Diffy: Inductive Reasoning of Array Programs Using Difference Invariants

Supratik Chakraborty[1], Ashutosh Gupta[1], and Divyesh Unadkat[1,2](✉)

[1] Indian Institute of Technology Bombay, Mumbai, India
{supratik,akg}@cse.iitb.ac.in
[2] TCS Research, Pune, India
divyesh.unadkat@tcs.com

Abstract. We present a novel verification technique to prove properties of a class of array programs with a symbolic parameter N denoting the size of arrays. The technique relies on constructing two slightly different versions of the same program. It infers difference relations between the corresponding variables at key control points of the joint control-flow graph of the two program versions. The desired post-condition is then proved by inducting on the program parameter N, wherein the difference invariants are crucially used in the inductive step. This contrasts with classical techniques that rely on finding potentially complex loop invaraints for each loop in the program. Our synergistic combination of inductive reasoning and finding simple difference invariants helps prove properties of programs that cannot be proved even by the winner of Arrays sub-category in SV-COMP 2021. We have implemented a prototype tool called Diffy to demonstrate these ideas. We present results comparing the performance of Diffy with that of state-of-the-art tools.

1 Introduction

Software used in a wide range of applications use arrays to store and update data, often using loops to read and write arrays. Verifying correctness properties of such array programs is important, yet challenging. A variety of techniques have been proposed in the literature to address this problem, including inference of quantified loop invariants [20]. However, it is often difficult to automatically infer such invariants, especially when programs have loops that are sequentially composed and/or nested within each other, and have complex control flows. This has spurred recent interest in mathematical induction-based techniques for verifying parametric properties of array manipulating programs [11,12,42,44]. While induction-based techniques are efficient and quite powerful, their Achilles heel is the automation of the inductive argument. Indeed, this often becomes the limiting step in applications of induction-based techniques. Automating the induction step and expanding the class of array manipulating programs to which induction-based techniques can be applied forms the primary motivation for our work. Rather than being a stand-alone technique, we envisage our work being used as part of a portfolio of techniques in a modern program verification tool.

© The Author(s) 2021
A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 911–935, 2021.
https://doi.org/10.1007/978-3-030-81688-9_42

We propose a novel and practically efficient induction-based technique that advances the state-of-the-art in automating the inductive step when reasoning about array manipulating programs. This allows us to automatically verify interesting properties of a large class of array manipulating programs that are beyond the reach of state-of-the-art induction-based techniques, viz. [12,42]. The work that comes closest to us is Vajra [12], which is part of the portfolio of techniques in VeriAbs [1] – the winner of SV-COMP 2021 in the Arrays Reach sub-category. Our work addresses several key limitations of the technique implemented in Vajra, thereby making it possible to analyze a much larger class of array manipulating programs than can be done by VeriAbs. Significantly, this includes programs with nested loops that have hitherto been beyond the reach of automated techniques that use mathematical induction [12,42,44].

A key innovation in our approach is the construction of two slightly different versions of a given program that have identical control flow structures but slightly different data operations. We automatically identify simple relations, called *difference invariants*, between corresponding variables in the two versions of a program at key control flow points. Interestingly, these relations often turn out to be significantly simpler than inductive invariants required to prove the property directly. This is not entirely surprising, since the difference invariants depend less on what individual statements in the programs are doing, and more on the difference between what they are doing in the two versions of the program. We show how the two versions of a given program can be automatically constructed, and how differences in individual statements can be analyzed to infer simple difference invariants. Finally, we show how these difference invariants can be used to simplify the reasoning in the inductive step of our technique.

We consider programs with (possibly nested) loops manipulating arrays, where the size of each array is a symbolic integer parameter N (> 0)[1]. We verify (a sub-class of) quantified and quantifier-free properties that may depend on the symbolic parameter N. Like in [12], we view the verification problem as one of proving the validity of a parameterized Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ for all values of N (> 0), where arrays are of size N in the program P_N, and N is a free variable in $\varphi(\cdot)$ and $\psi(\cdot)$.

To illustrate the kind of programs that are amenable to our technique, consider the program shown in Fig. 1(a), adapted from an SV-COMP benchmark. This program has a couple of sequentially composed loops that update arrays and scalars. The scalars `S` and `F` are initialized to 0 and 1 respectively before the first loop starts iterating. Subsequently, the first loop computes a recurrence in variable `S` and initializes elements of the array `B` to 1 if the corresponding elements of array `A` have non-negative values, and to 0 otherwise. The outermost branch condition in the body of the second loop evaluates to true only if the program parameter N and the variable `S` have same values. The value of `F` is reset based on some conditions depending on corresponding entries of arrays `A` and `B`. The pre-condition of this program is `true`; the post-condition asserts that `F` is never reset in the second loop.

[1] For a more general class of programs supported by our technique, please see [13].

```
// assume(true)
1. S = 0; F = 1;
2. for(i = 0; i< N; i++) {
3.  S = S + 1;
4.  if ( A[i] >= 0 ) B[i] = 1;
5.  else B[i] = 0;
6. }
7. for(j = 0; j< N; j++) {
8.  if(S == N) {
9.   if ( A[j] >= 0 && !B[j] ) F = 0;
10.  if ( A[j] < 0 && B[j] ) F = 0;
11. }
12.}
// assert(F == 1)
```

(a)

```
// assume(true)
1. S = 0;
2. for(i=0; i<N; i++) A[i] = 0;
3. for(j=0; j<N; j++) S = S + 1;
4. for(k=0; k<N; k++) {
5.  for(l=0; l<N; l++) A[l] = A[l] + 1;
6.  A[k] = A[k] + S;
7. }
// assert(forall x in [0,N), A[x]==2*N)
```

(b)

Fig. 1. Motivating examples

State-of-the-art techniques find it difficult to prove the assertion in this program. Specifically, VAJRA [12] is unable to prove the property, since it cannot reason about the branch condition (in the second loop) whose value depends on the program parameter N. VERIABS [1], which employs a sequence of techniques such as loop shrinking, loop pruning, and inductive reasoning using [12] is also unable to verify the assertion shown in this program. Indeed, the loops in this program cannot be merged as the final value of `S` computed by the first loop is required in the second loop; hence loop shrinking does not help. Also, loop pruning does not work due to the complex dependencies in the program and the fact that the exact value of the recurrence variable `S` is required to verify the program. Subsequent abstractions and techniques applied by VERIABS from its portfolio are also unable to verify the given post-condition. VIAP [42] translates the program to a quantified first-order logic formula in the theory of equality and uninterpreted functions [32]. It applies a sequence of tactics to simplify and prove the generated formula. These tactics include computing closed forms of recurrences, induction over array indices and the like to prove the property. However, its sequence of tactics is unable to verify this example within our time limit of 1 min.

Benchmarks with nested loops are a long standing challenge for most verifiers. Consider the program shown in Fig. 1(b) with a nested loop in addition to sequentially composed loops. The first loop initializes entries in array `A` to 0. The second loop aggregates a constant value in the scalar `S`. The third loop is a nested loop that updates array `A` based on the value of `S`. The entries of `A` are updated in the inner as well as outer loop. The property asserts that on termination, each array element equals twice the value of the parameter N.

While the inductive reasoning of VAJRA and the tactics in VIAP do not support nested loops, the sequence of techniques used by VERIABS is also unable to prove the given post-condition in this program. In sharp contrast, our prototype tool DIFFY is able to verify the assertions in both these programs automatically within a few seconds. This illustrates the power of the inductive technique proposed in this paper.

The technical contributions of the paper can be summarized as follows:

- We present a novel technique based on mathematical induction to prove interesting properties of a class of programs that manipulate arrays. The crucial inductive step in our technique uses difference invariants from two slightly different versions of the same program, and differs significantly from other induction-based techniques proposed in the literature [11,12,42,44].
- We describe algorithms to transform the input program for use in our inductive verification technique. We also present techniques to infer simple difference invariants from the two slightly different program versions, and to complete the inductive step using these difference invariants.
- We describe a prototype tool Diffy that implements our algorithms.
- We compare Diffy vis-a-vis state-of-the-art tools for verification of C programs that manipulate arrays on a large set of benchmarks. We demonstrate that Diffy significantly outperforms the winners of SV-COMP 2019, 2020 and 2021 in the Array Reach sub-category.

2 Overview and Relation to Earlier Work

In this section, we provide an overview of the main ideas underlying our technique. We also highlight how our technique differs from [12], which comes closest to our work. To keep the exposition simple, we consider the program P_N, shown in the first column of Fig. 2, where N is a symbolic parameter denoting the sizes of arrays `a` and `b`. We assume that we are given a parameterized pre-condition $\varphi(N)$, and our goal is to establish the parameterized post-condition $\psi(N)$, for all $N > 0$. In [12,44], techniques based on mathematical induction (on N) were proposed to solve this class of problems. As with any induction-based technique, these approaches consist of three steps. First, they check if the *base case* holds, i.e. if the Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ holds for small values of N, say $1 \leq N \leq M$, for some $M > 0$. Next, they assume that the *inductive hypothesis* $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds for some $N \geq M+1$. Finally, in the *inductive step*, they show that if the inductive hypothesis holds, so does $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$. It is not hard to see that the inductive step is the most crucial step in this style of reasoning. It is also often the limiting step, since not all programs and properties allow for efficient inferencing of $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ from $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$.

Like in [12,44], our technique uses induction on N to prove the Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ for all $N > 0$. Hence, our base case and inductive hypothesis are the same as those in [12,44]. However, our reasoning in the crucial inductive step is significantly different from that in [12,44], and this is where our primary contribution lies. As we show later, not only does this allow a much larger class of programs to be efficiently verified compared to [12,44], it also permits reasoning about classes of programs with nested loops, that are beyond the reach of [12,44]. Since the work of [12] significantly generalizes that of [44], henceforth, we only refer to [12] when talking of earlier work that uses induction on N.

