Formal Methods in System Design (2026) 68:7
https://doi.org/10.1007/s10703-026-00493-w

RESEARCH

A program instrumentation framework for automatic verification

Jesper Amilon[1] · Zafer Esen[2] · Dilian Gurov[1] · Christian Lidström[4] · Philipp Rümmer[2,3] · Marten Voorberg[1]

Received: 15 March 2024 / Accepted: 22 February 2026
© The Author(s) 2026

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 this equivalent program instead. In this article, we propose *program 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.

Keywords Program instrumentation · Automatic verification · Ghost code

1 Introduction

Program specifications are often written in expressive, high-level languages: for instance, in temporal logic [1], in first-order logic with quantifiers [2], in separation logic [3], or in specification languages that provide extended quantifiers for computing the sum or maximum value of array elements [4, 5]. 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 pos-

Jesper Amilon, Zafer Esen, Dilian Gurov, Christian Lidström, Philipp Rümmer and Marten Voorberg contributed equally to this work.

Extended author information available on the last page of the article

Published online: 20 March 2026

sible 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 [1]; quantified properties can be replaced with non-determinism, ghost variables, or loops [6, 7]; sets used to specify the absence of data-races can be represented using non-deterministically initialized variables [8]. By adding ghost variables and bespoke ghost code to programs [9], 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 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 [10], 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 article 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 can be 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.

Comparison to CAV paper

This article extends our previous work [11] by including new instrumentation operators, now covering the full set of extended quantifiers in the specification language ACSL [4]. Instrumentation operators for extended quantifiers are presented more systematically using monoid homomorphisms. A more comprehensive correctness result is included, where *completeness* in the typical sense is shown for certain classes of programs. Several existing sections of the CAV paper have been extended by adding more thorough explanations or discussions.

1.1 Motivating examples

We illustrate our approach on two simple examples.

1.1.1 Array aggregation

Consider first the program shown in Fig. 1, which operates on a battery pack powering the electric engine of a vehicle. Although this particular implementation is written by us, it is conceptually close to (significantly more complicated) code we encountered in collaborations with automotive companies. The battery pack is represented as a C struct, consisting of an array of batteries (representing the measured voltage at each battery) and a field for the maximum voltage value among the batteries. In the program, the `calc_pack_max` function calculates the maximum value by looping through the array, and in the main function, correctness of `calc_pack_max` is asserted by relying on the `\max` operator. Intuitively, $\texttt{\max}(\texttt{a}, \texttt{l}, \texttt{u})$ extracts the maximum value of the array `a` in the interval $[\texttt{l}, \texttt{u})$.

Although simple in principle, the program in Fig. 1 is non-trivial to handle for automatic approaches. Software model checkers, in particular, tend to infer loop invariants for the program that explicitly refer to all array elements, resulting in high computational complexity. Hardness is increased further when considering larger arrays, or even unbounded arrays, where more compact representations of the loop invariant are needed. In the industry, programs calculating some aggregate measure over an array are ubiquitous, wherefore there is a clear need for verification tools to support aggregation operators like `\max`.

To verify the program, we define the instrumentation operator Ω_{max}, shown in Fig. 2. For the sake of presentation, all instrumentation operators in this article are defined using functional arrays, using `select` and `store` to access and update the fields of an array, respectively. It is clear that the program in Fig. 1 can be rewritten to this style, but we leave

```
1  typedef struct {
2      int batt_volt[10];
3      int max_voltage;  //Maximum voltage value in pack
4  } BATTERY_PACK;
5
6  extern BATTERY_PACK nondet_BATTERY_PACK();
7  BATTERY_PACK bpack;
8
9  void calc_pack_max() {
10   bpack.max_voltage = bpack.batt_volt[0];
11   for (int i = 1; i < 10; i++)
12     if (bpack.batt_volt[i] > bpack.max_voltage)
13       bpack.max_voltage = bpack.batt_volt[i];
14 }
15 void main() {
16   bpack = nondet_BATTERY_PACK(); //non-det initialization
17   calc_pack_max();
18   assert(bpack.max_voltage == \max(bpack.batt_volt, 0, 10));
19 }
```

Fig. 1 Example program calculating the maximum voltage in a battery pack consisting of an array of batteries

Ω_{max} (Instrumentation operator)

G_{max} (Ghost variables)

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

$\texttt{init(ag\_lo)} = \texttt{init(ag\_hi)} = 0, \texttt{init(ag\_max)} = -\infty, \texttt{init(ag\_ar)} = \_$

R_{max} (Rewrite rules)

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

```
if (i == ag_lo - 1 && ag_lo != ag_hi) {
    assert(ag_ar == a);
    ag_lo = ag_lo = i;
    ag_max = max(ag_max, x);
} else if (i == ag_hi && ag_lo != ag_hi) {
    assert(ag_ar == a);
    ag_hi = i;
    ag_max = max(ag_val, x);
} else {
    ag_max = -∞;
    ag_lo = i;
    ag_hi = i + 1;
}
a' = store(a, i, x);
ag_ar = a'
```

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

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

```
if (u <= l) {
  r = −∞;
} else {
  assert(ag_ar == a && l == ag_lo && u == ag_hi);
  r = ag_max;
}
```

I_{max} (Instrumentation invariant)

$\texttt{ag\_lo} = \texttt{ag\_hi} \vee \texttt{ag\_max} = \texttt{\max(ag\_ar, ag\_lo, ag\_hi)}$

Fig. 2 Definition of the instrumentation operator Ω_{max}

this transformation implicit. Like all instrumentation operators, Ω_{max} is comprised of three parts, the *ghost variables*, the *rewrite rules*, and the *instrumentation invariant*. While instrumentation operators for array properties are discussed at length in Sect. 3, we summarise here the key features of Ω_{max} in the context of Fig. 1.

The essential function of Ω_{max} is to induce, by applying the rewrite rules, a program transformation that mitigates the verification tasks. For this, assignments in a program that include the functions `select`, `store`, or `\max` can be rewritten to more complex pieces of code that, in addition to accessing the array, also update the ghost variables of the instrumentation operator. The goal is to rewrite the program in such a way that its correctness can be checked without having matic functto evaluate the probleion `\max`; to this end, the instrumentation operator introduces ghost variables that track the maximum value of the array in an interval $[\texttt{ag\_lo}, \texttt{ag\_hi})$. The rewrite rules have to be applied to some selection of the read/write statements, so that the ghost variables are updated appropriately. For example, if the program reads from position `ag_hi` in the array, then the tracked interval $[\texttt{ag\_lo}, \texttt{ag\_hi})$ can be extended by incrementing the ghost variable `ag_hi` for the upper interval bound, and similarly for the lower bound for accesses at position $\texttt{ag\_lo} - 1$.

Moreover, if the value read is greater than the previously tracked maximum value `ag_max`, this ghost variable is updated as well.

Instrumentation operators do not, for a given program, produce a unique instrumentation, but instead give rise to a set of instrumented programs, which we call the *instrumentation space*. Essentially, the instrumentation space is formed by considering all possible sets of statements to which the rewrite rules can be applied. In this example, we need to rewrite the array reads at lines 10 and 13. Thereafter, we also rewrite the `\max` aggregation in terms of the ghost variables. As manifested in our experiments (Sect. 7), this shifting of verification domain, from array elements to ghost variables, significantly reduces the complexity for automatic verification tools to infer loop invariants, thus allowing programs such as this one to be verified.

The soundness of the instrumentation operator is tied tightly to the instrumentation invariant. 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 program in the instrumentation space.* To maintain their invariants, instrumentation operators use shadow variables that duplicate the values of program variables. For Ω_{max}, the invariant is that, if the ghost variables track some non-empty interval, then the tracked maximum value is indeed the maximum value of the shadow ghost array `ag_ar` in that interval. The connection between the shadow variable and the underlying program variable is established by letting the rewrite rules assert their equality.

1.1.2 Non-linear arithmetic

Our main focus is instrumentation operators for verification of properties over arrays. However, instrumentation operators also generalise to other verification tasks, as illustrated here for the purpose of verifying non-linear arithmetic properties.

Consider the program on the left-hand side of Fig. 3, computing the *triangular numbers* $s_N = (N^2 + N)/2$. For reasons of presentation, the program has been normalised by representing the square $\mathtt{N} * \mathtt{N}$ using an auxiliary variable `NN`. While mathematically simple, verifying the post-condition $\mathtt{s} == (\mathtt{NN} + \mathtt{N})/2$ in the program turns out to be challenging even for state-of-the-art model checkers, since such tools are usually thrown off course by the non-linear term $\mathtt{N} * \mathtt{N}$. 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 a non-linear invariant $0 <= \mathtt{i} <= \mathtt{N} \;\&\&\; 2 * \mathtt{s} == \mathtt{i} * \mathtt{i} + \mathtt{i}$ to be derived for the loop in lines 4–12.

