Template-Based Verification of Array-Manipulating Programs*

Viktor Malík[1 [0000−0002−0608−0748]], Peter Schrammel[2,3 [0000−0002−5713−1381]], and Tomáš Vojnar[1 [0000−0002−2746−8792]]

[1] Brno University of Technology, Faculty of Information Technology, Brno, CZ
[2] Diffblue Ltd., Oxford, UK
[3] University of Sussex, Brighton, UK

Abstract. This work deals with the 2LS program verification framework that combines several verification techniques—namely, abstract domains, templated invariants, k-induction, bounded model checking, and SAT/SMT solving. A distinguishing feature of the approach used by 2LS is that it allows for seamless combinations of various program abstractions. In this work, we introduce a novel abstract template domain allowing 2LS to reason about arrays, using an arbitrary abstract domain to describe values that are stored inside the arrays (including nested arrays and dynamic linked data structures), and with the arrays possibly nested inside other structures. The approach uses array index expressions to split each array into multiple contiguous, non-overlapping segments and computes a different invariant for each of them. We illustrate the approach on a program dealing with a list of arrays and subsequently present how the new domain allowed 2LS to improve in the international software verification competition SV-COMP.

Keywords: Formal software verification · Arrays · Abstract domains · Templates of invariants · k-Induction · Bounded model checking · SAT/SMT solving.

1 Introduction

Arrays are arguably one of the most fundamental data structures in software engineering. A majority of programs use them in one way or the other. This means that verification of array-manipulating programs has many potential applications. On the other hand, it still faces a lot of challenges which often stem from the very essential features of arrays.

One of such problems is that arrays are by definition compound data structures which have an arbitrary underlying (element) data type, but at the same time their number of elements may be parametric and not bounded in advance. Moreover, the size of an array can also change at runtime using low-level operations such as `realloc`. Therefore, it is often not sufficient to directly use existing scalar value analyses, but it is necessary to combine these with specialised techniques reflecting the way how arrays are structured.

* The work was supported by the Czech Science Foundation project GA23-06506S, the FIT BUT internal project FIT-S-23-8151, and the Horizon Europe Chess project.

In this work, we address this problem in a way which allows us to reuse as many existing program analyses as possible. To this end, we develop our approach within the framework of so-called *template-based verification* implemented in the 2LS tool [31, 9, 24]. One of the main ideas of this framework is that it uses abstract interpretation with domains that are all required to have the same form of parametrized, fixed, quantifier-free, first-order logic formulae. This allows 2LS to use an SMT solver to reason about program properties and at the same time enables a straightforward combination of abstract domains with delegating a lot of the complexity to the solver itself.

Our main contribution is a proposal of an abstract domain for reasoning about the contents of arrays in 2LS. Thanks to the fact that 2LS already contains a number of existing abstract domains (such as the template polyhedra domain or the heap shape domain), our domain can be easily combined with them and can be used to reason about arrays containing various kinds of elements as well as about complex data structures containing arrays. One of such structures are *unrolled linked lists*, which we describe in the running example introduced at the end of this section.

Since 2LS uses an original approach to implement its abstract interpretation, we present its most important concepts in Section 2. After that, we propose our new array abstract domain in Section 3 and show how the results of 2LS in the International Competition on Software Verification (SV-COMP) improved once we implemented the proposed domain (Section 4). Finally, Section 5 gives an overview of other existing approaches to verification of array-manipulating programs, some of which were a great inspiration for our method.

Running Example In Figure 1, we introduce an example program, which we will use to illustrate the mechanisms proposed in this paper. The program features initialization of a data structure called an *unrolled linked list* [32], which is a linked list whose nodes contain arrays of values (in our case, up to 1000 integer values with the number of cells really in use given by the field `size`). To check correctness of operations over such a data structure, the verification tool needs to be able to reason over linked heap structures as well as the integer contents of arrays at the same time. 2LS already contains abstract domains for analyzing integer values as well as the shape of the heap, and, in this paper, we complement these with a new abstract domain for analyzing arrays.

2 Template-Based Verification of Programs

In this section, we present the most important concepts of 2LS that our work builds upon—for more details, see [31, 9, 24]. In its main verification loop, 2LS gradually unfolds program loops. Let k be the applied number of unfolding steps. Each version of the program obtained this way is translated into a *static single assignment (SSA)* form while approximating the (partially unrolled) program loops as described below. The SSA form is then translated into a first-order logical formula, and an attempt to find a k-inductive program invariant is done. If the discovered invariant is sufficient to show safety of the program, the verification succeeds. Otherwise, bounded model checking is used to see whether a real error can be reached in k steps. If so, the verification fails. Otherwise, k gets increased, and another iteration of the main verification loop follows.

```
1  struct node {
2    int data[1000];
3    int size;
4    struct node *next;
5  };
6
7  int main() {
8    struct node *list = NULL;
9    while (nondet()) {
10     struct node *n = malloc(sizeof(*n));
11     for (int i = 0; i < 1000; i++)
12       n->data[i] = 0;
13     n->size = 0;
14     n->next = list;
15     list = n;
16   }
17   int x = nondet(0,1000); // 0 <= x < 1000
18   assert(!list || list->data[x] == 0);
19 }
```

Fig. 1. Running example

Since the SSA form used in 2LS has several unusual features that cannot be found in other verification approaches and since their understanding is important for understanding the rest of this work, we present it in detail in Section 2.1. Likewise, in Sections 2.2 and 2.3, we present in more details the algorithm for inference of inductive invariants, which is at the heart of the 2LS' approach to abstract interpretation. The most important one is that all abstract domains are required to have the same form of so-called *templates* (hence, we denote the approach as *template-based verification*).

Templates are fixed, parametrized, first-order logic formulae, which allows 2LS to use an SMT solver to reason about them. Additionally, the unified form of the templates makes it easy to combine multiple abstract domains together (in the simplest form by just taking a conjunction of their templates) since the heavy-lifting of abstract operation combinators can be left to the underlying solver. Thanks to this feature, our newly proposed domain for reasoning about array contents can be easily combined with other domains already present in 2LS, which will, e.g., allow us to verify the running example program from Figure 1.

