - quantified properties: $(\forall x \in [0, 1, i), false)$,$(\forall x \in [0, 1, i+1), A[x] = 0)$.

    Since $[0, 1, i) = \emptyset$, $[0, 1, i+1)$ is equal to $\{i\}$. Thus, $(\forall x \in [0, 1, i+1), A[x] = 0)$ is equal to $A[i] = 0$. Thus, $(\forall x \in [0, 1, i+1), A[x] = 0)$ holds.
  - After line 6:
    - basic properties: nothing.
    - quantified properties: $(\forall x \in [0, 1, i-1), false)$, $(\forall x \in [0, 1, i), A[x] = 0)$.

    After $i := i + 1$, $[0, 1, i-1)$ and $[0, 1, i)$ are respectively equal to the value of $[0, 1, i)$ and $[0, 1, i+1)$ before $i := i + 1$. Thus, $(\forall x \in [0, 1, i-1), false)$ and $(\forall x \in [0, 1, i), A[x] = 0)$ hold after $i := i + 1$.
- At the second iteration:
  - At line 4:
    - basic properties: nothing.
    - quantified properties: $(\forall x \in [0, 1, i), A[x] = 0)$.

    $(\forall x \in [0, 1, i), A[x] = 0)$ holds since $(\forall x \in [0, 1, i), false) \sqcup (\forall x \in [0, 1, i), A[x] = 0) = (\forall x \in [0, 1, i), A[x] = 0)$
  - After line 5:
    - basic properties: $(i < size), (A[i] = 0)$.
    - quantified properties: $(\forall x \in [0, 1, i), A[x] = 0)$, $(\forall x \in [0, 1, i+1), A[x] = 0)$.

    Since $[0, 1, i+1)$ is equal to $[0, 1, i) \cup \{i\}$, thus, $(A[i] = 0)$ and $(\forall x \in [0, 1, i), A[x] = 0)$ imply $(\forall x \in [0, 1, i+1), A[x] = 0)$ holds.
  - After line 6:
    - basic properties: nothing.
    - quantified properties: $(\forall x \in [0, 1, i-1), A[x] = 0), (\forall x \in [0, 1, i), A[x] = 0)$.

    The iteration converges.
- So, the final result at the end of the program is
  - basic properties: $(size \leq i)$.
  - quantified properties: $(\forall x \in [0, 1, i), A[x] = 0)$, which implies $(\forall x \in [0, 1, size), A[x] = 0)$ holds.

III. Language and Preliminaries

We first define the induction-loop programs in which we formalize our technique.

A. Induction-loop Programs

Fig. 1 shows the syntax of statements. In this syntax, both lcv and $cvar$ are program variables. Here, lcv is a loop control variable, $cvar$ denotes a non-loop-control variable. In this paper, a pre-analysis will identify loop control variables from program variables. A statement st can be an assignment, a while-statement, if-statement or a sequential composition of statements. There are three kinds of assignments: (1) the initialization of a loop control variable of the form "lcv=$init$", where lcv does not occur in the expression $init$, (2) the update statement of a loop control variable of the form "$lcv = lcv + c$",

and (3) the assignment to a left-hand expression lh, where lh is a left-hand expression. A condition $cond$ is a conjunction and/or a disjunction of atomic conditions.

$$
\begin{aligned}
st &::= assign \\
&\mid \textbf{while}(cond)\ st \\
&\mid \textbf{if}(cond)\ st_1\ \textbf{else}\ st_2 \\
&\mid st_1; st_2 \\
assign &::= lh\ :=\ e \\
&\mid lcv\ :=\ init\ \ (lcv\ does\ not\ occur\ in\ init) \\
&\mid lcv\ :=\ lcv + c \\
lh &::= cvar \mid e_1[e_2] \\
init &::= e
\end{aligned}
$$

Figure 1: Syntax of statements

B. Iterative Data-Flow Analysis Framework

Iterative data-flow analysis framework [18] can be used to solve data-flow problems. An iterative data-flow analysis framework can be characterized as a four-tuple $(D, L_{\mathcal{G}}, \sqcup_{\mathcal{G}}, F_{\mathcal{G}})$, where $\mathcal{G}$ represents a control flow graph (CFG), and:

- D is the direction of analysis, which is forward or backward.
- $L_{\mathcal{G}}$ is a description of a join semi-lattice that represents the data flow values relevant to the problem. The height of $L_{\mathcal{G}}$ must be finite.
- $\sqcup_{\mathcal{G}}$ is a description of the join operator of the semi-lattice. $\sqcup_{\mathcal{G}}$ is derivable from $L_{\mathcal{G}}$.
- $F_{\mathcal{G}}$ is a description of the set of admissible flow functions from $L_{\mathcal{G}}$ to $L_{\mathcal{G}}$. $F_{\mathcal{G}}$ must be monotonic.

For a data-flow analysis, as long as the four elements are given, the analysis can be performed by an iterative algorithm. If the flow function of a data flow analysis is monotonic, and the height of the lattice is finite, then the analysis will terminate [18].

Our analysis is a *forward* data-flow analysis. The *forward* data flow equation is the following forms.

$$
In_n = \begin{cases} BI & n\ is\ Start\ statement \\ \bigsqcup\limits_{p \in pred(n)} F_p(In_p) & otherwise \end{cases}
$$

where *n* and *p* are statements, In_n is data flow value before the node *n*. For a statement n, let $F_n : L_{\mathcal{G}} \to L_{\mathcal{G}}$ denote the flow function that applies the statement. *BI* is the data flow values before the *Start* statement. $F_{\mathcal{G}}$ consists of flow functions F_p. $\sqcup$ is the join operator.

IV. Pre-analysis to Identify Loop Control Variables

The pre-analysis is used to identify loop control variables and obtain facts about these variables. We first give a definition of loop control variables.

Definition IV.1. *A variable* v *is a loop control variable at a program point* u *if for every execution path reaching* u*, every assignment statement for* v *is either* $v := init$ *or* $v := v + c$*, where* v *does not occur in the expression* init*,* c *is a fixed constant.*

The pre-analysis first identifies the set of loop control variables, and for each loop control variable, it identifies the step and the initial value of the loop control variable. Besides, it also finds the $<, \leq, =$ and $\neq$ relations between loop control variables and other program variables or expressions at every program point, e.g., $i < size$, $i = j$. These relations are used for two purposes.

1) For a quantified property p , an assignment $e_1[e_2] := e$, these relations are used to check whether p will be killed by $e_1[e_2]$.