The insight needed to elegantly verify this program is that the value $\mathtt{i} * \mathtt{i}$ can be tracked during the program execution using a ghost variable `x_sq`. An instrumentation operator for this purpose is defined in Fig. 4, instrumenting the program to maintain the relationship $\mathtt{x\_sq} == \mathtt{i} * \mathtt{i}$. The instrumented program is shown in the right-hand side of Fig. 3. Initially, $\mathtt{i} == \mathtt{x\_sq} == 0$, and each time the value of `i` is modified, also the variable `x_sq` is updated accordingly. In particular, 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. With the value $\mathtt{x\_sq} == \mathtt{i} * \mathtt{i}$ available, both the loop invariant and the post-condition turn into formulas over linear arithmetic, and automatic program verification becomes largely straightforward.

```
// Triangular numbers
i = 0; /*A*/ s = 0; /*B*/
assume(N>0);
while(i < N) {




    i = i + 1; /*C*/


    s = s + i;
}


NN = N*N; /*D*/

assert(s == (NN+N)/2);
```

```
// Instrumented program
i=0; s=0; x_sq=0; x_shad=0;
assume(N>0);
while(i < N) {
    // Begin-instrumentation
    assert(i == x_shad);
    x_sq    = x_sq + 2*i + 1;
    i       = i + 1;
    x_shad  = i;
    // End-instrumentation
    s       = s + i;
}
// Begin-instrumentation
assert(N == x_shad);
NN = x_sq;
// End-instrumentation
assert(s == (NN+N)/2);
```

Fig. 3 Program computing triangular numbers, and its instrumented counterpart

Ω_{square} (Instrumentation operator)

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

$\texttt{x\_sq}, \texttt{x\_shad} : \texttt{Int}$

$\texttt{init}(\texttt{x\_sq}) = 0,\ \texttt{init}(\texttt{x\_shad}) = 0$

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

$y = \alpha \quad \rightsquigarrow \quad y = \alpha;\ \texttt{x\_sq} = \alpha^2;\ \texttt{x\_shad} = y$ (R1)

$y = x + \alpha \quad \rightsquigarrow \quad \texttt{assert}(x == \texttt{x\_shad});$
$\texttt{x\_sq} = \texttt{x\_sq} + 2\alpha * x + \alpha^2;\ y = x + \alpha;\ \texttt{x\_shad} = y$ (R2)

$y = \alpha * x \quad \rightsquigarrow \quad \texttt{assert}(x == \texttt{x\_shad});$
$\texttt{x\_sq} = \alpha^2 * \texttt{x\_sq};\ y = \alpha * x;\ \texttt{x\_shad} = y$ (R3)

$y = x * x \quad \rightsquigarrow \quad \texttt{assert}(x == \texttt{x\_shad});\ y = \texttt{x\_sq}$ (R4)

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

$\texttt{x\_sq} = \texttt{x\_shad}^2$

Fig. 4 Instrumentation operator Ω_{square} for tracking squares

1.2 Automatic program instrumentation

While the above examples illustrate the applicability of our instrumentation idea, we are still left with a major challenge, namely how to automatically discover the appropriate program transformation, without having to select manually the subset of statements to which the rewrite rules should be applied. To this end, we split the process of program instrumentation into two parts:

(i) Choosing an *instrumentation operator,* which is defined manually, designed to be program-independent, and inducing a space of possible program transformations.
(ii) Carrying out an automatic *application strategy* to find, among the possible program transformations, one that enables verification of the program at hand.

1.3 Contributions and outline

The two operators shown so far are simple and do not apply to all programs, but they can easily be generalised. The framework presented in this article provides the foundation for developing a library of formally verified instrumentation operators. In the scope of this article, 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 [12] 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 article are: (i) a general *framework for program instrumentation*, which defines a space of program transformations that work by rewriting individual statements (Sects. 2 and 4); (ii) a library of *concrete operators* defined within the framework, obtaining full coverage of the quantifiers (Sect. 3.1) and extended quantifiers (Sect. 3.2) in ACSL [4]; (iii) machine-checked proofs of the correctness of the instrumentation operators for quantifiers and extended quantifiers; (iv) correctness results showing *soundness* and *weak completeness* for all instrumentation operators, and *completeness* for certain classes of programs (Sect. 5); (v) an application strategy *search algorithm* in this space, for a given program (Sect. 6); (vi) a new *verification tool*, MonoCera, that is tailored to the verification of programs with aggregation; and (vii) an *evaluation* of our method and tool on a set of examples, including such from SV-COMP [12] (Sect. 7).

2 Instrumentation framework

We now formally introduce the instrumentation framework. The instrumentation of a program requires two main ingredients:

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 is responsible for selecting the right (if any) program instrumentation from the many possible ones. Section 6 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 pro-

gram, regardless of the chosen application strategy; that is, instrumentation of an incorrect program will never yield a correct program. This aspect is discussed in Sect. 5.

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.

Finally, for certain classes of programs, in Sect. 5.2 we shall guarantee *completeness* in the typical, stricter sense. That is, if a program is correct and adheres to certain conditions, then there is an instrumented program that is also correct, such that automatic verification has been made easier.

2.1 The core language

A simple programming language, containing loops and arrays, is used to present the instrumentation framework.

2.1.1 Syntax

While our implementation works on programs represented as constrained Horn clauses [13], i.e., is language-agnostic, for readability purposes we present our approach in the setting of an imperative core programming language with data-types 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. A notable deviation from C-like, imperative-style arrays is that the equality of two arrays depends on the array contents, and not on a pointer value. 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 [14]:

• Constructing a *constant* array filled with `x`	$\texttt{const}(\texttt{x})$;
• *Reading* the value of an array `a` at index `i`	$\texttt{select}(\texttt{a}, \texttt{i})$;
• *Updating* an array `a` at index `i` with a new value `x`	$\texttt{store}(\texttt{a}, \texttt{i}, \texttt{x})$;
• *Comparing* array `a` and `b` for equality	$\texttt{a} == \texttt{b}$.

The complete syntax of the core language is given in Fig. 5. Programs are written using a vocabulary $\mathcal{X}$ of typed program variables. We use $\alpha(x)$ to denote the type of a variable $x \in \mathcal{X}$. The typing rules of the language are shown in Fig. 6. We use $S : \texttt{wfs}$ to denote that a statement S is well-formed, meaning that it respects the typing rules. As syntactic sugar, we sometimes write $\texttt{a}[\texttt{i}]$ instead of $\texttt{select}(\texttt{a}, \texttt{i})$, and $\texttt{a}[\texttt{i}] = \texttt{x}$ instead of $\texttt{a} = \texttt{store}(\texttt{a}, \texttt{i}, \texttt{x})$. We also use $\texttt{a} = \_$ to mean that `a` is assigned to some constant array, for which we do not care about the value of the elements.

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

Fig. 5 Syntax of the core language

$$
\frac{}{\langle DecimalNumber \rangle : \texttt{Int}} \qquad \frac{}{\texttt{true} : \texttt{Bool}} \qquad \frac{}{\texttt{false} : \texttt{Bool}} \qquad \frac{x \in \mathcal{X} \quad \alpha(x) = \sigma}{x : \sigma}
$$

$$
\frac{s : \sigma \quad t : \sigma}{s \texttt{ == } t : \texttt{Bool}} \qquad \frac{s : \texttt{Int} \quad t : \texttt{Int}}{s \texttt{ <= } t : \texttt{Bool}} \qquad \frac{s : \texttt{Int} \quad t : \texttt{Int}}{s \texttt{ + } t : \texttt{Int}} \qquad \frac{s : \texttt{Int} \quad t : \texttt{Int}}{s \texttt{ * } t : \texttt{Int}}
$$

$$
\frac{s : \texttt{Int} \quad t : \texttt{Int}}{s \texttt{ / } t : \texttt{Int}} \qquad \frac{c : \texttt{Bool}}{\texttt{!}c : \texttt{Bool}} \qquad \frac{s : \texttt{Bool} \quad t : \texttt{Bool}}{s \texttt{ \&\& } t : \texttt{Bool}} \qquad \frac{s : \texttt{Bool} \quad t : \texttt{Bool}}{s \texttt{ || } t : \texttt{Bool}}
$$

$$
\frac{t : \sigma}{\texttt{const}(t) : \texttt{Array } \sigma} \qquad \frac{a : \texttt{Array } \sigma \quad t : \texttt{Int}}{\texttt{select}(a,t) : \sigma} \qquad \frac{a : \texttt{Array } \sigma \quad t : \texttt{Int} \quad s : \sigma}{\texttt{store}(a,t,s) : \texttt{Array } \sigma}
$$

$$
\frac{-}{\texttt{skip} : \texttt{wfs}} \qquad \frac{x : \sigma \quad t : \sigma}{x = t : \texttt{wfs}} \qquad \frac{b : \texttt{Bool} \quad S_1 : \texttt{wfs} \quad S_2 : \texttt{wfs}}{\texttt{if (}b\texttt{)}\ S_1\ \texttt{else}\ S_2 : \texttt{wfs}} \qquad \frac{b : \texttt{Bool} \quad S : \texttt{wfs}}{\texttt{while (}b\texttt{)}\ S : \texttt{wfs}}
$$

$$
\frac{b : \texttt{Bool}}{\texttt{assume}(b) : \texttt{wfs}} \qquad \frac{b : \texttt{Bool}}{\texttt{assert}(b) : \texttt{wfs}} \qquad \frac{S_1 : \texttt{wfs} \quad S_2 : \texttt{wfs}}{S_1\texttt{;}\ S_2 : \texttt{wfs}}
$$

Fig. 6 Typing rules of the core language

2.1.2 Semantics

We assume the Flanagan-Saxe *extended execution model* of programs with `assume` and `assert` statements (see, e.g. [15]), 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. 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 $f : \mathbb{Z} \to D_\sigma$. 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 (but not the left-hand side variable r), 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 conditions 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 $r = t$, in the sense that the Hoare triple $\{I\}\ z = t; s'\ \{z = r\}$, where 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. The conditions can themselves be deductively verified to hold for each concrete instrumentation operator, and this check is *independent* of the programs to be instrumented, so that an instrumentation operator can be proven correct once and for all.