2.1 Internal Program Representation

The 2LS framework is built upon the CPROVER infrastructure [14] and therefore uses the same intermediate representation called *GOTO programs*. In this procedural lan-

guage, any non-linear control flow, such as `if` or `switch` statements, loops, or jumps, is translated to equivalent *guarded goto* statements. These statements are branch instructions that include (optional) conditions. CPROVER generates one GOTO program per C function found in the parse tree. Furthermore, it adds a new main function that first calls an initialization function for global variables and then calls the original program entry function.

After obtaining a GOTO representation of the analyzed program from CPROVER, 2LS performs a light-weight static analysis to resolve function pointers to a case split over all candidate functions, resulting in a static call graph. Furthermore, assertions guarding against invalid pointer operations or memory leaks are inserted. In addition, 2LS uses a local constant propagation and expression simplification to increase efficiency.

After running the mentioned transformations, 2LS performs a static analysis to derive data flow equations for each function of the GOTO program. Struct types are decomposed into their members. The result is a static single assignment (SSA) form which we describe in detail in the following section.

The Static Single Assignment Form Program verification in 2LS is based on generating program abstractions using a constraint solver. In order to simplify the generation of a formula representing the program semantics, 2LS uses the *static single assignment form* (SSA) to represent programs. SSA is a standard program representation used in many contexts including compilers as well as various program verifiers or analysers. We use common concepts of SSA—introducing a fresh copy (version) x_i of each variable x at program location i in case x is assigned to at i, using the last version of x whenever x is read, and introducing a *phi* variable x_i^{phi} at a program join point i in case different versions of x come from the joined program branches. For an acyclic program, SSA yields a formula that represents exactly the post condition of running the code.

In 2LS, the traditional SSA is extended by two new concepts: (1) an over-approximation of loops in order to make the SSA acyclic and (2) a special encoding of the control-flow [9]. These concepts allow a straightforward transformation of the SSA form into a formula that can be passed to an SMT solver. In addition, 2LS leverages *incremental SMT solving*, which means that it tries to reuse as large parts of the generated formulae as possible for successive solver invocations.

Over-Approximation of Loops In order to be able to use a solver for reasoning about program abstractions, 2LS extends the SSA by *over-approximating* the effect of loops. As was said above, the value of a variable x is represented at the loop head by a phi variable x_i^{phi} joining the value of x from before the loop and from the end of the loop body (here, we assume that all paths in the loop join before its end, and the same holds for the paths before the loop). However, instead of using the version of x from the loop end, it is replaced by a *free "loop-back" variable* x_i^{lb}. This way, the SSA remains acyclic, and, since the value of x_i^{lb} is initially unconstrained, the effect of the loop is over-approximated. To improve the precision, the value of x_i^{lb} can be later constrained using a *loop invariant* that will be inferred during the analysis. A loop invariant is a property that holds at the end of the loop body after any iteration and can therefore be assumed to hold on the loop-back variable.

To illustrate, let us take the variable `list` from the example in Figure 1. It is updated in the outer loop, and so the value at the loop head is a join of the corresponding SSA variables from before the loop (denoted $list_8$) and from the end of the loop (denoted $list_{15}$). In the loop head phi node $list_9^{phi}$, we, however, use an unconstrained loop-back variable $list_{16}^{lb}$ instead of $list_{15}$ and join it non-deterministically with the value from before the loop using a new free Boolean *loop-select* variable g_{16}^{ls}. Overall, the final phi node expression will have the form $list_9^{phi} = g_{16}^{ls} \ ? \ list_8 : list_{16}^{lb}$. 2LS will then constrain the value of $list_{16}^{lb}$ by inferring an invariant in an appropriate domain.

Encoding the Control-Flow In 2LS, the program is represented by a single monolithic formula. It is thus required that the formula encodes control-flow information. This is achieved using so-called *guard* variables that track the reachability information for each program location. In particular, for each program location i, we introduce a Boolean variable g_i whose value encodes whether i is reachable.

Memory Model As we have already indicated, an important property of the analysis in 2LS is that it leverages incremental solving. This is a great benefit to the verification performance, however, it comes with several drawbacks. The main one is related to the encoding of pointer dereferencing operations since on-demand concretization of heap objects is not possible (as the formula representing the program cannot change). To overcome this limitation, 2LS uses a special memory model and representation of memory-manipulating operations [29].

The model is object-based, and it distinguishes objects allocated *statically* (i.e., variables on the stack and global variables) and *dynamically* (i.e., on the heap). Whereas 2LS has a mode using summarization for handling procedures [2], we consider here non-recursive programs with all functions inlined and, therefore, it does not need to consider the stack. Hence, the set of static memory objects corresponds, in fact, to the set of all program variables.

To represent dynamic memory objects (i.e., those allocated using `malloc` or some of its variants), 2LS uses *abstract dynamic objects*. An abstract dynamic object represents a set of concrete dynamic objects allocated by the same `malloc` call. We refer to a `malloc` call at a program location i as to an *allocation site* i.

Generally, a single abstract dynamic object is not sufficient to reasonably precisely represent all concrete objects allocated by a single `malloc` call. This is due to the fact that the analyzed program may use several concrete objects allocated at the same allocation site at the same time. If such objects are, e.g., compared, the memory model must allow us to distinguish them. This can be done either by concretization on demand or by pre-materialization of a sufficient number of objects at the beginning of the analysis. We have opted for the latter possibility since it matches better with the use of incremental solving in 2LS[4].

[4] When the pre-materialization is used, all the variables needed to represent the semantics are known at the start of the analysis, and the representation of the semantics of the program stays constant. On the other hand, the approach used in 2LS would require new variables and a new representation of the program be generated every time an on-demand concretization was found necessary, making the solver to discard everything it discovered so far.

