Abstraction Refinement for Quantified Array Assertions

Abstraction Refinement for Quantified Array Assertions

Mohamed Nassim Seghir[1,⋆], Andreas Podelski[1], and Thomas Wies[1,2]

[1] University of Freiburg, Germany
[2] EPFL, Switzerland

Abstract. We present an abstraction refinement technique for the verification of universally quantified array assertions such as *"all elements in the array are sorted"*. Our technique can be seamlessly combined with existing software model checking algorithms. We implemented our technique in the ACSAR software model checker and successfully verified quantified array assertions for both text book examples and real-life examples taken from the Linux operating system kernel.

1 Introduction

Among the most promising approaches to the verification of software systems is the combination of predicate abstraction [10] with automated abstraction refinement [6]. This approach is commonly referred to as software model checking. Software model checking offers a high degree of automation and has been successfully applied to non-trivial programs such as device drivers. Existing software model checkers (e.g., SLAM [2], BLAST [13], MAGIC [5], and ARMC [21]) have shown to be suitable for the verification of control-oriented properties, but they are limited when it comes to richer properties that involve data structures. A prominent class of such properties are universally quantified assertions over arrays (e.g., sortedness). We show that careful adaptation of existing software model checking techniques is sufficient to verify many interesting programs over arrays.

In order to verify quantified assertions, a program analysis needs to infer inductive invariants that are itself quantified. This contradicts the basic idea of predicate abstraction which is to construct an invariant from small pieces, since quantified assertions cannot be easily split into simpler predicates. In other words, finding the right predicates for verifying quantified assertions becomes as difficult as finding an inductive invariant. Recently, various techniques have been developed that either generalize or extend existing abstract domains (including the predicate abstraction domain) to abstract domains that can express quantified properties [3, 11, 17, 22, 25]. However, none of these approaches can be easily

⋆ The first author was supported in part by the German Federal Ministry of Education and Research (BMBF) in the framework of the VerisoftXT project under grant 01 IS 07 008.

integrated into existing software model checkers without major changes to the underlying implementation or making the analysis less scalable.

A simpler approach towards verification of quantified assertions is due to Flanagan and Qadeer [8] and based on *ghost variables*. A ghost variable is an auxiliary program variable that is never modified by the program. It is only used for the purpose of verification. The idea in [8] is to replace each quantified variable in an assertion by a ghost variable. Thus, the ghost variables fix one instantiation for each quantified variable throughout the whole execution of the program. The transformed program can be analyzed using standard predicate abstraction and the inferred inductive invariant is implicitly universally quantified. While this approach is strictly weaker than an approach based on quantified abstract domains, it has shown to be suitable for verifying quantified array assertions with vanilla predicate abstraction, i.e., where all predicates have been provided by the user [8].

Problems arise when this approach is used together with automated abstraction refinement. Standard techniques for extracting predicates from spurious counterexamples such as (weakest) preconditions [1, 6] and interpolants [12] are insufficient. The reason is that these techniques do not infer predicates that allow the analysis to perform the necessary widening, i.e., to compute an invariant that states properties of unbounded intervals in the array. Therefore, the refinement loop often diverges, establishing the invariant, one by one, for all the individual entries in the unbounded intervals.

We adapt an existing abstraction refinement method to handle universally quantified assertions over arrays. Our technique is based on the idea of using ghost variables to eliminate universal quantifiers in assertions, but overcomes the limitation of standard abstraction refinement techniques described above. The technique is tailored towards assertions that quantify over index variables of arrays. It uses a theorem prover to derive consequences from spurious error paths. These consequences determine entries in the array that violate the target property. From these consequences our technique derives predicates that describe unbounded intervals in the array. These predicates enable the analysis to perform the necessary widening that results in a sufficiently strong inductive invariant.

Despite its simplicity our technique is surprisingly effective. We have implemented our technique in the ACSAR software model checker [24]. Using our implementation we successfully verified quantified array assertions for both text book examples such as sorting algorithms and real-life examples taken from the Linux operating system kernel and the Xen hypervisor.

2 Related Work

There have been various attempts to account for the verification of quantified properties including approaches based on predicate abstraction [8, 14, 17], first-order theorem provers [15,20], templates [3,11,25], and shape analysis [9,22]. Our approach is able to handle all array related examples that have been analyzed in [3, 8, 9, 11, 14, 17, 20]. Some of the examples in [15, 25] involve properties with

alternating universal and existential quantifiers such as permutation of arrays. These properties are outside the scope of our approach. In the following, we make a more detailed comparison.