In order to better understand our contribution and its difference vis-a-vis the work of [12], a quick recap of the inductive step used in [12] is essential. The

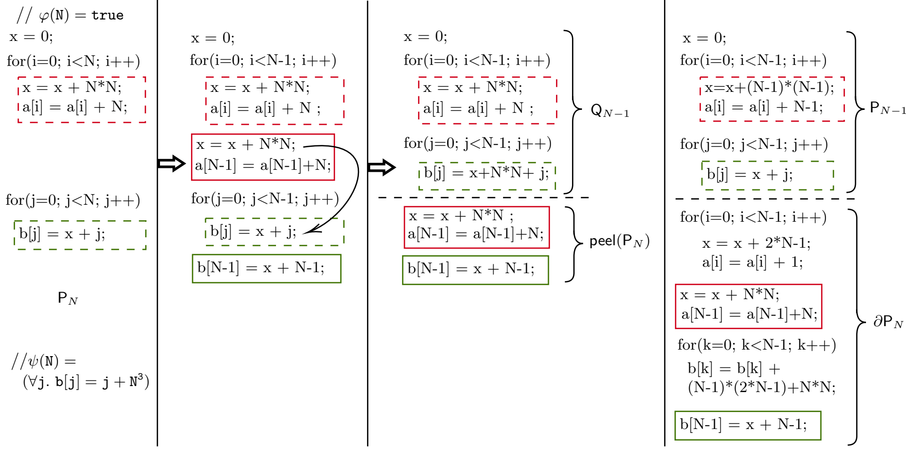


Fig. 2. Pictorial depiction of our program transformations

inductive step in [12] crucially relies on finding a "difference program" $\partial\mathsf{P}_N$ and a "difference pre-condition" $\partial\varphi(N)$ such that: (i) P_N is semantically equivalent to $\mathsf{P}_{N-1}; \partial\mathsf{P}_N$, where ';' denotes sequential composition of programs[2], (ii) $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$, and (iii) no variable/array element in $\partial\varphi(N)$ is modified by P_{N-1}. As shown in [12], once $\partial\mathsf{P}_N$ and $\partial\varphi(N)$ satisfying these conditions are obtained, the problem of proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ can be reduced to that of proving $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$. This approach can be very effective if (i) $\partial\mathsf{P}_N$ is "simpler" (e.g. has fewer loops or strictly less deeply nested loops) than P_N and can be computed efficiently, and (ii) a formula $\partial\varphi(N)$ satisfying the conditions mentioned above exists and can be computed efficiently.

The requirement of P_N being semantically equivalent to $\mathsf{P}_{N-1}; \partial\mathsf{P}_N$ is a very stringent one, and finding such a program $\partial\mathsf{P}_N$ is non-trivial in general. In fact, the authors of [12] simply provide a set of syntax-guided conditionally sound heuristics for computing $\partial\mathsf{P}_N$. Unfortunately, when these conditions are violated (we have found many simple programs where they are violated), there are no known algorithmic techniques to generate $\partial\mathsf{P}_N$ in a sound manner. Even if a program $\partial\mathsf{P}_N$ were to be found in an ad-hoc manner, it may be as "complex" as P_N itself. This makes the approach of [12] ineffective for analyzing such programs. As an example, the fourth column of Fig. 2 shows P_{N-1} followed by one possible $\partial\mathsf{P}_N$ that ensures P_N (shown in the first column of the same figure) is semantically equivalent to $\mathsf{P}_{N-1}; \partial\mathsf{P}_N$. Notice that $\partial\mathsf{P}_N$ in this example has two sequentially composed loops, just like P_N had. In addition, the assignment statement in the body of the second loop uses a more complex expression than that present in the corresponding loop of P_N. Proving $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$

[2] Although the authors of [12] mention that it suffices to find a $\partial\mathsf{P}_N$ that satisfies $\{\varphi(N)\}\ \mathsf{P}_{N-1}; \partial\mathsf{P}_N\ \{\psi(N)\}$, they do not discuss any technique that takes $\varphi(N)$ or $\psi(N)$ into account when generating $\partial\mathsf{P}_N$.

may therefore not be any simpler (perhaps even more difficult) than proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$.

In addition to the difficulty of computing $\partial\mathsf{P}_N$, it may be impossible to find a formula $\partial\varphi(N)$ such that $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$, as required by [12]. This can happen even for fairly routine pre-conditions, such as $\varphi(N) \equiv \big(\bigwedge_{i=0}^{N-1} A[i] = N\big)$. Notice that there is no $\partial\varphi(N)$ that satisfies $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$ in this case. In such cases, the technique of [12] cannot be used at all, even if P_N, $\varphi(N)$ and $\psi(N)$ are such that there exists a trivial proof of $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$.

The inductive step proposed in this paper largely mitigates the above problems, thereby making it possible to efficiently reason about a much larger class of programs than that possible using the technique of [12]. Our inductive step proceeds as follows. Given P_N, we first algorithmically construct two programs Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$, such that P_N is semantically equivalent to $\mathsf{Q}_{N-1}; \mathsf{peel}(\mathsf{P}_N)$. Intuitively, Q_{N-1} is the same as P_N, but with all loop bounds that depend on N now modified to depend on $N-1$ instead. Note that this is different from P_{N-1}, which is obtained by replacing *all uses* (not just in loop bounds) of N in P_N by $N-1$. As we will see, this simple difference makes the generation of $\mathsf{peel}(\mathsf{P}_N)$ significantly simpler than generation of $\partial\mathsf{P}_N$, as in [12]. While generating Q_{N-1} and $\mathsf{peel}(P_N)$ may sound similar to generating P_{N-1} and $\partial\mathsf{P}_N$ [12], there are fundamental differences between the two approaches. First, as noted above, P_{N-1} is semantically different from Q_{N-1}. Similarly, $\mathsf{peel}(\mathsf{P}_N)$ is also semantically different from $\partial\mathsf{P}_N$. Second, we provide an algorithm for generating Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ that works for a significantly larger class of programs than that for which the technique of [12] works. Specifically, our algorithm works for all programs amenable to the technique of [12], and also for programs that violate the restrictions imposed by the grammar and conditional heuristics in [12]. For example, we can algorithmically generate Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ even for a class of programs with arbitrarily nested loops – a program feature explicitly disallowed by the grammar in [12]. Third, we guarantee that $\mathsf{peel}(\mathsf{P}_N)$ is "simpler" than P_N in the sense that the maximum nesting depth of loops in $\mathsf{peel}(\mathsf{P}_N)$ is *strictly less* than that in P_N. Thus, if P_N has no nested loops (all programs amenable to analysis by [12] belong to this class), $\mathsf{peel}(\mathsf{P}_N)$ is guaranteed to be loop-free. As demonstrated by the fourth column of Fig. 2, no such guarantees can be given for $\partial\mathsf{P}_N$ generated by the technique of [12]. This is a significant difference, since it greatly simplifies the analysis of $\mathsf{peel}(\mathsf{P}_N)$ vis-a-vis that of $\partial\mathsf{P}_N$.

We had mentioned earlier that some pre-conditions $\varphi(N)$ do not admit any $\partial\varphi(N)$ such that $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$. It is, however, often easy to compute formulas $\varphi'(N-1)$ and $\Delta\varphi'(N)$ in such cases such that $\varphi(N) \Rightarrow \varphi'(N-1) \wedge \Delta\varphi'(N)$, and the variables/array elements in $\Delta\varphi'(N)$ are not modified by either P_{N-1} or Q_{N-1}. For example, if we were to consider a (new) pre-condition $\varphi(N) \equiv \big(\bigwedge_{i=0}^{N-1} A[i] = N\big)$ for the program P_N shown in the first column of Fig. 2, then we have $\varphi'(N-1) \equiv \big(\bigwedge_{i=0}^{N-2} A[i] = N\big)$ and $\Delta\varphi'(N) \equiv \big(A[N-1] = N\big)$. We assume the availability of such a $\varphi'(N-1)$ and $\Delta\varphi'(N)$ for the given $\varphi(N)$. This significantly relaxes the requirement on pre-conditions and allows a much larger class of Hoare triples to be proved using our technique vis-a-vis that of [12].

The third column of Fig. 2 shows Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ generated by our algorithm for the program P_N in the first column of the figure. It is illustrative to compare these with P_{N-1} and $\partial\mathsf{P}_N$ shown in the fourth column of Fig. 2. Notice that Q_{N-1} has the same control flow structure as P_{N-1}, but is not semantically equivalent to P_{N-1}. In fact, Q_{N-1} and P_{N-1} may be viewed as closely related versions of the same program. Let V_{Q} and V_{P} denote the set of variables of Q_{N-1} and P_{N-1} respectively. We assume V_{Q} is disjoint from V_{P}, and analyze the joint execution of Q_{N-1} starting from a state satisfying the pre-condition $\varphi'(N-1)$, and P_{N-1} starting from a state satisfying $\varphi(N-1)$. The purpose of this analysis is to compute a difference predicate $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ that relates corresponding variables in Q_{N-1} and P_{N-1} at the end of their joint execution. The above problem is reminiscent of (yet, different from) translation validation [4,17,24,40,46,48,49], and indeed, our calculation of $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ is motivated by techniques from the translation validation literature. An important finding of our study is that corresponding variables in Q_{N-1} and P_{N-1} are often related by simple expressions on N, regardless of the complexity of P_N, $\varphi(N)$ or $\psi(N)$. Indeed, in all our experiments, we didn't need to go beyond quadratic expressions on N to compute $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$.