The number of abstract dynamic objects pre-materialized for each allocation size is determined using a simple *may-alias analysis* followed by choosing a sufficient number for the particular site such that the analysis is guaranteed to remain sound. For details on the exact algorithm, cf. [29].

Once the numbers of sufficient abstract memory objects are computed, each `malloc` call (or any of its variants) is replaced by a non-deterministic choice among one of these objects. Afterwards, 2LS performs a static *may-points-to analysis* which over-approximates—for each program location i and for each pointer p—the set of memory objects that p may point to at i. A dereference of p at i is then represented by a choice among the pointed objects. This way, the memory manipulating operations are pre-materialized and the formula representing the program may remain static for the rest of the analysis.

2.2 Template-Based Predicate Inference

A key phase of the program verification in 2LS is the generation of Inv, an inductive invariant. Instead of using algorithms for solving the semantical fixed point equations [16, 17], e.g. as implemented in off-the-shelf abstract interpreters, 2LS implements an algorithm for inferring such an invariant that exploits the power of incremental SMT solving.

When directly using a solver, 2LS would need to handle (the existential fragment of) second-order logic. As such solvers with reasonable efficiency are not currently available, the problem is reduced to a problem that can be solved by iteratively applying a first-order solver. We restrict ourselves to finding *inductive* invariants Inv of the form $\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})$ where $\mathcal{T}$ is a fixed expression, a so-called *template*, over program variables $\boldsymbol{x}$ and template parameters $\boldsymbol{\delta}$. Fixing a template reduces the second-order search for an invariant to a first-order search for template *parameters*:

$$\begin{aligned} \exists \boldsymbol{\delta}. \forall \boldsymbol{x}, \boldsymbol{x'}.\ & (Init(\boldsymbol{x}) \wedge \Rightarrow \mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})) \wedge \\ & (\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x'}) \Rightarrow \mathcal{T}(\boldsymbol{x'}, \boldsymbol{\delta})) \end{aligned} \tag{1}$$

where $Trans$ is the transition relation representing the semantics of the program, generated from the SSA form described in Section 2.1.

We resolve the $\exists\forall$ problem by an iterative solving of the negated formula, particularly of the second conjunct of (1), for different choices of constants $\boldsymbol{d}$ as the values of the parameter $\boldsymbol{\delta}$:

$$\exists \boldsymbol{x}, \boldsymbol{x'}. \neg\big(\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \wedge Trans \Rightarrow \mathcal{T}(\boldsymbol{x'}, \boldsymbol{d})\big). \tag{2}$$

The reason why we concentrate on iteratively solving (2) instead of the entire (1) is the usage of incremental solving combined with the internal program representation of 2LS. Since the SSA of the program is pre-computed, $Init$ and $Trans$ are "learned" by the solver and hence do not need to be re-solved for every iteration of the invariant inference algorithm.

The formula (2) can be expressed in quantifier-free logics and efficiently solved by SMT solvers. Using this as a building block, one can decide the mentioned $\exists\forall$ problem for finite types (e.g. fixed-size bitvectors).

From the abstract interpretation point of view, $\boldsymbol{d}$ is an abstract value, i.e., it represents (*concretizes to*) the set of all program states $\boldsymbol{s}$ that satisfy the formula $\mathcal{T}(\boldsymbol{s}, \boldsymbol{d})$ where a state is a vector of values of variables from $\boldsymbol{x}$. The choice of the template is hence analogous to choosing an abstract domain in abstract interpretation. The abstract values representing the infimum $\bot$ and supremum $\top$ of the abstract domain denote the empty set and the whole state space, respectively: $\mathcal{T}(\boldsymbol{s}, \bot) \equiv \mathit{false}$ and $\mathcal{T}(\boldsymbol{s}, \top) \equiv \mathit{true}$ [9].

Formally, the concretization function γ is:

$$\gamma(\boldsymbol{d}) = \{\boldsymbol{s} \mid \mathcal{T}(\boldsymbol{s}, \boldsymbol{d}) \equiv \mathit{true}\}. \tag{3}$$

In the abstraction function, to get the most precise abstract value representing the given concrete program state $\boldsymbol{s}$, we let

$$\alpha(\boldsymbol{s}) = \{\min(\boldsymbol{d}) \mid \mathcal{T}(\boldsymbol{s}, \boldsymbol{d}) \equiv \mathit{true}\}. \tag{4}$$

for templates that are monotonic in $\boldsymbol{d}$, for instance. If the abstract domain forms a complete lattice, existence of such a minimal value $\boldsymbol{d}$ is guaranteed.

The algorithm for the invariant inference takes an initial value of $\boldsymbol{d} = \bot$ and iteratively solves (2) using an SMT solver. If the formula is unsatisfiable, then an invariant has been found, otherwise a model of satisfiability $\boldsymbol{d}'$ is returned by the solver. The model represents a counterexample to the current instantiation of the template being an invariant. The value of the template parameter $\boldsymbol{d}$ is then updated by combining the current value with the obtained model of satisfiability using a domain-specific join operator [9].

For example, assume we have a program with a loop that counts from 0 to 10 in a variable x, and we have a template $x \leq d$. Let us assume that the current value of the parameter d is 3, and we get a new model $d' = 4$. Then we update the parameter to 4 by computing $d \sqcup d' = \max(d, d')$ because $\max$ is the join operator for the domain that tracks numerical upper bounds.

In 2LS, we use a single template to compute all invariants of the analyzed program. Therefore, typically, a template is composed of multiple parts, each part describing an invariant for a set of program variables. With respect to this, we expect a template $\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})$ to be composed of so-called *template rows* $\mathcal{T}_r(\boldsymbol{x}_r, \delta_r)$, each row r describing an invariant for a subset $\boldsymbol{x}_r$ of variables $\boldsymbol{x}$ and having its own row parameter δ_r. The overall invariant is then a composition of individual template rows with computed values of the corresponding row parameters. The kind of the composition (it can be, e.g., a simple conjunction) is defined by each domain.

