Automatic Program Instrumentation for Automatic Verification

Jesper Amilon[1], Zafer Esen[2], Dilian Gurov[1], Christian Lidström[1], and Philipp Rümmer[2,3]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
[2] Uppsala University, Sweden
[3] University of Regensburg, Germany

Abstract. In deductive verification and software model checking, dealing with certain specification language constructs can be problematic when the back-end solver is not sufficiently powerful or lacks the required theories. One way to deal with this is to transform, for verification purposes, the program to an equivalent one not using the problematic constructs, and to reason about its correctness instead. In this paper, we propose instrumentation as a unifying verification paradigm that subsumes various existing ad-hoc approaches, has a clear formal correctness criterion, can be applied automatically, and can transfer back witnesses and counterexamples. We illustrate our approach on the automated verification of programs that involve quantification and aggregation operations over arrays, such as the maximum value or sum of the elements in a given segment of the array, which are known to be difficult to reason about automatically. We implement our approach in the MonoCera tool, which is tailored to the verification of programs with aggregation, and evaluate it on example programs, including SV-COMP programs.

1 Introduction

Overview Program specifications are often written in expressive, high-level languages: for instance, in temporal logic [14], in first-order logic with quantifiers [28], in separation logic [40], or in specification languages that provide extended quantifiers for computing the sum or maximum value of array elements [33,7]. Specifications commonly also use a rich set of theories; for instance, specifications could be written using full Peano arithmetic, as opposed to bit-vectors or linear arithmetic used in the program. Rich specification languages make it possible to express intended program behaviour in a succinct form, and as a result reduce the likelihood of mistakes being introduced in specifications.

There is a gap, however, between the languages used in specifications and the input languages of automatic verification tools. Software model checkers, in particular, usually require specifications to be expressed using program assertions and Boolean program expressions, and do not directly support any of the more sophisticated language features mentioned. In fact, rich specification languages are challenging to handle in automatic verification, since satisfiability checks

can become undecidable (i.e., it is no longer decidable whether assertion failures can occur on a program path), and techniques for inferring program invariants usually focus on simple specifications only.

To bridge this gap, it is common practice to *encode* high-level specifications in the low-level assertion languages understood by the tools. For instance, temporal properties can be translated to Büchi automata, and added to programs using ghost variables and assertions [14]; quantified properties can be replaced with non-determinism, ghost variables, or loops [13,37]; sets used to specify the absence of data-races can be represented using non-deterministically initialized variables [18]. By adding ghost variables and bespoke ghost code to programs [22], many specifications can be made effectively checkable.

The translation of specifications to assertions or ghost code is today largely designed, or even carried out, by hand. This is an error-prone process, and for complex specifications and programs it is very hard to ensure that the low-level encoding of a specification faithfully models the original high-level properties to be checked. Mistakes have been found even in industrial, very carefully developed specifications [39], and can result in assertions that are vacuously satisfied by any program. Naturally, the manual translation of specifications also tends to be an ad-hoc process that does not easily generalise to other specifications.

This paper proposes the first general framework to automate the translation of rich program specifications to simpler program assertions, using a process called *instrumentation.* Our approach models the semantics of specific complex operations using program-independent *instrumentation operators,* consisting of (manually designed) rewriting rules that define how the evaluation of the operator can be achieved using simpler program statements and ghost variables. The instrumentation approach is flexible enough to cover a wide range of different operators, including operators that are best handled by weaving their evaluation into the program to be analysed. While instrumentation operators are manually written, their application to programs can be performed in a fully automatic way by means of a search procedure. The soundness of an instrumentation operator is shown formally, once and for all, by providing an *instrumentation invariant* that ensures that the operator can never be used to show correctness of an incorrect program.

Additional instrumentation operator definitions, correctness proofs, and detailed evaluation results can be found in the accompanying extended report [4].

Motivating Example We illustrate our approach on the computation of *triangular numbers* $s_N = (N^2 + N)/2$, see left-hand side of Figure 1. For reasons of presentation, the program has been normalised by representing the square `N*N` using an auxiliary variable `NN`. While mathematically simple, verifying the post-condition `s == (NN+N)/2` in the program turns out to be challenging even for state-of-the-art model checkers, as such tools are usually thrown off course by the non-linear term `N*N`. Computing the value of `NN` by adding a loop in line 16 is not sufficient for most tools either, since the program in any case requires

```
1  // Triangular numbers
2  i = 0; /*A*/ s = 0; /*B*/
3  assume(N>0);
4  while(i < N) {
5
6
7
8      i = i + 1; /*C*/
9
10
11     s = s + i;
12 }
13
14
15 NN = N*N; /*D*/
16
17 assert(s == (NN+N)/2);
```

```
1  // Instrumented program
2  i=0; s=0; x_sq=0; x_shad=0;
3  assume(N>0);
4  while(i < N) {
5      // Begin-instrumentation
6      assert(i == x_shad);
7      x_sq   = x_sq + 2*i + 1;
8      i      = i + 1;
9      x_shad = i;
10     // End-instrumentation
11     s      = s + i;
12 }
13 // Begin-instrumentation
14 assert(N == x_shad);
15 NN = x_sq;
16 // End-instrumentation
17 assert(s == (NN+N)/2);
```

Fig. 1: Program computing triangular numbers, and its instrumented counterpart

a non-linear invariant `0 <= i <= N && 2*s == i*i + i` to be derived for the loop in lines 4–12.

The insight needed to elegantly verify the program is that the value `i*i` can be tracked during the program execution using a ghost variable `x_sq`. For this, the program is instrumented to maintain the relationship `x_sq == i*i`: initially, `i == x_sq == 0`, and each time the value of `i` is modified, also the variable `x_sq` is updated accordingly. With the value `x_sq == i*i` available, both the loop invariant and the post-condition turn into formulas over linear arithmetic, and program verification becomes largely straightforward. The challenge, of course, is to discover this program transformation automatically, and to guarantee the soundness of the process. For the example, the transformed program is shown on the right-hand side of Figure 1, and discussed in the next paragraphs.

Our method splits the process of program instrumentation into two parts: (i) choosing an *instrumentation operator,* which is defined manually, designed to be program-independent, and induces a space of possible program transformations; and (ii) carrying out an automatic *application strategy* to find, among the possible program transformations, one that enables verification of a program.

An instrumentation operator for tracking squares is shown in Figure 2, and consists of the declaration of two ghost variables (`x_sq`, `x_shad`) with initial value 0, respectively; four rules for rewriting program statements; and the instrumentation invariant witnessing correctness of the operator. The rewrite rules use formal variables x, y, which can represent arbitrary variables in the program (`i`, `N`, `NN`). An application of the operator to a program will declare the ghost variables in the form of global variables, and then rewrite some chosen set of program statements using the provided rules. Since the statements to be rewritten

Ω_{square} (Instrumentation operator)

$\mathbf{G}_{square}$ (Ghost variables)

`x_sq, x_shad : Int`

`init(x_sq)` $=$ 0, `init(x_shad)` $=$ 0

$\mathbf{R}_{square}$ (Rewrite rules)

x = α	$\rightsquigarrow$	x = α; `x_sq` = α^2; `x_shad` = x	(R1)
x = x + α	$\rightsquigarrow$	`assert(`x `== x_shad);`	(R2)
		`x_sq = x_sq +` 2α `*` x + α^2; x = x + α; `x_shad` = x	
x = α * x	$\rightsquigarrow$	`assert(`x `== x_shad);`	(R3)
		`x_sq` = α^2 `* x_sq;` x = α * x; `x_shad` = x	
y = x * x	$\rightsquigarrow$	`assert(`x `== x_shad);` y = `x_sq`	(R4)

$\mathbf{I}_{square}$ (Instrumentation invariant)

`x_sq` $=$ `x_shad`^{2}

Fig. 2: Definition of an instrumentation operator Ω_{square} for tracking squares

can be chosen arbitrarily, and since moreover multiple rewrite rules might apply to some statements, rewriting can result in many different variants of a program. In the example, we rewrite the assignments C, D of the left-hand side program using rewrite rules (R2) and (R4), respectively, resulting in the instrumented and correct program on the right-hand side.

