Lemmaless Induction in Trace Logic

Ahmed Bhayat[1], Pamina Georgiou[2], Clemens Eisenhofer[2], Laura Kovács[2], and Giles Reger[1]

[1] University of Manchester, Manchester, UK
[2] TU Wien, Vienna, AT

Abstract. We present a novel approach to automate the verification of first-order inductive program properties capturing the partial correctness of imperative program loops with branching, integers and arrays. We rely on trace logic, an instance of first-order logic with theories, to express first-order program semantics by quantifying over program execution timepoints. Program verification in trace logic is translated into a first-order theorem proving problem where, to date, effective reasoning has required the introduction of so-called trace lemmas to establish inductive properties. In this work, we extend trace logic with generic induction schemata over timepoints and loop counters, reducing reliance on trace lemmas. Inferring and proving loop invariants becomes an inductive inference step within superposition-based first-order theorem proving. We implemented our approach in the RAPID framework, using the first-order theorem prover VAMPIRE. Our extensive experimental analysis shows that automating inductive verification in trace logic is an improvement compared to existing approaches.

1 Introduction

Automating the verification of programs containing loops and recursive data structures is an ongoing research effort of growing importance. While different techniques for proving the correctness of such programs are in place [5, 6, 10, 13], most existing tools in this realm are heavily based on *satisfiability modulo theories* (SMT) backends [4, 8] that come with strong theory reasoning but have limitations in quantified reasoning. In contrast, first-order theorem provers enable quantified reasoning modulo theories [20, 27, 28], such as linear integer arithmetic and arrays. First-order reasoning can thus complement the aforementioned verification efforts when it comes to proving program properties with complex quantification, as evidenced in our original work on the RAPID framework [11] which utilised the VAMPIRE theorem prover [2, 21].

At a high level, the RAPID framework [11] works by translating a program into *trace logic*, adding a number of ad hoc trace lemmas, asserting a desired property, and then running an automated theorem prover on the result. The effectiveness of this approach depends on the underlying trace lemmas. This paper focuses on building induction support into the VAMPIRE theorem prover to reduce reliance on these lemmas.

To understand the role of these trace lemmas (and therefore, what support must be added to the theorem prover) we briefly overview trace logic and the RAPID framework in a little more detail. Trace logic is an instance of first-order logic with theories, such that the program semantics of imperative programs with loops, branching, integers, and arrays can be directly encoded in trace logic. A key feature of this encoding is tracking program executions by quantifying over execution *timepoints* (rather than only over single states), which may themselves be parameterised by *loop iterations*. In principle, we can check whether a translated program entails the desired property in trace logic using an automated theorem prover for first-order logic. In our case, we make use of the saturation-based theorem prover VAMPIRE which implements the superposition calculus [3]. However, a straightforward use of theorem proving often fails in establishing validity of program properties in trace logic, as the proof requires some specific induction, in general not supported by superposition-based reasoning.

In our previous work [11], we overcame this challenge by introducing so-called *trace lemmas* capturing common patterns of inductive loop properties over arrays and integers. Inductive loop reasoning in trace logic is then achieved by generating and adding trace lemma instances to the translated program. However, there are two significant limitations to using trace lemmas:

1. Trace lemmas capture inductive patterns/templates that need to be manually identified, as induction is not expressible in first-order logic. As such, they cannot be inferred by a first-order reasoner, implying that the effectiveness of trace logic reasoning depends on the expressiveness of manually supplied trace lemmas.
2. When instantiating trace lemmas with appropriate inductive program variables, a large number of inductive properties are generated, causing saturation-based proof search to diverge and fail to find program correctness proofs in reasonable time.

In this paper we address these limitations by reducing the need for trace lemmas. We achieve this by introducing a couple of novel induction inferences. Firstly, *multi-clause goal induction* which applies induction in a goal oriented fashion as many safety program assertions are structurally close to useful loop invariants. Secondly, *array mapping induction* which covers certain cases where the required loop invariant does not stem from the goal. Specifically, we make the following contributions:

Contribution 1. We introduce two new inference rules, *multi-clause goal* and *array mapping* induction, for *lemmaless induction* over loop iterations (Sections 5–6). The inference rules are compatible with any saturation-based inference system used for first-order theorem proving and work by carrying out induction on terms corresponding to final loop iterations.

Contribution 2. We implemented our approach in the first-order theorem prover VAMPIRE [21]. Further, we extended the RAPID framework [11] to support inductive reasoning in the automated backend (Section 7). We carry out an extensive evaluation of the new method (Section 8) comparing against state-of-the-art approaches SEAHORN [12, 13] and VAJRA/DIFFY [5, 6].