Importantly, an instrumentation operator Ω does itself not define which occurrences of program statements are to be rewritten, but only how they are rewritten.

Definition 2 (Rewriting a program statement). A program statement $v = e$ matches a rewrite rule $r = t \rightsquigarrow s$ if there is a substitution σ, replacing each meta-variable with a program expression, that maps r to $\sigma(r) = v$ and t to $\sigma(t) = e$.

The result of rewriting $v = e$ using the rule $r = t \rightsquigarrow s$ is defined as follows:

(i) If v does not occur in e, the result of rewriting $v = e$ is the program $\sigma(s)$, i.e., the right-hand side s of the rule, with meta-variables substituted with the expressions occurring in the assignment $v = e$.
(ii) If v occurs in e, we first replace $v = e$ with the two statements $v' = e;\ v = v'$, for some fresh variable v' that has the same type as v, and then rewrite the new assignment $v' = e$ using $r = t \rightsquigarrow s$.

The procedure in second case (ii) has the purpose of ensuring that the right-hand side $\mathtt{e}$ is stable in s; otherwise, carrying out the assignment could have the effect of changing the value of expressions in $\mathtt{e}$.

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 $\mathtt{x}_1 = init_1; \ldots; \mathtt{x}_k = init_k$ are added at the beginning of the program, and (ii) some of the assignments in P that match rewrite rules $r = t \rightsquigarrow s$ in Ω are rewritten as defined in Definition 2. We denote by $\Omega(P)$ the set of all instrumented programs P' that can be derived in this way.

Two simplified examples of instrumentation operators were already given in Sect. 1. The reader is advised to return to those examples, and convince herself that the operators indeed satisfy the conditions required in Definition 1.

3 Instrumentation operators for arrays

We will now instantiate our framework for several classes of functions over arrays, targeting, in particular, the different kinds of quantifiers over arrays provided by ACSL [4]. Among others, we will obtain a more general and more precise version of the operator for the extended quantifier $\backslash\mathtt{max}$ from Sect. 1.

3.1 Instrumentation operators for quantification over arrays

To handle quantifiers in a programming setting, we extend the language defined in Fig. 5 by adding quantifier expressions over arrays, as shown in Fig. 7, where Q ranges over $\{\forall, \exists\}$. We shall often use $\backslash\mathtt{forall}$ for $\mathtt{quant}_\forall$, and $\backslash\mathtt{exists}$ for $\mathtt{quant}_\exists$. We also extend the language with a lambda expression over two variables. The rationale is that quantified properties can often be expressed as a binary predicate, with the first argument corresponding to the value of an element and the second to the index.

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

$$[\![\, \mathtt{quant}_Q\,(\mathtt{a}, \mathtt{1}, \mathtt{u}, \lambda\,(\mathtt{x}, \mathtt{i})\,.\mathtt{P})\,]\!]_s \quad = \quad Q\mathtt{i} \in [\mathtt{l}, \mathtt{u})\,.[\![\mathtt{P}\,(\mathtt{a}\,[\mathtt{i}]\,, \mathtt{i})]\!]_s$$

$$\begin{aligned}\langle \mathit{Expr}\rangle \ ::= \ & (\lambda(\langle \mathit{Variable}\rangle, \langle \mathit{Variable}\rangle)\,.\langle \mathit{Expr}\rangle)\ (\langle \mathit{Expr}\rangle,\ \langle \mathit{Expr}\rangle)\ \mid \\ & \mathtt{quant}_Q(\langle \mathit{Expr}\rangle, \langle \mathit{Expr}\rangle, \langle \mathit{Expr}\rangle, \lambda(\langle \mathit{Variable}\rangle, \langle \mathit{Variable}\rangle)\,.\langle \mathit{Expr}\rangle)\end{aligned}$$

Fig. 7 Extension of the core language with quantified expressions

As an example, consider the program in Fig. 8, which uses `forall` to specify that every element in the array should equal its index. To verify this program, we define the instrumentation operator $\Omega_\forall$ in Fig. 9. The operator is similar to Ω_{max}, which was introduced in Sect. 1.1, but, instead of tracking the maximum value, we use the ghost variable `qu_P` to track if the given property P holds for all elements in the tracked interval [`qu_lo`, `qu_hi`). Naturally, an instrumentation operator for existential quantification can be defined in a similar fashion. When applying instrumentation operators for array properties, such as $\Omega_{\forall,P}$, we assume for simplicity 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, `store` is restricted to occur in statements of the form: $\mathtt{a}' = \mathtt{store}(\mathtt{a}, \mathtt{i}, \mathtt{x})$.

The rewrite rules can be justified as follows. For `store`, the first if-branch resets the tracking to the singleton interval $[\mathtt{i}, \mathtt{i}+1)$ 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 $\mathtt{i} = \mathtt{qu\_lo} - 1$), then that element is added to the tracked interval, and the 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 `select` are similar to `store`, 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 $\Omega_{\forall,\lambda(i,x).x=i}$ to the program in Fig. 8 is shown in Fig. 10. In the process of rewriting, we renamed the left-hand side of $\mathtt{a} = \mathtt{store}(\mathtt{a}, \mathtt{i}, \mathtt{i})$ by introducing a fresh variable $\mathtt{a}'$, following case (ii) of Definition 2. As exhibited by the experiments in Sect. 7, the resulting program is, in many cases, easier to handle for state-of-the-art verification tools.

Note that the proposed instrumentation operator is only one possibility among many. For example, one could track simultaneously several ranges 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 store operations outside of the tracked interval.

```
1  Int N = nondet;
2  assume(N > 0);
3  Array Int a = const(0);
4  Int i = 0;
5  while(i < N) {
6      a = store(a, i, i);
7      i = i + 1;
8  }
9  Bool b = \forall(a, 0, N, λ(i,x).(x == i));
10 assert(b);
```

Fig. 8 Example of program to be verified using a quantified assert statement

$\Omega_{\forall,\mathbf{P}}$ (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);` $\rightsquigarrow$

```
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 {               // - might overwrite 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
  }
}
a' = store(a, i, x);
qu_ar = a';
```

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

`b = \forall(a, l, u, λ(x,i).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. 9 Instrumentation operator for universal quantification

3.2 Instrumentation operators for aggregations

We now turn to defining instrumentation operators for any *aggregation* that can be formalised using *monoid homomorphisms*. As we shall show, this not only generalises $\forall$ and $\exists$, but also allows defining instrumentation operators for the aggregations `\sum`, `\product`, `\numof`, `\max`, and `\min`. Such aggregation is supported in the specification languages JML [5] and ACSL [4] in the form of *extended quantifiers*, and is frequently needed for the specification of functional correctness properties. Although such constructs are commonly available in specification languages, most verification tools do not support the verification of properties specified using aggregation. Such properties instead typically have to be manually

```
Int qu_lo = 0; qu_hi = 0; Int qu_ar = []; Bool qu_P = true;
Int N = nondet; assume(N > 0); Array Int a = const(0); Int i = 0;
while(i < N) {
    // Begin-instrumentation
    if (qu_lo == qu_hi || i < qu_lo - 1 || i > qu_hi ||
        (i == i && !qu_P && qu_lo <= i && i < qu_hi)) {
      qu_lo = i; qu_hi = i + 1; qu_P = (i == i);
    } else {
      assert(qu_ar == a); qu_P = qu_P && i == i;
      if (qu_lo - 1 == i) {
          qu_lo = i;
      } else if (qu_hi == i) {
          qu_hi = i + 1;
      }
    }
    a' = store(a, i, i);
    qu_ar = a';
    // End-instrumentation
    a = a'; i = i + 1;
}
Bool b;
// Begin-instrumentation
if (N <= 0) {
  b = true;
} else {
  if (qu_P) {
    assert(qu_ar == a && 0 >= qu_lo && N <= qu_hi);
  } else {
    assert(qu_ar == a && 0 <= qu_lo && N >= qu_hi);
  }
  b = qu_P;
}
// End-instrumentation
assert(b);
```

Fig. 10 Resulting program from applying the instrumentation $\Omega_{\forall,\lambda(i,x).x=i}$ to the program in Fig. 8

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 elegantly handled using our proposed instrumentation framework.

Definition 3 (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 called *commutative* if $\circ$ is commutative.

We model finite intervals of arrays as strings over the monoid $(D^*, \cdot, \epsilon)$ of finite sequences over some data domain D. The concatenation operator is non-commutative. The aggregation or quantification of finite intervals of an array is then defined as a monoid homomorphism from $(D^*, \cdot, \epsilon)$ to some target monoid.

Definition 4 (Monoid Homomorphism). A *monoid homomorphism* is a function $h : M_1 \to M_2$ between monoids $(M_1, \circ_1, e_1)$ and $(M_2, \circ_2, e_2)$ such that $h(x \circ_1 y) = h(x) \circ_2 h(y)$ for all $x, y \in M_1$ and $h(e_1) = e_2$.