Instrumentation operators are designed to be *sound,* which means that rewriting a wrong selection of program statements might lead to an instrumented program that cannot be verified, i.e., in which assertions might fail, but instrumentation can never turn an incorrect source program into a correct instrumented program. This opens up the possibility to systematically search for the right program instrumentation. We propose a counterexample-guided algorithm for this purpose, which starts from some arbitrarily chosen instrumentation, checks whether the instrumented program can be verified, and otherwise attempts to fix the instrumentation using a refinement loop. As soon as a verifiable instrumented program has been found, the search can stop and the correctness of the original program has been shown.

The concept of instrumentation invariants is essential for guaranteeing soundness of an operator. Instrumentation invariants are formulas that can (only) refer to the ghost variables introduced by an instrumentation operator, and are formulated in such a way that they hold *in every reachable state of every instrumented program.* To maintain their invariants, instrumentation operators use shadow variables that duplicate the values of program variables. In the operator in Figure 2, the purpose of the shadow variable `x_shad` is to reproduce the value of the program variable whose square is tracked (`i`). The rewriting rules intro-

duce guards to detect incorrect instrumentation (the assertions in (R2), (R3), (R4)), which are particular cases in which some update of a relevant variable was missed and not correctly instrumented. The use of shadow variables and guards make instrumentation operators very flexible; in our example, note that instrumentation tracks the square of the value of `i` during the loop, but is also used later to simplify the expression `N*N`. This is possible because of the instrumentation invariant and because `i == N` holds after termination of the loop, which is verified through the assertion introduced in line 14.

Contributions and Outline The operator shown in Figure 2 is simple, and does not apply to all programs, but it can easily be generalised to other arithmetic operators and program statements. The framework presented in this paper provides the foundation for developing a (extendable) library of formally verified instrumentation operators. In the scope of this paper, we focus on two specification constructs that have been identified as particularly challenging in the literature: existential and universal *quantifiers* over arrays, and *aggregation* (or *extended quantifiers*), which includes computing the sum or maximum value of elements in an array. Our experiments on benchmarks taken from the SV-COMP [8] show that even relatively simple instrumentation operators can significantly extend the capabilities of a software model checker, and often make the automatic verification of otherwise hard specifications easy.

The contributions of the paper are: (i) a general *framework for program instrumentation*, which defines a space of program transformations that work by rewriting individual statements (Section 2); (ii) an application strategy *search algorithm* in this space, for a given program (Section 3); (iii) two *instantiations* of the framework—one for instrumentation operators to handle specifications with *quantifiers* (Section 4.1), and one for *extended quantifiers* (Section 4.2); (iv) machine-checked proofs of the correctness of the instrumentation operators for quantifiers $\forall$ and the extended quantifier `\max`; (v) a new *verification tool*, MonoCera, that is tailored to the verification of programs with aggregation; and (vi) an *evaluation* of our method and tool on a set of examples, including such from SV-COMP [8] (Section 5).

2 Instrumentation Framework

The next two sections formally introduce the instrumentation framework. Later, we instantiate the framework for quantification and aggregation over arrays. We split the instrumentation process into two parts:

1. An *instrumentation operator* that defines how to rewrite program statements with the purpose of eliminating language constructs that are difficult to reason about automatically, but leaves the choice of which occurrences of these statements to rewrite to the second part (this section).
2. An *application strategy* for the instrumentation operator, which can be implemented using heuristics or systematic search, among others. The strategy

Table 1: Syntax of the core language.

$$
\begin{array}{rcl}
\langle \mathit{Type} \rangle & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{Array}\ \langle \mathit{Type} \rangle \\
\langle \mathit{Expr} \rangle & ::= & \langle \mathit{DecimalNumber} \rangle \mid \texttt{true} \mid \texttt{false} \mid \langle \mathit{Variable} \rangle \\
& \mid & \langle \mathit{Expr} \rangle\ \texttt{==}\ \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle\ \texttt{<=}\ \langle \mathit{Expr} \rangle \mid \texttt{!}\langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle\ \texttt{\&\&}\ \langle \mathit{Expr} \rangle \\
& \mid & \langle \mathit{Expr} \rangle\ \texttt{||}\ \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle\ \texttt{+}\ \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle\ \texttt{*}\ \langle \mathit{Expr} \rangle \\
& \mid & \texttt{select(}\langle \mathit{Expr} \rangle\texttt{,}\langle \mathit{Expr} \rangle\texttt{)} \mid \texttt{store(}\langle \mathit{Expr} \rangle\texttt{,}\langle \mathit{Expr} \rangle\texttt{,}\langle \mathit{Expr} \rangle\texttt{)} \\
\langle \mathit{Prog} \rangle & ::= & \texttt{skip} \mid \langle \mathit{Variable} \rangle\ \texttt{=}\ \langle \mathit{Expr} \rangle \mid \langle \mathit{Prog} \rangle\texttt{;}\ \langle \mathit{Prog} \rangle \mid \texttt{while (}\langle \mathit{Expr} \rangle\texttt{)}\ \langle \mathit{Prog} \rangle \\
& \mid & \texttt{assert(}\langle \mathit{Expr} \rangle\texttt{)} \mid \texttt{assume(}\langle \mathit{Expr} \rangle\texttt{)} \mid \texttt{if (}\langle \mathit{Expr} \rangle\texttt{)}\ \langle \mathit{Prog} \rangle\ \texttt{else}\ \langle \mathit{Prog} \rangle
\end{array}
$$

is responsible for selecting the right (if any) program instrumentation from the many possible ones, Section 3 is dedicated to the second part.

Even though instrumentation operators are non-deterministic, we shall guarantee their *soundness:* if the original program has a failing assertion, so will any instrumented program, regardless of the chosen application strategy; that is, instrumentation of an incorrect program will never yield a correct program.

We shall also guarantee a weak form of *completeness*, to the effect that if an assertion that has not been added to the program by the instrumentation fails in the instrumented program, then it will also fail in the original program. As a result, any counterexample (for such an assertion) produced when verifying the instrumented program can be transformed into a counterexample for the original program.

2.1 The Core Language

While our implementation works on programs represented as constrained Horn clauses [12], i.e., is language-agnostic, for readability purposes we present our approach in the setting of an imperative core programming language with datatypes for unbounded integers, Booleans, and arrays, and `assert` and `assume` statements. The language is deliberately kept simple, but is still close to standard C. The main exception is the semantics of arrays: they are defined here to be *functional* and therefore represent a value type. Arrays have integers as index type and are unbounded, and their signature and semantics are otherwise borrowed from the SMT-LIB theory of extensional arrays [6]:

- *Reading* the value of an array `a` at index `i`: `select(a, i)`;
- *Updating* an array `a` at index `i` with a new value `x`: `store(a, i, x)`.

The complete syntax of the core language is given in Table 1. Programs are written using a vocabulary $\mathcal{X}$ of typed program variables; the typing rules of the language are given in [4]. As syntactic sugar, we sometimes write `a[i]` instead of `select(a, i)`, and `a[i] = x` instead of `a = store(a, i, x)`.