   Example IV.1. *Consider the following property* $\forall x \in [0, 1, i), A[x] = 0$ *before an assignment* $A[j] := 1$*. If* $i < j$ *holds before this assignment, this property still holds after the assignment since* $i < j \Rightarrow j \notin [0, 1, i)$.
2) Deducing new quantified properties, e.g., $size \leq i \Rightarrow [0, 1, size) \subseteq [0, 1, i)$, so we can deduce $\forall x(x \in [0, 1, size) \Rightarrow p)$ from $\forall x(x \in [0, 1, i) \Rightarrow p)$.

The facts about loop control variables can be easily obtained by some existing analysis, such as induction variable analysis, difference-bound matrices (DBM) analysis [29].

The pre-analysis depicts and records simultaneously an expression set $\mathbb{IE}$xpr:

- $init$, lcv, $lcv - c$, and $lcv + c$ belong to $\mathbb{IE}$xpr, where lcv is a loop control variable, $init$ and c respectively are the initial value and the step of lcv.
- e_1 and e_2 belong to $\mathbb{IE}$xpr, where $(e_1\ op\ e_2)$ is a relation obtained by the pre-analysis, op is $<, \leq, =$ or $\neq$.

$\mathbb{IE}$xpr is used in the upper and lower bounds of intervals in our data flow analysis.

V. Quantified Abstract Domain

Throughout the paper, we assume the existence of one basic analysis, which our framework can leverage on. We use $(L_B, \sqsubseteq_B, \sqcup_B)$ to denote the lattice of basic analyses. Elements of L_B will be noted as ϕ.

The quantified abstract domain $L_\forall$ is parametrized by pre-analysis and basic analysis. An element of $L_\forall$ is a set of quantified properties in the following form:

$$\forall x_1 \in I_1, \cdots, \forall x_n \in I_n, \varphi(x_1, \cdots, x_n)$$

where

1) $I_1, \cdots, I_n$ are intervals. An interval can be (e_1, c, e_2), $[e_1, c, e_2)$, $(e_1, c, e_2]$ or $[e_1, c, e_2]$. The interval starts from e_1 (included or excluded) to the e_2 (included or excluded), and the difference between the consecutive terms is constant c. For any interval, the e_1 and e_2 are from $\mathbb{IE}$xpr.
2) φ is ϕ (i.e., an element of L_B) or $false$.

The data flow values before the *Start* statement (i.e., BI) is $\emptyset$.

A. Quantified Partial Order on $L_\forall$

The partial order $\sqsubseteq_\forall$ on $L_\forall$ is defined as follows. For any $L_1, L_2 \in L_\forall$, we say

$$L_1 \sqsubseteq_\forall L_2$$

iff for each quantified property $p_2 \in L_2$, there is a quantified property $p_1 \in L_1$ such that $p_1 \preceq p_2$.

The $\preceq$ can be seen as logical implication. $\preceq$ is defined as follows:

For any quantified properties $p_1 := (\forall x_1 \in I_{1_1}, \cdots, \forall x_n \in I_{1_n}, \varphi_1)$ and $p_2 := (\forall x_1 \in I_{2_1}, \cdots, \forall x_m \in I_{2_m}, \varphi_2)$, we say

$$p_1 \preceq p_2$$

when the following criteria are satisfied:

1) $n = m$ and $I_{1_1} = I_{2_1} \wedge \cdots \wedge I_{1_n} = I_{2_m}$, and
2) φ_1 and φ_2 meet the following conditions:
   - φ_1 is $false$, or
   - φ_1 and φ_2 both are elements of L_B, and $\varphi_1 \sqsubseteq_B \varphi_2$

Informally, $p_1 \preceq p_2$ means the intervals are same and either (1) φ_1 is $false$, or (2) $\varphi_1 \sqsubseteq_B \varphi_2$.

B. Join Operation $\sqcup_\forall$ on $L_\forall$

For any $L_1, L_2 \in L_\forall$,

$$L_1 \sqcup_\forall L_2 = \{p_1 \sqcup p_2 \mid p_1 \in L_1 \wedge p_2 \in L_2 \wedge p_1 \sqcup p_2 \neq \bot\}$$

The $\sqcup$ is defined as follows:

For any quantified properties

$$p_1 := \forall x_1 \in I_{1_1}, \cdots, \forall x_n \in I_{1_n}, \varphi_1$$
$$p_2 := \forall x_1 \in I_{2_1}, \cdots, \forall x_m \in I_{2_m}, \varphi_2$$

- if $n = m$ and $I_{1_1} = I_{2_1} \wedge \cdots \wedge I_{1_n} = I_{2_m}$,

$$p_1 \sqcup p_2 = \begin{cases} p_1 & if\ \varphi_2 = false \\ p_2 & if\ \varphi_1 = false \\ \forall x_1 \in I_{1_1}, \cdots, \forall x_n \in I_{1_n}, \varphi_1 \sqcup_B \varphi_2 & otherwise \end{cases}$$

  Note that, if $\varphi_1 \sqcup_B \varphi_2 = \bot$, $p_1 \sqcup p_2 = \bot$.
- otherwise, $p_1 \sqcup p_2 = \bot$.

VI. Abstract Interpreter for $L_\forall$

This section presents the principal flow functions for $L_\forall$. Let a in $L_\forall$ be the data flow value before a statement n. The flow function $F_n(a)$ generates a quantified property set belonging to $L_\forall$. All quantified properties in the set shall hold after statement n. The flow function $F_n(a)$ is defined as:

$$a' = \begin{cases} transfer(n,a) \cup genSpecial(n) & \\ \qquad\qquad if\ n\ is\ lcv\ initialization & \\ transfer(n,a) \cup handleInterval(n,a) & \\ \qquad\qquad if\ n\ is\ lcv\ update\ stmt & \\ transfer(n,a) \quad otherwise & \end{cases}$$
$$F_n(a) = genAQ(a')$$

For any statement n, $transfer$ function transfers all quantified properties not killed by n. If n is a loop control variable initialization statement, $genSpecial$ function generates a special quantified property. If n is a loop control variable update statement, $handleInterval$ function handles the intervals of quantified properties. After above operations, $genAQ$ is performed to generate more quantified properties.

We give the definitions of these operations in the following sections.

A. $transfer$ Function

To check whether a quantified property p not killed by assignment $t := e$, we need to define what it means for the term t to not occur in p. A term t can be lh or lcv. We first define a predicate $NotIn$.