2 Motivating Example

We motivate our work with the example program in Figure 1. The program iterates over two arrays `a` and `b` of arbitrary, but fixed length `length` and copies array elements into a new array `c`. Each even position in `c` contains an element of `a`, while each odd position an element of `b`. Our task is to prove the safety assertion at line 14: at the end of the program, every element in `c` is an element from `a` or `b`. This property involves (i)

```
func main() {
  const Int[] a;
  const Int[] b;
  Int[] c;
  const Int length;
  Int i = 0;

  while (i < length) {
    c[2*i] = a[i]
    c[(2*i) + 1] = b[i]
    i = i + 1;
  }
}
assert (∀pos𝕀.∃l𝕀.((0 ≤ pos < (2 × length))
   → c(main_end, pos) = a(l) ∨ c(main_end, pos) = b(l)))
```

Fig. 1: Copying elements from arrays `a` and `b` to even/odd positions in array `c`.

alternation of quantifiers and (ii) is expressed in the first-order theories of linear integer arithmetic and arrays. Note that in the safety assertion, the program variable `length` is modeled as a logical constant of the same name of sort integer, whilst the constant arrays `a` and `b` are modeled as logical functions from integers to integers. The mutable array variable `c` is additionally equipped with a timepoint argument `main_end`, indicating that the assertion is referring to the value of the variable at the end of program execution.

Proving the correctness of this example program remains challenging for most state-of-the-art approaches, such as [5, 6, 10, 12], mainly due to the complex quantified structure of our assertion. Moreover, it cannot be achieved in the current RAPID framework either, as existing trace lemmas do not relate the values of multiple program variables, notably equality over multiple array variables. In fact, to automatically prove the assertion, we need an inductive property/trace lemma formalizing that each element at an even position in `c` is an element of `a` or `b` at each valid loop iteration, thereby also restricting the bounds of the loop counter variable `i`. Naïvely adding such a trace lemma would be highly inefficient as automated generation of verification conditions would introduce many instances that are not required for the proof.

3 Related Work

Most of recent research in verifying inductive properties of array-manipulating programs focuses on quantified invariant generation and/or is mostly restricted to proving universally quantified program properties. The works [10, 13] generate universally quantified inductive invariants by iteratively inferring and strengthening candidate invariants. These methods use SMT solving and as such are restricted to first-order theories with a finite model property. Similar logical restrictions also apply to [26], where linear recurrence solving is used in combination with array-specific proof tactics to prove quantified program properties. A related approach is described in [6], where relational invariants instead of recurrence equations are used to handle universal and quantifier-free inductive properties. Unlike these works, our work is not limited to universal invariants but can both infer and prove inductive program properties with alternations of quantifiers.

With the use of extended expressions and induction schemata, our work shares some similarity with template-based approaches [17, 22, 29]. These works [17, 22, 29] infer and prove universal inductive properties based on Craig interpolation, formula slicing and/or SMT generalizations over quantifier-free formulas. Unlike these works, we do not require any assumptions on the syntactic shape of the first-order invariants. Moreover, our invariants are not restricted to the shape of our induction schemata. Rather, we treat inductive (invariant) inferences as additional rules of first-order theorem provers, maintaining thus the efficient handling of arbitrary first-order quantifiers. Our framework can be used in arbitrary first-order theories, even with theories that have no interpolation property and/or a finite axiomatization, as exemplified by our experimental results using inductive reasoning over arrays and integers.

Inductive theorem provers, such as ACL2 [18] and HipSpec [7], implement powerful induction schemata and heuristics. However these provers, to the best of our knowledge, automate inductive reasoning for only universally quantified inductive formulas using a goal/subgoal architecture, for which user-guidance is needed to split conjectures into subgoals. In contrast, our work can prove formulas of full first-order logic by integrating and fully automating induction in saturation-based proof search. By combining induction with saturation, we allow these techniques to interleave and complement each other, something that pure induction provers cannot do. Unlike tools such as Dafny [23], our approach is fully automated requiring no user annotations.

First-order theorem proving has previously been used to derive invariants with alternations of quantifiers in our previous work [11]. Our current work generalizes the inductive capabilities of [11] by reducing the expert knowledge of [11] in introducing inductive lemmas to guide the process of proving inductive properties.

4 Preliminaries

Many-Sorted First-Order Logic. We consider standard many-sorted first-order logic with built-in equality, denoted by $\simeq$. By $s = F[u]$ we indicate that the term u is a subterm of s surrounded by (a possibly empty) context F.

We use x, y to denote variables, l, r, s, t for terms and sk for Skolem symbols. A *literal* is an atom A or its negation $\neg A$. A *clause* is a disjunction of literals $L_1 \vee ... \vee L_n$, for $n \geq 0$. Given a formula F, we denote by CNF(F) the clausal normal form of F.