For example, universal quantification can be modelled as a homomorphism to the monoid $(\mathbb{B}, \wedge, true)$. A second 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 $-\infty$.[1] A third example is the computation of the element *sum* of an integer sequence, modelled by the homomorphism h_{sum} to the target monoid $(\mathbb{Z}, +, 0)$. The monoid in the last example, the computation of element sums, has the special property of being *cancellative.*

Definition 5 (Cancellative Monoid). A monoid $(M, \circ, e)$ is *cancellative* if one of the following equivalent properties holds:

(1) $x \circ y = x \circ z$ implies $y = z$, and $y \circ x = z \circ x$ implies $y = z$, for all $x, y, z \in M$.
(2) there are partial operations $\circ_l^{-1}, \circ_r^{-1} : M \times M \rightharpoonup M$ such that $(x \circ y) \circ_r^{-1} y = x$ and $x \circ_l^{-1} (x \circ y) = y$, for all $x, y \in M$.

In the case of a *commutative* cancellative monoid, it is not necessary to distinguish two separate functions $\circ_l^{-1}, \circ_r^{-1}$ for left- and right-cancellation. We use the notation $(x \circ y) \circ^{-1} y$ in the commutative case.

Cancellative monoids generalise the notion of a group, and are a useful special case in the context of instrumentation: aggregation operators based on cancellative monoids can be represented using a particularly efficient kind of instrumentation. An example of a cancellative (commutative) monoid is the group $(\mathbb{Z}, +, 0)$ used to model `\sum`, which calculates the sum of the elements in an array.

3.2.1 Programming language with aggregation

We extend our core programming language a second time, this time adding support for aggregation of array values. We introduce the expressions $\texttt{aggregate}_{M,h}(\langle Expr\rangle,\langle Expr\rangle,\langle Expr\rangle)$, and use monoid homomorphisms to formalise them. Recall that we use D_σ to denote the domain of a program type σ.

Definition 6 Let $Array\ \sigma$ 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 $Array\ \sigma$, and l and u be integer expressions. Then, $aggregate_{M,h}(ar,l,u)$ is an expression of type σ_M, defined as:

$$[\![aggregate_{M,h}(ar,l,u)]\!]_s = h(\langle [\![ar]\!]_s([\![l]\!]_s), [\![ar]\!]_s([\![l]\!]_s + 1), \ldots, [\![ar]\!]_s([\![u]\!]_s - 1)\rangle) \quad (1)$$

Intuitively, the expression $\texttt{aggregate}_{M,h}(ar,l,u)$ denotes the result of applying the homomorphism h to the slice $ar[l \,..\, u-1]$ of the array *ar*. As a convention, in case $u < l$ we

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

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 $\texttt{t} = \texttt{aggregate}_{M,h}(ar,l,u)$.

We define aggregation operators corresponding to all extended quantifiers found in ACSL, with the homomorphism h being uniquely defined by how it acts on singleton sequences. For example, for `sum`, the homomorphism h_{sum} is generated by the mapping $\langle n\rangle \mapsto n$. Table 1 shows, for each operator, the homomorphism and its domain monoid.

One may observe here that `aggregate`, in a slightly extended form, subsumes the operator $\texttt{quant}_Q$ previously introduced for ordinary quantifiers. For this, consider a generalised $\texttt{aggregate}'$ operator that passes not only array elements, but also their indexes to a homomorphism:

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

For the quantifier `\forall`, we can then let $M_\forall$ be the monoid $(\mathbb{B}, \wedge, true)$ and $h_\forall$ the homomorphism generated by mapping value-index pairs $\langle(x,i)\rangle$ to the Boolean $P(x,i)$.

3.2.2 An instrumentation operator for non-cancellative monoids

We are now ready to define instrumentation operators that work for any aggregation that is based on a monoid homomorphism. We start by considering the general case of monoids that are not (necessarily) cancellative, and define an instrumentation operator that is *parameterised* on the target monoid $(M, \circ, e)$ and homomorphism $h : D^*_\sigma \to M$, as shown in Fig. 11. The operator can be instantiated to obtain instrumentation operators for any ACSL extended quantifier, with the respective target monoid and homomorphism being as defined in Table 1. In such instances of the parameterised instrumentation operator, $(M, \circ, e)$ and h have to be replaced with their respective programming language implementation.

The operator Ω_{nc} is largely similar to the ones already defined for `\max` and `\forall`, wherefore we refer the reader to Sect. 3.1 for an explanation of the intuition behind the operator. The main difference is that the statements on lines 4 and 8 now use $\circ$ and h.

Theorem 1 *(Correctness of Ω_{nc}). For any target monoid $(M, \circ, e)$ and monoid homomorphism $h : D^*_\sigma \to M$ with implementations of $\circ$ and h that terminate and do not assign to any variables, the instrumentation operator $\Omega_{nc}((M, \circ, e), h)$ is correct, i.e., it adheres to the constraints imposed by Definition 1.*

Table 1 Target monoid and homomorphism generator for all ACSL (extended) quantifiers

(Extended) Quantifier	Target Monoid	h generated by
`\sum(ar,l,u)`	$(\mathbb{Z}, +, 0)$	$\langle x\rangle \mapsto x$
`\min(ar,l,u)`	$(\mathbb{Z}_{+\infty}, min, +\infty)$	$\langle x\rangle \mapsto x$
`\max(ar,l,u)`	$(\mathbb{Z}_{-\infty}, max, -\infty)$	$\langle x\rangle \mapsto x$
`\product(ar,l,u)`	$(\mathbb{Z}, \cdot, 1)$	$\langle x\rangle \mapsto x$
`\numof(ar,l,u,λ(x,i).P)`	$(\mathbb{Z}, +, 0)$	$\langle b\rangle \mapsto b?1:0$
`\forall(ar,l,u,λ(x,i).P)`	$(\mathbb{B}, \wedge, true)$	$\langle(x,i)\rangle \mapsto P(x,i)$
`\exist(ar,l,u,λ(x,i).P)`	$(\mathbb{B}, \vee, false)$	$\langle(x,i)\rangle \mapsto P(x,i)$

$\Omega_{nc}((M, \circ, e), h)$ (Instrumentation operator)

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

`ag_lo, ag_hi : Int, ag_ar : Array` σ`, ag_val :` M

`init(ag_lo) = init(ag_hi) = 0, init(ag_val) =` e`, init(ag_ar) = _`

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

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

```
1  if (i == ag_lo - 1 && ag_lo != ag_hi) {
2      assert(ag_ar == a);
3      ag_lo  = ag_lo = i;
4      ag_val = h(x) ∘ ag_val;
5  } else if (i == ag_hi && ag_lo != ag_hi) {
6      assert(ag_ar == a);
7      ag_hi  = i;
8      ag_val = ag_val ∘ h(x);
9  } else {
10     ag_val = h(x);
11     ag_lo  = i;
12     ag_hi  = i + 1;
13 }
14 a' = store(a, i, x);
15 ag_ar = a'
```

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

`r = \aggregate`$_{M,h}$`(a, l, u)` $\rightsquigarrow$

```
1  if (u <= l) {
2    t = e;
3  } else {
4    assert(ag_ar == a && l == ag_lo && u == ag_hi);
5    t = gh_val;
6  }
```

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

`ag_lo` $\leq$ `ag_hi` $\wedge$ `ag_val` $=$ `aggregate`$_{M,h}$`(ag_ar, ag_lo, ag_hi)`

Fig. 11 Instrumentation operator for aggregation parameterised on a non-cancellative target monoid $(M, \circ, e)$ and homomorphism h

Proof Let $(M, \circ, e)$ and h be such a target monoid and homomorphism. We will show that the instrumentation operator created by implementing $\Omega_{nc}((M, \circ, e), h)$ with terminating implementations of $\circ$ and h that do not assign to any variables is correct.

First, observe that `ag_lo` and `ag_hi` are initialised to the same value. Since the value of aggregation over an empty interval is the monoid identity e, and `ag_val` is initialised to e, the invariant I is established by the initial values assigned by G. It now remains to show that the rewrite rules adhere to the remaining constraints. We will first show that a rewritten `store` terminates and satisfies the constraints in Definition 1. It is easy to see that the instrumentation terminates, since no loop or recursion is introduced and the implementations of $\circ$ and h are assumed to terminate. It is also easy to see that we only assign to ghost variables and `a'`, and not to any other program variables, since $\circ$ and h do not assign to any variables. Thus, the semantics is preserved, since the original `store` statement remains unchanged on line 14 and the instrumentation code only assigns to ghost variables. Lastly, we must establish that the invariant is preserved, i.e., that the new values

of ag_val, ag_hi, ag_lo, and ag_ar satisfy the invariant. For this, we distinguish three cases, each of which corresponds to a conditional block. In the first case, we assume that $\texttt{i} = \texttt{ag\_lo} - 1$ and $\texttt{ag\_lo} \neq \texttt{ag\_hi}$. Since the value of ag_lo is decreased, the first conjunct, $\texttt{ag\_lo} \leq \texttt{ag\_hi}$, is preserved. The following derivation shows that the second conjunct of the invariant, $\texttt{ag\_val} = \texttt{aggregate}_{M,h}(\texttt{ag\_ar}, \texttt{ag\_lo} - 1, \texttt{ag\_hi})$, is also preserved:

$$
\begin{aligned}
\texttt{ag\_val}_{\texttt{new}} &= h(\texttt{x}) \circ \texttt{ag\_val\_old} \\
&= h(\texttt{x}) \circ \texttt{aggregate}_{M,h}(\texttt{ag\_ar}, \texttt{ag\_lo}, \texttt{ag\_hi}) && \{By\ I_{nc}\} \\
&= h(\texttt{x}) \circ h(\langle \texttt{ag\_ar}[\texttt{ag\_lo}], \ldots, \texttt{ag\_ar}[\texttt{ag\_hi} - 1] \rangle) && \{By\ Eq.\ (1)\} \\
&= h(\langle \texttt{x} \rangle \cdot \langle \texttt{ag\_ar}[\texttt{ag\_lo}], \ldots, \texttt{ag\_ar}[\texttt{ag\_hi} - 1] \rangle) && \{h\ homomorphism\} \\
&= h(\langle \texttt{ag\_ar}[\texttt{ag\_lo} - 1], \ \ldots \ , \texttt{ag\_ar}[\texttt{ag\_hi} - 1] \rangle) && \{\texttt{x} = \texttt{ag\_ar}[\texttt{i}] \text{ and } i = \texttt{ag\_lo} - 1\} \\
&= \texttt{aggregate}_{M,h}(\texttt{ag\_ar}, \texttt{ag\_lo} - 1, \texttt{ag\_hi}) && \{By\ Eq.\ (1)\}
\end{aligned}
$$