Let E be the pre-analysis result at a program point under concern. For any quantified property $p := (\forall x_1 \in I_1, \cdots, \forall x_n \in I_n, \varphi)$ at the program point, a term t is provably not in p, denoted by $NotIn_E(t, p)$, if

1) t is a variable and t does not syntactically occur in p, or
2) t is of the form $e_1[e_2]$ and for every term $e'_1[e'_2]$ that (syntactically) occurs in p, it is the case that either $E \wedge (x_1 \in I_1 \wedge \cdots \wedge x_n \in I_n) \Rightarrow e_1 \neq e'_1$, or $E \wedge (x_1 \in I_1 \wedge \cdots \wedge x_n \in I_n) \Rightarrow e_2 \neq e'_2$.

Example VI.1. *This example is from program $find$ (Program 2). Let $E := \{j > i\}$ be the pre-analysis result before an assignment $n : A[i-1] := A[j]$. $p := (\forall k \in [size-1, -1, j-1), A[k] \geq x)$. $NotIn_E(A[i-1], p)$ is true since $j > i \wedge k \in [size-1, -1, j-1) \Rightarrow k \neq i-1$.*

Let E be the pre-analysis result before statement n. Function $transfer(n, a)$ is defined as:

$$transfer(n,a) = \begin{cases} \{p \mid p \in a \wedge NotIn_E(t,p)\} & if\ n\ is\ t := e, \\ a & if\ n\ is\ cond \end{cases}$$

where t can be lh or lcv.

Lemma VI.1. *Let E be pre-analysis result before statement n. For any $x, y \in L_\forall$, if $x \sqsubseteq_\forall y$, then $transfer(n, x) \sqsubseteq_\forall transfer(n, y)$.*

Proof. Because $x \sqsubseteq_\forall y$, thus, for each property $p_2 \in y$, there is a property $p_1 \in x$ such that $p_1 \preccurlyeq p_2$. According to the definition of $\preccurlyeq$, the intervals of p_1 are equal to the intervals of p_2. Thus, for a term t, $NotIn_E(t, p_2)$ implies $NotIn_E(t, p_1)$. Thus, $transfer(n, x) \sqsubseteq_\forall transfer(n, y)$. □

B. $genSpecial$ Function

For loop control variable initialization $lcv := init$, $genSpecial$ is defined as:

$$genSpecial(lcv := init) = \{\forall x \in [init, c, lcv), false\}$$

where c is step of lcv, c is obtained by pre-analysis. After $lcv := init$, $(\forall x \in [init, c, lcv), false)$ holds since $[init, c, lcv)$ is equal to $\emptyset$.

The special quantified property $\forall x \in [init, c, lcv), false$ is used to generate other normal quantified properties (the detailed description is given in $genAQ$ function).

Corollary VI.1. *If n is a loop control variable initialization, $genSpecial(n)$ is monotonic.*

C. $handleInterval$ Function

For a loop control variable update statement $lcv := lcv + c$, $handleInterval$ handles the intervals of quantified properties. After $lcv := lcv + c$, $[init, c, lcv)$ is equal to the value of $[init, c, lcv + c)$ before $lcv := lcv + c$. If $(\forall x \in [init, c, lcv + c), q)$ holds before $lcv := lcv + c$ and lcv does not occur in q, $(\forall x \in [init, c, lcv), q)$ holds after $lcv := lcv + c$. $(\forall x \in [init, c, lcv - c), q)$ can be treated similarly.

For loop control variable update statement $lcv := lcv + c$, $handleInterval$ is defined as:

$$handleInterval(n, a) = \{H(n, p) \mid p \in a \wedge H(n, p) \neq \bot\}$$

where function H is defined as follows:

$$H(lcv := lcv + c, p) = \begin{cases} \forall x \in [init, c, lcv - c), q \quad if\ lcv\ not\ occur\ in\ q, \\ \qquad and\ p\ is\ of\ form\ \forall x \in [init, c, lcv), q \\ \forall x \in [init, c, lcv), q \qquad if\ lcv\ not\ occur\ in\ q, \\ \qquad and\ p\ is\ of\ form\ \forall x \in [init, c, lcv + c), q \\ \bot \qquad otherwise \end{cases}$$

where $init$ and c respectively are the initial value and the step of lcv, q can be a quantified property or ϕ (i.e., an element of L_B) or $false$.

Example VI.2. *Let n be a lcv update statement $i := i + 1$, $a := \{(\forall x \in [0, 1, i), false), (\forall x \in [0, 1, i + 1), A[x] = 0)\}$ is the data flow value before $i := i + 1$. From the definition,*

$$H(i := i + 1, (\forall x \in [0, 1, i), false)) = (\forall x \in [0, 1, i - 1), false)$$
$$H(i := i + 1, (\forall x \in [0, 1, i + 1), A[x] = 0)) = (\forall x \in [0, 1, i), A[x] = 0)$$

Thus, $handleInterval(i := i + 1, a) = \{(\forall x \in [0, 1, i - 1), false), (\forall x \in [0, 1, i), A[x] = 0)\}$.

Lemma VI.2. *For any $x, y \in L_\forall$, if n is $lcv := lcv + c$ and $x \sqsubseteq_\forall y$, then $handleInterval(n, x) \sqsubseteq_\forall handleInterval(n, y)$.*

Proof. Because $x \sqsubseteq_\forall y$, thus, for each property $p_2 \in y$, there is a property $p_1 \in a$ such that $p_1 \preccurlyeq p_2$. According to the definition of $\preccurlyeq$, the interval of p_1 is equal to the interval of p_2. Thus, $H(n, p_1) \preccurlyeq H(n, p_2)$. Thus, $handleInterval(n, x) \sqsubseteq_\forall handleInterval(n, y)$. □

D. genAQ function

Function $genAQ$ generates quantified properties. For any $a \in L_\forall$, $genAQ$ is formally defined as:

$$genAQ(a) = intervalNarrow(a) \cup intervalIncrease(a)$$

where these two functions represent two ways to generate new quantified properties. The function $intervalNarrow$ generates a new quantified property by reducing the interval of a quantified property. The function $intervalIncrease$ generates a new quantified property by increasing the interval of a quantified property on some conditions.

Let E be pre-analysis result at a program point under concern, $intervalNarrow(a)$ is formally defined as:

For any p in a,

- if p is of form $(\forall x \in [e_1, c, e_2), q)$ and $c > 0$ and $(e_1' > e_1)$ is in E, then $(\forall x \in [e_1, c, e_2), q)$ is in $intervalNarrow(a)$;
- if p is of form $(\forall x \in [e_1, c, e_2), q)$ and $c > 0$ and $(e_2' \leq e_2)$ is in E, then $(\forall x \in [e_1, c, e_2), q)$ is in $intervalNarrow(a)$.

where q can be a quantified property or ϕ (i.e., an element of L_B) or $false$. Note that, there are other conditions to generate quantified properties (e.g., conditions for $c < 0$). Other conditions are similar to these conditions. We just list two conditions here.