We denote by D_σ the domain of a program type σ. The domain of an array type `Array` σ is the set of functions $f : \mathbb{Z} \to D_\sigma$.

Semantics. We assume the Flanagan-Saxe *extended execution model* of programs with `assume` and `assert` statements (see, e.g., [23]), in which executing an `assert` statement with an argument that evaluates to false *fails*, i.e., terminates abnormally. An `assume` statement with an argument that evaluates to false has the same semantics as a non-terminating loop. Partial correctness properties of programs are expressed using *Hoare triples* $\{Pre\}\ P\ \{Post\}$, which state that an execution of P, starting in a state satisfying Pre, never fails, and may only terminate in states that satisfy $Post$. As usual, a program P is considered *(partially) correct* if the Hoare triple $\{true\}\ P\ \{true\}$ holds.

The evaluation of program expressions is modelled using a function $[\![\cdot]\!]_s$ that maps program expressions t of type σ to their value $[\![t]\!]_s \in D_\sigma$ in the state s.

2.2 Instrumentation Operators

An instrumentation operator defines schemes to rewrite programs while preserving the meaning of the existing program assertions. Without loss of generality, we restrict program rewriting to assignment statements. Instrumentation can introduce *ghost state* by adding arbitrary fresh variables to the program. The main part of an instrumentation consists of *rewrite rules*, which are schematic rules $r := t \rightsquigarrow s$, where the meta-variable r ranges over program variables, t is an expression that can contain further meta-variables, and s is a schematic program in which the meta-variables from $r := t$ might occur. Any assignment that matches $r := t$ can be rewritten to s.

Definition 1 (Instrumentation Operator). *An* instrumentation operator *is a tuple $\Omega = (G, R, I)$, where:*

(i) $G = \langle(\mathtt{x}_1, init_1), \ldots, (\mathtt{x}_k, init_k)\rangle$ *is a tuple of pairs of ghost variables and their initial values;*
(ii) R *is a set of rewrite rules $r := t \rightsquigarrow s$, where s is a program operating on the ghost variables $\mathtt{x}_1, \ldots, \mathtt{x}_k$ (and containing meta-variables from $r := t$);*
(iii) I *is a formula over the ghost variables $\mathtt{x}_1, \ldots, \mathtt{x}_k$, called the* instrumentation invariant.

The rewrite rules R and the invariant I must adhere to the following constraints:

1. *The instrumentation invariant I is satisfied by the initial ghost values, i.e., it holds in the state $\{\mathtt{x}_1 \mapsto init_1, \ldots, \mathtt{x}_k \mapsto init_k\}$.*
2. *For all rewrites $r := t \rightsquigarrow s \in R$ the following hold:*
   (a) *s terminates (normally or abnormally) for pre-states satisfying I, assuming that all meta-variables are ordinary program variables.*
   (b) *s does not assign to variables other than r or the ghost variables $\mathtt{x}_1, \ldots, \mathtt{x}_k$.*
   (c) *s preserves the instrumentation invariant: $\{I\}\ s'\ \{I\}$, where s' is s with every `assert(`e`)` statement replaced by an `assume(`e`)` statement.*
   (d) *s preserves the semantics of the assignment $r := t$: the Hoare triple $\{I\}\ \mathtt{z} := t;\ s'\ \{\mathtt{z} = r\}$, where $\mathtt{z}$ is a fresh variable, holds.*

The conditions imposed in the definition ensure that all instrumentations are *correct*, in the sense that they are sound and weakly complete, as we show below. In particular, the instrumentation invariant guarantees that the rewrites of program statements are *semantics-preserving* w.r.t. the original program, and thus, the execution of any `assert` statement of the original program has the same effect before and after instrumentation. Observe that the conditions can themselves be deductively verified to hold for each concrete instrumentation operator, and that this check is *independent* of the programs to be instrumented, so that an instrumentation operator can be proven correct once and for all.

An instrumentation operator Ω does itself not define which occurrences of program statements are to be rewritten, but only how they are rewritten. Given a program P and the operator Ω, an instrumented program P' is derived by carrying out the following two steps: (i) variables $\mathtt{x}_1, \ldots, \mathtt{x}_k$ and the assignments $\mathtt{x}_1 = \mathit{init}_1; \ \ldots; \ \mathtt{x}_k = \mathit{init}_k$ are added at the beginning of the program, and (ii) some of the assignments in P, to which a rewriting rule $r = t \rightsquigarrow s$ in Ω is applicable, are replaced by s, substituting meta-variables with the actual terms occurring in the assignment. We denote by $\Omega(P)$ the set of all instrumented programs P' that can be derived in this way. An example of an instrumentation operator and its application was shown Figure 1 and Figure 2.

2.3 Instrumentation Correctness

Verification of an instrumented program produces one of two possible results: a *witness* if verification is successful, or a *counterexample* otherwise. A witness consists of the inductive invariants needed to verify the program, and is presented in the context of the programming language: it is translated back from the back-end theory used by the verification tool, and is a formula over the program variables and the ghost variables added during instrumentation. A counterexample is an execution trace leading to a failing assertion.

Definition 2 (Soundness). *An instrumentation operator Ω is called* sound *if for every program P and instrumented program $P' \in \Omega(P)$, whenever there is an execution of P where some* `assert` *statement fails, then there also is an execution of P' where some* `assert` *statement fails.*

Equivalently, existence of a witness for an instrumented program entails existence of a witness for the original program, in the form of a set of inductive invariants solely over the program variables. Notably, because of the semantics-preserving nature of the rewrites under the instrumentation invariant, a witness for the original program can be derived from one for the instrumented program. One such back-translation is to add the instrumentation invariant as a conjunct to the original witness, and to existentially quantify over the ghost variables.

Example. To illustrate the back-translation, we return to the instrumentation operator from Figure 2 and the example program from Figure 1. The witness produced by our verification tool in this case is the formula:

$$\mathtt{i} = \mathtt{x\_shad} \wedge \mathtt{x\_sq} + \mathtt{x\_shad} = 2s \wedge \mathtt{N} \geq \mathtt{i} \wedge \mathtt{N} \geq 1 \wedge 2\mathtt{s} \geq \mathtt{i} \wedge \mathtt{i} \geq 0$$

After conjoining the instrumentation invariant $\texttt{x\_sq} = \texttt{x\_shad}^2$ and existentially quantifying over the involved ghost variables, we obtain an inductive invariant that is sufficient to verify the original program:

$$\exists x_{\text{sq}}, x_{\text{shad}}.\ (\texttt{i} = x_{\text{shad}} \wedge x_{\text{sq}} + x_{\text{shad}} = 2s \wedge \\ \texttt{N} \geq \texttt{i} \wedge \texttt{N} \geq \texttt{1} \wedge \texttt{2s} \geq \texttt{i} \wedge \texttt{i} \geq 0 \wedge x_{\text{sq}} = x_{\text{shad}}^2)$$

Definition 3 (Weak Completeness). *The operator Ω is called* weakly complete *if for every program P and instrumented program $P' \in \Omega(P)$, whenever an* `assert` *statement that has not been added to the program by the instrumentation fails in the instrumented program P', then it also fails in the original program P.*

Similarly to the back-translation of invariants, when verification fails, counterexamples for assertions of the original program, found during verification of the instrumented program, can be translated back to counterexamples for the original program. We thus obtain the following result.