Range predicates [14] describe properties of unbounded array segments which enables the verification of universally quantified array assertions using predicate abstraction with abstraction refinement. In the refinement phase, an axiom-based algorithm is applied to infer new range predicates as Craig interpolants for the spurious counterexample. Range predicates refer to an implicitly quantified variable that ranges over array indices. However, this approach does not handle properties that require quantification over more than one variable, such as properties of multidimensional arrays. Our approach does not have these restrictions.

Lahiri and Bryant proposed an extension of predicate abstraction to infer universally quantified invariants [17]. Their technique is based on index predicates which are predicates that contain free index variables. These index variables are implicitly universally quantified at each program location. Heuristics for inferring index predicates based on counterexample-guided abstraction refinement are described in [18]. This approach is more general than an approach based on ghost variables because the index variables occuring in the computed invariant are quantified per program location rather than globally for the entire program. However, the computation of abstract transformers is more involved than in classical predicate abstraction and requires theorem provers that can effectively deal with quantified formulas.

Several template-based techniques for generation of quantified invariants have been developed recently [3, 11, 25]. 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. These techniques can handle more complex properties than our approach. In particular, Srivastava and Gulwani [25] have used their approach to verify properties of arrays with alternating quantifiers. On the other hand, techniques that can effectively compute these templates and thus provide the same degree of automation as predicate abstraction refinement have not yet been developed.

Another interesting direction is the recent deployment of resolution-based first-order theorem provers for inferring quantified invariants over arrays. Existing approaches include [20] and [15]. McMillan's approach is based on the computation of quantified interpolants. The idea in [15] is to generate a set of clauses from quantified formulas that encode changes to arrays in the analyzed program, saturate the set under resolution, and then mine the saturated set for interesting quantified invariants. Currently these approaches are still limited due to the missing inbuilt support for arithmetic theories in the underlying theorem provers.

Abstract domains that are used in shape analyses such as in three-valued shape analysis [23] and Boolean heaps [22] can express quantified properties of unbounded data structures (namely, shape analysis constraints [16, 26] in the case of [23] and their universal fragment in the case of [22]). In particular, Gopan

et al. [9] have used three-valued shape analysis to verify properties of arrays. However, the abstract domains in these shape analyses are exponentially more succinct than the one used in predicate abstraction [19]. While this additional precision is needed for the analysis of programs manipulating linked data structures, our experience shows that it is not necessarily required for the verification of array related properties.

3 Experimental Results

Our work gives a positive answer to the question whether existing software model checking technology can be adapted to effectively verify quantified assertions over arrays and whether such an approach works well in practice. The most important contribution of our work are the experimental results confirming this answer. We start by presenting these results.

Implementation. We integrated our technique into the ACSAR software model checker [24]. The system implements a backward reachability analysis based on predicate abstraction refinement with lazy abstraction [13]. The implementation is done in C++. We performed tests using an X41 Thinkpad laptop with 1 GB of RAM and a 1.6 GHz CPU, running Linux. ACSAR uses the Yices theorem prover [7] for computing the abstraction and analyzing spurious counterexamples. The communication with Yices is performed through its API Lite. The input to ACSAR is a C program annotated with assertions to be verified. The output is either an invariant that implies the correctness of the annotated assertions or a counterexample trace.

Experiments. The results of our experiments are illustrated in Table 1. The column "Property" contains an informal description of the universally quantified assertion that we verified. Column "Iter" refers to the number of refinement steps performed until a safe invariant is computed. Finally, column "Pred" refers to the number of inferred predicates. Our tool is based on lazy abstraction [13], we therefore provide the average number of predicates per location instead of the total number of predicates. The size of examples varies from 10 to 200 lines of code. Although scalability is an important issue, the decisive factor here is the complexity of the property of interest.

Out of all examples, only find, cyber_init, perfect_copy_info, do_enoprof_op and selection_sort take more than 4 seconds verification time. The time includes all verification phases (parsing, theorem prover requests, etc.). The checked assertion for cyber_init is a conjunction of four assertions. The average time for checking each individual assertion is less than 4 seconds.

We divide our tests into two classes. The first class concerns academic examples taken from literature, their names appear without superscript. The example find was proposed by Qadeer and Flanagan [8]. Our tool automatically proves the postcondition specified in their paper. The example array_init is a simple array initialization program which is considered in most papers on array verification [3, 11, 14]. Programs num_index and part_init were proposed by Gopan *et*

al. [9]. The first one illustrates numeric constraints on the value of array elements. The second one aims to show the handling of multiple arrays as well as partial array initialization. Finally, partition was proposed by Henzinger *et al* [4]. It partitions a given array into two arrays `a` and `b` by copying the positive array entries into `a` and negative ones into `b`.

The second class of examples covers typical uses of arrays in real world system code. The programs are code fragments taken from the Linux kernel and driver code as well as the Xen hypervisor[3] code.