Example VI.3. *Let $E := \{size \leq i\}$ be the pre-analysis result at a program point, $a := \{(\forall x \in [0, 1, i), A[x] = 0)\}$ holds at the program point. We have $intervalNarrow(a) = \{(\forall x \in [0, 1, size), A[x] = 0)\}$ since $E \Rightarrow [0, 1, size) \subseteq [0, 1, i)$.*

Let C be basic analysis result at a program point under concern. The function $intervalIncrease(a)$ is formally defined as:

For any p in a,

1) if the following conditions are met,
   - p is of form $(\forall x \in [init, c, lcv), false)$, and
   - $\psi(\cdots, e_1[f_1(lcv)], \cdots)$ is in $C \cup a$

   then $(\forall x \in [init, c, lcv + c), \psi(\cdots, e_1[f_1(x)], \cdots))$ is in $intervalIncrease(a)$.
2) if the following conditions are met,
   - p is of form $(\forall x \in [init, c, lcv - c), false)$, and
   - $\psi(\cdots, e_1[f_1(lcv - c)], \cdots)$ is in $C \cup a$

   then $(\forall x \in [init, c, lcv), \psi(\cdots, e_1[f_1(x)], \cdots))$ is in $intervalIncrease(a)$ (Note that this rule is similar to rule (1)).
3) if the following conditions are met,
   - p is of form $(\forall x \in [init, c, lcv), \psi_1(\cdots, e_1[f_1(x)], \cdots))$ and $\psi_2(\cdots, e_1[f_1(lcv)], \cdots)$ is in $C \cup a$, and
   - $\psi(\cdots, e_1[f_1(x)], \cdots) = \psi_1(\cdots, e_1[f_1(x)], \cdots) \sqcup \psi_2(\cdots, e_1[f_1(x)], \cdots)$

   then $(\forall x \in [init, c, lcv + c), \psi(\cdots, e_1[f_1(x)], \cdots))$ is in $intervalIncrease(a)$.
4) if the following conditions are met,

- p is of form $(\forall x \in [init, c, lcv - c), \psi_1(\cdots, e_1[f_1(x)], \cdots))$ and $\psi_2(\cdots, e_1[f_1(lcv - c)], \cdots)$ is in $C \cup a$, and
- $\psi(\cdots, e_1[f_1(x)], \cdots) = \psi_1(\cdots, e_1[f_1(x)], \cdots) \sqcup \psi_2(\cdots, e_1[f_1(x)], \cdots)$

then $(\forall x \in [init, c, lcv), \psi(\cdots, e_1[f_1(x)], \cdots))$ is in $intervalIncrease(a)$ (Note that this rule is similar to rule (3)).

where lcv is a loop control variable, $init$ and c are respectively the initial value and the step of lcv. ψ is a basic property in C or a quantified property in a, f_1 is a general function about lcv.

Let $\psi(k)$ be abbreviation of $\psi(\cdots, e_1[f_1(k)], \cdots)$. For rule (1), since $(\forall x \in [init, c, lcv), false)$ means $[init, c, lcv) = \emptyset$, $[init, c, lcv + c)$ is equal to $\{lcv\}$. Thus, $\psi(lcv)$ implies $(\forall x \in [init, c, lcv + c), \psi(x))$ holds. For rule (3), since $\psi(x) = \psi_1(x) \sqcup \psi_2(x)$, $(\forall x \in [init, c, lcv), \psi_1(x))$ and $\psi_2(lcv)$ respectively imply $(\forall x \in [init, c, lcv), \psi(x))$ and $\psi(lcv)$ hold. Since $[init, c, lcv + c)$ is equal to $[init, c, lcv) \cup \{lcv\}$, thus, $(\forall x \in [init, c, lcv), \psi(x))$ and $\psi(lcv)$ imply $(\forall x \in [init, c, lcv + c), \psi(x))$ holds.

Rules (2) and (4) are respectively similar to rules (1) and (3). The formers are to add $\psi(\cdots, e_1[f_1(lcv - c)], \cdots)$ to a quantified property whose interval is $[init, c, lcv - c)$, the latters are to add $\psi(\cdots, e_1[f_1(lcv)], \cdots)$ to a quantified property whose interval is $[init, c, lcv)$.

Example VI.4. *Consider program "init" (Program 1). Variable i is a loop control variable, the initial value is* 0*, the step is* 1*.*

1) $C := \{A[i] = 0\}$ *is the basic analysis result after statement* $A[i] := 0$*,* $a := \{(\forall x \in [0, 1, i), false)\}$ *is the data flow value after* $A[i] := 0$*, (at the first iteration).* $intervalIncrease(a) = \{(\forall x \in [0, 1, i + 1), A[x] = 0)\}$.
2) $C := \{A[i] = 0\}$ *is the basic analysis result after statement* $A[i] := 0$*,* $a := \{(\forall x \in [0, 1, i), A[x] = 0)\}$ *is the data flow value after* $A[i] := 0$*, (at the second iteration).* $intervalIncrease(a) = \{(\forall x \in [0, 1, i + 1), A[x] = 0)\}$.

Lemma VI.3. *Let E be pre-analysis result at a program point under concern. Let C be basic analysis result at the program point. For any $x, y \in L_\forall$, if $x \sqsubseteq_\forall y$, then $genAQ(x) \sqsubseteq_\forall genAQ(y)$.*

Proof. To prove $genAQ(x) \sqsubseteq_\forall genAQ(y)$, we just need to prove that: (1) $intervalNarrow(x) \sqsubseteq_\forall intervalNarrow(y)$, (2) $intervalIncrease(x) \sqsubseteq_\forall intervalIncrease(y)$.

Because $x \sqsubseteq_\forall y$, thus, for each property $p_2 \in y$, there is a property $p_1 \in x$ such that $p_1 \preccurlyeq p_2$.