Theorem 1 (Soundness and weak completeness). *Every instrumentation operator Ω is sound and weakly complete.*

Proof. Let $\Omega = (G, R, I)$ be an instrumentation operator. Since I is a formula over ghost variables only, which holds initially and is preserved by all rewrites, I is an invariant of the fully instrumented program. This entails that rewrites of assignments are semantics-preserving. Furthermore, since instrumentation code only assigns to ghost variables or to r (i.e., the left-hand side of the original statement), program variables have the same valuation in the instrumented program as in the original one. Furthermore, since all rewrites are terminating under I, the instrumented program will terminate if and only if the original program does.

In the case when verification succeeds, and a witness is produced, weak completeness follows vacuously. A witness consists of the inductive invariants sufficient to verify the instrumented program. Thus, they are also sufficient to verify the assertions existing in the original program, since assertions are not rewritten and all program variables have the same valuation in the original and the instrumented programs. Since a witness for the instrumented program can be back-translated to a witness for the original program, any failing assertion in the original program must also fail after instrumentation, and Ω is therefore sound.

In the case when verification fails, soundness follows vacuously, and if the failing assertion was added during instrumentation, also weak completeness follows. If the assertion existed in the original program, since such assertions are not rewritten, and since program variables have the same valuation in the instrumented program as in the original program, then any counterexample for the instrumented program is also a counterexample for the original program, when projected onto the program variables. □

Input: Program P; statements S; instrumentation space R; oracle *IsCorrect*.
Result: Instrumentation $r \in R$ with $\mathit{IsCorrect}(P_r)$; *Incorrect*; or *Inconclusive*.

```
1  begin
2      Cand ← R;
3      while Cand ≠ ∅ do
4          pick r ∈ Cand;
5          if IsCorrect(P_r) then
6              return r;
7          else
8              cex ← counterexample path for P_r;
9              if failing assertion in cex also exists in P then
                   /* cex is also a counterexample for P                */
10                 return Incorrect;
11             else
                   /* instrumentation on cex may have been incorrect    */
12                 C' ← {p ∈ C | ins_r(p) occurs on cex};
13                 Cand ← Cand \ {r' ∈ Cand | r(s) = r'(s) for all p ∈ C'};
14             end
15         end
16     end
17     return Inconclusive;
18 end
```

Algorithm 1: Counterexample-guided instrumentation search

3 Instrumentation Application Strategies

We will now define a counterexample-guided search procedure to discover applications of instrumentation operators that make it possible to verify a program.

For our algorithm, we assume that we are given an oracle *IsCorrect* that is able to check the correctness of programs after instrumentation. Such an oracle could be approximated, for instance, using a software model checker. The oracle is free to ignore the complex functions we are trying to eliminate by instrumentation; for instance, in Figure 1, the oracle can over-approximate the term `N*N` by assuming that it can have any value. We further assume that C is the set of control points of a program P corresponding to the statements to which a given set of instrumentation operators can be applied. For each control point $p \in C$, let $Q(p)$ be the set of rewrite rules applicable to the statement at p, including also a distinguished value $\perp$ that expresses that p is not modified. For the program in Figure 1, for instance, the choices could be defined by $Q(\texttt{A}) = Q(\texttt{B}) = \{(\text{R1}), \perp\}$, $Q(\texttt{C}) = \{(\text{R2}), \perp\}$, and $Q(\texttt{D}) = \{(\text{R4}), \perp\}$, referring to the rules in Figure 2. Any function $r : C \to \bigcup_{p \in C} Q(p)$ with $r(p) \in Q(p)$ will then define one possible program instrumentation. We will denote the set of well-typed functions $C \to \bigcup_{p \in C} Q(p)$ by R, and the program obtained by rewriting P according to $r \in R$ by P_r. We further denote the control point in P_r corresponding to some $p \in C$ in P by $\mathit{ins}_r(p)$.

Table 2: Extension of the core language with quantified expressions.

$$
\begin{aligned}
\langle Expr \rangle \ ::= \ & \texttt{(}\lambda\texttt{(}\langle Variable \rangle\texttt{,}\langle Variable \rangle\texttt{).}\langle Expr \rangle\texttt{)}\ \texttt{(}\langle Expr \rangle\texttt{,}\ \langle Expr \rangle\texttt{)} \mid \\
& \texttt{forall(}\langle Expr \rangle\texttt{,}\langle Expr \rangle\texttt{,}\langle Expr \rangle\texttt{,}\lambda\texttt{(}\langle Variable \rangle\texttt{,}\langle Variable \rangle\texttt{).}\langle Expr \rangle\texttt{)} \mid \\
& \texttt{exists(}\langle Expr \rangle\texttt{,}\langle Expr \rangle\texttt{,}\langle Expr \rangle\texttt{,}\lambda\texttt{(}\langle Variable \rangle\texttt{,}\langle Variable \rangle\texttt{).}\langle Expr \rangle\texttt{)}
\end{aligned}
$$

Algorithm 1 presents our algorithm to search for instrumentations that are sufficient to verify a program P. The algorithm maintains a set $Cand \subseteq R$ of remaining ways to instrument P, and in each loop considers one of the remaining elements $r \in Cand$ (line 4). If the oracle manages to verify P_r in line 5, due to soundness of instrumentation the correctness of P has been shown (line 6); if P_r is incorrect, there has to be a counterexample ending with a failing assertion (line 8). There are two possible causes of assertion failures: if the failing assertion in P_r already existed in P, then due to the weak completeness of instrumentation also P has to be incorrect (line 10). Otherwise, the program instrumentation has to be refined, and for this from $Cand$ we remove all instrumentations r' that agree with r regarding the instrumentation of the statements occurring in the counterexample (line 13).

Since R is finite, and at least one element of $Cand$ is eliminated in each iteration, the refinement loop terminates. The set $Cand$ can be exponentially big, however, and therefore should be represented symbolically (using BDDs, or using an SMT solver managing the set of blocking constraints from line 13).

We can observe soundness and completeness of the algorithm w.r.t. the considered instrumentation operators (proof in [4]):

Lemma 1 (Correctness of Algorithm 1). *If Algorithm 1 returns an instrumentation $r \in R$, then P_r and P are correct. If Algorithm 1 returns Incorrect, then P is incorrect. If there is $r \in R$ such that P_r is correct, then Algorithm 1 will return r' such that $P_{r'}$ is correct.*

4 Instrumentation Operators for Arrays

4.1 Instrumentation Operators for Quantification over Arrays

To handle quantifiers in a programming setting, we extend the language defined in Table 1 by adding quantified expressions over arrays, as shown in Table 2. As seen, we also extend the language with a lambda expression over two variables. The rationale for this is that many quantified properties can be expressed as a binary predicate with the first argument corresponding to the value of an element and the second to the index. This allows us to express properties over both the value of an element and its index. For example, we can express that each element should be equal to its index, as is done in the example program in Figure 3. In the program, each element in the array is assigned the value corresponding to its index, after which it is asserted that this property indeed holds.

```
Int N = nondet;
assume(N > 0);
Array Int a = const(0, N);
Int i = 0;
while(i < N) {
    a = store(a, i, i);
    i = i + 1;
}
Bool b = forall(a, 0, N, λ(i,x).(x == i));
assert(b);
```

Fig. 3: Example of program to be verified using a quantified assert statement.

Using $\texttt{P(x}_0\texttt{,i}_0\texttt{)}$ as shorthand for $(\lambda\texttt{(x,i).P)(x}_0\texttt{,i}_0\texttt{)}$, the new expressions can be defined formally as:

$$[\![\texttt{forall(a, l, u, } \lambda\texttt{(x,i).P)}]\!]_s = \forall \texttt{i} \in [\texttt{l},\texttt{u}).\ [\![\texttt{P(a[i],i)}]\!]_s$$
$$[\![\texttt{exists(a, l, u, } \lambda\texttt{(x,i).P)}]\!]_s = \exists \texttt{i} \in [\texttt{l},\texttt{u}).\ [\![\texttt{P(a[i],i)}]\!]_s$$

Note that the types of `x` and `a` must be compatible and `P` be a Boolean-valued expression.