Once the steps described above are completed, we have $\Delta\varphi'(N)$, $\mathsf{peel}(\mathsf{P}_N)$ and $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$. It can now be shown that if the inductive hypothesis, i.e. $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds, then proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ reduces to proving $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$, where $\psi'(N-1) \equiv \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$. A few points are worth emphasizing here. First, if $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ is obtained as a set of equalities, the existential quantifier in the formula $\psi'(N-1)$ can often be eliminated simply by substitution. We can also use quantifier elimination capabilities of modern SMT solvers, viz. Z3 [39], to eliminate the quantifier, if needed. Second, recall that unlike $\partial\mathsf{P}_N$ generated by the technique of [12], $\mathsf{peel}(\mathsf{P}_N)$ is guaranteed to be "simpler" than P_N, and is indeed loop-free if P_N has no nested loops. Therefore, proving $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$ is typically significantly simpler than proving $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$. Finally, it may happen that the pre-condition in $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$ is not strong enough to yield a proof of the Hoare triple. In such cases, we need to strengthen the existing pre-condition by a formula, say $\xi'(N-1)$, such that the strengthened pre-condition implies the weakest pre-condition of $\psi(N)$ under $\mathsf{peel}(\mathsf{P}_N)$. Having a simple structure for $\mathsf{peel}(\mathsf{P}_N)$ (e.g., loop-free for the entire class of programs for which [12] works) makes it significantly easier to compute the weakest pre-condition. Note that $\xi'(N-1)$ is defined over the variables in V_{Q}. In order to ensure that the inductive proof goes through, we need to strengthen the post-condition of the original program by $\xi(N)$ such that $\xi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1) \Rightarrow \xi'(N-1)$. Computing $\xi(N-1)$ requires a special form of logical abduction that ensures that $\xi(N-1)$ refers only to variables in V_P. However, if $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ is given as a set of equalities (as is often the case), $\xi(N-1)$ can be computed from $\xi'(N-1)$ simply by substitution. This process of strengthening the pre-condition and post-condition may need to iterate a few times until a fixed point is reached, similar to what

happens in the inductive step of [12]. Note that the fixed point iterations may not always converge (verification is undecidable in general). However, in our experiments, convergence always happened within a few iterations. If $\xi'(N-1)$ denotes the formula obtained on reaching the fixed point, the final Hoare triple to be proved is $\{\xi'(N-1) \wedge \Delta\varphi'(N) \wedge \psi'(N-1)\}$ $\mathsf{peel}(\mathsf{P}_N)$ $\{\xi(N) \wedge \psi(N)\}$, where $\psi'(N-1) \equiv \exists V_{\mathsf{P}} \big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$. Having a simple (often loop-free) $\mathsf{peel}(\mathsf{P}_N)$ significantly simplifies the above process.

We conclude this section by giving an overview of how Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ are computed for the program P_N shown in the first column of Fig. 2. The second column of this figure shows the program obtained from P_N by peeling the last iteration of each loop of the program. Clearly, the programs in the first and second columns are semantically equivalent. Since there are no nested loops in P_N, the peels (shown in solid boxes) in the second column are loop-free program fragments. For each such peel, we identify variables/array elements modified in the peel and used in subsequent non-peeled parts of the program. For example, the variable `x` is modified in the peel of the first loop and used in the body of the second loop, as shown by the arrow in the second column of Fig. 2. We replace all such uses (if needed, transitively) by expressions on the right-hand side of assignments in the peel until no variable/array element modified in the peel is used in any subsequent non-peeled part of the program. Thus, the use of `x` in the body of the second loop is replaced by the expression `x + N * N` in the third column of Fig. 2. The peeled iteration of the first loop can now be moved to the end of the program, since the variables modified in this peel are no longer used in any subsequent non-peeled part of the program. Repeating the above steps for the peeled iteration of the second loop, we get the program shown in the third column of Fig. 2. This effectively gives a transformed program that can be divided into two parts: (i) a program Q_{N-1} that differs from P_N only in that all loops are truncated to iterate $N-1$ (instead of N) times, and (ii) a program $\mathsf{peel}(\mathsf{P}_N)$ that is obtained by concatenating the peels of loops in P_N in the same order in which the loops appeared in P_N. It is not hard to see that P_N, shown in the first column of Fig. 2, is semantically equivalent to $\mathsf{Q}_{N-1}; \mathsf{peel}(\mathsf{P}_N)$. Notice that the construction of Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ was fairly straightforward, and did not require any complex reasoning. In sharp contrast, construction of $\partial\mathsf{P}_N$, as shown in the bottom half of fourth column of Fig. 2, requires non-trivial reasoning, and produces a program with two sequentially composed loops.

3 Preliminaries and Notation

We consider programs generated by the grammar shown below:

$$
\begin{aligned}
\mathsf{PB} &::= \mathsf{St} \\
\mathsf{St} &::= \mathsf{St}\ ;\ \mathsf{St} \mid v := \mathsf{E} \mid A[\mathsf{E}] := \mathsf{E} \mid \textbf{if}(\mathsf{BoolE})\ \textbf{then}\ \mathsf{St}\ \textbf{else}\ \mathsf{St} \mid \\
&\quad\ \ \textbf{for}\ (\ell := 0;\ \ell < \mathsf{UB};\ \ell := \ell{+}1)\ \ \{\mathsf{St}\} \\
\mathsf{E} &::= \mathsf{E}\ \mathsf{op}\ \ \mathsf{E} \mid A[\mathsf{E}] \mid v \mid \ell \mid \mathsf{c} \mid N \\
\mathsf{op} &::= + \mid \mathsf{-} \mid * \mid / \\
\mathsf{UB} &::= \mathsf{UB}\ \mathsf{op}\ \ \mathsf{UB} \mid \ell \mid \mathsf{c} \mid N \\
\mathsf{BoolE} &::= \mathsf{E}\ \mathsf{relop}\ \mathsf{E} \mid \mathsf{BoolE\ AND\ BoolE} \mid \mathsf{NOT\ BoolE} \mid \mathsf{BoolE\ OR\ BoolE}
\end{aligned}
$$

Formally, we consider a program P_N to be a tuple $(\mathcal{V}, \mathcal{L}, \mathcal{A}, \mathsf{PB}, N)$, where $\mathcal{V}$ is a set of scalar variables, $\mathcal{L} \subseteq \mathcal{V}$ is a set of scalar loop counter variables, $\mathcal{A}$ is a set of array variables, PB is the program body, and N is a special symbol denoting a positive integer parameter of the program. In the grammar shown above, we assume that $A \in \mathcal{A}$, $v \in \mathcal{V} \setminus \mathcal{L}$, $\ell \in \mathcal{L}$ and $\mathsf{c} \in \mathbb{Z}$. We also assume that each loop L has a unique loop counter variable ℓ that is initialized at the beginning of L and is incremented by 1 at the end of each iteration. We assume that the assignments in the body of L do not update ℓ. For each loop L with termination condition $\ell < \mathsf{UB}$, we require that UB is an expression in terms of N, variables in $\mathcal{L}$ representing loop counters of loops that nest L, and constants as shown in the grammar. Our grammar allows a large class of programs (with nested loops) to be analyzed using our technique, and that are beyond the reach of state-of-the-art tools like [1,12,42].

We verify Hoare triples of the form $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$, where the formulas $\varphi(N)$ and $\psi(N)$ are either universally quantified formulas of the form $\forall I\ (\alpha(I, N) \Rightarrow \beta(\mathcal{A}, \mathcal{V}, I, N))$ or quantifier-free formulas of the form $\eta(\mathcal{A}, \mathcal{V}, N)$. In these formulas, I is a sequence of array index variables, α is a quantifier-free formula in the theory of arithmetic over integers, and β and η are quantifier-free formulas in the combined theory of arrays and arithmetic over integers.

For technical reasons, we rename all scalar and array variables in the program in a pre-processing step as follows. We rename each scalar variable using the well-known Static Single Assignment (SSA) [43] technique, such that the variable is written at (at most) one location in the program. We also rename arrays in the program such that each loop updates its own version of an array and multiple writes to an array element within the same loop are performed on different versions of that array. We use techniques for array SSA [30] renaming studied earlier in the context of compilers, for this purpose. In the subsequent exposition, we assume that scalar and array variables in the program are already SSA renamed, and that all array and scalar variables referred to in the pre- and post-conditions are also expressed in terms of SSA renamed arrays and scalars.

4 Verification Using Difference Invariants

The key steps in the application of our technique, as discussed in Sect. 2, are

A1: Generation of Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ from a given P_N.
A2: Generation of $\varphi'(N-1)$ and $\Delta\varphi'(N)$ from a given $\varphi(N)$.
A3: Generation of the difference invariant $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$, given $\varphi(N-1)$, $\varphi'(N-1)$, Q_{N-1} and P_{N-1}.
A4: Proving $\{\Delta\varphi'(N)\ \wedge\ \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$, possibly by generation of $\xi'(N-1)$ and $\xi(N)$ to strengthen the pre- and post-conditions, respectively.

We now discuss techniques for solving each of these sub-problems.

4.1 Generating Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$

The procedure illustrated in Fig. 2 (going from the first column to the third column) is fairly straightforward if none of the loops have any nested loops within them. It is easy to extend this to arbitrary sequential compositions of non-nested loops. Having all variables and arrays in SSA-renamed forms makes it particularly easy to carry out the substitution exemplified by the arrow shown in the second column of Fig. 2. Hence, we don't discuss any further the generation of Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ when all loops are non-nested.