The second case, in which $\texttt{i} = \texttt{ag\_hi}$ and $\texttt{ag\_lo} \neq \texttt{ag\_hi}$, follows symmetrically. In the last case, in which i is different from ag_hi, ag_lo − 1, or it is the case that $\texttt{ag\_lo} = \texttt{ag\_hi}$, we reset the ghost variables to track the singleton sequence. The invariant clearly holds afterwards.

We omit the case for select statements as it is similar to store.

Lastly, the instrumentation code for aggregate also satisfies the constraints in Definition 1: (a) The instrumentation code obviously terminates. (b) It only assigns to the variable t. (c) Since the ghost variables are not updated, the invariant is trivially preserved. (d) To show that the semantics is preserved, we distinguish two cases. In the case u <= l, we have defined the value of the aggregation $\texttt{aggregate}_{M,h}(\texttt{a}, \texttt{u}, \texttt{l})$ as equal to e in Sect. 3.2.1, and therefore the semantics is preserved. In the other case, the value of $\texttt{aggregate}_{M,h}(\texttt{a}, \texttt{u}, \texttt{l})$ is equal to ag_val by the instrumentation invariant and therefore, by assigning ag_val to t, the semantics is preserved. □

3.2.3 An instrumentation operator for cancellative monoids

When the target monoid is commutative and cancellative, we can track aggregate values faithfully even when storing *inside* of the tracked interval, unlike in the non-cancellative case. Examples of cancellative operators are the aggregates \sum and \numof.

To this end, we have defined the instrumentation operator Ω_c in Fig. 12. This instrumentation operator can be instantiated with a cancellative monoid $(M, \circ, e)$ and a homomorphism h. The operator differs from $\Omega_n c$ in the rewrite rule for store operations, by adding a new else if branch (lines 9–12). Since the monoid is cancellative and commutative, when storing inside of the tracked interval, we first remove the current value at the index being written to using $\circ^{-1}$, and then add the new value using $\circ$. Instrumentation operators for \sum and \numof can be defined in terms of Ω_c, using the definitions in Table 1. The following result establishes correctness.

Theorem 2 *(Correctness of Ω_c). $\Omega_c((M, \circ, e), h]$ is an instrumentation operator, i.e., it adheres to the constraints imposed in Definition 1.*

Proof Similar to the proof of Theorem 1. □

$\Omega_c((M, \circ, e), h)$ (Instrumentation operator)

G_c (Ghost variables)

`ag_lo, ag_hi : Int, ag_ar : Array` σ, `ag_val :` M

`init(ag_lo) = init(ag_hi) =` 0, `init(ag_val) =` e, `init(ag_ar) = _`

R_c (Rewrite rules)

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

```
if (i == ag_lo - 1 && ag_lo != ag_hi) {
    assert(ag_ar == a);
    ag_lo  = i;
    ag_val = ag_val ∘ h(x);
} else if (i == ag_hi  && ag_lo != ag_hi) {
    assert(ag_ar == a);
    ag_hi  = i;
    ag_val = ag_val ∘ h(x);
} else if (ag_lo <= i && i < ag_hi) {
    assert(ag_ar == a);
    ag_val =
      ag_val ∘^{-1} h(select(a, i)) ∘ (h(x));
} else {
    ag_val = h(x);
    ag_lo  = i;
    ag_hi  = i + 1;
}
a' = store(a, i, x);
ag_ar = a'
```

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

`r = \aggregate`$_{M,h}$`(a, l, u)` $\rightsquigarrow$

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

I_c (Instrumentation invariant)

`ag_lo` $\leq$ `ag_hi` $\wedge$ `ag_val` $=$ `aggregate`$_{M,h}$`(ag_ar, ag_lo, ag_hi)`

Fig. 12 Instrumentation operator for aggregation parameterised on a *cancellative* target monoid $(M, \circ, e)$ and homomorphism h

Corollary 1 *There are correct instrumentation operators for all ACSL (extended) quantifiers.*

Proof The non-cancellative cases (`\forall`, `\exists`, `\product`, `\min`, `\max`) follow from Theorem 1, by instantiating $(M, \circ, e)$ and h according to Table 1. The cancellative cases (`\sum`, `\numof`) follow similarly from Theorem 2. □

It is worth pointing out that we view Ω_{nc} and Ω_c as a *base recipe* for how to implement instrumentation operators for aggregation, but that one can further *tailor* the instrumentation to specific aggregations. For example, the instrumentation operator for `\forall` in Fig. 9 specialises the rewriting of the aggregation, utilising that if a property holds for some portion of the array, then it also holds for a subset of that portion.

3.2.4 Cancelling the non-cancellative

Up until now, we have been using the same monoid to define the instrumentation operator, henceforth denoted as $(M_{aggr}, \circ_{aggr}, e_{aggr})$, as we used to define the semantics of the extended quantifier itself. In this section, we create instrumentation operators using a different, *cancellative* monoid, denoted as $(M_{instr}, \circ_{instr}, e_{instr})$, and parameterise our instrumentation operator on another homomorphism $g : M_{instr} \to M_{aggr}$ to map from the instrumentation monoid to the aggregation monoid when instrumenting an assignment using `aggregate`. This means we slightly alter the definition of Ω_c in Fig. 12 to be parameterised on a homomorphism g and we change line 5 of the rewrite rule for $\texttt{r} = \texttt{aggregate}_{M,h}(\texttt{a}, \texttt{u}, \texttt{l})$ to $\texttt{t} = \texttt{g}(\texttt{ag\_val})$.

We can use this technique to create a cancellative instrumentation operator for universal and existential quantification, instead of the non-cancellative operator we defined in Sect. 3.2.2. By noting that an existential quantifier for a predicate P is true for an interval if the number of elements satisfying P in that interval is larger than 0, we can reuse the cancellative monoid we defined for `\numof`. Whenever we instrument an assignment using `\exists`, we use the homomorphism $g_{\exists,P}$ (see Eq. 2) to map the number of elements satisfying the predicate to the result of the existential quantifier. This now allows us to define a stronger instrumentation operator for existential quantification as in Eq. 2.

$$g_{\exists,P}\,(c) \stackrel{\text{def}}{=} \begin{cases} \texttt{true} & \text{if } c > 0 \\ \texttt{false} & \text{otherwise} \end{cases} \tag{2}$$

$$\Omega_{\exists,P} \stackrel{\text{def}}{=} \Omega_c((\mathbb{Z}, +, 0)\,,\, h_{numof,P},\, g_{\exists,P}) \tag{3}$$

A similar technique can be used to create a cancellative operator for `\product`. By noting that multiplication is cancellative over integers if we explicitly exclude 0, it makes sense to define a monoid for multiplication in which the elements are tuples (p, c) where p is the product of the non-zero elements and c is the number of elements equal to zero. This cancellative monoid can then be used to define Ω_{prod}:

$$M_{prod} \stackrel{\text{def}}{=} \left\{(p, c) \,\middle|\, p \in \mathbb{Z}^{\neq 0}\,, c \in \mathbb{N}\right\}$$

$$(p_1,\ c_1) \circ_{prod} (p_2,\ c_2) \stackrel{\text{def}}{=} (p_1 \cdot p_2,\ c_1 + c_2)$$

$$e_{prod} \stackrel{\text{def}}{=} (1,\ 0)$$

$$h_{prod} \text{ homomorphism generated by } \langle x \rangle \mapsto \begin{cases} (x,\ 0) & \text{if } x \neq 0 \\ (1,\ 1) & \text{otherwise} \end{cases}$$

$$g_{prod}\ ((p,\ c)) \stackrel{\text{def}}{=} \begin{cases} p & \text{if } c = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\Omega_{prod} = \Omega_c\left((M_{prod},\ \circ_{prod},\ e_{prod})\,,\ h_{prod},\ g_{prod}\right)$$

3.3 Deductive verification of instrumentation operators

As stated in Sect. 2.2, instrumentation operators may be verified independently of the programs to be instrumented. The operators described in this article, i.e., operators for the (extended) quantifiers listed in Table 1 and the operator in Fig. 4, have been verified in the verification tool Frama-C [16]. The verification checks that the operators adhere to the constraints listed in Definition 1. 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.