1) Because $p_1 \preccurlyeq p_2$, the interval of p_1 is equal to the interval of p_2. Thus, $intervalNarrow(x) \sqsubseteq_\forall intervalNarrow(y)$.
2) Let $\psi(k)$ be abbreviation of $\psi(\cdots, e_1[f_1(k)], \cdots)$,
   - if rule (1) holds in x, i.e., $\psi(lcv)$ is in $C \cup x$ and $(\forall k \in [init, c, lcv), false)$ is in x, then $(\forall k \in [init, c, lcv + c), \psi(k))$ is in $intervalIncrease(x)$. Because $x \sqsubseteq_\forall y$, $(\forall k \in [init, c, lcv), false)$ is in y.
     - if $\psi(lcv)$ is in C, we have: $(\forall k \in [init, c, lcv + c), \psi(k))$ is in y.
     - if $\psi(lcv)$ is in x, because $x \sqsubseteq_\forall y$, there exists $\psi'(lcv)$ in y such that $\psi(k) \preccurlyeq \psi'(k)$. $(\forall k \in [init, c, lcv + c), \psi'(k))$ is in y. Thus, $(\forall k \in [init, c, lcv + c), \psi(k)) \preccurlyeq (\forall k \in [init, c, lcv + c), \psi'(k))$.

     Thus, $intervalIncrease(x) \sqsubseteq_\forall$ *intervalIncrease* (y).
   - if rule (3) holds in x, i.e., $(\forall k \in [init, c, lcv), \psi_1(k))$ is in x and $\psi_2(lcv)$ is in $C \cup x$, and $\psi(k) = \psi_1(k) \sqcup \psi_2(k)$, then $(\forall k \in [init, c, lcv + c), \psi(k))$ is in $intervalIncrease(x)$. Because $x \sqsubseteq_\forall y$, there exists $(\forall k \in [init, c, lcv), \psi_1'(k))$ in y such that $\psi_1 \preccurlyeq \psi_1'$ holds.
     - if $\psi_2(lcv)$ is in C, let $\psi'(k)$ be $\psi_1'(k) \sqcup \psi_2(k)$, we have: $\psi(k) \sqsubseteq_B \psi'(k)$ and $(\forall k \in [init, c, lcv + c), \psi'(k))$ is in y. Thus, $(\forall k \in [init, c, lcv + c), \psi(k)) \preccurlyeq (\forall k \in [init, c, lcv + c), \psi'(k))$.
     - if $\psi(lcv)$ is in x, because $x \sqsubseteq_\forall y$, there exists $\psi_2'(lcv)$ in y such that $\psi_2(k) \preccurlyeq \psi_2'(k)$. Let $\psi'(k)$ be $\psi_1'(k) \sqcup \psi_2'(k)$, we have: $\psi(k) \preccurlyeq \psi'(k)$ and $(\forall k \in [init, c, lcv + c), \psi'(k))$ is in y. Thus, $(\forall k \in [init, c, lcv + c), \psi(k)) \preccurlyeq (\forall k \in [init, c, lcv + c), \psi'(k))$

     Thus, $intervalIncrease(x) \sqsubseteq_\forall$ *intervalIncrease* (y).
   - The proofs of rules (2) and (4) are similar to the proofs of rules (1) and (3).

Thus, $genAQ(x) \sqsubseteq_\forall genAQ(y)$. □

E. Termination

Lemma VI.4. *Let* n *be a statement in a CFG $\mathcal{G}$. F_n is monotonic.*

Proof. F_n is monotonic iff $\forall x, y \in L_\forall : x \sqsubseteq_\forall y \Rightarrow F_n(x) \sqsubseteq_\forall F_n(y)$. According to the definition of F_n, we just need to prove that all functions occurring in F_n are monotonic. We have proved that all functions occurring in F_n are monotonic (Lemma VI.1, Lemma VI.2, Corollary VI.1, Lemma VI.3). □

Lemma VI.5. *The height of $(L_\forall, \sqsubseteq_\forall)$ is finite.*

Proof. An element of quantified abstract domain $L_\forall$ is the set of quantified properties. A quantified property is of the form:

$$\forall x_1 \in I_1, \cdots, \forall x_n \in I_n, \varphi$$

For any interval (e_1, c, e_2), $[e_1, c, e_2)$, $(e_1, c, e_2]$ or $[e_1, c, e_2]$, c is a constant occurring in program, the e_1 and e_2 are from $\mathbb{IE}xpr$. φ is an element of L_B or $false$. Because $\mathbb{IE}xpr$ is finite and the height of L_B is finite, thus, the height of $L_\forall$ is finite. □

Theorem VI.1. *Analysis will terminate.*

$$
\begin{aligned}
&[\![.]\!]: \ st \to States \to States\\
&\qquad e \to States \to L_c \cup \mathbb{Z}\\
&\qquad cond \to States \to \mathbb{B}\\
&\\
&[\![cvar := e]\!](\rho_v,\rho_a) = (\rho_v[[\![e]\!](\rho_v,\rho_a)/cvar],\rho_a)\\
&[\![lcv := init]\!](\rho_v,\rho_a) = (\rho_v[[\![init]\!](\rho_v,\rho_a)/lcv],\rho_a)\\
&[\![lcv := lcv + c]\!](\rho_v,\rho_a) = (\rho_v[[\![lcv+c]\!](\rho_v,\rho_a)/lcv],\rho_a)\\
&[\![A[e_1] := e_2]\!](\rho_v,\rho_a) = (\rho_v,\rho_a[F/A])\\
&\qquad where\ F = \lambda z.\begin{cases}\rho_a(A)(z) & if\ z \neq [\![e_1]\!](\rho_v,\rho_a)\\ [\![e_2]\!](\rho_v,\rho_a) & otherwise\end{cases}\\
&[\![\textbf{while}(cond)\ stmt]\!](\rho_v,\rho_a) =\\
&\qquad \begin{cases}(\rho_v,\rho_a) & if\ [\![cond]\!](\rho_v,\rho_a) = false\\ [\![stmt; while(cond)\ stmt]\!](\rho_v,\rho_a) & otherwise\end{cases}\\
&[\![\textbf{if}(cond)\ stmt_1\ \textbf{else}\ stmt_2]\!](\rho_v,\rho_a) =\\
&\qquad \begin{cases}[\![stmt_1]\!](\rho_v,\rho_a) & if\ [\![cond]\!](\rho_v,\rho_a) = false\\ [\![stmt_2]\!](\rho_v,\rho_a) & otherwise\end{cases}\\
&[\![st_1; st_2]\!](\rho_v,\rho_a) = [\![st_2]\!]([\![st_1]\!](\rho_v,\rho_a))
\end{aligned}
$$

Figure 2: Semantics of induction-loop Programs

Proof. The flow function F_n is monotonic (Lemma VI.4), and the height $(L_\forall, \sqsubseteq_\forall)$ is finite (Lemma VI.5). According to iterative data-flow analysis framework [18], analysis will terminate. □

F. Soundness of Operations

Let *States* denote the set of states of a program. A state is a tuple (ρ_v, ρ_a), where ρ_v maps scalar variables names to their values, ρ_a maps array names to their values. The semantics of statements of "induction-loop programs" are described in Fig. 2 as functions from *States* to *States*. We use "$A[e_1] := e_2$" to denote an assignment to an array element. Here, we simplify arrays for one dimension arrays.