The case of nested loops is, however, challenging and requires additional discussion. Before we present an algorithm for handling this case, we discuss the intuition using an abstract example. Consider a pair of nested loops, L_1 and L_2, as shown in Fig. 3. Suppose that `B1` and `B3` are loop-free code fragments in the body of L_1 that precede and succeed the nested loop L_2. Suppose further that the loop body, `B2`, of L_2 is loop-free. To focus on the key aspects of computing peels of nested loops, we make two simplifying assumptions: (i) no scalar variable or array element modified in `B2` is used subsequently (including transitively) in either `B3` or `B1`, and (ii) every scalar variable or array element that is modified in `B1` and used subsequently in `B2`, is not modified again in either `B1`, `B2` or `B3`. Note that these assumptions are made primarily to simplify the exposition. For a detailed discussion on how our technique can be used even with some relaxations of these assumptions, the reader is referred to [13]. The peel of the abstract loops L_1 and L_2 is as shown in Fig. 4. The first loop in the peel includes the last iteration of L_2 in each of the $N-1$ iterations of L_1, that was missed in Q_{N-1}. The subsequent code includes the last iteration of L_1 that was missed in Q_{N-1}.

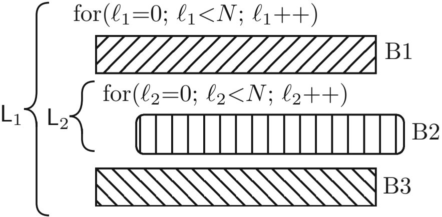


Fig. 3. A generic nested loop

Formally, we use the notation L_1`(N)` to denote a loop L_1 that has no nested loops within it, and its loop counter, say ℓ_1, increases from 0 to an upper bound that is given by an expression in N. Similarly, we use L_1`(N,` L_2`(N))` to denote a loop L_1 that has another loop L_2 nested within it. The loop counter ℓ_1 of L_1 increases from 0 to an upper bound expression in N, while the loop counter ℓ_2 of L_2 increases from 0 to an upper bound expression in ℓ_1 and N. Using this notation, L_1`(N,` L_2`(N,` L_3`(N)))` represents three nested loops, and so on. Notice that the upper bound expression for a nested loop can depend not only on N but also on the loop counters of other loops nesting it. For notational clarity, we also use `LPeel(`L_i`, a, b)` to denote the peel of loop L_i

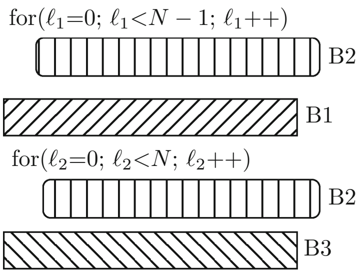


Fig. 4. Peel of the nested loop

consisting of all iterations of L_i where the value of ℓ_i ranges from `a` to `b-1`, both inclusive. Note that if `b-a` is a constant, this corresponds to the concatenation of `(b-a)` peels of L_i.

We will now try to see how we can implement the transformation from the first column to the second column of Fig. 2 for a nested loop L_1`(N,` L_2`(N))`. The first step is to truncate all loops to use $N-1$ instead of N in the upper bound expressions. Using the notation introduced above, this gives the loop L_1`(N-1,` L_2`(N-1))`. Note that all uses of N other than in loop upper bound expressions stay unchanged as we go from L_1`(N,` L_2`(N))` to L_1`(N-1,` L_2`(N-1))`. We now ask: *Which are the loop iterations of* L_1`(N,` L_2`(N))` *that have been missed (or skipped) in going to* L_1`(N-1,` L_2`(N-1))`*?* Let the upper bound expression of L_1 in L_1`(N,` L_2`(N))` be $U_{\mathsf{L}_1}(N)$, and that of L_2 be $U_{\mathsf{L}_2}(\ell_1, N)$. It is not hard to see that in every iteration ℓ_1 of L_1, where $0 \leq \ell_1 < U_{\mathsf{L}_1}(N-1)$, the iterations corresponding to $\ell_2 \in \{U_{\mathsf{L}_2}(\ell_1, N-1), \ldots, U_{\mathsf{L}_2}(\ell_1, N)-1\}$ have been missed. In addition, all iterations of L_1 corresponding to $\ell_1 \in \{U_{\mathsf{L}_1}(N-1), \ldots, U_{\mathsf{L}_1}(N)-1\}$ have also been missed. This implies that the "peel" of L_1`(N,` L_2`(N))` must include all the above missed iterations. This peel therefore is the program fragment shown in Fig. 5.

```
for(ℓ1=0; ℓ1<U_L1(N-1); ℓ1++)
  LPeel(L2, U_L2(ℓ1,N-1), U_L2(ℓ1,N))
LPeel(L1, U_L1(N-1), U_L1(N))
```

Fig. 5. Peel of L_1`(N,` L_2`(N))`

Notice that if U_{L_2} $(\ell_1$ `,N)` `-` U_{L_2} $(\ell_1$ `,N-1)` is a constant (as is the case if $U_{\mathsf{L}_2}(\ell_1$`,N)` is any linear function of ℓ_1 and N), then the peel does not have any loop with nesting depth 2. Hence, the maximum nesting depth of loops in the peel is strictly less than that in L_1`(N,` L_2`(N))`, yielding a peel that is "simpler" than the original program. This argument can be easily generalized to loops with arbitrarily large nesting depths. The peel of L_1`(N,` L_2`(N,` L_3`(N)))` is as shown in Fig. 6.

```
for(ℓ1=0; ℓ1<U_L1(N-1); ℓ1++) {
  for(ℓ2=0; ℓ2<U_L2(ℓ1,N-1); ℓ2++)
    LPeel(L3, U_L3(ℓ1,ℓ2,N-1), U_L3(ℓ1,ℓ2,N))
  LPeel(L2, U_L2(ℓ1,N-1), U_L2(ℓ1,N))
}
LPeel(L1, U_L1(N-1), U_L1(N))
```

Fig. 6. Peel of L_1`(N,` L_2`(N,` L_3`(N)))`

As an illustrative example, let us consider the program in Fig. 7(a), and suppose we wish to compute the peel of this program containing nested loops. In this case, the upper bounds of the loops are $U_{\mathsf{L}_1}(N) = U_{\mathsf{L}_2}(N) = N$. The peel is shown in Fig. 7(b) and consists of two sequentially composed non-nested loops. The first loop takes into account the missed iterations of the inner loop (a single iteration in this example) that are executed in P_N but are missed in Q_{N-1}. The

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    A[i][j] = N;
```

(a)