Guarded Templates Since we use the SSA form rather than control flow graphs, we cannot use templates directly. Instead we use *guarded templates*. As described above, a template is composed of multiple template rows, each row describing an invariant for a subset of program variables. In a guarded template, each row r is of the form:

$$G_r(\boldsymbol{x}_r) \Rightarrow \widehat{\mathcal{T}}_r(\boldsymbol{x}_r, \delta_r) \tag{5}$$

for the r^{th} row $\widehat{\mathcal{T}}_r$ of the base template domain (e.g., template polyhedra). G_r is the conjunction of the SSA guards g_r associated with the definition of variables $\boldsymbol{x}_r$ occurring

in $\widehat{\mathcal{T}_r}$. Since we intend to infer loop invariants, $G_r(\boldsymbol{x}_r)$ denotes the guard associated to variables $\boldsymbol{x}_r$ appearing at the loop head. Hence, template rows for different loops have different guards.

We illustrate the above on the variable `i` from the inner loop of the example program from Figure 1. Let the phi node for `i` have the form $i_{11}^{phi} = g_{12}^{ls} \ ? \ i_{11} : i_{12}^{lb}$. We use a guarded interval template which has the form:

$$\mathcal{T}(i_{12}^{lb}, (\delta_1, \delta_2)) = \begin{array}{l} g_{11} \wedge g_{12}^{ls} \Rightarrow i_{12}^{lb} \leq \delta_1 \wedge \\ g_{11} \wedge g_{12}^{ls} \Rightarrow -i_{12}^{lb} \leq \delta_2. \end{array} \tag{6}$$

Here, g_{11} is a boolean guard expressing the reachability of line 11 (i.e., the loop head), and g_{12}^{ls} is a non-deterministic guard expressing that i_{12}^{lb} is chosen as the value of i_{11}^{phi}. In this example, the inferred values of the template parameters would be $\delta_1 = 1000$ and $\delta_2 = -1$.

2.3 Abstract Domains in 2LS

Reviewer 1: Give more complex concrete examples of domain

Over the past years, several abstract domains were introduced in 2LS. All of these have the same form of templates as described in Section 2.2 and hence can be arbitrarily combined. The list of the most important domains contains the *template polyhedra domain* [9], the *heap shape domain* [29], and the *ranking domains* [13].

In this work, we will combine our newly proposed array abstract domain with the *interval domain* (being the specialization of the template polyhedra domain) and with the *heap shape* domain. The former is used for invariants over numerical variables and the latter is used for pointer-typed variables (and numerical and pointer-typed fields of dynamic objects, respectively). As for the combination itself, it is done using a so-called *product domain* which allows to combine templates from multiple domains by taking a conjunction of the corresponding formulae.

3 Abstract Domain for Arrays

In this section, we introduce our abstract domain for analyzing the contents of arrays—we refer to it as *the array domain*.

Similarly to all other domains in 2LS, the array domain has the form of a template. An important property of arrays is that they may have an arbitrary element type. Therefore, using a simple domain is not sufficient and our array domain is a so-called *combination domain* where the domain itself describes only the form (the memory layout) of the array and it delegates reasoning about the actual array element values to another abstract domain, which we denote as the *inner domain*. Note that the inner domain may be any domain present in 2LS, including the array domain itself, which can be used to analyze arrays with multiple dimensions.

The domain is intended to be used to deal with variables from the set Arr of all array-typed variables of the given program. In particular, since we deal with loop invariants, we concentrate on arrays that are updated inside loops. In our SSA representation described in Section 2.1, such arrays are abstracted using so-called loop-back

array variables. We denote Arr^{lb} the set of such variables, and our array domain is then limited to this set.

The primary idea of our array domain, inspired by [15], is that each array $a \in Arr^{lb}$ is split into several *segments*, and an invariant is computed for each segment in the appropriate inner domain (based on the element type of a). For each a, the set of segments is determined using so-called *segment borders* that we infer at the beginning of the analysis (using the set of index expressions that the analyzed program uses to write into a—cf. Section 3.2 for details). A segment border can be any valid SSA expression.

In the rest of this section, we describe different aspects of the array abstract domain and invariant inference using it. First, we show in Section 3.1 how, given a set of borders, an array is split into segments and how the array domain template looks like. Next, we introduce the way we determine array segment borders in Section 3.2. Last, in Section 3.3, we present how invariants are computed in the array domain and how they can be used to verify program properties. To facilitate understanding of the presented concepts, we illustrate all of them on our running example in Section 3.4.

In the rest of this chapter, let us assume that we compute a loop invariant for an array $a \in Arr^{lb}$ updated in a loop l. We use N_a to denote the size (number of elements) of a.

3.1 Array Domain Template

We now describe the form of our array domain template. As outlined before, each array is split into multiple segments, and an invariant for each segment is computed in the array inner domain. Hence, the form of the array template is given by *array segmentation*, i.e., the way that each array is split into segments.

Let us denote B_a the set of segment borders for the array a. Prior to creating the segments, we perform two pre-processing steps: (1) making the borders unique and (2) ordering the borders.

Making the Borders Unique In order to decrease the number of segment borders and avoid empty segments, we first remove duplicate borders. This is done using an SMT solver—in particular, for each pair of segment borders $b_1, b_2 \in B_a$, we check whether the formula

$$b_1 \neq b_2 \wedge \mathit{Trans} \tag{7}$$

is satisfiable. In Eq. (7), Trans denotes the transition relation formula created from the SSA form of the analyzed program and representing the (over-approximated) program semantics. If Eq. (7) is unsatisfiable, then the values of the borders are always equal and hence one of them can be removed from B_a. By repeating this process for each pair of borders, we obtain the set of unique borders.