For an abstract value a, $[\![a]\!](\rho_v,\rho_a)$ is defined as follows: $\forall p \in a$, $[\![p]\!](\rho_v,\rho_a)$.

We use c to denote a concrete program state (ρ_v,ρ_a) in subsequent paragraphs.

Theorem VI.2. *For all statement* n*, let c_i denote the concrete program states immediately before executing statement n, and let a_i denote the data flow value immediately before statement n. If the pre-analysis is sound, basic analysis is sound, $c_{i+1} = [\![n]\!](c_i)$ and $[\![a_i]\!](c_i)$ holds, the following conditions are satisfied:*

1) *If* n *is $lh := e$, $[\![transfer(n,a)]\!](c_{i+1})$ holds.*
2) *If* n *is cond, $[\![transfer(n,a)]\!](c_{i+1})$ holds.*
3) *If* n *is $lcv := init$, $[\![transfer(n,a) \cup genSpecial(n)]\!](c_{i+1})$ holds.*
4) *If* n *is $lcv := lcv + c$, $[\![transfer(n,a) \cup handleInterval(n,a)]\!](c_{i+1})$ holds.*
5) *If $[\![a']\!](c_{i+1})$ holds, $[\![genAQ(a')]\!](c_{i+1})$ holds.*

Proof. 1) To prove conclusion (1), we need to prove :

$$[\![\{p \mid p \in a \wedge NotIn_E(lh,p)\}]\!](c_{i+1})$$

where E is the pre-analysis result before n. $NotIn_E(lh,p)$ implies that all values of memory units in p remain unchanged, thus, $[\![p]\!](c_{i+1})$ holds. Thus, conclusion (1) holds.

2) The conclusion (2) follows directly from the definition of $transfer(cond,a)$.
3) To prove conclusion (3), we need to prove :

$$[\![transfer(n,a) \cup \{(\forall x \in [init,c,lcv), false)\}]\!](c_{i+1})$$

$[\![\forall x \in [init,c,lcv), false)]\!](c_{i+1})$ holds because n is $lcv = init$. $[\![transfer(n,a)]\!](c_{i+1})$ holds (the proofs are similar to conclusion (1)).
4) To prove conclusion (4), we need to prove :

$$[\![transfer(n,a) \cup \{H(p) \mid p \in a \wedge H(p) \neq \bot\}]\!](c_{i+1})$$

We need to prove:

a) if $[\![\forall x \in [init,c,lcv), q]\!](c_i)$ holds, $[\![\forall x \in [init,c,lcv-c), q]\!](c_{i+1})$ holds.
b) if $[\![\forall x \in [init,c,lcv+c), q]\!](c_i)$ holds, $[\![\forall x \in [init,c,lcv), q]\!](c_{i+1})$ holds.

Because n is $lcv := lcv + c$, lcv doesn't occur in q and $init$, the conclusions (a) and (b) hold.

5) Let $\psi(k)$ be abbreviation of $\psi(\cdots, e_1[f_1(k)], \cdots)$. To prove conclusion (5), we need to prove :

a) if $[\![\forall x \in [e_1,c,e_2), q]\!](c_{i+1})$ and $[\![c > 0 \wedge (e_1' > e_1)]\!](c_{i+1})$, then $[\![\forall x \in [e_1',c,e_2), q]\!](c_{i+1})$;
b) if $[\![\forall x \in [e_1,c,e_2), q]\!](c_{i+1})$ and $[\![c > 0 \wedge (e_2' \leq e_2)]\!](c_{i+1})$, then $[\![\forall x \in [e_1,c,e_2'), q]\!](c_{i+1})$;
c) if $[\![\forall x \in [init,c,lcv), false]\!](c_{i+1})$ and $[\![\psi(lcv)]\!](c_{i+1})$, then $[\![\forall x \in [init,c,lcv+c), \psi(x)]\!](c_{i+1})$.
d) if $[\![\forall x \in [init,c,lcv-c), false]\!](c_{i+1})$ and $[\![\psi(lcv-c)]\!](c_{i+1})$, then $[\![\forall x \in [init,c,lcv), \psi(x)]\!](c_{i+1})$.
e) if $[\![\forall x \in [init,c,lcv), \psi_1(x)]\!](c_{i+1})$ and $[\![\psi_2(lcv)]\!](c_{i+1})$ and $\psi(x) = \psi_1(x) \sqcup \psi_2(x)$ then $[\![\forall x \in [init,c,lcv+c), \psi(x)]\!](c_{i+1})$.
f) if $[\![\forall x \in [init,c,lcv-c), \psi_1(x)]\!](c_{i+1})$ and $[\![\psi_2(lcv-c)]\!](c_{i+1})$ and $\psi(x) = \psi_1(x) \sqcup \psi_2(x)$ then $[\![\forall x \in [init,c,lcv), \psi(x)]\!](c_{i+1})$.

- The conclusions (a) and (b) follow directly from the definition.
- For (c), $[\![\forall x \in [init,c,lcv), false]\!](c_{i+1})$ implies $[\![lcv = init]\!](c_{i+1})$, thus, $[\![\forall x \in [init,c,lcv+c), \psi(x)]\!](c_{i+1})$ is equal $[\![\psi(lcv)]\!](c_{i+1})$.
- For (d), the proofs are similar to the proofs of (c).
- For (e), because $[\![\forall x \in [init,c,lcv), \psi_1(x)]\!](c_{i+1})$ and $[\![\psi_2(lcv)]\!](c_{i+1})$ and $\psi(x) \preceq \psi_1(x)$, and because basic analysis is sound, thus, $[\![\forall x \in [init,c,lcv), \psi(x)]\!](c_{i+1})$ and $[\![\psi(lcv)]\!](c_{i+1})$ hold. Thus, $[\![\forall x \in [init,c,lcv+c), \psi(x)]\!](c_{i+1})$ holds.
- For (f), the proofs are similar to the proofs of (e).