Selection sort. The most challenging benchmark that we considered is the selection sort example. We refer to Section 4.2, for the source code and a detailed description of this example. We verified that upon termination the array a is sorted in ascending order. The sortedness property was stated in the form

$$\forall x, y \in [0, n-1].\ x < y \Rightarrow a[x] \leq a[y]$$

ACSAR successfully verifies this property. The verification time is significantly larger than in our remaining benchmarks ($\sim$7 minutes). Inspection of the generated predicates revealed that the refinement loop generates many redundant predicates. We therefore believe that the verification time can be significantly reduced by implementing certain redundancy checks.

4 Examples

We now explain our approach and discuss two of the examples from the previous section in more detail: array initialization and selection sort. The first example illustrates the basic idea of our approach. The second example shows that it also works for challenging examples.

4.1 Array Initialization

Our first example is the simple procedure `array_init` shown in Figure 1. The procedure takes two arguments, an integer array `a` and an integer `n` denoting the length of `a`. The procedure initializes all entries of `a` to 0. We prove the assertion stating that after termination of the loop all array entries are indeed properly initialized. We use standard notation and formally represent programs in terms of transition constraints over primed and unprimed program variables. Figure 2 shows the corresponding transition constraints for procedure `array_init`. The program counter is modeled explicitly using the variable pc that ranges over control locations (ℓ_0 stands for the initial location and ℓ_E for the error location). Array a is represented by an uninterpreted function symbol. The notation $a[x := e]$ stands for a function update. The set of initial states of the program is described by the formula $pc = \ell_0$ and the set of error locations by the formula $pc = \ell_E$.

[3] A hypervisor is a software that permits hardware virtualization. It allows multiple operating systems to run in parallel on a computer. The Xen hypervisor is available at http://www.xen.org/

Program	Property	Iter.	Pred.	Time (s)
string_copy	0 terminal string s_1 is copied to s_2	2	4	0.63
scan	array entries before actual entry are not null	3	3	0.54
array_init	array entries are initialized	3	6	0.83
loop1	each array entry is initialized with its index	3	5	0.71
copy1	array a is copied to array b	2	6	0.84
partition	array a contains positive entries and array b contains negative ones	8	7	1.94
num_index	for every array entry i of array a we have $a[i] = 2 * i + 3$	2	6	0.89
part_init	all array entries are initialized to values between 0 and $n - 1$	5	9	3.17
find	every array entry whose index is less than the returned value contains false	8	13	8.81
insertion_sort (inner loop)	entry $a[j]$ is less or equal than all entries of the segment $a[j \dots i]$	2	14	2.45
selection_sort	array is sorted	3	39	409.87
cyber_init*	for every i, if i modulo 4 is equal to 0, 1, 2 or 3 then $a[i]$ is initialized to v_0, v_1, v_2 or v_3 respectively	8	13	10.36
i2o_device_parse_lct*	entries preceding the actual entry are different from a given value	5	4	1.64
ixj_pad_fsk*	after the execution of 2 loops entries in a given range are initialized	6	7	1.53
ixj_daa_cid_read*	all entries with odd index are equal to v_1 all entries with even index are equal to v_2	5	14	3.81
snd_atiixp_mixer_new*	entries having property p in their pre-state are set to NULL	3	4	1.25
dvb_net_feed_stop*	entries different from 0 in their pre-state are set to 0	3	7	3.41
perfc_copy_info**	for each entry i of array a if $a[i]$ has some property then $b[i]$ and $c[i]$ should be equal	4	12	10.57
do_enoprof_op**	if variable op has value v_1 and variable s has value v_2 then array a is copied to array b	26	3	34.17

Table 1. Experimental results for academic and industrial examples. The upper half of the table refers to examples taken from literature. The lower half refers to examples taken from system code. Examples marked with superscript $*$ are from the Linux kernel and driver code. Examples marked with $**$ are taken from the Xen hypervisor code.

```
      void array_init (int a[], int n)
      {
        int i;
ℓ0:
ℓ1:   for(i = 0; i < n; ++i)
        {
          a[i] = 0;
        }

ℓ2:   assert(∀ x. x ≥ 0 ∧ x < n ⇒ a[x] = 0);
      }
```

Fig. 1. Array initialization

$$\tau_0 : pc = \ell_0 \wedge pc' = \ell_1 \wedge a' = a \wedge i' = 0 \wedge k' = k$$
$$\tau_1 : pc = \ell_1 \wedge i < n \wedge pc' = \ell_1 \wedge a' = a[i := 0] \wedge i' = i + 1 \wedge k' = k$$
$$\tau_2 : pc = \ell_1 \wedge i \geq n \wedge pc' = \ell_2 \wedge a' = a \wedge i' = i \wedge k' = k$$
$$\tau_3 : pc = \ell_2 \wedge 0 \leq k \wedge k < n \wedge a(k) \neq 0 \wedge pc' = \ell_E \wedge a' = a \wedge i' = i \wedge k' = k$$