Ordering Borders After making B_a contain unique indices only, we try to order them. Again, we query the SMT solver, this time using two formulae:

$$\neg(b_1 \leq b_2) \wedge \mathit{Trans}, \tag{8}$$

$$\neg(b_2 \leq b_1) \wedge \mathit{Trans}. \tag{9}$$

If exactly one of Equations (8) and (9) is unsatisfiable for each pair of $b_1, b_2 \in B_a$, then a total ordering over B_a can be found. Otherwise, B_a is left unordered.

Array Segmentation Once the array segment borders are unique and possibly ordered, we create the array segmentation. We distinguish two situations:

1. B_a cannot be totally ordered. In such a case, we create multiple segmentations, one for each $b \in B_a$:
$$\{0\}\ S_1^b\ \{b\}\ S_2^b\ \{b+1\}\ S_3^b\ \{N_a\}. \tag{10}$$
The idea here is that if $a[b]$ is written to in a loop with gradually incrementing b, then, for any iteration b of the loop, S_1^b will abstract all array elements that have already been traversed, S_2^b will be the element accessed in the current iteration, and S_3^b will abstract elements to be traversed in the following iterations.
2. B_a can be totally ordered s.t. $b_1 \leq \ldots \leq b_n$ for $B_a = \{b_1, \ldots, b_n\}$. In such a case, we create a single segmentation for the entire array a:
$$\{0\}\ S_1\ \{b_1\}\ S_2\ \{b_1+1\}\ \cdots\ \{b_n\}\ S_{2n}\ \{b_n+1\}\ S_{2n+1}\ \{N_a\}. \tag{11}$$

Array Segments A single array segment S denoted
$$\{b_l\}\ S\ \{b_u\} \tag{12}$$
is an expression abstracting the elements of a between the indices b_l (inclusive) and b_u (exclusive). We refer to b_l and b_u as to the *lower* and *upper segment bounds*, respectively. In addition, for each S, we define two special variables: (1) the *segment element variable* $elem^S$ being an abstraction of the array elements contained in S and (2) the *segment index variable* idx^S being an abstraction of the indices of the array elements contained in S.

Template Form Having the set of loop-back arrays Arr^{lb} and a set of segments $\mathcal{S}^a$ for each $a \in Arr^{lb}$, we define the array domain template as:
$$\mathcal{T}^A \equiv \bigwedge_{a \in Arr^{lb}} \bigwedge_{S \in \mathcal{S}^a} \left(G^S \Rightarrow \mathcal{T}^{in}(elem^S) \right) \tag{13}$$
where $\mathcal{T}^{in}$ is the inner domain template and G^S is the conjunction of guards associated with the segment S.

The inner domain template abstracts the elements within a segment S. It is typically chosen based on the data type of $elem^S$ (e.g., we usually use the interval domain for numerical types and the shape domain for pointer types).

The purpose of G^S is to make sure that the inner invariant is limited to the elements of the given segment $\{b_l\}\ S\ \{b_u\}$. In particular, G^S is a conjunction of several guards:
$$b_l \leq idx^S < b_u \wedge \tag{14}$$
$$0 \leq idx^S < N_a \wedge \tag{15}$$
$$elem^S = a[idx^S] \tag{16}$$

where Eq. (14) makes sure that the segment index variable stays between the segment borders, Eq. (15) makes sure that the segment index variable stays between the array borders (since segment borders are generic expressions, they may lie outside of the array, hence Eq. (14) is not sufficient), and Eq. (16) binds the segment element variable with the segment index variable.

Using the above template, 2LS is able to compute a different invariant for each segment. For example, for a typical array iteration loop, this would allow to infer a different invariant for the part of the array that has already been traversed than for the part of the array that is still to be visited.

3.2 Computing Array Segment Borders

In the previous section, we assumed that we already have the set of segment borders for each array. In this section, we describe how this set is obtained. As we outlined earlier, the verification approach of 2LS requires the domain template to be a fixed, parametrized, first-order formula. To be able to fulfil the "fixed" property, we need to determine the set of segments at the beginning of the analysis so that we are able to create a finite set of array segments which will form the array domain template.

The main idea of our approach is that the segment borders should be closely related to the expressions that are used to access array elements in the analyzed program (we denote these as *array index expressions*). Therefore, we perform a static *array index analysis* which collects the set of all expressions occurring as array access indices (i.e., expressions that appear inside the square bracket operators). In addition, we distinguish between *read* accesses (occurring on the right-hand side of assignments and in conditions) and *write* accesses (occurring on the left-hand side of assignments).

Once the array index analysis is complete, for each loop-back array a, we determine the set of its segment borders by taking the set of all index expressions used to write into a in the corresponding loop. In addition, if some of those expressions contain a variable whose value is updated inside the same loop, we also take the pre-loop value of the expression as a segment border.

To illustrate the above, let us have a simple loop initializing the second half of an array:

```
1 for (int i = N / 2; i < N; i++)
2     a[i] = 0;
```

The set of index expressions used to write into the array is $\{\mathtt{i}\}$, but we would also use `N/2` (the initial value of `i`) as a segment border. Hence, the segmentation of `a` would be:

$$\{0\} \cdots \{\mathtt{N}/2\} \cdots \{\mathtt{N}/2+1\} \cdots \{\mathtt{i}\} \cdots \{\mathtt{i}+1\} \cdots \{\mathtt{N}\}. \tag{17}$$

Thanks to this segmentation, 2LS is able to differentiate three important parts of the array: (1) the first half of the array (which is untouched in the loop), (2) the part between `N/2` and `i` which in any iteration represents the already initialized part, and (3) the part from `i` to the array end which represents the part to be initialized in future

iterations. In particular, 2LS would be able to infer an invariant stating that all elements in part (2) are equal to 0, which would mean that the entire second half of the array is set to 0 once the loop ends.

3.3 Array Domain Invariant Inference

Once the array domain template is created, the invariant inference algorithm of 2LS (see Section 2.2) is used to compute loop invariants for individual segments. As we already described, most of the work is delegated to the inner domain, and the array domain is mainly responsible for making sure that the segment element variables, for which the inner invariants are computed, are properly constrained.