To handle programs such as the one in Figure 3, we turn to the instrumentation framework outlined in Section 2.2, which we use here to define an instrumentation operator for universal quantification. The general idea is to instrument programs with a ghost variable, tracking if some predicate holds for all elements in an interval of the array, with shadow variables representing the tracked array, and the bounds of the interval. Naturally, an instrumentation operator for existential quantification can be defined in a similar fashion. For simplicity, we shall assume a *normal form* of programs, into which every program can be rewritten by introducing additional variables. In the normal form, `store`, `select` and `forall` can only occur in simple assignment statements. For example, stores are restricted to occur in statements of the form: `a' = store(a, i, x)`.

Over such normalised programs, and for a universally quantified expression `forall(a, l, u, λ(x,i)(P))`, we define the instrumentation operator $\Omega_{\forall,P} = (G_{\forall,P}, R_{\forall,P}, I_{\forall,P})$ as shown in Figure 4 over four ghost variables. The array over which quantification occurs is tracked by `qu_ar` and the variables `qu_lo`, `qu_hi` represent the bounds of the currently tracked interval. The result of the quantified expression is tracked by `qu_P`, whose value is *true* iff `P` holds for all elements in `a` in the interval $[\texttt{qu\_lo}, \texttt{qu\_hi})$. The rewrite rules for stores, selects and assignments of universally quantified expressions are then defined as follows. For stores, the first if-branch resets the tracking to the one element interval $[\texttt{i}, \texttt{i} + 1)$ when accessing elements far outside of the currently tracked interval, or if we are tracking the empty interval (as is the case at initialisation). If an access occurs immediately adjacent to the currently tracked interval (e.g., if $\texttt{i} = \texttt{qu\_lo} - 1$), then that element is added to the tracked interval, and the

$\Omega_{\forall,\mathbf{P}}$ (Instrumentation operator)

$\mathbf{G}_{\forall,\mathbf{P}}$ (Ghost variables)

`qu_ar : Array Int, qu_lo, qu_hi : Int, qu_P : Bool`

`init(qu_ar)` $=$ `[]`, `init(qu_lo)` $=$ 0, `init(qu_hi)` $=$ 0
`init(qu_P)` $=$ `true`

$\mathbf{R}_{\forall,\mathbf{P}}$ (Rewrite rules)

`a' = store(a, i, x);` $\rightsquigarrow$

```
a' = store(a, i, x);
if (qu_lo == qu_hi || i < qu_lo - 1 || i > qu_hi ||
    (P(x, i) && !qu_P && qu_lo <= i && i < qu_hi)) {
  qu_lo = i;              // Reset, because either:
  qu_hi = i + 1;          // - tracking empty interval
  qu_P  = P(x, i);        // - storing far outside interval
} else {                  // - possibly overwriting sole false
  assert(qu_ar == a);
  qu_P = qu_P && P(x, i);
  if (qu_lo - 1 == i) {
      qu_lo = i;           // Decrement lower bound by 1
  } else if (qu_hi == i) {
      qu_hi = i + 1;       // Increment upper bound by 1
  }
}
qu_ar = a';
```

`x = select(a, i);` $\rightsquigarrow$ similar to store

`b = forall(a, l, u,`λ`x.P);` $\rightsquigarrow$

```
if (u <= l) {
  b = true;
} else {
  if (qu_P) {
    assert(qu_ar == a && l >= qu_lo && u <= qu_hi);
  } else {
    assert(qu_ar == a && l <= qu_lo && u >= qu_hi);
  }
  b = qu_P;
}
```

$\mathbf{I}_{\forall,\mathbf{P}}$ (Instrumentation Invariant)

$\texttt{qu\_lo} = \texttt{qu\_hi} \vee$
$(\texttt{qu\_lo} < \texttt{qu\_hi} \wedge \texttt{qu\_P} = \texttt{forall}(\texttt{qu\_ar}, \texttt{qu\_lo}, \texttt{qu\_hi}, \lambda(\texttt{x}, \texttt{i}).\texttt{P}))$

Fig. 4: Definition of an instrumentation operator for universal quantification

value of `qu_P` is updated to also account for the value of `P` at index `i`. If instead the access is within the tracked interval, then we either reset the interval (if `qu_P` is `false`) or keep the interval unchanged (if `qu_P` is `true`). Rewrites of selects are similar to stores, except tracking does not need to be reset when reading inside the tracked interval. For rewrites of quantified expressions, if the quantified interval is empty, `b` is assigned `true`. Otherwise, assertions check that the tracked interval matches the quantified interval before assigning `t` to `qu_P`.

If `qu_P` is `true`, then it is sufficient that quantification occurs over a sub-interval of the tracked interval, and vice versa if `qu_P` is `false`.

The result of applying $\Omega_{\forall,P}$ to the program in Figure 3 is shown in [4]. As exhibited by the experiments in Section 5, the resulting program is in many cases easier to verify by state-of-the-art verification tools. Note that the instrumentation operator defined is only one possibility among many. For example, one could track several ranges simultaneously over the array in question, or also track the index of some element in the array over which `P` holds, or make different choices on stores outside of the tracked interval.

The following lemma establishes correctness of the instrumentation operator. The proof can be found in [4].

Lemma 2 (Correctness of $\Omega_{\forall,P}$). *$\Omega_{\forall,P}$ is an instrumentation operator, i.e., it adheres to the constraints imposed in Definition 1.*

4.2 Instrumentation Operators for Aggregation over Arrays

We now turn to the verification of safety properties with *aggregation.* As examples of aggregation, we consider in particular the operators `\sum` and `\max`, calculating the sum and maximum value of an array, respectively. Aggregation is supported in the form of *extended quantifiers* in the specification languages JML [33] and ACSL [7], and is frequently needed for the specification of functional correctness properties. Although commonly used, most verification tools do not support aggregation, so that properties involving aggregation have to be manually rewritten using standard quantifiers, pure recursive functions, or ghost code involving loops. This reduction step is error-prone, and represents an additional complication for automatic verification approaches, but can be handled elegantly using the instrumentation framework. For generality, we formalise aggregation over arrays with the help of monoid homomorphisms.

Definition 4 (Monoid). *A* monoid *is a structure $(M, \circ, e)$ consisting of a non-empty set M, a binary associative operation $\circ$ on M, and a neutral element $e \in M$. A* monoid *is* commutative *if $\circ$ is commutative. A monoid is* cancellative *if $x \circ y = x \circ z$ implies $y = z$, and $y \circ x = z \circ x$ implies $y = z$, for all $x, y, z \in M$.*

For aggregation, we model finite intervals of arrays using the cancellative monoid $(D^*, \cdot, \epsilon)$ of finite sequences over some data domain D. The concatenation operator $\cdot$ is non-commutative.