□

VII. Implementation and Experiments

The method has been implemented using clang [2] and Z3 [9]. In the prototype tool, clang is used to analyze source code.

Z3 is used to check whether quantified properties are still valid after an assignment, which only uses linear constraint solver, which is well supported by Z3. We use the difference-bound matrices (DBM) analysis [29] to obtain the facts about loop control variables.

In our experiments, we instantiate our quantified analysis by two basic analysis. The first basic analysis is difference-bound matrices with disequality constraints ($dDBM$) analysis [27]. It is an extension of the DBM one. It can handle disequations between pairs of expressions. The second basic analysis is polynomial equations analysis [28]. It can generate polynomial equations invariants in programs.

A. The $dDBM$ Analysis Result

In this experiment, we apply our tool on 10 challenging array benchmarks shown in Figure I. The functions $init$ (Program 1) and $init\_noncost$ initialize all elements of an array to a constant and an iteration-dependent value respectively. $init\_partial$ initializes part of the array, and $init\_even$ initializes even positions. $2D\_array\_init$ initializes a 2-dimensional array using a nested loop. Various versions of $copy$ copy all or some elements of an array to another array. $find\_first\_nonzero$ looks for a non-zero element and returns its index (or -1 if the element is not found). $partition$ copies the zero and non-zero elements of a source array into two different arrays. $find$ (Program 2) is used for segmenting arrays in $QuickSort$. Our tool can automatically discover expected properties of these programs. Due to space limitations, this paper just gives the results of this analysis on two programs (shown in Program 3 and Program 4). All of the benchmarks and the results are available at https://github.com/libin049/QDInvSynthesis.

Results for *find(QuickSort)*: Program 2 shows the program. At the end of the program, we get the expected result (i.e., the array is segmented):

- $\forall k \in [1, 1, i), A[k-1] < x$
- $\forall k \in [size-1, -1, i], A[k] \geq x$

Results for *2D_array_init*: Program 3 shows the program. At the end of the program, we get the expected result (i.e., all elements of array A are equal to zero):

- $\forall x_1 \in [0, 1, row), \forall x_2 \in [0, 1, col), A[x_1][x_2] = 0)$

Results for *partition*: Program 4 shows the program. At the end of the program, we get the expected result (i.e., all elements of array B are equal to zero, while all elements of array C are equal to non-zero):

- $\forall x \in [0, 1, j), B[x] = 0)$
- $\forall x \in [0, 1, k), C[x] \neq 0)$

Program 3: 2D_array_init

```
i:=0;
while(i<row) {
    j:=0;
    while(j<col) {
        A[i][j] := 0;
        j:=j+1;
    }
```

Program	time (s)
init	0.61
init_nonconst	0.72
init_partial	0.60
init_even	0.77
copy	0.69
copy_partial	0.67
2D_array_init	3.71
find_first_nonzero	0.57
partition	2.63
find(QuickSort)	4.58

Table I: Case Study of $dDBM$ Basic Analysis

Program	time (s)
matrix_addition	4.04
matrix_dot_product	4.06
array_arithmetic1	0.83
array_arithmetic2	0.89
array_arithmetic3	2.55
array_arithmetic4	2.69

Table II: Case Study of $dDBM$ Basic Analysis

```
    i:=i+1;
}
```

Program 4: partition

```
i:=0; j:=0; k:=0;
while(i<size){
    if(A[i]==0){
        B[j]:=A[i];
        j:=j+1;
    }
    else{
        C[k]:=A[i];
        k:=k+1;
    }
    i:=i+1;
}
```

B. Polynomial Equations Analysis Result

In this experiment, we apply our tool on 6 challenging array benchmarks shown in Table II. The functions $matrix\_addition$ copies the results of adding two matrices to another matrix. $matrix\_dot\_product$ is the operation of multiplying two matrices by multiplying the corresponding elements. $array\_arithmetic1$, $array\_arithmetic2$, $array\_arithmetic3$ and $array\_arithmetic4$ do complex arithmetic operations for different arrays. Due to space limitations, this paper just gives the results of this analysis on one program (shown in Program 5).

Results for *array_arithmetic3*: Program 5 shows the program. At the end of the program, we get the expected result:

- $\forall x \in [1, 4, size), A[(x-1)/4] = 4 * B[x] + 1$

Program 5: array_arithmetic3

```
i := 1;
j := 0;
while( i < size ) {
    A[j] := i;
    B[i] := j;
    i := i+4;
    j := j+1;
}
```

tool	synthesis or verification	time(s)	sound or not sound
our tool	47	627	sound
SMACK+Corral	45	10310	not sound
ceagle	53	936	not sound
esbmc	39	10989	not sound
CppInv	27	15163	sound

Table III: Verification or synthesis results of SMACK+Corral, ceagle, esbmc, CppInv and our tool

Performances: All experiments were run on a 2.4 GHz Intel processor with 6 GB of RAM. Table I and II show the experimental results. The results show that it takes less than 1 seconds on many small programs, while multidimensional array programs and some complex programs require more time to be analyzed. The results show that it takes less than 5 seconds on all programs.

C. Analysis of array-examples benchmark of SV-COMP

Competition on Software Verification (SV-COMP)[4] provides an array program benchmark *array-examples* [3], which is designed to check (un)reachability. The benchmark is written in GNU C or ANSI C. It has 88 files, 2299 line codes. The total number of loops is 375. In this benchmark, there are 28 files with false specifications, which is designed to make the tools produce the violation witness. Other 60 files have correct specifications. All of these assertions are atomic assertions. However, an atomic assertion about array elements and loop-control-variable in the body of a loop statement is equivalent to a quantified property about the whole array. There are 56 such atomic assertions in the 60 files with correct specifications.

In *array-examples* of SV-COMP, all of the loops are induction-loops. Using $dDBM$ analysis and polynomial equations analysis, our tool finds quantified properties in 304 loops. The total execution time is 881 seconds. The average time for each of loops is 2.3 seconds. We also compare properties synthesized by out tool with the 56 assertions in the 60 files. Our tool finds 47 equivalent universally quantified properties.

We compared our synthesis results with several tools, which represent the state of the art. SMACK+Corral [16], ceagle [1] and esbmc [26] are program verification tools. They are the top tools w.r.t. correct rate in SV-COMP17 for *array-examples*. CppInv[23] can generate array universally quantified invariants in program. There are also other methods which aim to generate array universally quantified invariants, such as [8], [14], [15] and [20]. Unfortunately, we can not compare our approach with these tools, since they are not publicly available.