Additionally, there is one more necessary step after the array invariants are computed. The problem is that the invariants describe properties of the segment element and index variables, however, these variables are not actually used inside the analyzed program. Therefore, in order for the invariant to properly constrain the program semantics, we *bind* the computed invariants with all index expressions used to read from the arrays. We do not need to constrain the array elements that are written by the program since their value gets overridden, hence binding with read elements is sufficient. The set of expressions to bind the invariant with is obtained using the array index analysis introduced in Section 3.2.

In particular, for each segment S of each loop-back array a, we create a binding between the segment element and index variables $elem^S$ and idx^S and each index expression i_r used to read from a as follows. We take the computed invariant for S and replace all occurrences of idx^S by i_r and all occurrences of $elem^S$ by $a[i_r]$. Then, the obtained formula is passed to the solver which constrains the values of a for the given access through i_r. This process is done for the final invariant as well as for each candidate invariant found during the analysis to allow the invariant inference algorithm to account for the already computed constraints.

3.4 Running Example

Reviewer 1: Demonstrate impact of incrementality

We now illustrate usage of the array domain on the running example from Figure 1. The arrays in the program are contained within dynamically allocated objects. For simplicity, let us assume that all objects allocated by the `malloc` on line 10 are represented by a single abstract dynamic object ao_{10}[5]. We demonstrate inference of a loop invariant for the inner loop of the program, hence the SSA object that we work with is $ao_{10}.data^{lb}_{12}$.

First, 2LS runs the array index analysis to determine the set of indices used to access the array. In this case, there is a single index used for writing (`i` on line 12) and one index used for reading (`x` on line 18).

After the array index analysis is run, the analyzed array must be segmented. There is a single written index, hence there will be three segments in total. In addition, the index

[5] In practice, we would need at least 2 abstract dynamic objects to distinguish between the current node pointed by `n` and the next node pointed by `n->next`.

is updated inside the same loop, hence we will use its loop-back variant (i_{12}^{lb}) inside the segmentation

$$\{0\}\ S_1\ \{i_{12}^{lb}\}\ S_2\ \{i_{12}^{lb}+1\}\ S_3\ \{1000\}. \tag{18}$$

For each segment S_j, $j \in \{1,2,3\}$, we introduce a segment element variable $elem^j$ and a segment index variable idx^j.

Since the array is of integer type, we will use the interval abstract domain as the inner domain. The interval domain has two template rows for each variable, hence our template will contain 6 rows in total (two for each segment element variable). For the sake of legibility, we only give the two rows for $elem^1$:

$$\begin{aligned} &g_{11} \wedge g_{12}^{ls} \ \wedge\ 0 \le idx^1 < i_{12}^{lb} \ \wedge\ 0 \le idx^1 < 1000 \ \wedge\ elem^1 = ao_{10}.data_{12}^{lb}[idx^1] \\ &\qquad \Rightarrow elem^1 \le d_1 \ \wedge \\ &g_{11} \wedge g_{12}^{ls} \ \wedge\ 0 \le idx^1 < i_{12}^{lb} \ \wedge\ 0 \le idx^1 < 1000 \ \wedge\ elem^1 = ao_{10}.data_{12}^{lb}[idx^1] \\ &\qquad \Rightarrow -elem^1 \le d_2. \end{aligned} \tag{19}$$

Both rows have the same guard (the implication antecedent) consisting of multiple parts:

- The first two conjuncts ($g_{11} \wedge g_{12}^{ls}$) are standard row guards used in other domains that guard the reachability of the loop and the definition of the loop-back variable.
- The second part ($0 \le idx^1 < i_{12}^{lb}$) guards that the segment index variable stays within the segment bounds.
- The third part ($0 \le idx^1 < 1000$) guards that the segment index variable stays within the array bounds.
- The last part ($elem^1 = ao_{10}.data_{12}^{lb}[idx^1]$) binds the segment element variable with the analyzed array object and the segment index variable.

The properties of interest to be computed (the implication consequences) are determined from the inner domain, in this case the interval abstract domain.

Using the above template in the invariant inference algorithm of 2LS, we will obtain values of $d_1 = d_2 = 0$ (i.e., values of all array elements in the segment S_1 are equal to 0). After the loop ends, the value of i_{12}^{lb} will be 1000 (thanks to the loop condition and the invariant computed for i_{12}^{lb} from the template given in (6)), which will effectively prove that all elements of the given array are equal to 0 at that moment.

The remaining part of the array domain usage is binding of the invariant onto indices used to read from it. Our example program features one array read at line 18 (`list->data[x]`). At this point of the program, `list` may point to the dynamic object ao_{10}, hence an access to $ao_{10}.data$ is possible in this expression. Therefore, we bind the invariant computed from the template from Eq. (19) with the read index x_{17} as follows:

$$\begin{aligned} &g_{11} \wedge g_{12}^{ls} \ \wedge\ 0 \le x_{17} < i_{12}^{lb} \ \wedge\ 0 \le x_{17} < 1000 \Rightarrow ao_{10}.data_{12}^{lb}[x_{17}] \le 0 \ \wedge \\ &g_{11} \wedge g_{12}^{ls} \ \wedge\ 0 \le x_{17} < i_{12}^{lb} \ \wedge\ 0 \le x_{17} < 1000 \Rightarrow -ao_{10}.data_{12}^{lb}[x_{17}] \le 0. \end{aligned} \tag{20}$$

The equation has been obtained from Eq. (19) by supplying the actual computed values of d_1 and d_2 and by replacing occurrences of idx^1 by x_{17} and the occurrences of $elem^1$

by $ao_{10}.data^{lb}_{12}[x_{17}]$. We removed the last part of each row guard as $elem^1$ is no longer used. Also, note that we bind the invariant to $ao_{10}.data^{lb}_{12}$ rather than to the SSA version of $ao_{10}.data$ valid on line 18 (which would be $ao_{10}.data^{phi}_{9}$) because we only want to constrain the value of the array coming from the loop that the invariant is computed for. In other words, Eq. (20) allows to leverage the computed array invariant during further program analysis.