Definition 5 (Monoid Homomorphism). *A* monoid homomorphism *is a function $h : M_1 \to M_2$ between monoids $(M_1, \circ_1, e_1)$ and $(M_2, \circ_2, e_2)$ with the properties $h(x \circ_1 y) = h(x) \circ_2 h(y)$ and $h(e_1) = e_2$.*

Ordinary quantifiers can be modelled as homormorphisms $D^* \to \mathbb{B}$, so that the instrumentation in this section strictly generalizes Section 4.1. A second classical example is the computation of the *maximum* (similarly, *minimum*) value in a sequence. For the domain of integers, the natural monoid to use is the

algebra $(\mathbb{Z}_{-\infty}, \max, -\infty)$ of integers extended with $-\infty$,[4] and the homomorphism $h_{\max}$ is generated by mapping singleton sequences $\langle n \rangle$ to the value n. A third example is the computation of the element *sum* of an integer sequence, corresponding to the monoid $(Z, +, 0)$ and the homomorphism h_{sum}. Similarly, the *number of occurrences* of some element can be computed. The considered monoid in the last two cases of aggregation is even cancellative.

Programming Language with Aggregation We extend our core programming language with expressions $\texttt{aggregate}_{M,h}\texttt{(}\langle Expr \rangle\texttt{,}\langle Expr \rangle\texttt{,}\langle Expr \rangle\texttt{)}$, and use monoid homomorphisms to formalise them. Recall that we denote by D_σ the domain of a program type σ.

Definition 6. *Let* $\texttt{Array}\ \sigma$ *be an array type,* σ_M *a program type,* M *a commutative monoid that is a subset of* D_{σ_M}*, and* $h : D_\sigma^* \to M$ *a monoid homomorphism. Let furthermore* ar *be an expression of type* $\texttt{Array}\ \sigma$*, and* l *and* u *integer expressions. Then,* $\texttt{aggregate}_{M,h}\texttt{(}ar\texttt{,}l\texttt{,}u\texttt{)}$ *is an expression of type* σ_M*, with semantics defined by:*

$$[\![\texttt{aggregate}_{M,h}\texttt{(}ar\texttt{,}l\texttt{,}u\texttt{)}]\!]_s \;=\; h(\langle [\![ar]\!]_s([\![l]\!]_s), [\![ar]\!]_s([\![l]\!]_s{+}1), \ldots, [\![ar]\!]_s([\![u]\!]_s{-}1)\rangle)$$

Intuitively, the expression $\texttt{aggregate}_{M,h}\texttt{(}ar\texttt{,}l\texttt{,}u\texttt{)}$ denotes the result of applying the homomorphism h to the slice $ar[l \mathinner{..} u-1]$ of the array ar. As a convention, in case $u < l$ we assume that the result of `aggregate` is $h(\langle\rangle)$. As with array accesses, we assume also that `aggregate` only occurs in normalised statements of the form $\texttt{t = aggregate}_{M,h}\texttt{(}ar\texttt{,}l\texttt{,}u\texttt{)}$.

In our examples, we use derived operations as found in ACSL: `\max` as shorthand notation for $\texttt{aggregate}_{(\mathbb{Z}_{-\infty},\max,-\infty),h_{\max}}$[5], and `\sum` as short-hand notation for $\texttt{aggregate}_{(\mathbb{Z},+,0),h_{\text{sum}}}$.

An Instrumentation Operator for Maximum For `\max`, an operator $\Omega_{max} = (G_{max}, R_{max}, I_{max})$ can be defined similarly to the operator $\Omega_{\forall,P}$ from Section 4.1, in that the maximum value in a particular interval of the array is tracked. One key difference is that an extra ghost variable `ag_max_idx` is added to track an array index where the maximum value of the array interval is stored, in order to not have to reset tracking on every store inside of the tracked interval. A complete definition is proposed in [4].

An instrumentation operator for Sum Cancellative aggregation is aggregation based on a cancellative monoid. Cancellative aggregation makes it possible to track aggregate values faithfully even when storing *inside* of the tracked interval, unlike `\max` and universal quantification. An example of a cancellative operator is the aggregate `\sum`.

[4] For machine integers, $-\infty$ could be replaced with `INT_MIN`.

[5] With a slight abuse of the framework, we assume that $\mathbb{Z}_{-\infty}$ is represented by the program type `Int`, mapping $-\infty$ to some fixed integer number. More elegant solutions are not difficult to devise, but add unnecessary complexity.

Ω_{sum} (Instrumentation operator)

G_{sum} (Ghost variables)

`ag_lo, ag_hi, ag_sum : Int, ag_ar : Array Int`

`init(ag_lo) = init(ag_hi) = init(ag_sum) = 0, init(ag_ar) = []`

R_{sum} (Rewrite rules)

`a' = store(a, i, x)` $\rightsquigarrow$

```
a' = store(a, i, x);
if (ag_lo == ag_hi || i < ag_lo - 1 || i > ag_hi) {
    ag_lo      = i;          // Reset, because either:
    ag_hi      = i + 1;      // - tracking empty interval
    ag_sum     = x;          // - storing far outside interval
} else {
    assert(ag_ar == a);
    if (ag_lo <= i && i < ag_hi) {
        // Subtract previous value from sum
        ag_sum = ag_sum - select(ag_ar, i);
    } else if (ag_lo - 1 == i) {
        ag_lo = i;           // Decrease lower bound by 1
    } else if (ag_hi == i) {
        ag_hi = i + 1;       // Increase upper bound by 1
    }
    ag_sum = ag_sum + x;     // Add new value to sum
}
ag_ar = a';
```

`x = select(a, i)` $\rightsquigarrow$ code similar to rewrites of `store`

`r = \sum(a, l, u)` $\rightsquigarrow$

```
if (u <= l) {
  t = 0;
} else {
  assert(ag_ar == a && l == ag_lo && u == ag_hi);
  t = ag_sum;
}
```

I_{sum} (Instrumentation invariant)

$\texttt{ag\_lo} = \texttt{ag\_hi} \vee \texttt{ag\_sum} = \texttt{sum}(\texttt{ag\_ar}, \texttt{ag\_lo}, \texttt{ag\_hi})$

Fig. 5: Definition of an instrumentation operator Ω_{sum} for Sum

The instrumentation operator $\Omega_{sum} = (G_{sum}, R_{sum}, I_{sum})$ is defined in Figure 5. The instrumentation code tracks the sum of values in the interval, and when increasing the bounds of the tracked interval, the new values are simply added to the tracked sum. Since `\sum` is cancellative, when storing inside of the tracked interval, the previous value at the index being written to is first subtracted from the sum, before adding the new value, ensuring that the correct aggregate value is computed. The following correctness result is proved in [4].

Lemma 3 (Correctness of Ω_{sum}). *Ω_{sum} is an instrumentation operator, i.e., it adheres to the constraints imposed in Definition 1.*

Deductive Verification of Instrumentation Operators As stated in Section 2.2, instrumentation operators may be verified independently of the programs to be instrumented. The operators described in this paper, i.e. square, universal quantification, maximum, and sum, have been verified in the verification tool Frama-C [15]. The verified instrumentations are adaptations for the C language semantics and execution model. More specifically, the adapted operators assume C native arrays, rather than functional ones.

5 Evaluation

5.1 Implementation

To evaluate our instrumentation framework, we have implemented the instrumentation operators for quantifiers and aggregation over arrays. The implementation is done over constrained Horn clauses (CHCs), by adding the rewrite rules defined in Section 4 to Eldarica [30], an open-source solver for CHCs. We also implemented the automatic application of the instrumentation operators, largely following Algorithm 1 but with a few minor changes due to the CHC setting. The CHC setting makes our implementation available to various CHC-based verification tools, for instance JayHorn (Java) [32], Korn (C) [19], RustHorn (Rust) [36], SeaHorn (C/LLVM) [26] and TriCera (C) [20].

In order to evaluate our approach at the level of C programs, we extended TriCera, an open-source assertion-based model checker that translates C programs into a set of CHCs and relies on Eldarica as back-end solver. TriCera is extended to parse quantifiers and aggregation operators in its input C programs and to encode them as part of the translation into CHCs. We call the resulting toolchain MonoCera. An artefact that includes MonoCera and the benchmarks is available online [5].