4 Application of multiple instrumentation operators

The instrumentation operators considered so far have instrumented programs for a single, specific type of property, such as the Ω_{max} operator for programs containing the `\max` extended quantifier. In many cases this is sufficient, and the appropriate operator can be selected manually or by some simple syntactic analysis of the program to be verified. It is also common, however, that programs have to be rewritten multiple times to obtain an instrumented program that can be verified easily. This happens, for instance, whenever a program contains occurrences of multiple different extended quantifiers, or multiple occurrences of the same quantifier.

There are several ways to handle such cases in our framework. Naturally, instrumentation can be iterated: starting with a program $P = P_0$, a chain $P_0, P_1, P_2, \ldots, P_k$ can be constructed in which each P_{i+1} is obtained by applying some instrumentation operator to P_i. If the final program P_k can successfully be verified, the correctness of $P = P_0$ follows.

Multiple instrumentation operators can also be composed to create new operators, which can instrument a program for many different properties, or many different occurrences of the same type of property, at once.

Definition 7 (Composition of Instrumentation Operators). Two instrumentation operators $\Omega = (G, R, I)$ *and* $\Omega' = (G', R', I')$, with distinct ghost variables, can be composed to create a new instrumentation operator $\Omega'' = \Omega||\Omega'$:

$$\Omega'||\Omega \stackrel{\text{def}}{=} (G \cup G', R \cup R', I \wedge I') \tag{4}$$

The operators having distinct ghost variables means that, if $vars(G)$ is a function returning all the variable names in G (i.e., the first element of each tuple), then $vars(G) \cap vars(G') = \varnothing$. Note that it is possible to compose two operators of the same type, as long as the ghost variable names are substituted with fresh ones.

Lemma 1 *(Correctness of $\Omega||\Omega'$). If $\Omega = (G, R, I)$ and $\Omega' = (G', R', I')$ are both correct instrumentation operators (i.e., they adhere to Definition 1), then the composed instrumentation operator $\Omega||\Omega'$ is also correct, i.e., it adheres to the conditions imposed in Definition 1.*

Proof Let $\Omega = (G, R, I)$, $\Omega' = (G', R', I')$ and $\Omega'' = \Omega||\Omega' = (G'', R'', I'')$.

By Definition 1, I and I' are invariants over only the variables in G and G', respectively. Since the sets of variables defined G and G' are disjoint, then it follows that their initial values also establish $I \wedge I'$.

Since the rewrite rules are not changed, it follows that they still terminate and assign only to the allowed variables.

For the remaining two constraints, we consider the rewrite rules from R (the proof is symmetrical for the rules from R'). Since the rules in R preserve the invariant I and we start from a stronger invariant $I \wedge I'$, it follows I must hold afterwards. Since the rules in R do not mention the variables in G', I' must also still hold, and $I \wedge I'$ is preserved. Similarly, since the rules in R preserve the semantics of the original statement under I, it must also do so under the stronger $I \wedge I'$.

5 Correctness of instrumentation operators

This section explains how we can reason about the *correctness* of instrumentation operators. For this, in Sect. 5.1 we introduce the two notions of *soundness* and *weak completeness,* which are properties that hold by construction for every instrumentation operator. In Sect. 5.2 we show that for certain classes of programs, the instrumentation operators are also *complete* in a stronger sense.

5.1 Soundness and weak completeness

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 8 (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 add existential quantifiers for the ghost variables.

Example 1 To illustrate the back-translation, we return to the instrumentation operator from Fig. 4 and the example program from Fig. 3. The witness produced by our verification tool in this case is the formula:

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

After conjoining the instrumentation invariant $\mathtt{x\_sq} = \mathtt{x\_shad}^2$ and adding existential quantifiers, we obtain an inductive invariant that is sufficient to verify the original program:

$$\begin{aligned}\exists x_{\mathrm{sq}}, x_{\mathrm{shad}}.\ (i = x_{\mathrm{shad}} \wedge x_{\mathrm{sq}} + x_{\mathrm{shad}} = 2s \wedge \\ \mathtt{N} \geqslant \mathtt{i} \wedge \mathtt{N} \geqslant 1 \wedge 2\mathtt{s} \geqslant \mathtt{i} \wedge \mathtt{i} \geqslant 0 \wedge x_{\mathrm{sq}} = x_{\mathrm{shad}}^2)\end{aligned}$$

Definition 9 (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 3 *(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. □

5.2 Completeness

Weak completeness of an instrumentation operator only guarantees that instrumentation does not affect the validity of assertions that are left intact by the instrumentation. A "proper" notion of completeness would state something considerably stronger, namely, that for any correct program, there is a program in the instrumentation space of the operator that is also correct – with the intention that the instrumented program can be automatically verified. This latter aspect can be captured by restricting the operators or instructions that the instrumented program is allowed to contain.

To formalise this notion of completeness, let $\mathcal{A}$ denote a class of programs; $\mathcal{A}$ could, for instance, contain all programs that do not use certain aggregation operators, and which will therefore be handled well by a given back-end solver. For a program P and an instrumentation operator Ω, we then denote by $\Omega_{\mathcal{A}}(P) = \Omega(P) \cap \mathcal{A}$ the set of instrumented versions of P that belong to $\mathcal{A}$.

Definition 10 ($\mathcal{A}$-Relative Completeness). Let Ω be an instrumentation operator, $\mathcal{A}$ a class of programs, and P a program. We say that Ω is complete for P and $\mathcal{A}$ if and only if the correctness of P implies the correctness of some instrumented program $P' \in \Omega_{\mathcal{A}}(P)$:

$$\mathsf{correct}(P) \Rightarrow \exists P' \in \Omega_{\mathcal{A}}(P).\, \mathsf{correct}(P')$$

We denote with $\mathcal{P}_{\Omega_{\mathcal{A}}}$ the class of programs for which Ω is complete for $\mathcal{A}$.
Note that the above notion of $\mathcal{A}$-relative completeness assumes that a single application of an instrumentation operator suffices to eliminate, in the program P, all problematic constructs of the type for which the operator is defined. This, of course, is a strong assumption. In many cases, a single application will not suffice, but a series of applications of the same operator may achieve the task. Fortunately, it is straightforward to generalise the definition of $\mathcal{A}$-relative completeness for this case. Let $\Omega^*(P)$ denote the set of programs that result from applying the instrumentation operator Ω to the program P any finite number of times in a sequence. One can then define $\Omega^*_{\mathcal{A}}(P) = \Omega^*(P) \cap \mathcal{A}$, and adapt Definition 10 accordingly. We do not lose any generality by iteratively applying the *same* operator a finite number of times, since we can leverage Definition 7 to compose multiple, different operators into a single operator that subsumes its parts.

The notion of $\mathcal{A}$-relative completeness can only be of practical value if one can formulate *conditions on programs* that are sufficient for a program to belong to the class $\mathcal{P}_{\Omega_{\mathcal{A}}}$. The approach we take here is to formulate, for concrete instrumentation operators Ω and program classes $\mathcal{A}$, conditions on programs P that ensure that there is a program P' in the instrumentation space $\Omega_{\mathcal{A}}(P)$ in which none of the assertions added to P by the instrumentation fails. By virtue of weak completeness, Ω must then be complete for P and $\mathcal{A}$.

5.2.1 Completeness conditions for non-cancellative operators

As the base case, we focus on the class $\mathcal{A}_B$ of programs in our core language that do not contain any aggregation expressions. Completeness with respect to the class $\mathcal{A}_B$ captures the ability of an operator to *eliminate* aggregation expressions from a program; the resulting

program can then be processed by many existing software verification tools, which may be unable to handle aggregation, but are otherwise fully automatic.

Definition 11 (Class $\mathcal{P}_{nc}((M,\circ,e),h)$). For given target monoid $(M,\circ,e)$ and monoid homomorphism $h : D_\sigma^* \to M$, we define $\mathcal{P}_{nc}((M,\circ,e),h)$ as the class of programs P satisfying the following conditions:

1. The program P contains exactly one statement $\mathtt{r} = \backslash\mathtt{aggregate}_{\mathtt{M},\mathtt{h}}(\mathtt{a},\mathtt{l},\mathtt{u})$ involving aggregation, for an array variable $\mathtt{a} : \mathtt{Array}\ \sigma$.
2. In any execution of P, prior to aggregation the array elements $\mathtt{a}[\mathtt{i}]$ between $\mathtt{l}$ and $\mathtt{u}-1$ are either read using $\mathtt{select}(\mathtt{a},\mathtt{i})$ or updated with $\mathtt{a} = \mathtt{store}(\mathtt{a},\mathtt{i},\mathtt{x})$.
3. These selects and stores occur sequentially, starting at $\mathtt{l}$ and ending at $\mathtt{u}-1$.
4. No other accesses or assignments to $\mathtt{a}$ occur in P.

It should be pointed out that the class $\mathcal{P}_{nc}$ is a *semantic* class; in fact, it is not decidable whether a given program belongs to $\mathcal{P}_{nc}$ or not, since conditions 2–4 cannot be checked effectively in the general case. The criteria are still useful to characterise cases in which the non-cancellative instrumentation operator can succeed. For instance, our motivating example in Fig. 1 satisfies our conditions, which tells us that we successfully apply the instrumentation operator Ω_{max} to obtain an instrumented program without extended quantifiers.