Fig. 2. Transition constraints for array initialization

Transition τ_0 models the initialization of the loop counter in the `for` loop of procedure `array_init`, transition τ_1 models the loop body, and τ_2 the loop exit. The assert statement is reflected by transition τ_3 that goes from the loop exit location ℓ_2 to the error location ℓ_E. We use the idea from [8] and replace the quantified variable x in the original assertion by a ghost variable k. Our goal is to prove that the program represented by the transition constraints is safe, i.e., that no error state is reachable from an initial state by consecutive execution of the transitions represented by the transition constraints. If no error state is reachable then the assertion in procedure `array_init` is never violated.

Our algorithm performs a backward reachability analysis starting from the error states and computes an inductive backward invariant, i.e., an overapproximation of the set of states that are backward-reachable from an error state. If the computed invariant is disjoint from the initial states then the program is safe. An inductive backward invariant for the array initialization program that is disjoint from the initial states is given by the following formula φ:

$$\varphi \stackrel{\text{def}}{=} \quad pc \neq \ell_0 \wedge (pc = \ell_1 \;\Rightarrow\; 0 \leq k \wedge k < n \wedge a(k) \neq 0 \wedge k < i)$$

Note that due to the fact that φ is a backward invariant the ghost variable k is implicitly *existentially* quantified. Our analysis is based on predicate abstraction with counterexample guided abstraction refinement. Thus, if the refinement loop is able to infer predicates whose Boolean combination can express φ then the backward analysis will construct a sufficiently strong invariant.

The basis of our refinement procedure is a predicate extraction function that syntactically extracts predicates from preconditions that are computed from spurious error paths. For instance, if we start with an empty set of predicates then the first iteration of the refinement process that goes through the program loop produces the spurious error path $\tau_0; \tau_1; \tau_2; \tau_3$. It then extracts all atomic subformulas from the precondition of the feasible part of the error path: $\mathsf{pre}(\tau_1; \tau_2; \tau_3, pc = \ell_E)$. This formula is given by

$$pc = \ell_1 \wedge 0 \leq k \wedge k < n \wedge a[i := 0](k) \neq 0 \wedge i < n \wedge i + 1 \geq n \qquad (1)$$

Note that function updates such as $a[x := 0]$ can be eliminated via case splits. If we only extracted atomic formulas from preconditions then the analysis would unroll the loop in procedure `array_init` and enumerate all predicates that occur in preconditions of the form

$$\mathsf{pre}((\tau_1)^+; \tau_2; \tau_3, pc = \ell_E)$$

but never infer the predicate $k < i$. The refinement would fail to perform the necessary widening that ensures termination of the analysis. We developed a simple technique that realizes this kind of widening.

First, our technique extracts all ghost variables and index expressions that occur as indices of arrays in the precondition (1) of the counterexample path. Then it determines all disjunctions of inequalities $s_i \neq t_i$ over pairs (s_i, t_i) of index expressions that are consequences of the formula (1). The individual disjuncts $s_i \neq t_i$ of such consequences are then split into inequalities $s_i < t_i$, $s_i > t_i$ and added as additional abstraction predicates. The intuition behind this technique is that the considered disequalities determine the boundaries of intervals in the array that violate the target property. Splitting the disequality into inequalities allows the analysis to perform the necessary widening to infer a sufficiently strong invariant.

In our example the only candidate disequality is given by $k \neq i$ which is indeed a consequence of the formula (1). We therefore add the inequalities $k < i$ and $k > i$ to the set of abstraction predicates which ensures that the refinement loop terminates.

4.2 Selection Sort

Our second example is the procedure `selection_sort` shown in Figure 3. This example is more challenging because it has the so-called *write-many* property, i.e., an array entry can be updated more than once. We show that upon termination of the outer loop, all elements of array `a` are sorted in ascending order.

The set of transition constraints encoding procedure `selection_sort` is given in Figure 4. Constraints τ_0 models the initialization of the outer for loop, τ_1 models the statement before location ℓ_2 and the initialization of the inner for loop, τ_2 and τ_3 model the body of the inner loop, τ_4 the exit of the inner loop and the remaining body of the outer loop, and τ_5 the exit of the outer loop. The assert statement checking the sortedness property in the original program is model by

```
      void selection_sort (int a[], int n)
      {
        int i, j, s;
ℓ0:
ℓ1:     for(i = 0; i < n; ++i)
        {
          s = i;
ℓ2:       for(j = i+1; j < n; ++j)
          {
            if(a[j] < a[s])
            {
              s = j;
            }
          }
          t = a[i];
          a[i] = a[s];
          a[s] = t;
        }

ℓ3:     assert(∀ x y. 0 ≤ x < n ∧ 0 ≤ y < n ∧ x < y ⇒ a[x] ≤ a[y]);
      }
```