```
for(i=0; i<N-1; i++)
  A[i][N-1] = N;
for(j=0; j<N; j++)
  A[N-1][j] = N;
```

(b)

Fig. 7. (a) Nested Loop & (b) Peel

Algorithm 1. GenQandPeel(P_N: program)

1: Let sequentially composed loops in P_N be in the order $\mathsf{L}_1, \mathsf{L}_2, \ldots, \mathsf{L}_m$;
2: **for** each loop $\mathsf{L}_i \in$ TopLevelLoops(P_N) **do**
3: $\quad\langle \mathsf{Q}_{\mathsf{L}_i}, \mathsf{R}_{\mathsf{L}_i} \rangle \leftarrow$ GenQandPeelForLoop(L_i);
4: **while** $\exists v.use(v) \in \mathsf{Q}_{\mathsf{L}_i} \wedge def(v) \in \mathsf{R}_{\mathsf{L}_j}$, for some $1 \leq j < i \leq N$ **do** ▷ v is var/array element
5: $\quad$Substitute rhs expression for v from $\mathsf{R}_{\mathsf{L}_j}$ in $\mathsf{Q}_{\mathsf{L}_i}$; ▷ If $\mathsf{R}_{\mathsf{L}_j}$ is a loop, abort
6: $\mathsf{Q}_{N-1} \leftarrow \mathsf{Q}_{\mathsf{L}_1}; \mathsf{Q}_{\mathsf{L}_2}; \ldots; \mathsf{Q}_{\mathsf{L}_m}$;
7: $\mathsf{peel}(\mathsf{P}_N) \leftarrow \mathsf{R}_{\mathsf{L}_1}; \mathsf{R}_{\mathsf{L}_2}; \ldots; \mathsf{R}_{\mathsf{L}_m}$;
8: **return** $\langle \mathsf{Q}_{N-1}, \mathsf{peel}(\mathsf{P}_N) \rangle$;

9: **procedure** GenQandPeelForLoop(L: loop)
10: $\quad$Let $U_{\mathsf{L}}(N)$ be the UB expression of loop L;
11: $\quad\mathsf{Q}_{\mathsf{L}} \leftarrow \mathsf{L}$ with $N-1$ substituted for N in all UB expressions (including for nested loops);
12: $\quad$**if** L has subloops **then**
13: $\quad\quad t \leftarrow$ nesting depth of inner-most nested loop in L;
14: $\quad\quad \mathsf{R}_{t+1} \leftarrow$ empty program with no statements;
15: $\quad\quad$**for** $k = t; k \geq 2; k$-- **do**
16: $\quad\quad\quad$**for** each subloop SL_j in L_i at nesting depth k **do** ▷ Ordered $SL_1, SL_2, \ldots, SL_j$
17: $\quad\quad\quad\quad \mathsf{R}_{SL_j} \leftarrow$ `LPeel`$(SL_j, U_{SL_j}(\ell_1, \ldots, \ell_{k-1}, N-1), U_{SL_j}(\ell_1, \ldots, \ell_{k-1}, N))$;
18: $\quad\quad\quad \mathsf{R}_k \leftarrow$ `for (i=0; i<`$U_{\mathsf{L}_{k-1}}(N-1)$`; i++) {` R_{k+1};R_{SL_1};R_{SL_2};...;R_{SL_j} `}`;
19: $\quad\quad \mathsf{R}_{\mathsf{L}} \leftarrow \mathsf{R}_2;$ `LPeel`$(\mathsf{L}, U_{\mathsf{L}}(N-1), U_{\mathsf{L}}(N))$;
20: $\quad$**else**
21: $\quad\quad \mathsf{R}_{\mathsf{L}} \leftarrow$ `LPeel`$(\mathsf{L}, U_{\mathsf{L}}(N-1), U_{\mathsf{L}}(N))$;
22: $\quad$**return** $\langle \mathsf{Q}_{\mathsf{L}}, \mathsf{R}_{\mathsf{L}} \rangle$;

second loop takes into account the missed iterations of the outer loop in Q_{N-1} compared to P_N.

Generalizing the above intuition, Algorithm 1 presents function GenQand-Peel for computing Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ for a given P_N that has sequentially composed loops with potentially nested loops. Due to the grammar of our programs, our loops are well nested. The method works by traversing over the structure of loops in the program. In this algorithm $\mathsf{Q}_{\mathsf{L}_i}$ and $\mathsf{R}_{\mathsf{L}_i}$ represent the counterparts of Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ for loop L_i. We create the program Q_{N-1} by peeling each loop in the program and then propagating these peels across subsequent loops. We identify the missed iterations of each loop in the program P_N from the upper bound expression UB. Recall that the upper bound of each loop L_k at nesting depth k, denoted by U_{L_k} is in terms of the loop counters ℓ of outer loops and the program parameter N. We need to peel $U_{\mathsf{L}_k}(\ell_1, \ell_2, \ldots, \ell_{k-1}, N) - U_{\mathsf{L}_k}(\ell_1, \ell_2, \ldots, \ell_{k-1}, N-1)$ number of iterations from each loop, where $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_{k-1}$ are counters of the outer nesting loops. As discussed above, whenever this difference is a constant value, we are guaranteed that the loop nesting depth reduces by one. It may so happen that there are multiple sequentially composed loops SL_j at nesting depth k and not just a single loop L_k. At line 2, we iterate over top level loops and call function GenQandPeelForLoop(L_i) for each sequentially composed loop L_i in P_N. At line 11 we construct Q_{L} for loop L. If the loop L has no nested loops, then the peel is the last iterations computed using the upper bound in line 21 For nested loops, the loop at line 15 builds the peel for all loops inside L following the above intuition. The peels of all sub-loops are collected and inserted in the peel of L at line 19. Since all the peeled iterations are moved after Q_{L} of each loop, we

need to repair expressions appearing in Q_{L}. The repairs are applied by the loop at line 4. In the repair step, we identify the right hand side expressions for all the variables and array elements assigned in the peeled iterations. Subsequently, the uses of the variables and arrays in $\mathsf{Q}_{\mathsf{L}_i}$ that are assigned in $\mathsf{R}_{\mathsf{L}_j}$ are replaced with the assigned expressions whenever $j < i$. If $\mathsf{R}_{\mathsf{L}_j}$ is a loop, this step is more involved and hence currently not considered. Finally at line 8, the peels and Qs of all top level loops are stitched and returned.

Note that lines 4 and 5 of Algorithm 1 implement the substitution represented by the arrow in the second column of Fig. 2. This is necessary in order to move the peel of a loop to the end of the program. If either of the loops L_i or L_j use array elements as index to other arrays then it can be difficult to identify what expression to use in $\mathsf{Q}_{\mathsf{L}_i}$ for the substitution. However, such scenarios are observed less often, and hence, they hardly impact the effectiveness of the technique on programs seen in practice. The peel $\mathsf{R}_{\mathsf{L}_j}$, from which the expression to be substituted in $\mathsf{Q}_{\mathsf{L}_i}$ has to be taken, itself may have a loop. In such cases, it can be significantly more challenging to identify what expression to use in $\mathsf{Q}_{\mathsf{L}_i}$. We use several optimizations to transform the peeled loop before trying to identify such an expression. If the modified values in the peel can be summarized as closed form expressions, then we can replace the loop in the peel with its summary. For example, consider the peeled loop, `for (` ℓ_1 `=0;` $\ell_1 <$ `N;` ℓ_1 `++) { S = S + 1; }`. This loop is summarized as `S = S + N;` before it can be moved across subsequent code. If the variables modified in the peel of a nested loop are not used later, then the peel can be trivially moved. In many cases, the loop in the peel can also be substituted with its conservative over-approximation. We have implemented some of these optimizations in our tool and are able to verify several benchmarks with sequentially composed nested loops. It may not always be possible to move the peel of a nested loop across subsequent loops but we have observed that these optimizations suffice for many programs seen in practice.

Theorem 1. *Let* Q_{N-1} *and* $\mathsf{peel}(\mathsf{P}_N)$ *be generated by application of function* GENQANDPEEL *from Algorithm 1 on program* P_N. *Then* P_N *is semantically equivalent to* $\mathsf{Q}_{N-1}; \mathsf{peel}(\mathsf{P}_N)$.

Lemma 1. *Suppose the following conditions hold;*

- *Program* P_N *satisfies our syntactic restrictions (see Sect. 3).*
- *The upper bound expressions of all loops are linear expressions in* N *and in the loop counters of outer nesting loops.*

Then, the max nesting depth of loops in $\mathsf{peel}(\mathsf{P}_N)$ *is strictly less than that in* P_N.