To handle complicated access patterns, for instance a program processing an array from the beginning and end at the same time, the implementation can apply multiple instrumentation operators simultaneously; the number of operators is incremented when Algorithm 1 returns *Inconclusive*.

5.2 Experiments and Comparisons

To assess our implementation, we assembled a test suite and carried out experiments comparing MonoCera with the state-of-the-art C model checkers CPAchecker 2.1.1 [11], SeaHorn 10.0.0 [26] and TriCera 0.2. It should be noted that deductive verification frameworks, such as Dafny and Frama-C, can handle, for example, the program in Figure 3 if they are provided with a manually written loop invariant; however, since MonoCera relies on automatic techniques for invariant inference, we only benchmark against tools using similar automatic techniques. We also excluded VeriAbs [1], since its licence does not permit its use for scientific evaluation.

The tools were set up, as far as possible, with equivalent configurations; for instance, to use the SMT-LIB theory of arrays [6] in order to model C arrays, and

	Verification results					Ver. time			Inst. space		Inst. steps	
	#Tests	Mono	Tri	Sea	CPA	Min	Max	Avg	Max	Avg	Max	Avg
`min`	17	9	2	2	2	22	59	33	27	11	55	24
`max`	12	8	2	3	3	21	285	76	108	21	96	30
`sum`	26	16	3	3	3	26	245	78	2916	188	284	36
`forall`	96	30	1	0	2	14	236	91	59049	2446	334	59

Table 3: Results for MonoCera (Mono), TriCera (Tri), SeaHorn (Sea), and CPAchecker (CPA). For MonoCera, also statistics are given for verification time (s), size of the instrumentation search space, and search iterations.

a mathematical (as opposed to machine) semantics of integers. CPAchecker was configured to use k-induction [10], which was the only configuration that worked in our tests using mathematical integers. SeaHorn was run using the default settings. All tests were run on a Linux machine with AMD Opteron 2220 SE @ 2.8 GHz and 6 GB RAM with a timeout of 300 seconds.

Test Suite. The comparison includes a set of programs calculating properties related to the quantification and aggregation properties over arrays. The benchmarks and verification results are summarised in Table 3. The benchmark suite contains programs ranging between 16 to 117 LOC and is comprised of two parts: (i) 117 programs taken from the SV-COMP repository [9], and (ii) 26 programs crafted by the authors (`min`: 6, `max`: 8, `sum`: 9, `forall`: 3).

To construct the SV-COMP benchmark set for MonoCera we gathered all test files from the directories prefixed with `array` or `loop`, and singled out programs containing some assert statement that could be rewritten using a quantifier or an aggregation operator over a single array. For example, loops

```
for (int i = 0; i < N; i++) assert(a[i] <= 0);
```

can be rewritten using `forall` or `max` operators. We created a benchmark for each possible rewriting; for instance, in the case of `max`, by rewriting the loop into `assert(\max(a, 0, N) <= 0)`. The original benchmarks were used for the evaluation of the other tools, none of which supported (extended) quantifiers.

In (ii), we crafted 9 programs that make use of aggregation or quantifiers, and derived further benchmarks by considering different array sizes (10, 100 and unbounded size); one combination (unbounded array inside a struct) had to be excluded, as it is not valid C. In order to evaluate other tools on our crafted benchmarks, we reversed the process described for the SV-COMP benchmarks and translated the operators into corresponding loop constructs.

Results. In Table 3, we present the number of verified programs per instrumentation operator for each tool, as well as further statistics for MonoCera regarding verification times and instrumentation search space. The "Inst. space" column indicates the size of the instrumentation search space (i.e., number of instrumentations producible by applying the non-deterministic instrumentation operator).

"Inst. steps" column indicates the number of attempted instrumentations, i.e., number of iterations in the while-loop in Algorithm 1. In our implementation, the check in Algorithm 1 line 5 can time out and cause the check to be repeated at a later time with a greater timeout, which can lead to more iterations than the size of the search space. In [4], we list results per benchmark for each tool.

For the SV-COMP benchmarks, CPAchecker managed to verify 1 program, while SeaHorn and TriCera could not verify any programs. MonoCera verified in total 42 programs from SV-COMP. Regarding the crafted benchmarks, several tools could verify the examples with array size 10. However, when the array size was 100 or unbounded, only MonoCera succeeded.

6 Related Work

It is common practice, in both model checking and deductive verification, to translate high-level specifications to low-level specifications prior to verification (e.g., [14,18,13,37]). Such translations often make use of ghost variables and ghost code, although relatively little systematic research has been done on the required properties of ghost code [22]. The addition of ghost variables to a program for tracking the value of complex expressions also has similarities with the concept of term abstraction in Horn solving [3]. To the best of our knowledge, we are presenting the first general framework for automatic program instrumentation.

A lot of research in *software model checking* considered the handling of standard quantifiers $\forall, \exists$ over arrays. In the setting of constrained Horn clauses, properties with universal quantifiers can sometimes be reduced to quantifier-free reasoning over non-linear Horn clauses [13,37]. Our approach follows the same philosophy of applying an up-front program transformation, but in a more general setting. Various direct approaches to infer quantified array invariants have been proposed as well: e.g., by extending the IC3 algorithm [27], syntax-guided synthesis [21], learning [24], by solving recurrence equations [29], backward reachability [3], or superposition [25]. To the best of our knowledge, such methods have not been extended to aggregation.

Deductive verification tools usually have rich support for quantified specifications, but rely on auxiliary assertions like loop invariants provided by the user, and on SMT solvers or automated theorem provers for quantifier reasoning. Although several deductive verification tools can parse extended quantifiers, few offer support for reasoning about them. Our work is closest to the method for handling comprehension operators in Spec# [35], which relies on code annotations provided by the user, but provides heuristics to automatically verify such annotations. The code instrumentation presented in this paper has similarity with the proof rules in Spec#; the main differences are that our method is based on an upfront program transformation, and that we aim at automatically finding required program invariants, as opposed to only verifying their correctness. The KeY tool provides proof rules similar to the ones in Spec# for some of the JML extended quantifiers [2]; those proof rules can be applied manually to verify human-written invariants. The Frama-C system [15] can parse ACSL extended

quantifiers [7], but, to the best of our knowledge, none of the Frama-C plug-ins can automatically process such quantifiers. Other systems, e.g., Dafny [34], require users to manually define aggregation operators as recursive functions.

In the theory of *algebraic data-types*, several transformation-based approaches have been proposed to verify properties that involve recursive functions or catamorphisms [31,17]. Aggregation over arrays resembles the evaluation of recursive functions over data-types; a major difference is that data-types are more restricted with respect to accessing and updating data than arrays.

Array folds logic (AFL) [16] is a decidable logic in which properties on arrays beyond standard quantification can be expressed: for instance, counting the number of elements with some property. Similar properties can be expressed using automata on data words [41], or in variants of monadic second-order logic [38]. Such languages can be seen as alternative formalisms to aggregation or extended quantifiers; they do not cover, however, all kinds of aggregation we are interested in. Array sums cannot be expressed in AFL or data automata, for instance.

7 Conclusion

We have presented a framework for automatic and provably correct program instrumentation, allowing the automatic verification of programs containing certain expressive language constructs, which are not directly supported by the existing automatic verification tools. Our experiments with a prototypical implementation, in the tool MonoCera, show that our method is able to automatically verify a significant number of benchmark programs involving quantification and aggregation over arrays that are beyond the scope of other tools.