Fig. 3. Selection sort

$$\tau_0 : pc = \ell_0 \wedge pc' = \ell_1 \wedge i' = 0 \wedge j' = j \wedge s' = s \wedge k' = k$$
$$\tau_1 : pc = \ell_1 \wedge i < n \wedge pc' = \ell_2 \wedge s' = i \wedge j' = i + 1 \wedge i' = i \wedge k' = k$$
$$\tau_2 : pc = \ell_2 \wedge j < n \wedge a(j) \geq a(s) \wedge pc' = \ell_2 \wedge a' = a \wedge i' = i \wedge j' = j + 1 \wedge s' = s \wedge k' = k$$
$$\tau_3 : pc = \ell_2 \wedge j < n \wedge a(j) < a(s) \wedge pc' = \ell_2 \wedge a' = a \wedge i' = i \wedge j' = j + 1 \wedge s' = j \wedge k' = k$$
$$\tau_4 : pc = \ell_2 \wedge j \geq n \wedge pc' = \ell_1 \wedge a' = a[i := a(s), s := a(i)] \wedge i' = i + 1 \wedge k' = k$$
$$\tau_5 : pc = \ell_1 \wedge i \geq n \wedge pc' = \ell_3 \wedge a' = a \wedge k' = k$$
$$\tau_6 : pc = \ell_3 \wedge 0 \leq k < n \wedge 0 \leq l < n \wedge l < k \wedge a(k) < a(l) \wedge pc' = \ell_E \wedge a' = a \wedge k' = k$$

Fig. 4. Transition constraints for selection sort

τ_6. We introduce the two ghost variables k and l for the universally quantified variables x and y in the original assertion. The following formula shows one of the disjuncts of a safe inductive backward invariant. The shown disjunct covers all backward-reachable states at program location ℓ_1, i.e., the loop cut point of the outer loop in procedure `selection_sort`:

$$pc = \ell_1 \wedge 0 \leq l \wedge l < i \wedge l < k \wedge k < n \wedge a(k) < a(l)$$

We sketch how the analysis infers the predicate $l < i$. After several iterations our analysis returns the spurious counterexample $\tau_0; \pi$ where

$$\pi \stackrel{\text{def}}{=} \tau_1; \tau_2; \tau_4; \tau_1; \tau_4; \tau_5; \tau_6$$

Again we extract atomic predicates from the preconditions of the error path and infer additional predicates by checking disequalities that are implied by preconditions of the feasible part of the counterexample. For instance, consider the precondition $\mathsf{pre}(\pi, pc = \ell_E)$ which is given by

$$\begin{array}{l} 0 \leq k < n \wedge 0 \leq l < n \wedge l < k \wedge i + 1 < n \wedge a(i) \leq a(i+1) \wedge \\ n \leq i + 2 \wedge a[i := a(i), i := a(i)](k) < a[i := a(i), i := a(i)](l) \end{array} \tag{2}$$

Note that the updated function $a[i := a(i), i := a(i)]$ is equal to a. Furthermore, it is easy to see that the implication

$$k = i + 1 \wedge l = i \;\Rightarrow\; a(i) > a(i+1) \vee a(k) \geq a(l)$$

is valid. Thus, by contraposition (2) implies the disjunction of inequalities

$$k \neq i + 1 \vee l \neq i$$

From this disjunction we extract the predicates

$$l < i,\ l > i,\ i + 1 > k, \text{ and } i + 1 < k\ .$$

5 Predicate Abstraction Refinement

In this section, we describe the by now classical setting of predicate abstraction refinement. The method is parameterized by the procedure extract that takes a formula and returns a set of *predicates*. We use a minimal notational setting (following, e.g., [1]) and ignore details (in particular, the concrete programming language and the use of concrete counterexamples for refinement). These details are irrelevant for our main purpose, which is to introduce the specific procedure extract used in our analysis of array programs (in the next section). Everything in this setting is standard up to the syntax of the formulas that we use to denote sets of states, in the concrete as well as in the abstract domain.

Concrete domain of formulas. We assume a (generally infinite) set of quantifier-free formulas which we call *base formulas*. We represent an (in general infinite) set of states by a first-order formula φ built up from such base formulas. In our setting, φ is of the form

$$\varphi \;\equiv\; \bigvee_{i \in I} \bigwedge_{j \in J_i} \varphi_{ij} \tag{3}$$

where the φ_{ij}'s are base formulas.