As we already mentioned, the last step would be done after each round of the invariant inference algorithm, however, we omit that here for the sake of simplicity and give the binding for the final invariant only.

When combined with a heap invariant computed for the `next` field of ao_{10}, our array invariant will allow us to prove that all array elements of all the unrolled list nodes are equal to zero.

4 Experimental Evaluation

We have implemented our proposed array domain in the 2LS framework. In this chapter, we evaluate the impact of this implementation. Since 2LS regularly competes in the *International Competition on Software Verification (SV-COMP)*, we present results of 2LS from this competition. In particular, we check results in the *ReachSafety-Arrays* category which features verification tasks requiring reasoning about (mainly numerical) contents of arrays.

2LS has traditionally received negative score in this category (-28 in 2020–2022) which has only changed in 2023 with the introduction of our proposed domain into 2LS [30]. Using the new array domain, 2LS was able to successfully verify 17[6] and 16 error-free tasks from this category in 2023 and 2024, respectively (as opposed to 2 from the previous years).

While these are not large numbers, they show that our new domain is a good first step towards better analysis of array contents in 2LS. In addition, due to the nature of program analysis in 2LS, the domain may leverage from combination with other abstract domains (e.g., the shape domain), however, SV-COMP benchmarks do not yet feature tasks requiring such a combination.

Reviewer 2: provide experimental results with comparison to other tools of the SV-COMP

5 Related Work

There exists a vast body of works aimed at analysis of array contents and verification of programs manipulating arrays. We describe the most important works in this section.

[6] The given number for 2023 is different from the official results of 2LS in SV-COMP 2023. The reason is that a number of tasks was last-minute disqualified due to past-deadline changes which were often related to the tasks being added to new categories (e.g., *NoOverflows*) rather than real modifications of the tasks or their verdicts. Hence, we present results from the entire benchmark instead of the competition benchmark set as those results are more representative and can be better compared to the previous year results.

Many of the works are related and use a similar principle, hence we divide the overview into three categories.

Reviewer 2: distribute some references from section 5 to section 3

5.1 Methods Based on Array Segmentation

One of the first works in the area [8] introduced two basic techniques for reasoning about the contents of arrays: (1) *array expansion* where each array element is represented using a single abstract value and (2) *array smashing* (also presented in [19]) where all elements of the array are abstracted using a single value. These approaches represent two extremes in approaching the arrays—while the first one often does not scale due to unbounded nature of the arrays, the second one abstracts away too much information that is often crucial for proving the required array properties.

An approach to overcome these problems, which we also take in our work, is to split arrays into multiple parts, usually called segments. This technique was first introduced in [20] where it was combined with simple numerical domains and was mainly able to reason about array initialization loops. This method was improved in [22] by extending it to handle relational abstract properties and consequently in [15] which proposed to use an arbitrary abstract domain for reasoning about array elements. Our approach is heavily based on the latter work, mainly due to the fact that it is compatible with the verification approach in 2LS. The proposed method uses automatic inference of segment bounds based on semantic pre-analysis of the array usage in the program.

Compared to all of these works, which are mainly aimed at analysis of numerical contents of arrays, we leverage other domains present in 2LS. In particular, the combination with our newly introduced shape domain allows us to expand program verification to analysis of structures combining arrays and linked structures on the heap. In addition, it was necessary to formulate our approach in the very specific 2LS framework, hence the introduced domain has several unique features that cannot be found in other approaches.

The segmentation-based approaches were further extended to non-contiguous and overlapping segments [28, 10] but these are much more difficult to be described using first-order logic formulae, and therefore we did not consider them for our approach.

5.2 Methods Based on Analysis of Array-Manipulating Loops

A completely different approach to the typical array verification problems is taken by the VERIABS verification tool [1]. Instead of trying to describe the contents of (potentially huge) arrays in an abstract way, the tool focuses on analysis of loops manipulating the arrays. In particular, VERIABS features two important techniques related to verification of array contents: (1) *loop shrinking* [25] and (2) *full-program induction* [11].

The first technique automatically analyzes loops that manipulate program arrays and for each loop it determines the so-called *shrink factor* k—the sufficient number of iterations that are necessary to prove the property being checked. After k is obtained, the processed array is reduced to the size k and filled with k non-deterministically chosen elements of the original array. The reduced program is then verified using state-of-the-art BMC tools.

In some cases, the shrink factor is not sufficiently low for BMC to scale and prove or refute program correctness. In such a case, VERIABS transforms the arrays to be of symbolic size N and performs so-called full-program induction. This technique, given a program P_N parametrized by the array size N and pre- and post-conditions denoted $\varphi(N)$ and $\psi(M)$, respectively, is able to very efficiently check validity of the Hoare triple $\{\varphi(N)\ P_N\ \psi(N)\}$ for all values of $N > 0$.

The above methods have proven very effective since VERIABS has been the most successful tool in the *ReachSafety-Arrays* sub-category of SV-COMP in the recent years [4–6]. On the other hand, VERIABS does not compete in memory safety, so the effectiveness of these methods on programs combining arrays and linked structures remains questionable. Still, we may consider implementing modified versions of the proposed techniques in future to improve efficiency of verification of array-manipulating programs in 2LS.

5.3 Predicate Abstraction and Non-Automatic Methods

In the last group of works, we present those that use completely different verification approaches than 2LS does, and hence were not considered for our case.

First, there is a large group of works [27, 26] based on predicate abstraction [18], possibly improved by counterexample guided refinement [7] and Craig interpolants [23]. It is, however, not clear how to combine predicate abstraction efficiently with the computation loop of 2LS. Moreover, we note that methods based on predicate abstraction make use of the property to be proved while our approach allows one to also discover (previously unknown) existing properties.