Proof. Let $U_{\mathsf{L}_k}(\ell_1, \ldots, \ell_{k-1}, N)$ be the upper bound expression of a loop L_k at nesting depth k. Suppose $U_{\mathsf{L}_k} = c_1.\ell_1 + \cdots c_{k-1}.\ell_{k-1} + C.N + D$, where $c_1, \ldots c_{k-1}, C$ and D are constants. Then $U_{\mathsf{L}_k}(\ell_1, \ldots, \ell_{k-1}, N) - U_{\mathsf{L}_k}(\ell_1, \ldots \ell_{k-1}, N-1) = C$, i.e. a constant. Now, recalling the discussion in Sect. 4.1, we see that `LPeel(`L_k`,` U_k`(`$\ell_1,\ldots,\ell_{k-1}$`,N-1),` U_k`(`$\ell_1,\ldots,\ell_{k-1}$`,N))` simply results in concatenating a constant number of peels of the loop L_k. Hence,

the maximum nesting depth of loops in $\texttt{LPeel(}\ \mathsf{L}_k,\ U_k(\ell_1,\ldots,\ell_{k-1},\texttt{N}-\texttt{1}),\ U_k(\ell_1,\ldots,\ell_{k-1},\texttt{N}))$ is strictly less than the maximum nesting depth of loops in L_k.

Suppose loop L with nested loops (having maximum nesting depth t) is passed as the argument of function GenQandPeelForLoop (see Algorithm 1). In line 15 of function GenQandPeelForLoop, we iterate over all loops at nesting depth 2 and above within L. Let L_k be a loop at nesting depth k, where $2 \leq k \leq t$. Clearly, L_k can have at most $t-k$ nested levels of loops within it. Therefore, when LPeel is invoked on such a loop, the maximum nesting depth of loops in the peel generated for L_k can be at most $t-k-1$. From lines 18 and 19 of function GenQandPeelForLoop, we also know that this LPeel can itself appear at nesting depth k of the overall peel R_L. Hence, the maximum nesting depth of loops in R_L can be $t-k-1+k$, i.e. $t-1$. This is strictly less than the maximum nesting depth of loops in L. □

Corollary 1. *If* P_N *has no nested loops, then* $\mathsf{peel}(\mathsf{P}_N)$ *is loop-free.*

4.2 Generating $\varphi'(N-1)$ and $\Delta\varphi'(N)$

Given $\varphi(N)$, we check if it is of the form $\bigwedge_{i=0}^{N-1} \rho_i$, where ρ_i is a formula on the ith elements of one or more arrays, and scalars used in P_N. If so, we infer $\varphi'(N-1)$ to be $\bigwedge_{i=0}^{N-2} \rho_i$ and $\Delta\varphi'(N)$ to be ρ_{N-1} (assuming variables/array elements in ρ_{N-1} are not modified by Q_{N-1}). Note that all uses of N in ρ_i are retained as is (i.e. not changed to $N-1$) in $\varphi'(N-1)$. In general, when deriving $\varphi'(N-1)$, we do not replace any use of N in $\varphi(N)$ by $N-1$ unless it is the limit of an iterated conjunct as discussed above. Specifically, if $\varphi(N)$ doesn't contain an iterated conjunct as above, then we consider $\varphi'(N-1)$ to be the same as $\varphi(N)$ and $\Delta\varphi'(N)$ to be True. Thus, our generation of $\varphi'(N-1)$ and $\Delta\varphi'(N)$ differs from that of [12]. As discussed earlier, this makes it possible to reason about a much larger class of pre-conditions than that admissible by the technique of [12].

4.3 Inferring Inductive Difference Invariants

Once we have P_{N-1}, Q_{N-1}, $\varphi(N-1)$ and $\varphi'(N-1)$, we infer *difference invariants*. We construct the standard cross-product of programs Q_{N-1} and P_{N-1}, denoted as $\mathsf{Q}_{N-1} \times \mathsf{P}_{N-1}$, and infer difference invariants at key control points. Note that P_{N-1} and Q_{N-1} are guaranteed to have synchronized iterations of corresponding loops (both are obtained by restricting the upper bounds of all loops to use $N-1$ instead of N). However, the conditional statements within the loop body may not be synchronized. Thus, whenever we can infer that the corresponding conditions are equivalent, we synchronize the branches of the conditional statement. Otherwise, we consider all four possibilities of the branch conditions. It can be seen that the net effect of the cross-product is executing the programs P_{N-1} and Q_{N-1} one after the other.

We run a dataflow analysis pass over the constructed product graph to infer difference invariants at loop head, loop exit and at each branch condition. The only dataflow values of interest are differences between corresponding variables in Q_{N-1} and P_{N-1}. Indeed, since structure and variables of Q_{N-1} and P_{N-1} are similar, we can create the correspondence map between the variables. We start the difference invariant generation by considering relations between corresponding variables/array elements appearing in pre-conditions of the two programs. We apply static analysis that can track equality expressions (including disjunctions over equality expressions) over variables as we traverse the program. These equality expressions are our difference invariants.

We observed in our experiments the most of the inferred equality expressions are simple expressions of N (atmost quadratic in N). This not totally surprising and similar observations have also been independently made in [4,15,24]. Note that the difference invariants may not always be equalities. We can easily extend our analysis to learn inequalities using interval domains in static analysis. We can also use a library of expressions to infer difference invariants using a guess-and-check framework. Moreover, guessing difference invariants can be easy as in many cases the difference expressions may be independent of the program constructs, for example, the equality expression $\mathtt{v} = \mathtt{v}'$ where $\mathtt{v} \in \mathsf{P}_{N-1}$ and $\mathtt{v}' \in \mathsf{Q}_{N-1}$ does not depend on any other variable from the two programs.

For the example in Fig. 2, the difference invariant at the head of the first loop of $Q_{N-1} \times P_{N-1}$ is $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1) \equiv (\mathtt{x}' - \mathtt{x} = \mathtt{i} \times (2 \times \mathtt{N} - 1) \wedge \forall \mathtt{i} \in [0, \mathtt{N}-1),\ \mathtt{a}'[\mathtt{i}] - \mathtt{a}[\mathtt{i}] = 1)$, where $\mathtt{x}, \mathtt{a} \in V_{\mathsf{P}}$ and $\mathtt{x}', \mathtt{a}' \in V_{\mathsf{Q}}$. Given this, we easily get $\mathtt{x}' - \mathtt{x} = (\mathtt{N}-1) \times (2 \times \mathtt{N} - 1)$ when the first loop terminates. For the second loop, $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1) \equiv (\forall \mathtt{j} \in [0, \mathtt{N}-1),\ \mathtt{b}'[\mathtt{j}] - \mathtt{b}[\mathtt{j}] = (\mathtt{x}' - \mathtt{x}) + \mathtt{N}^2 = (\mathtt{N}-1) \times\ (2 \times \mathtt{N} - 1) + \mathtt{N}^2)$.

Note that the difference invariants and its computation are agnostic of the given post-condition. Hence, our technique does not need to re-run this analysis for proving a different post-condition for the same program.

4.4 Verification Using Inductive Difference Invariants

We present our method DIFFY for verification of programs using inductive difference invariants in Algorithm 2. It takes a Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ as input, where $\varphi(N)$ and $\psi(N)$ are pre- and post-condition formulas. We check the base in line 1 to verify the Hoare triple for $N = 1$. If this check fails, we report a counterexample. Subsequently, we compute Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ as described in Sect. 4.1 using the function GENQANDPEEL from Algorithm 1. At line 4, we compute the formulas $\varphi'(N-1)$ and $\Delta\varphi'(N)$ as described in Sect. 4.2. For automation, we analyze the quantifiers appearing in $\varphi(N)$ and modify the quantifier ranges such that the conditions in Sect. 4.2 hold. We infer difference invariants $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ on line 5 using the method described in Sect. 4.3, wherein V_{Q} and V_{P} are sets of variables from Q_{N-1} and P_{N-1} respectively. At line 6, we compute $\psi'(N-1)$ by eliminating variables V_{P} from P_{N-1} from $\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$. At line 7, we check the inductive step of our analysis. If the inductive step succeeds, then we conclude that the assertion holds.

Algorithm 2. DIFFY($\{\varphi(N)\}$ P_N $\{\psi(N)\}$)