We assume a partial order on formulas $\varphi' \leq \varphi$. The partial order is usually a sound but possibly incomplete implementation (by a theorem prover) of the test of validity of implication.

Pre. A program is a set $\mathcal{P}$ of statements st. For the purpose of the formal presentation, we assume that a statement comes as a transition constraint

$$\mathit{st} \;\equiv\; \psi \wedge x_1' = e_1 \wedge \ldots \wedge x_m' = e_m$$

where $x_1, x_2, \ldots, x_m$ are variables (including a program counter pc); as usual, the variable x' stands for the value of x in the successor state. The guard ψ is a conjunction of base formulas. The update formula comes as a conjunction of logical equalities between primed variables and expressions over unprimed variables.

For a statement st, the application of the operator $\mathsf{pre}_{\mathit{st}}$ on a formula φ returns a formula representing the set of all predecessor states of φ under the statement st. The definition extends canonically to a sequence of statements. For a statement st as above, the application of the operator $\mathsf{pre}_{\mathit{st}}$ to the formula φ is implemented by the projection (on unprimed variables) of the conjunction the transition constraint with the renaming of φ (from unprimed to primed variables). The operator pre for a program (a set of statements) is simply the disjunction of the $\mathsf{pre}_{\mathit{st}}$ over all statements.

$$\mathsf{pre}_{\mathit{st}}(\varphi) \equiv \exists x_1' \ldots \exists x_m' (\varphi[x_1'/x_1, \ldots, x_m'/x_m] \wedge \psi \wedge x_1' = e_1 \wedge \ldots \wedge x_m' = e_m)$$
$$\mathsf{pre}(\varphi) \equiv \bigvee_{\mathit{st} \in \mathcal{P}} \mathsf{pre}_c(\varphi)$$

Invariants. In order to specify correctness, we fix formulas $\mathsf{nonInit}$ and unsafe denoting the complement of the set of *initial* and *safe* states, respectively. We define the given program to be *correct* if no unsafe state is reachable from an initial state. In our setting, $\mathsf{nonInit}$ is quantifier-free (but unsafe is not).

The correctness can be proven by showing the condition below. Here, $\mathsf{lfp}(\mathsf{pre}, \varphi)$ stands for the least fixpoint of the operator pre above φ.

$$\mathsf{lfp}(\mathsf{pre}, \mathsf{unsafe}) \leq \mathsf{nonInit}$$

A *backward invariant* is an invariant that is *inductive* under pre and implies $\mathsf{nonInit}$, i.e. a formula ψ such that

- $\mathsf{unsafe} \leq \psi$,
- $\mathsf{pre}(\psi) \leq \psi$,
- $\psi \leq \mathsf{nonInit}$.

Predicate abstraction. A possible approach to establish correctness is to find an upper abstraction $\mathsf{pre}^{\#}$ of the operator pre (i.e. where $\mathsf{pre}(\varphi) \leq \mathsf{pre}^{\#}(\varphi)$ holds for all formulas φ) such that $\mathsf{lfp}(\mathsf{pre}^{\#}, \mathsf{unsafe})$, the least fixpoint of $\mathsf{pre}^{\#}$ above unsafe, can be computed and is contained in $\mathsf{nonInit}$. Then, $\mathsf{lfp}(\mathsf{pre}^{\#}, \mathsf{unsafe})$ is a backward invariant because of the simple fact that $\mathsf{pre}^{\#}(\varphi) \leq \varphi$ entails $\mathsf{pre}(\varphi) \leq \varphi$. We use predicate abstraction with abstraction refinement to find such an upper abstraction $\mathsf{pre}^{\#}$.

The method generates a sequence of finite sets $\mathcal{P}_n$ of predicates over states (for $n = 0, 1, \ldots$). Since we identify a predicate with the base formula φ defining it, we have that $\mathcal{P}_n$ is a *finite* subset of the given set of base formulas.

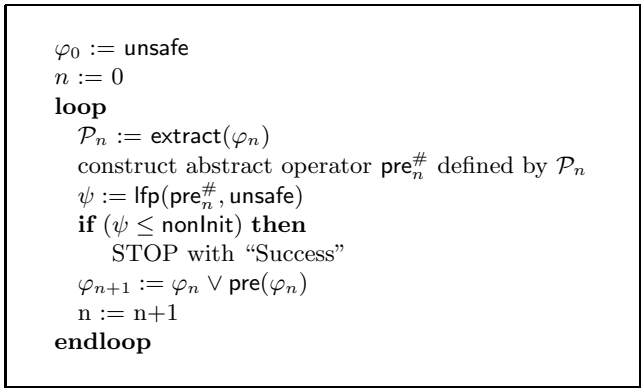

Fig. 5. Abstract fixpoint checking with iterative abstraction refinement, where extract is a parameterized procedure that infers a finite set of predicates from a formula and $\mathsf{pre}_n^\#$ is a predicate abstraction of pre for the set of predicates $\mathcal{P}_n$.