Theorem 4 *(Completeness of $\mathcal{P}_{nc}((M,\circ,e),h)$). For a given target monoid $(M,\circ,e)$ and monoid homomorphism $h : D_\sigma^* \to M$, let $\Omega = \Omega_{nc}((M,\circ,e),h)$. Then, Ω is complete for every program in the class $\mathcal{P}_{nc}((M,\circ,e),h)$ and $\mathcal{A}_B$, i.e., $\mathcal{P}_{nc}((M,\circ,e),h) \subseteq \mathcal{P}_{\Omega_{\mathcal{A}_B}}$.*

Proof Let $(M,\circ,e)$ be a target monoid, $h : D_\sigma^* \to M$ be a monoid homomorphism and $\Omega = \Omega_{nc}((M,\circ,e),h)$. Let P be a program satisfying the conditions of Definition 11, i.e., let $P \in \mathcal{P}_{nc}$. Assume that P is correct, i.e., that $\mathsf{correct}(P)$ holds.

Let $P' \in \Omega(P)$ be the instrumented program in which the statement $\mathtt{r} = \backslash\mathtt{aggregate}_{\mathtt{M},\mathtt{h}}(\mathtt{a},\mathtt{l},\mathtt{u})$ and all select and store operations have been rewritten. Clearly, $P' \in \mathcal{A}_B$, and hence $P' \in \Omega_{\mathcal{A}_B}(P)$.

Next assume, for the sake of obtaining a contradiction, that P' is not correct, i.e., that $\neg\mathsf{correct}(P')$ holds. By virtue of the definition of Ω (see Fig. 11), and by weak completeness of Ω, one of the following assertions, added by the instrumentation, must have failed:

1. an $\mathtt{assert}(\mathtt{ag\_ar} == \mathtt{a})$ in an instrumentation of `store`,
2. an $\mathtt{assert}(\mathtt{ag\_ar} == \mathtt{a})$ in an instrumentation of `select`, or
3. an $\mathtt{assert}(\mathtt{ag\_ar} == \mathtt{a}\ \&\&\ \mathtt{l} == \mathtt{ag\_lo}\ \&\&\ \mathtt{h} == \mathtt{ag\_hi})$.

The next step of the proof is a case analysis on each of the 5 conjuncts. We show here one case, as the other cases follow similarly.

Assume that the assertion in an instrumented `store` statement failed. This means that the ghost array and actual array must be unequal. Since the interval is non-empty whenever $\mathtt{ag\_ar} == \mathtt{a}$, this means that there must be some assignment to $\mathtt{a}$ that does not update the ghost variable. The former contradicts our choice of P' that says that all store and select

operations to a have been rewritten. Since the interval is non-empty and we only increase the size of the interval through store or select statements, the latter implies that there is an assignment to a after a store or select, which contradicts condition 4.

Hence, P' must be correct, i.e., $\mathsf{correct}(P')$ holds, and therefore, by Definition 10, $P \in \mathcal{P}_{\Omega_{\mathcal{A}_B}}$. □

5.2.2 Completeness conditions for cancellative operators

We can give a similar characterisation as in Definition 11 also for cancellative operators. As in the previous section, let $\mathcal{A}_B$ be the programs that do not contain aggregation expressions.

Definition 12 (Class $\mathcal{P}_c((M, \circ, e), h)$). For given cancellative and commutative target monoid $(M, \circ, e)$ and monoid homomorphism $h : D_\sigma^* \to M$, we define $\mathcal{P}_c((M, \circ, e), h)$ as the class of programs P satisfying the following conditions:

1. The program P contains exactly one statement $\mathtt{r} = \backslash\mathtt{aggregate}_{\mathtt{M},\mathtt{h}}(\mathtt{a}, \mathtt{l}, \mathtt{u})$ involving aggregation, for an array variable $\mathtt{a} : \mathtt{Array}\,\sigma$.
2. In any execution of P, prior to aggregation, each index i between 1 and $\mathtt{u} - 1$ is either read using $\mathtt{select}(\mathtt{a}, \mathtt{i})$ or written to using $\mathtt{a} = \mathtt{store}(\mathtt{a}, \mathtt{i}, \mathtt{x})$. We distinguish the first operation that writes to a specific index from all subsequent operations and call the first operation that writes to a specific index a *frontier-expanding operation*.
3. The indices that are accessed by frontier-expanding operations occur sequentially, starting at index 1 and ending at $\mathtt{u} - 1$.
4. All other accesses to a only operate on indices which have already been accessed by frontier-expanding operations.

Theorem 5 *(Completeness of $\mathcal{P}_c((M, \circ, e), h)$). For given cancellative and commutative target monoid $(M, \circ, e)$ and monoid homomorphism $h : D_\sigma^* \to M$, let $\Omega = \Omega_c((M, \circ, e), h)$. Then, Ω is complete for the class $\mathcal{P}_c((M, \circ, e), h)$ and $\mathcal{A}_B$, i.e., $\mathcal{P}_c((M, \circ, e), h) \subseteq \mathcal{P}_{\Omega_{\mathcal{A}_B}}$.*

Proof The proof of Theorem 5 follows closely that of Theorem 4. □

6 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 again that we are given an oracle correct 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 all functions, like aggregation, we are trying to eliminate by instrumentation; for instance, in Fig. 3, the oracle can over-approximate the term $\mathtt{N} * \mathtt{N}$ 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 Fig. 3, for instance, the choices could be defined by $Q(\mathtt{A}) = Q(\mathtt{B}) = \{(\mathrm{R1}), \perp\}$, $Q(\mathtt{C}) = \{(\mathrm{R2}), \perp\}$,

and $Q(\texttt{D}) = \{(\text{R4}), \bot\}$, referring to the rules in Fig. 4. 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 $ins_r(p)$.

Algorithm 1 Counterexample-guided instrumentation search

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

```
1  begin
2  |  Cand ← R;
3  |  while Cand ≠ ∅ do
4  |  |  pick r ∈ Cand;
5  |  |  if correct(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(p) = r'(p) for all p ∈ C'};
14 |  |  |  end
15 |  |  end
16 |  end
17 |  return Inconclusive;
18 end
```

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, we remove from $Cand$ 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.

Lemma 2 *(Correctness of Algorithm 1). Algorithm 1 is sound and, in a certain sense, complete:*

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

Proof Algorithm 1 will return an instrumentation r when it has derived that P_r is correct; due to the soundness of instrumentation operators, then also P is correct.

Algorithm 1 will return *Incorrect* only when it has discovered a counterexample for P_r that ends in a failing assertion that also occurs in P, i.e., that has not been introduced as a part of instrumentation. Due to the weak completeness of instrumentation operators, then also P is incorrect.

Assuming that there is an r such that P_r is correct, the correctness also of P follows. Algorithm 1 can then not return the result *Incorrect*. To see that Algorithm 1 will eventually find some instrumentation r' such that $P_{r'}$ is correct, note that the algorithm will in lines 12–13 only eliminate instrumentations r'' such that $P_{r''}$ is incorrect. □

7 Evaluation

To evaluate our instrumentation framework, we have implemented the instrumentation operators for quantifiers and aggregation over arrays. We evaluate this implementation on benchmarks taken from SV-COMP, as well as a set of C programs authored by us that use `\min`, `\max`, `\sum`, and `\forall`.

7.1 Implementation

Our implementation is done in the setting of constrained Horn clauses (CHCs), by adding the rewrite rules defined in Sect. 3 to Eldarica [17], 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) [18], Korn (C) [19], RustHorn (Rust) [20], SeaHorn (C/LLVM) [21], and TriCera (C) [22].

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 C programs and to encode them as part of the translation into CHCs. We call the resulting tool chain MonoCera. An artefact that includes MonoCera and the benchmarks is available online [23].

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

7.2 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 CPA-checker 2.1.1 [24], SeaHorn 10.0.0 [21] 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 Fig. 8, if they are provided with a manually written loop invariant. Since MonoCera relies on automatic techniques for invariant inference, we only benchmark against tools using similar automatic techniques. We also excluded VeriAbs [25], 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 [14] in order to model C arrays, and a mathematical (as opposed to machine-based) semantics of integers. CPAchecker was configured to use k-induction [26], 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.

7.2.1 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 2. The benchmark suite contains programs ranging from 16 to 117 LOC, and is comprised of two parts: (i) 117 programs taken from the SV-COMP repository [27], 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 of the shape:

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

can be rewritten using `\forall` or `\max`. 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.

Table 2 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

	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

In (ii), we crafted 12 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.

7.2.2 Results

In Table 2, 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). The "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. We also provide the results per benchmarks for each tool in an accompanying technical report [28].

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. This is because only MonoCera could infer the quantified invariants needed to prove unbounded safety of those benchmarks.

8 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. [1, 6–8]). 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 [9]. 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 [29]. To the best of our knowledge, we are presenting the first general framework for automatic program instrumentation.

Much 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 [6, 7]. 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 [30], syntax-guided synthesis [31], learning [32], by solving recurrence equations [33], backward reachability [29], or superposition [34]. 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 article 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 [36]; those proof rules can be applied manually to verify human-written invariants. The Frama-C system [16] can parse ACSL extended quantifiers [4], but, to the best of our knowledge, none of the Frama-C plugins can automatically process such quantifiers. Other systems, e.g., Dafny [37], 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 [38, 39]. 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) [40] 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 [42]. 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.