```
1: if {φ(1)} P_1 {ψ(1)} fails then                                    ▷ Base case for N=1
2:     return "Counterexample found!";

3: ⟨Q_{N−1}, peel(P_N)⟩ ← GenQAndPeel(P_N);
4: ⟨φ'(N − 1), Δφ'(N)⟩ ← FormulaDiff(φ(N));                   ▷ φ(N) ⇒ φ'(N − 1) ∧ Δφ'(N)
5: D(V_Q, V_P, N − 1) ← InferDiffInvs(Q_{N−1}, P_{N−1}, φ'(N − 1), φ(N − 1));
6: ψ'(N − 1) ← QE(V_P, ψ(N − 1) ∧ D(V_Q, V_P, N − 1));
7: if {ψ'(N − 1) ∧ Δφ'(N)} peel(P_N) {ψ(N)} then
8:     return True;                                                ▷ Verification Successful
9: else
10:     return Strengthen(P_N, peel(P_N), φ(N), ψ(N), ψ'(N − 1), Δφ'(N), D(V_Q, V_P, N));

11: procedure Strengthen(P_N, peel(P_N), φ(N), ψ(N), ψ'(N − 1), Δφ'(N), D(V_Q, V_P, N))
12:     χ(N) ← ψ(N);
13:     ξ(N) ← True;
14:     ξ'(N − 1) ← True;
15:     repeat
16:         χ'(N − 1) ← WP(χ(N), peel(P_N));                  ▷ Dijkstra's WP for loop free code
17:         if χ'(N − 1) = ∅ then
18:             if peel(P_N) has a loop then
19:                 return Diffy({ξ'(N − 1) ∧ Δφ'(N) ∧ ψ'(N − 1)} peel(P_N) {ξ(N) ∧ ψ(N)});
20:             else
21:                 return False;                                     ▷ Unable to prove
22:         χ(N) ← QE(V_Q, χ'(N) ∧ D(V_Q, V_P, N));
23:         ξ(N) ← ξ(N) ∧ χ(N);
24:         ξ'(N − 1) ← ξ'(N − 1) ∧ χ'(N − 1);
25:         if {φ(1)} P_1 {ξ(1)} fails then
26:             return False;                                         ▷ Unable to prove
27:         if {ξ'(N − 1) ∧ Δφ'(N) ∧ ψ'(N − 1)} peel(P_N) {ξ(N) ∧ ψ(N)} holds then
28:             return True;                                    ▷ Verification Successful
29:     until timeout;
30:     return False;
```

If that is not the case then, we try to iteratively strengthen both the pre- and post-condition of $\mathsf{peel}(\mathsf{P}_N)$ simultaneously by invoking STRENGTHEN.

The function STRENGTHEN first initializes the formula $\chi(N)$ with $\psi(N)$ and the formulas $\xi(N)$ and $\xi'(N-1)$ to True. To strengthen the pre-condition of $\mathsf{peel}(\mathsf{P}_N)$, we infer a formula $\chi'(N-1)$ using Dijkstra's weakest pre-condition computation of $\chi(N)$ over the $\mathsf{peel}(\mathsf{P}_N)$ in line 16. It may happen that we are unable to infer such a formula. In such a case, if the program $\mathsf{peel}(\mathsf{P}_N)$ has loops then we recursively invoke DIFFY at line 19 to further simplify the program. Otherwise, we abandon the verification effort (line 21). We use quantifier elimination to infer $\chi(N-1)$ from $\chi'(N-1)$ and $D(V_\mathsf{Q}, V_\mathsf{P}, N-1))$ at line 6.

The inferred pre-conditions $\chi(N)$ and $\chi'(N-1)$ are accumulated in $\xi(N)$ and $\xi'(N-1)$, which strengthen the post-conditions of P_N and Q_{N-1} respectively in lines 23–24. We again check the base case for the inferred formulas in $\xi(N)$ at line 25. If the check fails we abandon the verification attempt at line 26. If the base case succeeds, we then proceed to the inductive step. When the inductive step succeeds, we conclude that the assertion is verified. Otherwise, we continue in the loop and try to infer more pre-conditions untill we run out of time.

The pre-condition in Fig. 2 is $\phi(N) \equiv \mathsf{True}$ and the post-condition is $\psi(N) \equiv \forall \mathtt{j} \in [\mathtt{0}, \mathtt{N}),\ \mathtt{b[j]} = \mathtt{j} + \mathtt{N}^3)$. At line 4, $\phi'(N-1)$ and $\Delta\phi'(N-1)$ are computed to be True. $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$ is the formula computed in Sect. 4.3. At line 6,

Table 1. Summary of the experimental results. S is successful result. U is inconclusive result. TO is timeout.

Program Category		Diffy			Vajra		VeriAbs		VIAP		
		S	U	TO	S	U	S	TO	S	U	TO
Safe C1	110	110	0	0	110	0	96	14	16	1	93
Safe C2	24	21	0	3	0	24	5	19	4	0	20
Safe C3	23	20	3	0	0	23	9	14	0	23	0
Total	157	151	3	3	110	47	110	47	20	24	113
Unsafe C1	99	98	1	0	98	1	84	15	98	0	1
Unsafe C2	24	24	0	0	17	7	19	5	22	0	2
Unsafe C3	23	20	3	0	0	23	22	1	0	23	0
Total	146	142	4	0	115	31	125	21	120	23	3

$\psi'(N-1) \equiv (\forall \mathtt{j} \in [\mathtt{0}, \mathtt{N}-\mathtt{1}),\ \mathtt{b}'[\mathtt{j}] = \mathtt{j} + (\mathtt{N}-\mathtt{1})^3 + (\mathtt{N}-\mathtt{1}) \times (\mathtt{2} \times \mathtt{N} - \mathtt{1}) + \mathtt{N}^2 = \mathtt{j} + \mathtt{N}^3)$. The algortihm then invokes Strengthen at line 10 which infers the formulas $\chi'(N-1) \equiv (\mathtt{x}' = (\mathtt{N}-\mathtt{1})^3)$ at line 16 and $\chi(N) \equiv (\mathtt{x} = \mathtt{N}^3)$ at line 22. These are accumulated in $\xi'(N-1)$ and $\xi(N)$, simultaneosuly strengthening the pre- and post-condition. Verification succeeds after this strengthening iteration.

The following theorem guarantees the soundness of our technique.

Theorem 2. *Suppose there exist formulas $\xi'(N)$ and $\xi(N)$ and an integer $M > 0$ such that the following hold*

- $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N) \wedge \xi(N)\}$ *holds for* $1 \leq N \leq M$, *for some* $M > 0$.
- $\xi(N) \wedge D(V_\mathsf{Q}, V_\mathsf{P}, N) \Rightarrow \xi'(N)$ *for all* $N > 0$.
- $\{\xi'(N-1) \wedge \Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\xi(N) \wedge \psi(N)\}$ *holds for all* $N \geq M$, *where* $\psi'(N-1) \equiv \exists V_\mathsf{P}\big(\psi(N-1) \wedge D(V_\mathsf{Q}, V_\mathsf{P}, N-1)\big)$.

Then $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ *holds for all* $N > 0$.

5 Experimental Evaluation

We have instantiated our technique in a prototype tool called Diffy. It is written in `C++` and is built using the LLVM(v6.0.0) [31] compiler. We use the SMT solver Z3(v4.8.7) [39] for proving Hoare triples of loop-free programs. Diffy and the supporting data to replicate the experiments are openly available at [14].

Setup. All experiments were performed on a machine with Intel i7-6500U CPU, 16 GB RAM, running at 2.5 GHz, and Ubuntu 18.04.5 LTS operating system. We have compared the results obtained from Diffy with Vajra(v1.0) [12], VIAP(v1.1) [42] and VeriAbs(v1.4.1-12) [1]. We choose Vajra which also employs inductive reasoning for proving array programs and verify the benchmarks in its test-suite. We compared with VeriAbs as it is the winner of the arrays sub-category in SV-COMP 2020 [6] and 2021 [7]. VeriAbs applies a

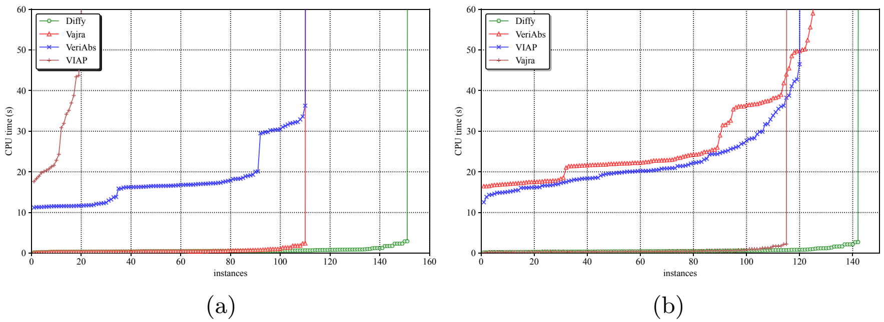


Fig. 8. Cactus Plots (a) All Safe Benchmarks (b) All Unsafe Benchmarks

sequence of techniques from its portfolio to verify array programs. We compared with VIAP which was the winner in arrays sub-category in SV-COMP 2019 [5]. VIAP also employs a sequence of tactics, implemented for proving a variety of array programs. Diffy does not use multiple techniques, however we choose to compare it with these portfolio verifiers to show that it performs well on a class of programs and can be a part of their portfolio. All tools take C programs in the SV-COMP format as input. Timeout of 60 s was set for each tool. A summary of the results is presented in Table 1.

Benchmarks. We have evaluated Diffy on a set of 303 array benchmarks, comprising of the entire test-suite of [12], enhanced with challenging benchmarks to test the efficacy of our approach. These benchmarks take a symbolic parameter N which specifies the size of each array. Assertions are (in-)equalities over array elements, scalars and (non-)linear polynomial terms over N. We have divided both the safe and unsafe benchmarks in three categories. Benchmarks in C1 category have standard array operations such as min, max, init, copy, compare as well as benchmarks that compute polynomials. In these benchmarks, branch conditions are not affected by the value of N, operations such as modulo and nested loops are not present. There are 110 safe and 99 unsafe programs in the C1 category in our test-suite. In C2 category, the branch conditions are affected by change in the program parameter N and operations such as modulo are used in these benchmarks. These benchmarks do not have nested loops in them. There are 24 safe and unsafe benchmarks in the C2 category. Benchmarks in category C3 are programs with atleast one nested loop in them. There are 23 safe and unsafe programs in category C3 in our test-suite. The test-suite has a total of 157 safe and 146 unsafe programs.

Analysis. Diffy verified 151 safe benchmarks, compared to 110 verified by Vajra as well as VeriAbs and 20 verified by VIAP. Diffy was unable to verify 6 safe benchmarks. In 3 cases, the smt solver timed out while trying to prove the induction step since the formulated query had a modulus operation and in 3 cases it was unable to compute the predicates needed to prove the assertions. Vajra was unable to verify 47 programs from categories C2 and

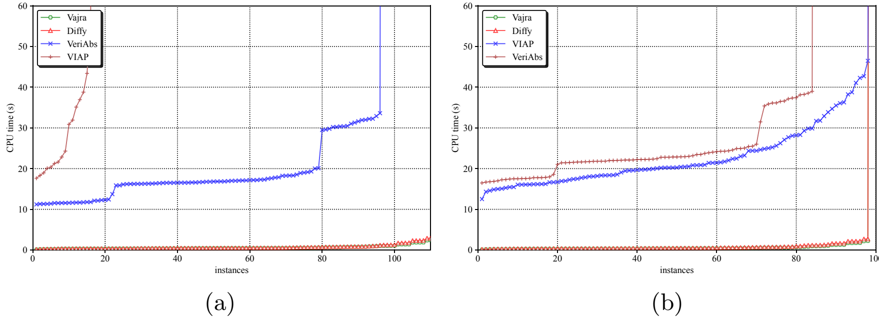


Fig. 9. Cactus plots (a) Safe C1 benchmarks (b) Unsafe C1 benchmarks

C3. These are programs with nested loops, branch conditions affected by N, and cases where it could not compute the difference program. The sequence of techniques employed by VeriAbs, ran out of time on 47 programs while trying to prove the given assertion. VeriAbs proved 2 benchmarks in category C2 and 3 benchmarks in category C3 where Diffy was inconclusive or timed out. VeriAbs spends considerable amount of time on different techniques in its portfolio before it resorts to Vajra and hence it could not verify 14 programs that Vajra was able to prove efficiently. VIAP was inconclusive on 24 programs which had nested loops or constructs that could not be handled by the tool. It ran out of time on 113 benchmarks as the initial tactics in its sequence took up the allotted time but could not verify the benchmarks. Diffy was able to verify all programs that VIAP and Vajra were able to verify within the specified time limit.

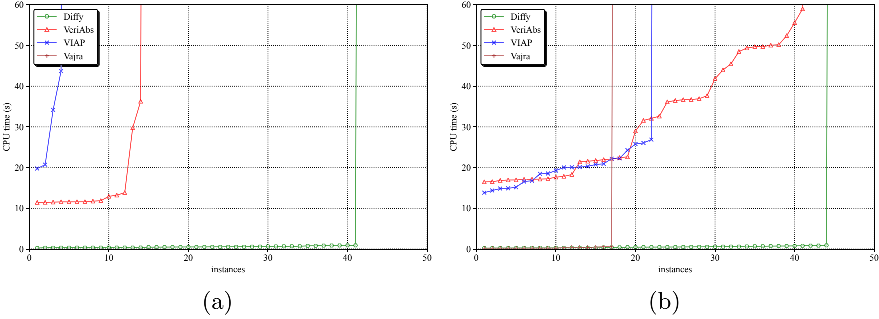
The cactus plot in Fig. 8(a) shows the performance of each tool on all safe benchmarks. Diffy was able to prove most of the programs within three seconds. The cactus plot in Fig. 9(a) shows the performance of each tool on safe benchmarks in C1 category. Vajra and Diffy perform equally well in the C1 category. This is due to the fact that both tools perform efficient inductive reasoning. Diffy outperforms VeriAbs and VIAP in this category. The cactus plot in Fig. 10(a) shows the performance of each tool on safe benchmarks in the combined categories C2 and C3, that are difficult for Vajra as most of these programs are not within its scope. Diffy out performs all other tools in categories C2 and C3. VeriAbs was an order of magnitude slower on programs it was able to verify, as compared to Diffy. VeriAbs spends significant amount of time in trying techniques from its portfolio, including Vajra, before one of them succeeds in verifying the assertion or takes up the entire time allotted to it. VIAP took 70 seconds more on an average as compared to Diffy to verify the given benchmark. VIAP also spends a large portion of time in trying different tactics implemented in the tool and solving the recurrence relations in programs.

Our technique reports property violations when the base case of the analysis fails for small fixed values of N. While the focus of our work is on proving assertions, we report results on unsafe versions of the safe benchmarks from our

test-suite. Diffy was able to detect a property violation in 142 unsafe programs and was inconclusive on 4 benchmarks. Vajra detected violations in 115 programs and was inconclusive on 31 programs. VeriAbs reported 125 programs as unsafe and ran out of time on 21 programs. VIAP reported property violation in 120 programs, was inconclusive on 23 programs and timed out on 3 programs.

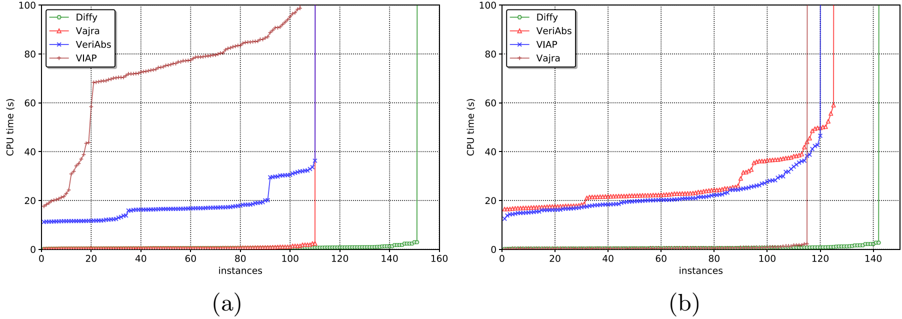
The cactus plot in Fig. 8(b) shows the performance of each tool on all unsafe benchmarks. Diffy was able to detect a violation faster than all other tools and on more benchmarks from the test-suite. Figure 9(b) and Fig. 10(b) give a finer glimpse of the performance of these tools on the categories that we have defined. In the C1 category, Diffy and Vajra have comparable performance and Diffy disproves the same number of benchmarks as Vajra and VIAP. In C2 and C3 categories, we are able to detect property violations in more benchmarks than other tools in less time.