We write $\mathcal{L}(\mathcal{P}_n)$ for the (finite!) sublattice of $\mathcal{L}$ that is generated by the set of predicates $\mathcal{P}_n$. We sometimes refer to conjunctions of predicates as "abstract states" (thus, abstract states are exactly the symbolic states in $\mathcal{L}(\mathcal{P}_n)$). We have that $\mathcal{L}(\mathcal{P}_n)$ contains unsafe, but generally $\mathcal{L}(\mathcal{P}_n)$ is not closed with respect to the operator pre. We define the operator $\mathsf{pre}_n^\#$ over $\mathcal{L}(\mathcal{P}_n)$ as an abstraction of pre.

The 'best' abstraction $\mathsf{pre}_n^\#$ of pre with respect to $\mathcal{P}_n$ is defined in terms of a Galois connection,

$$\mathsf{pre}_n^\# \equiv \alpha_n \circ \mathsf{pre} \circ \gamma$$

where the composition $f \circ g$ of two functions f and g is defined from right to left: $f \circ g(x) = f(g(x))$. The abstraction function α_n maps a formula φ to the smallest formula φ' in $\mathcal{L}(\mathcal{P}_n)$ that is larger (wrt. "$\leq$") than φ, formally

$$\alpha_n(\varphi) \equiv \mu\varphi' \in \mathcal{L}(\mathcal{P}_n)^{\sqsubseteq}.\ \varphi \leq \varphi'.$$

The meaning function γ is the identity.

The construction of the best abstraction is not practical. Hence, one uses a weaker abstraction of pre and one defines $\mathsf{pre}_n^\#$ not as the function above but, instead, as follows.

$$\mathsf{pre}_n^\#(\bigvee_{i\in I}\bigwedge_{j\in J_i}\varphi_{ij}) = \bigvee_{i\in I}\mathsf{pre}_n^\#(\bigwedge_{j\in J_i}\varphi_{ij})$$

and

$$\mathsf{pre}_n^\#(\bigwedge_{j\in J_i}\varphi_{ij}) = \bigwedge\{p \in \mathcal{P}_n \mid \mathsf{pre}(\bigwedge_{j\in J_i}\varphi_{ij}) \leq p\}.$$

Thus, the image of an abstract state (i.e., a conjunction of predicates) under $\mathsf{pre}_n^\#$ yields the smallest abstract state above its image under pre.

We will have that $\mathcal{P}_0 \subset \mathcal{P}_1 \subset \ldots$ and hence $\mathcal{L}(\mathcal{P}_0) \subset \mathcal{L}(\mathcal{P}_1) \subset \ldots$ which means an increasing precision of the abstraction α_n for increasing n.

The iterative abstraction refinement method. The method in Figure 5 is parameterized by the refinement procedure extract which takes a formula and returns a finite set of base formulas ("the new predicates"). In each iteration, the method

- constructs the abstract operator $\mathsf{pre}_n^\#$ defined by $\mathcal{P}_n$,
- computes the abstract fixpoint $\mathsf{lfp}(\mathsf{pre}_n^\#, \mathsf{start})$,
- generates a new set of predicates $\mathcal{P}_{n+1}$

until the abstract proof succeeds, i.e., $\mathsf{lfp}(\mathsf{pre}_n^\#, \mathsf{unsafe}) \leq \mathsf{nonInit}$ for some n.

If the method terminates for some n, then $\mathsf{lfp}(\mathsf{pre}_n^\#, \mathsf{unsafe})$ is a backward invariant computed over a finite lattice.

6 Refinement for Arrays

The refinement scheme defined in Figure 5 is parameterized by the procedure extract. This procedure takes a conjunction φ of base formulas and returns a set of base formulas (which are then used to define a set of new predicates). In its most basic version, the procedure $\mathsf{extract}_0$ returns the set of conjuncts.

$$\mathsf{extract}_0(\varphi_1 \wedge \ldots \wedge \varphi_n) = \{\varphi_1, \ldots, \varphi_n\}$$

The rationale for our extension of the procedure $\mathsf{extract}_0$ stems from a result in [1]. This result formally evaluates the power of the refinement scheme with the procedure $\mathsf{extract}_0$ above (the power as a proof method for program correctness). The evaluation uses an idealized oracle-based proof method for comparison. This method works by backward iteration of the (concrete) *pre* operator; i.e., it starts with the formula unsafe and iteratively applies the operator pre. In order to accelerate the convergence towards a fixpoint, it judiciously applies a *syntactic widening* on the formula obtained. The syntactic widening applied to a conjunction φ drops one or more of its conjuncts in φ (for example, applied to the interval constraint $0 < x \wedge x < 1$ it may result in $0 < x$). It is the oracle which judiciously chooses what conjuncts to drop and what conjuncts to keep. The result in [1] states that the (realistic) refinement scheme with the procedure $\mathsf{extract}_0$ achieves the same power as the idealized oracle-based method with syntactic widening.