9 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.

Acknowledgements We want to thank the anonymous reviewers for helpful feedback. This work has been partially funded by the Swedish Vinnova FFI Programme under grant 2021-02519, the Swedish Research Council (VR) under grant 2021-06327, 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," and for the support through the COST action CA20111 EuroProofNet.

Author contributions All authors contributed equally to this work.

Funding Open access funding provided by Royal Institute of Technology.

Data availability Implementation and data used in the experiments is available on: https://zenodo.org/records/7875416.

Declarations

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, 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 licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence 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. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Clarke EM, Henzinger TA, Veith H et al. (eds) (2018) Handbook of model checking. Springer. https://doi.org/10.1007/978-3-319-10575-8
2. Harrison J (2009) Handbook of practical logic and automated reasoning. Cambridge University Press
3. Reynolds JC (2002) 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. IEEE Computer Society, pp 55–74. https://doi.org/10.1109/LICS.2002.1029817
4. Baudin P, Filliâtre JC, Marché C et al. (2024) ACSL: ANSI/ISO C specification language. Tech. Rep., CEA-List, https://frama-c.com/download/acsl.pdf, Inria
5. Leavens GT, Baker AL, Ruby C (1999) 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. Springer, pp 175–188. https://doi.org/10.1007/978-1-4615-5229-1_12
6. Bjørner NS, McMillan KL, Rybalchenko A (2013) 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. Springer, pp 105–125. https://doi.org/10.1007/978-3-642-38856-9_8
7. Monniaux D, Gonnord L (2016) 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. Springer, pp 361–382. https://doi.org/10.1007/978-3-662-53413-7_18
8. Donaldson AF, Kroening D, Rümmer P (2010) 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. Springer, pp 280–295. https://doi.org/10.1007/978-3-642-12002-2_24

9. Filliâtre J, Gondelman L, Paskevich A (2016) The spirit of ghost code. Formal Methods Syst Des 48(3):152–174. https://doi.org/10.1007/s10703-016-0243-x
10. Priya S, Zhou X, Su Y et al. (2021) 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. Springer, pp 187–202. https://doi.org/10.1007/978-3-030-88885-5_13
11. Amilon J, Esen Z, Gurov D et al. (2023) Automatic program instrumentation for automatic verification. In: Enea C, Lal A (eds) Computer aided verification. Springer Nature, Switzerland, Cham, pp 281–304. https://doi.org/10.1007/978-3-031-37709-9_14
12. Beyer D (2022) 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. Springer, pp 375–402. https://doi.org/10.1007/978-3-030-99527-0_20
13. Bjørner N, Gurfinkel A, McMillan KL et al. (2015) Horn clause solvers for program verification. In: Beklemishev LD, Blass A, Dershowitz N et al. (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. Springer, pp 24–51. https://doi.org/10.1007/978-3-319-23534-9_2
14. Barrett C, Fontaine P, Tinelli C (2017) The SMT-LIB standard: version 2.6. Tech. Rep., Department of Computer Science, The University of Iowa. Available at https://www.SMT-LIB.org
15. Flanagan C, Saxe JB (2001) Avoiding exponential explosion: generating compact verification conditions. In: Hankin C, Schmidt D (eds) Proceedings of: Symposium on Principles of Programming Languages (POPL'01), ACM, pp 193–205. https://doi.org/10.1145/360204.360220
16. Cuoq P, Kirchner F, Kosmatov N et al. (2012) 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. Springer, pp 233–247. https://doi.org/10.1007/978-3-642-33826-7_16
17. Hojjat H, Rümmer P (2018) The ELDARICA horn solver. In FMCAD 2018, pp 1–7. https://doi.org/10.23919/FMCAD.2018.8603013
18. Kahsai T, Kersten R, Rümmer P et al. (2017) 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. EasyChair, pp 368–384. https://easychair.org/publications/paper/Pmh
19. Ernst G (2023) 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. Springer, pp 559–564. https://doi.org/10.1007/978-3-031-30820-8_36
20. Matsushita Y, Tsukada T, Kobayashi N (2021) RustHorn: CHC-based verification for rust programs. ACM Trans Program Lang Syst 43(4):15:1–15:54. https://doi.org/10.1145/3462205
21. Gurfinkel A, Kahsai T, Komuravelli A et al. (2015) The SeaHorn verification framework. In: Kroening D, Pasareanu CS (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. Springer, pp 343–361. https://doi.org/10.1007/978-3-319-21690-4_20
22. Esen Z, Rümmer P (2022) Tricera: verifying C programs using the theory of heaps. In: Griggio A, Rungta N (eds) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022, IEEE, pp 380–391. https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_45
23. Amilon J, Esen Z, Gurov D et al. (2023) Artifact for the CAV 2023 paper "Automatic program instrumentation for automatic verification". https://doi.org/10.5281/zenodo.7875416
24. Beyer D, Keremoglu ME (2011) 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. Springer, pp 184–190. https://doi.org/10.1007/978-3-642-22110-1_16
25. Afzal M, Chakraborty S, Chauhan A et al. (2020) 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. Springer, pp 383–387. https://doi.org/10.1007/978-3-030-45237-7_25

26. Beyer D, Dangl M, Wendler P (2015) Boosting k-induction with continuously-refined invariants. In: Kroening D, Pasareanu CS (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. Springer, pp 622–640. https://doi.org/10.1007/978-3-319-21690-4_42
27. Beyer D (2022) SV-Benchmarks: benchmark set for software verification and testing (SV-COMP 2022 and test-comp 2022). https://doi.org/10.5281/zenodo.5831003
28. Amilon J, Esen Z, Gurov D et al. (2024) A program instrumentation framework for automatic verification. https://arxiv.org/abs/2412.06431, 2412.06431
29. Alberti F, Bruttomesso R, Ghilardi S et al. (2012) Lazy abstraction with interpolants for arrays. In: Bjørner NS, 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. Springer, pp 46–61. https://doi.org/10.1007/978-3-642-28717-6_7
30. Gurfinkel A, Shoham S, Vizel Y (2018) Quantifiers on demand. In: Lahiri SK, 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. Springer, pp 248–266. https://doi.org/10.1007/978-3-030-01090-4_15
31. Fedyukovich G, Prabhu S, Madhukar K et al. (2019) 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. Springer, pp 259–277. https://doi.org/10.1007/978-3-030-25540-4_14
32. Garg P, Löding C, Madhusudan P et al. (2013) 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. Springer, pp 813–829. https://doi.org/10.1007/978-3-642-39799-8_57
33. Henzinger TA, Hottelier T, Kovács L et al. (2010) Aligators for arrays (tool paper). In: Fermüller CG, 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. Springer, pp 348–356. https://doi.org/10.1007/978-3-642-16242-8_25
34. Georgiou P, Gleiss B, Kovács L (2020) Trace logic for inductive loop reasoning. In 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020, IEEE, pp 255–263. https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_33
35. Leino KRM, Monahan R (2009) Reasoning about comprehensions with first-order SMT solvers. In: Shin SY, Ossowski S (eds) Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009, ACM, pp 615–622. https://doi.org/10.1145/1529282.1529411
36. Ahrendt W, Beckert B, Bubel R et al. (2016) Deductive software verification - the KeY book - from theory to practice. In Lecture Notes in Computer Science, vol 10001. Springer. https://doi.org/10.1007/978-3-319-49812-6
37. Leino KRM (2010) Dafny: an automatic program verifier for functional correctness. In: Clarke EM, Voronkov A (eds) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers, Lecture Notes in Computer Science, vol 6355. Springer, pp 348–370. https://doi.org/10.1007/978-3-642-17511-4_20
38. De Angelis E, Proietti M, Fioravanti F et al. (2022) Verifying catamorphism-based contracts using constrained horn clauses. Theory Pract Log Program 22(4):555–572. https://doi.org/10.1017/S1471068422000175
39. Hgv K, Shoham S, Gurfinkel A (2022) Solving constrained horn clauses modulo algebraic data types and recursive functions. Proc ACM Program Lang 6(POPL):1–29. https://doi.org/10.1145/3498722
40. Daca P, Henzinger TA, Kupriyanov A (2016) 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. Springer, pp 230–248. https://doi.org/10.1007/978-3-319-41540-6_13
41. Segoufin L (2006) 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. Springer, pp 41–57. https://doi.org/10.1007/11874683_3
42. Neven F, Schwentick T, Vianu V (2004) Finite state machines for strings over infinite alphabets. ACM Trans Comput Log 5(3):403–435. https://doi.org/10.1145/1013560.1013562

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Jesper Amilon[1] · Zafer Esen[2] · Dilian Gurov[1] · Christian Lidström[4] · Philipp Rümmer[2,3] · Marten Voorberg[1]

✉ Jesper Amilon
jamilon@kth.se

Zafer Esen
zafer.esen@it.uu.se

Dilian Gurov
dilian@kth.se

Christian Lidström
clidstrom@fbk.eu

Philipp Rümmer
philipp.ruemmer@it.uu.se

Marten Voorberg
voorberg@kth.se

[1] EECS, KTH Royal Institute of Technology, Stockholm, Sweden

[2] IT Department, Uppsala University, Uppsala, Sweden

[3] FIDS, University of Regensburg, Regensburg, Germany

[4] FM, Fondazione Bruno Kessler, Trento, Italy