There are still various other benchmarks that MonoCera (as well as other tools) cannot verify. We believe that many of those benchmarks are in reach of our method, because of the generality of our approach. Ghost code is known to be a powerful specification mechanism; similarly, in our setting, more powerful instrumentation operators can be easily formulated for specific kinds of programs. In future work, we therefore plan to develop a library of instrumentation operators for different language constructs (including arithmetic operators), non-linear arithmetic, other types of structures with regular access patterns such as binary heaps, and general linked-data structures.

We also plan to refine our method for showing incorrectness of programs more efficiently, as the approach is currently applicable mainly for verifying correctness (experiments in [4]). Another line of work is the establishment of stronger completeness results than the weak completeness result presented here, for specific programming language fragments.

Acknowledgements. This work has been partially funded by the Swedish Vinnova FFI Programme under grant 2021-02519, the Swedish Research Council (VR) under grant 2018-04727, the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), and the Wallenberg project UPDATE. We are also grateful for the opportunity to discuss the research at the Dagstuhl Seminar 22451 on "Principles of Contract Languages."

References

1. Afzal, M., Chakraborty, S., Chauhan, A., Chimdyalwar, B., Darke, P., Gupta, A., Kumar, S., M, C.B., Unadkat, D., Venkatesh, R.: Veriabs : Verification by abstraction and test generation (competition contribution). In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 383–387. Springer (2020), https://doi.org/10.1007/978-3-030-45237-7_25
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016), https://doi.org/10.1007/978-3-319-49812-6
3. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy abstraction with interpolants for arrays. In: Bjørner, N.S., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7180, pp. 46–61. Springer (2012), https://doi.org/10.1007/978-3-642-28717-6_7
4. Amilon, J., Esen, Z., Gurov, D., Lidström, C., Rümmer, P.: Automatic program instrumentation for automatic verification (extended technical report). CoRR (2023), to appear.
5. Amilon, J., Esen, Z., Gurov, D., Lidström, C., Rümmer, P.: Artifact for the CAV 2023 paper "Automatic Program Instrumentation for Automatic Verification" (Apr 2023), https://doi.org/10.5281/zenodo.7875416
6. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at www.SMT-LIB.org
7. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html
8. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 375–402. Springer (2022), https://doi.org/10.1007/978-3-030-99527-0_20
9. Beyer, D.: SV-Benchmarks: Benchmark Set for Software Verification and Testing (SV-COMP 2022 and Test-Comp 2022) (Jan 2022), https://doi.org/10.5281/zenodo.5831003
10. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 622–640. Springer (2015), https://doi.org/10.1007/978-3-319-21690-4_42
11. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011), https://doi.org/10.1007/978-3-642-22110-1_16

12. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. Lecture Notes in Computer Science, vol. 9300, pp. 24–51. Springer (2015), https://doi.org/10.1007/978-3-319-23534-9_2
13. Bjørner, N.S., McMillan, K.L., Rybalchenko, A.: On solving universally quantified Horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7935, pp. 105–125. Springer (2013), https://doi.org/10.1007/978-3-642-38856-9_8
14. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018), https://doi.org/10.1007/978-3-319-10575-8
15. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - A software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer (2012), https://doi.org/10.1007/978-3-642-33826-7_16
16. Daca, P., Henzinger, T.A., Kupriyanov, A.: Array folds logic. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 230–248. Springer (2016), https://doi.org/10.1007/978-3-319-41540-6_13
17. De Angelis, E., Proietti, M., Fioravanti, F., Pettorossi, A.: Verifying catamorphism-based contracts using constrained Horn clauses. Theory Pract. Log. Program. **22**(4), 555–572 (2022), https://doi.org/10.1017/S1471068422000175
18. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 6015, pp. 280–295. Springer (2010), https://doi.org/10.1007/978-3-642-12002-2_24
19. Ernst, G.: Korn - software verification with Horn clauses (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 559–564. Springer (2023), https://doi.org/10.1007/978-3-031-30820-8_36
20. Esen, Z., Rümmer, P.: TriCera: Verifying C programs using the theory of heaps. In: 2022 Formal Methods in Computer Aided Design, FMCAD 2022, Trento, Italy, October 17 - October 21, 2022 (2022), (To appear)
21. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 259–277. Springer (2019), https://doi.org/10.1007/978-3-030-25540-4_14
22. Filliâtre, J., Gondelman, L., Paskevich, A.: The spirit of ghost code. Formal Methods Syst. Des. **48**(3), 152–174 (2016), https://doi.org/10.1007/s10703-016-0243-x

23. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: Hankin, C., Schmidt, D. (eds.) Proceedings of: Symposium on Principles of Programming Languages (POPL'01). pp. 193–205. ACM (2001), https://doi.org/10.1145/360204.360220
24. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 813–829. Springer (2013), https://doi.org/10.1007/978-3-642-39799-8_57
25. Georgiou, P., Gleiss, B., Kovács, L.: Trace logic for inductive loop reasoning. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 255–263. IEEE (2020), https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_33
26. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015), https://doi.org/10.1007/978-3-319-21690-4_20
27. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: Lahiri, S.K., Wang, C. (eds.) Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11138, pp. 248–266. Springer (2018), https://doi.org/10.1007/978-3-030-01090-4_15
28. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)
29. Henzinger, T.A., Hottelier, T., Kovács, L., Rybalchenko, A.: Aligators for arrays (tool paper). In: Fermüller, C.G., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6397, pp. 348–356. Springer (2010), https://doi.org/10.1007/978-3-642-16242-8_25
30. Hojjat, H., Rümmer, P.: The ELDARICA Horn solver. In: FMCAD 2018. pp. 1–7 (2018), https://doi.org/10.23919/FMCAD.2018.8603013
31. K., H.G.V., Shoham, S., Gurfinkel, A.: Solving constrained Horn clauses modulo algebraic data types and recursive functions. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022), https://doi.org/10.1145/3498722
32. Kahsai, T., Kersten, R., Rümmer, P., Schäf, M.: Quantified heap invariants for object-oriented programs. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017. EPiC Series in Computing, vol. 46, pp. 368–384. EasyChair (2017), https://easychair.org/publications/paper/Pmh
33. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, The Kluwer International Series in Engineering and Computer Science, vol. 523, pp. 175–188. Springer (1999), https://doi.org/10.1007/978-1-4615-5229-1_12
34. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Sene-

gal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010), https://doi.org/10.1007/978-3-642-17511-4_20
35. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: Shin, S.Y., Ossowski, S. (eds.) Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009. pp. 615–622. ACM (2009), https://doi.org/10.1145/1529282.1529411
36. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for Rust programs. ACM Trans. Program. Lang. Syst. **43**(4), 15:1–15:54 (2021), https://doi.org/10.1145/3462205
37. Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free Horn clauses. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 361–382. Springer (2016), https://doi.org/10.1007/978-3-662-53413-7_18
38. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Trans. Comput. Log. **5**(3), 403–435 (2004), https://doi.org/10.1145/1013560.1013562
39. Priya, S., Zhou, X., Su, Y., Vizel, Y., Bao, Y., Gurfinkel, A.: Verifying verified code. In: Hou, Z., Ganesh, V. (eds.) Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12971, pp. 187–202. Springer (2021), https://doi.org/10.1007/978-3-030-88885-5_13
40. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 55–74. IEEE Computer Society (2002), https://doi.org/10.1109/LICS.2002.1029817
41. Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In: Ésik, Z. (ed.) Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4207, pp. 41–57. Springer (2006), https://doi.org/10.1007/11874683_3