For all the tools, we compare the target properties number of verification or synthesis about the 56 assertions in the 60 files (the second column), the time of verification or synthesis (the third column), whether these tools are sound (the fourth column). Note that each file was configured with a timeout of 15 minutes.

Table III shows that though ceagle finds six more equivalent universally quantified properties than our tool, it is not sound. Except for ceagle, SMACK+Corral and esbmc are not sound. They are based on bounded model checking or testing techniques. Only our tool and CppInv are sound. Our tool is clearly superior to CppInv in terms of the number and time of synthesis.

VIII. Related Work

Several researchers have previously investigated the problem of generating array invariants.

Array expansion [6]. This method expands the cells of the array to local variables, and fully unrolls the loops. Array expansion is precise, but in practice can only be used for arrays of small size, and is not able to handle unbounded arrays.

Array smashing [7], [12]. All the cells of an array *A* are subsumed by one variable *a*. Initially, *a* is given the strongest known property satisfied by all the initial values of the cells of *A*. $A[i] := e$ is replaced by $a\sqcup := e$. However, weak assignment can only lose information tests on individual cells do not bring any information (do not process test condition). One needs to know an initial property satisfied by all the array cells. As a consequence, the results are generally unprecise.

Array partitioning [8], [13], [15]. Array partitioning method partitions the index domain (say, [1...n]) into several symbolic intervals (e.g., $I_1 = [1...i-1]$, $I_2 = [i, i]$, $I_3 = [i+1...n]$), and associates with each subarray $A[I_k]$ a summary auxiliary variable a_k. The partitioning is done either syntactically [13], [15] or by some pre-analysis [8]. Our approach does not associate with each subarray a summary auxiliary variable (or slice variables) to process. We directly process quantified properties, which are highly expressive. Thus, our method is very easy to deal with multidimensional arrays properties. Moreover, this method is difficult to deal with programs that the lcv update statement appears before the statement about array contexts in a loop. Such programs are very common (e.g., A[++i]). Our method can process them.

Predicate abstraction [10], [20], [22]. Predicate abstraction method uses some easy syntactic heuristics to derive the predicates used for the abstraction, or provided manually by the user. Moreover, counter-example guided refinement [5] and Craig interpolants [17] propose a significant improvement. However, some of these atomic predicates typically must be provided manually, as effective selection algorithms are lacking. Our approach is different in that we model the array properties directly. Unlike these predicate abstraction-based techniques, our approach does not require programs to be annotated with assertions.

Under-approximations and Templates [14], [29]. Under-approximations and templates method is extremely powerful yet expensive. The common idea behind these approaches is that the user provides templates that fix the structure of potential invariants. The analysis then searches for an invariant that instantiates the template parameters. Unlike our approach, the method may require the participation of users in the process.

Theorem prover-based [19], [24]. The method of [19] generates quantified invariants for loop programs without nesting. It uses a saturation theorem prover to generate loop invariants. The idea is to encode the changes to an array at the i-th iteration

as a quantified fact and then to systematically apply resolution to derive a closed form (one not mentioning the loop iteration i). Currently these approaches are still limited due to the missing inbuilt support for arithmetic theories in the underlying theorem provers(as opposed to SMT solvers, where arithmetic reasoning is hard-wired in the theory solvers). In [24], a related approach is presented, where invariants are generated by examining candidates supplied by an interpolating theorem prover. In addition to suffering from similar arithmetic reasoning problems as [19], the approach also requires program assertions. Unlike Theorem prover-based techniques, our approach only relies on the function of linear constraint solving, which is well supported by SMT solver (e.g., Z3).

IX. Conclusion

We propose a general framework for building universally quantified abstract domains that leverage existing quantifier-free domains in induction-loop programs. The method synthesizes properties by "iterating forward" analysis. It does not require the participation of users and automatically discovers quantified properties of programs by leverage existing quantifier-free analysis. Our method is sound and converges in finite time. The method has been implemented using clang and Z3. We also instantiate this framework using two basic analyses: difference-bound matrices with disequality constraints ($dDBM$) analysis and polynomial equations analysis. These analyses are used to successfully generate quantified invariants on challenging array benchmarks.

References

[1] http:// sts.thss.tsinghua.edu.cn/ceagle/.

[2] http://clang.llvm.org/.

[3] https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp17.

[4] http://sv-comp.sosy-lab.org/2017/index.php.

[5] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Acm Sigplan Notices*, volume 42, pages 300–309. ACM, 2007.

[6] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, pages 85–108. Springer, 2002.

[7] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN Notices*, volume 38, pages 196–207. ACM, 2003.

[8] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM SIGPLAN Notices*, volume 46, pages 105–118. ACM, 2011.

[9] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[10] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *ACM SIGPLAN Notices*, volume 37, pages 191–202. ACM, 2002.

[11] D. Gopan. *Numeric program analysis techniques with applications to array analysis and library summarization*. PhD thesis, Citeseer, 2007.

[12] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–529. Springer, 2004.

[13] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. *ACM SIGPLAN Notices*, 40(1):338–350, 2005.

[14] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *ACM SIGPLAN Notices*, volume 43, pages 235–246. ACM, 2008.

[15] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *ACM SIGPLAN Notices*, volume 43, pages 339–348. ACM, 2008.

[16] A. Haran, M. Carter, M. Emmi, A. Lal, S. Qadeer, and Z. Rakamarić. Smack+ corral: A modular verifier. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 451–454. Springer, 2015.

[17] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *Computer Aided Verification*, pages 193–206. Springer, 2007.

[18] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[19] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.

[20] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation*, pages 267–281. Springer, 2004.

[21] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *Computer Aided Verification*, pages 135–147. Springer, 2004.

[22] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *Computer Aided Verification*, pages 141–153. Springer, 2003.

[23] D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. Smt-based array invariant generation. In *VMCAI*, volume 7737, pages 169–188. Springer, 2013.

[24] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–427. Springer, 2008.

[25] A. Miné. The octagon abstract domain. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 310–319. IEEE, 2001.

[26] J. Morse, L. Cordeiro, D. Nicole, and B. Fischer. Handling unbounded loops with esbmc 1.20. In *Proc. of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795, pages 619–622. Springer, 2013.

[27] M. Péron and N. Halbwachs. An abstract domain extending difference-bound matrices with disequality constraints. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 268–282. Springer, 2007.

[28] E. Rodríguez-Carbonell and D. Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *International Static Analysis Symposium*, pages 280–295. Springer, 2004.

[29] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *ACM Sigplan Notices*, volume 44, pages 223–234. ACM, 2009.