Second, besides fully automated works, there exist approaches which require some user intervention. For instance, [21] also specifies abstract domains using templates, but their domains are universally quantified (as opposed to our quantifier-free templates). This naturally makes the domains much stronger, however, the verification approach requires all the abstract domains to be specified manually. In contrary, the verification approach of 2LS is fully automatic. Other techniques based on deductive methods [3, 12] suffer from a similar issue when they require users to provide loop invariants (which our method is able to infer automatically).

6 Conclusions and Future Work

We have presented a new abstract template domain for the 2LS verification framework. In particular, the domain allows 2LS to compute invariants of programs dealing with arrays that can be storing various types of data (including nested arrays and dynamic linked data structures) and that can be themselves nested in other structures. The domain is based on segmenting arrays according to array index expressions (computed statically) and on computing an invariant for each array segment independently. We have illustrated the approach on a program with linked lists whose cells contain arrays, and we have shown that the implementation of the approach within 2LS significantly improved its score on benchmark programs dealing with arrays.

Reviewer 3: give some example of array-manipulating program that cannot be dealt with easily in this framework

Nevertheless, the array abstract template domain and the techniques for dealing with it that we have introduced are so far still rather simple. Further improvements are needed to handle an even wider spectrum of array-manipulating programs. Also, in the future, it might be interesting to attempt to combine some of the techniques introduced in VERIABS [1] with those we proposed for 2LS in this work.

References

1. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: Veriabs: Verification by abstraction and test generation. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1138–1141 (2019). https://doi.org/10.1109/ASE.2019.00121
2. Alur, R., Bouajjani, A., Esparza, J.: Model checking procedural programs. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 541–572. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_17, https://doi.org/10.1007/978-3-319-10575-8_17
3. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. p. 49–69. CASSIS'04, Springer-Verlag, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30569-9_3
4. Beyer, D.: Advances in automatic software verification: Sv-comp 2020. In: Biere, A., Parker, D. (eds.) Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 347–367. Springer (2020)
5. Beyer, D.: Software verification: 10th comparative evaluation (sv-comp 2021). In: Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 401–422. Springer (2021)
6. Beyer, D.: Progress on software verification: Sv-comp 2022. In: Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 375–402. Springer (2022)
7. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 300–309. PLDI '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1250734.1250769
8. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, pp. 85–108. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-36377-7_5
9. Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety Verification and Refutation by k-Invariants and k-Induction. In: Proceedings of the 22nd Static Analysis Symposium. Lecture Notes in Computer Science, vol. 9291, pp. 145–161. Springer (2015)
10. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs by tiling. In: Proceedings of the 24th Static Analysis Symposium. pp. 428–449 (08 2017). https://doi.org/10.1007/978-3-319-66706-5_21
11. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs with full-program induction. In: Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 22–39. Springer (2020)

12. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with jml and esc/java2. In: Proceedings of the 4th International Conference on Formal Methods for Components and Objects. p. 342–363. FMCO'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11804192_16
13. Chen, H.Y., David, C., Kroening, D., Schrammel, P., Wachter, B.: Bit-Precise Procedure-Modular Termination Proofs. TOPLAS **40**, 1–38 (2017)
14. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)
15. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 105–118. POPL '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926399
16. Esparza, J., Kiefer, S., Luttenberger, M.: Newtonian program analysis. J. ACM **57**(6), 33:1–33:47 (2010). https://doi.org/10.1145/1857914.1857917, https://doi.org/10.1145/1857914.1857917
17. Esparza, J., Luttenberger, M., Schlund, M.: FPSOLVE: A generic solver for fix-point equations over semirings. Int. J. Found. Comput. Sci. **26**(7), 805–826 (2015). https://doi.org/10.1142/S0129054115400018, https://doi.org/10.1142/S0129054115400018
18. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 191–202. POPL '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/503272.503291
19. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Jensen, K., Podelski, A. (eds.) Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 512–529. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
20. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 338–350. Association for Computing Machinery, New York, NY, USA (2005)
21. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 235–246. POPL '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1328438.1328468
22. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 339–348. PLDI '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1375581.1375623
23. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Proceedings of the 19th International Conference on Computer-Aided Verification. p. 193–206. CAV'07, Springer-Verlag, Berlin, Heidelberg (2007)
24. Kroening, D., Malík, V., Schrammel, P., Vojnar, T.: 2LS for Program Analysis. Tech. rep. (2023)
25. Kumar, S., Sanyal, A., Venkatesh, R., Shah, P.: Property checking array programs using loop shrinking. In: Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 213–231. Springer (2018)
26. Lahiri, S.K., Bryant, R.E.: Indexed predicate discovery for unbounded system verification. In: Alur, R., Peled, D.A. (eds.) Proceedings of the 16th International Conference on

Computer-Aided Verification. pp. 135–147. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
27. Lahiri, S.K., Bryant, R.E., Cook, B.: A symbolic approach to predicate abstraction. In: Hunt, W.A., Somenzi, F. (eds.) Proceedings of the 15th International Conference on Computer-Aided Verification. pp. 141–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
28. Liu, J., Rival, X.: Abstraction of arrays based on non contiguous partitions. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Proceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 282–299. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
29. Malík, V., Hruška, M., Schrammel, P., Vojnar, T.: Template-based verification of heap-manipulating programs. In: Proceedings of the 2018 Formal Methods in Computer-Aided Design. pp. 103–111 (2018)
30. Malík, V., Nečas, F., Schrammel, P., Vojnar, T.: 2ls: Arrays and loop unwinding (competition contribution). In: Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 529–534. Springer (2023)
31. Schrammel, P., Kroening, D.: 2LS for Program Analysis - (Competition Contribution). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 9636, pp. 905–907. Springer (2016)
32. Shao, Z., Reppy, J.H., Appel, A.W.: Unrolling lists. In: Proceedings of the 1994 ACM Conference on LISP and Functional Programming. p. 185–195. Association for Computing Machinery, New York, NY, USA (1994). https://doi.org/10.1145/182409.182453