To observe any changes in the performance of these, we also ran them with an increased time out of 100 seconds (Fig. 11). Performance remains unchanged for Diffy, Vajra and VeriAbs on both safe and unsafe benchmarks, and of VIAP on unsafe benchmarks. VIAP was able to additionally verify 89 safe programs in categories C1 and C2 with the increased time limit.



(a) (b)

Fig. 10. Cactus plots (a) Safe C2 & C3 benchmarks (b) Unsafe C2 & C3 benchmarks



(a) (b)

Fig. 11. Cactus plots. TO = 100 s. (a) Safe benchmarks (b) Unsafe benchmarks

6 Related Work

Techniques Based on Induction. Our work is related to several efforts that apply inductive reasoning to verify properties of array programs. Our work subsumes the full-program induction technique in [12] that works by inducting on the entire program via a program parameter N. We propose a principled method for computation and use of difference invariants, instead of computing difference programs which is more challenging. An approach to construct safety proofs by automatically synthesizing squeezing functions that shrink program traces is proposed in [27]. Such functions are not easy to synthesize, whereas difference invariants are relatively easy to infer. In [11], the post-condition is inductively established by identifying a tiling relation between the loop counter and array indices used in the program. Our technique can verify programs from [11], when supplied with the *tiling* relation. [44] identifies recurrent program fragments for induction using the loop counter. They require restrictive data dependencies, called *commutativity of statements*, to move peeled iterations across subsequent loops. Unfortunately, these restrictions are not satisfied by a large class of programs in practice, where our technique succeeds.

Difference Computation. Computing differences of program expressions has been studied for incremental computation of expensive expressions [35,41], optimizing programs with arrays [34], and checking data-structure invariants [45]. These differences are not always well suited for verifying properties, in contrast with the difference invariants which enable inductive reasoning in our case.

Logic Based Reasoning. In [21], trace logic that implicitly captures inductive loop invariants is described. They use theorem provers to introduce and prove lemmas at arbitrary control locations in the program. Unlike their technique, we focus primarily on universally quantified and quantifier-free properties, although a restricted class of existentially quantified properties can be handled by our technique (see [13] for more details). VIAP [42] translates the program to an quantified first-order logic formula using the scheme proposed in [32]. It uses a portfolio of tactics to simplify and prove the generated formulas. Dedicated solvers for recurrences are used whereas our technique adapts induction for handling recurrences.

Invariant Generation. Several techniques generate invariants for array programs. QUIC3 [25], FreqHorn [9,19] infer universally quantified invariants over arrays for Constrained Horn Clauses (CHCs). Template-based techniques [8,23,47] search for inductive quantified invariants by instantiating parameters of a fixed set of templates. We generate relational invariants, which are often easier to infer compared to inductive quantified invariants for each loop.

Abstraction-Based Techniques. Counterexample-guided abstraction refinement using prophecy variables for programs with arrays is proposed in [36]. VERIABS [1] uses a portfolio of techniques, specifically to identify loops that can be soundly abstracted by a bounded number of iterations. VAPHOR [38] transforms array programs to array-free Horn formulas to track bounded number of array cells. BOOSTER [3] combines lazy abstraction based interpolation [2] and

acceleration [10,28] for array programs. Abstractions in [16,18,22,26,29,33,37] implicitly or explicitly partition the range array indices to infer and prove facts on array segments. In contrast, our method does not rely on abstractions.

7 Conclusion

We presented a novel verification technique that combines generation of difference invariants and inductive reasoning. Difference invariants relate corresponding variables and arrays from the two versions of a program and are often easy to infer and prove. We have instantiated these techniques in our prototype Diffy. Experiments shows that Diffy out-performs the tools that won the Arrays sub-category in SV-COMP 2019, 2020 and 2021. Although we have focused on universal and quantifier-free properties in this paper, the technique applies to some classes of existential properties as well. The interested reader is referred to [13] for more details. Investigations in using synthesis techniques for automatic generation of difference invariants to verify properties of array manipulating programs is a part of future work.

References

1. Afzal, M., et al.: Veriabs: verification by abstraction and test generation (competition contribution). In: TACAS 2020. LNCS, vol. 12079, pp. 383–387. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_25
2. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy abstraction with interpolants for arrays. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 46–61. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_7
3. Alberti, F., Ghilardi, S., Sharygina, N.: Booster: an acceleration-based verification framework for array programs. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 18–23. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_2
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
5. Beyer, D.: Competition on software verification (SV-COMP) (2019). http://sv-comp.sosy-lab.org/2019/
6. Beyer, D.: Competition on software verification (SV-COMP) (2020). http://sv-comp.sosy-lab.org/2020/
7. Beyer, D.: Competition on software verification (SV-COMP) (2021). http://sv-comp.sosy-lab.org/2021/
8. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69738-1_27
9. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_8

10. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_23
11. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs by tiling. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 428–449. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_21
12. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs with full-program induction. In: TACAS 2020. LNCS, vol. 12078, pp. 22–39. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_2
13. Chakraborty, S., Gupta, A., Unadkat, D.: Diffy: inductive reasoning of array programs using difference invariants (2021). https://arxiv.org/abs/2105.14748
14. Chakraborty, S., Gupta, A., Unadkat, D.: Diffy: inductive reasoning of array programs using difference invariants, April 2021. https://doi.org/10.6084/m9.figshare.14509467
15. Churchill, B., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: Proceedings of PLDI, pp. 1027–1040 (2019)
16. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of POPL, pp. 105–118 (2011)
17. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Chang, B.-Y.E. (ed.) APLAS 2017. LNCS, vol. 10695, pp. 127–147. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71237-6_7
18. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: beyond strong vs. weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_14
19. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 259–277. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_14
20. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proceedings of POPL, pp. 191–202 (2002)
21. Georgiou, P., Gleiss, B., Kovács, L.: Trace logic for inductive loop reasoning. In: Proceedings of FMCAD, pp. 255–263 (2020)
22. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: Proceedings of POPL, pp. 338–350 (2005)
23. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Proceedings of POPL, pp. 235–246 (2008)
24. Gupta, S., Rose, A., Bansal, S.: Counterexample-guided correlation algorithm for translation validation. Proc. OOPSLA **4**, 1–29 (2020)
25. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 248–266. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_15
26. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Proceedings of PLDI, pp. 339–348 (2008)
27. Ish-Shalom, O., Itzhaky, S., Rinetzky, N., Shoham, S.: Putting the squeeze on array programs: loop verification via inductive rank reduction. In: Proceedings of VMCAI, pp. 112–135 (2020)
28. Jeannet, B., Schrammel, P., Sankaranarayanan, S.: Abstract acceleration of general linear loops. In: Proceedings of POPL, pp. 529–540 (2014)

29. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_23
30. Knobe, K., Sarkar, V.: Array ssa form and its use in parallelization. In: Proceedings of POPL, pp. 107–120 (1998)
31. Lattner, C.: LLVM and clang: next generation compiler technology. In: The BSD Conference, pp. 1–2 (2008)
32. Lin, F.: A formalization of programs in first-order logic with a discrete linear order. Artif. Intell. **235**, 1–25 (2016)
33. Liu, J., Rival, X.: Abstraction of arrays based on non contiguous partitions. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 282–299. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_16
34. Liu, Y.A., Stoller, S.D., Li, N., Rothamel, T.: Optimizing aggregate array computations in loops. TOPLAS **27**(1), 91–125 (2005)
35. Liu, Y.A., Stoller, S.D., Teitelbaum, T.: Static caching for incremental computation. TOPLAS **20**(3), 546–585 (1998)
36. Mann, M., Irfan, A., Griggio, A., Padon, O., Barrett, C.: Counterexample-guided prophecy for model checking modulo the theory of arrays. In: TACAS 2021. LNCS, vol. 12651, pp. 113–132. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_7
37. Monniaux, D., Alberti, F.: A simple abstraction of arrays and maps by program translation. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 217–234. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9_13
38. Monniaux, D., Gonnord, L.: Cell morphing: from array programs to array-free horn clauses. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 361–382. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_18
39. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
40. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of PLDI, pp. 83–94 (2000)
41. Paige, R., Koenig, S.: Finite differencing of computable expressions. TOPLAS **4**(3), 402–454 (1982)
42. Rajkhowa, P., Lin, F.: Extending VIAP to handle array programs. In: Piskac, R., Rümmer, P. (eds.) VSTTE 2018. LNCS, vol. 11294, pp. 38–49. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_3
43. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of POPL, pp. 12–27 (1988)
44. Seghir, M.N., Brain, M.: Simplifying the verification of quantified array assertions via code transformation. In: Albert, E. (ed.) LOPSTR 2012. LNCS, vol. 7844, pp. 194–212. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38197-3_13
45. Shankar, A., Bodik, R.: Ditto: automatic incrementalization of data structure invariant checks (in Java). ACM SIGPLAN Not. **42**(6), 310–319 (2007)
46. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Data-driven equivalence checking. In: Proceedings of OOPSLA, pp. 391–406 (2013)
47. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. ACM SIGPLAN Not. **44**(6), 223–234 (2009)

48. Zaks, A., Pnueli, A.: CoVaC: compiler validation by program analysis of the cross-product. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 35–51. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_5
49. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B.: VOC: a translation validator for optimizing compilers. ENTCS **65**(2), 2–18 (2002)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.