In our setting, with programs over arrays, the backward invariants used in correctness proofs contain conjuncts that not syntactically appear in the iterates of the backward iteration procedure. This means that the syntactic widening is not sufficient (even in the idealized proof method above); we need to combine it with a semantic analysis in order to obtain a greater choice for the possible widening results. The *semantic widening* applied to a conjunction φ first *saturates* the conjunction, i.e., adds redundant conjuncts (logical consequences of a

certain form), and then applies the syntactic widening to the resulting conjunction.

The saturation consists of adding each disjunction of strict inequalities between index variables x_i and y_i that is entailed (in the theory of linear arithmetic with uninterpreted function symbols) by φ.

$$\mathsf{saturate}(\varphi) = \varphi \wedge \bigwedge \{ \bigvee_{i \in I} x_i < y_i \mid \varphi \models \bigvee_{i \in I} x_i < y_i \}$$

If a disjunction of disequalities $\bigvee_{i \in I} x_i \neq y_i$ is entailed by φ, as, for example, in

$$a[x] > a[y] \wedge a[z] > a[t] \models (x \neq y) \vee (z \neq t) \vee (x \neq t) \vee (z \neq y)$$

then one obtains the corresponding entailed disjunction of inequalities by replacing each of the disequalities by the disjunction of the two corresponding inequalities.

This leads us to define the predicate extraction procedure $\mathsf{extract}_1$ as the composition of the saturation with the syntactic widening.

$$\mathsf{extract}_1(\varphi) = \mathsf{extract}_0(\mathsf{saturate}(\varphi))$$

Our proof method is the instantiation of the refinement scheme of Figure 5 with the predicate extraction procedure $\mathsf{extract}_1$. By the above-mentioned result in [1], this proof method has the same power as the idealized oracle-based method with semantic widening. I.e., if the unrealistic oracle-based method succeeds in proving a program correct, then so does our method.

Practical optimizations. A naive implementation of the procedure saturate, which consists of enumerating all possible disjunctions of inequalities over all index expressions, requires exponentially many (in the number of occuring index expressions) theorem prover queries. In practice we can impose a polynomial bound by considering only disjunctions up to a fixed length. For further optimization, we only consider inequalities between index expressions associated to the same array (not blindly any pair of index expressions). In addition, we construct the checked disjunctions incrementally starting from disjunctions of length one and if a disjunction is entailed, do not consider any longer disjunction that includes it.

7 Conclusion

We presented an abstraction refinement technique for verifying quantified assertions over arrays that can be easily integrated into existing software model checkers. Using this technique we were able to verify almost all array related examples in the literature that have been verified using quantified abstract domains. Furthermore, we were able to verify various real-life examples taken from system code. Our results indicate that, at least for quantified assertions over arrays, the use of sophisticated techniques for dealing with quantified assertions can often be avoided if one instead carefully adapts existing techniques for quantifier-free assertions by using domain specific knowledge.

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*, volume 2280 of *LNCS*, pages 158–172, 2002.
2. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
3. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, volume 4349 of *LNCS*, pages 378–394, 2007.
4. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.
5. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, pages 385–395, 2003.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.
7. B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*, pages 81–94, 2006.
8. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.
9. D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.
10. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
11. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.
12. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
13. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
14. R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV*, pages 193–206, 2007.
15. L. Kovacs and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, 2009. To appear.
16. V. Kuncak and M. Rinard. Boolean Algebra of Shape Analysis Constraints. In *VMCAI*, volume 2937 of *LNCS*, pages 59–72, 2004.
17. S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, pages 267–281, 2004.
18. S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004.
19. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *VMCAI*, volume 3385 of *LNCS*, pages 181–198, 2005.
20. K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, volume 4963 of *LNCS*, pages 413–427, 2008.
21. A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, volume 4354 of *LNCS*, pages 245–259, 2007.
22. A. Podelski and T. Wies. Boolean heaps. In *SAS*, pages 268–283, 2005.
23. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 2002.

24. M. N. Seghir and A. Podelski. ACSAR: Software model checking with transfinite refinement. In *SPIN*, pages 274–278, 2007.
25. S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, 2009. To appear.
26. G. Yorsh, T. W. Reps, M. Sagiv, and R. Wilhelm. Logical Characterizations of Heap Abstractions. *ACM Transactions on Computational Logic*, 8(1), 2007.