For a logical variable x of sort S we write x_S. A *first-order theory* denotes the set of all valid formulas on a class of first-order structures. Any symbol in the signature of a theory is considered *interpreted*. All other symbols are *uninterpreted*. In particular, we use the theory of linear integer arithmetic denoted by $\mathbb{I}$ and the boolean sort $\mathbb{B}$. We consider natural numbers as the term algebra $\mathbb{N}$ with four symbols in the signature: the constructors `0` and successor `suc`, as well as `pred` and $<$ respectively interpreted as the predecessor function and less-than relation. Note that we do not define any arithmetic on naturals. We assume familiarity with the basics of saturation theorem proving.

```
   program ::= function
  function ::= func main() { subprogram }
subprogram ::= statement | context
   context ::= statement; ... ; statement
 statement ::= atomicStatement
             | if ( condition ) { context }  else  { context }
             | while ( condition ) { context }
```

Fig. 2: Grammar of $\mathcal{W}$.

4.1 Trace Logic $\mathcal{L}$

Trace logic, denoted as $\mathcal{L}$, is an instance of many-sorted first-order logic with theories. Its signature is $\Sigma(\mathcal{L}) := S_{\mathbb{N}} \cup S_{\mathbb{I}} \cup S_{\mathbb{L}} \cup S_V \cup S_n$, includes respectively the signatures of the theory of natural numbers $\mathbb{N}$ (as a term algebra), the in-built integer theory $\mathbb{I}$, a set $S_{\mathbb{L}}$ of timepoints (also referred to as *locations*), a set of symbols representing program variables S_V, as well as a set of symbols representing last iteration symbols S_n. For more details on trace logic, refer to [11].

4.2 Programming Model $\mathcal{W}$

We consider programs written in a WHILE-like programming language $\mathcal{W}$, as given in the (partial) language grammar of Figure 2. Programs in $\mathcal{W}$ contain mutable and immutable integer as well as integer-array program variables and consist of a single top-level function `main` comprising arbitrary nestings of while-loops and if-then-else branching. We consider expressions over booleans and integers without side effects.

4.3 Translating Expressions to Trace Logic

Locations and Timepoints. We consider programs as sets of locations over time: given a program statement `s`, we denote its location by l_s of type $\mathbb{L}$, the location/timepoint sort, corresponding to the line of the program where the statement appears. When `s` is a while-loop the corresponding location is revisited at multiple timepoints of the execution. Thus, we model such locations as functions over *loop iterations* $l_s : \mathbb{N} \mapsto \mathbb{L}$, where the argument of sort $\mathbb{N}$ intuitively corresponds to the number of loop iterations. Further, for each loop statement `s` we model the last loop iteration by a symbol $nl_s \in S_n$ of target sort $\mathbb{N}$. Let `p` be a program statement or context. We use $start_{\texttt{p}}$ to denote the location at which the execution of `p` has started and $end_{\texttt{p}}$ to denote the location that occurs just after the execution of `p`. We use $main\_end$ to denote the location at the end of the main function.

Example 1. Consider line 6 of our running example in Figure 1. Term l_6 corresponds to the timepoint of the first assignment of 0 to program variables `i` while $l_8(\texttt{0})$ and $l_8(nl_8)$ denote the timepoints of the loop at the first and last loop iteration respectively. Further, we can quantify over all executions of the loops by quantifying over all iterations smaller than the last e.g. $\forall it_{\mathbb{N}}.it < nl_8 \rightarrow F[l_8(it)]$ where $F[l_8(it)]$ is some first-order formula.

Program Variables. Program variable are expressed as functions over timepoints. We express an integer variable $\mathtt{v}$ as a function $v : \mathbb{L} \mapsto \mathbb{I}$, where $v \in S_V$. Let tp be a term of sort $\mathbb{L}$. Then, $v(tp)$ denotes the value of $\mathtt{v}$ at timepoint tp. We model numeric array variables $\mathtt{v}$ with an additional argument of sort $\mathbb{I}$ to denote the position of an array access. We obtain $v : \mathbb{L} \times \mathbb{I} \mapsto \mathbb{I}$. Immutable variables are modelled as per their mutable counterparts, but without the timepoint argument.

Example 2. To denote program variable $\mathtt{i}$ at the location of the assignment in line 6, we use the equation $i(l_6) \simeq 0$. For the first assignment of $\mathtt{c}$ within the loop, we write $c(l_8(it), 2 \times i(l_8(it))) \simeq a(i(l_8(it)))$ for some iteration it. As $\mathtt{a}$ is a constant array, the timepoint argument is omitted.

Program Expressions. Let $\mathtt{e}$ be an arbitrary program expression. We write $[\![\mathtt{e}]\!](tp)$ to denote the logical denotation of $\mathtt{e}$ at timepoint tp. We do not provide the full inductive definition of the denotation function $[\![\ ]\!](tp)$ here, just a few of its cases. If $\mathtt{e}$ is an integer variable $\mathtt{v}$, then $[\![\mathtt{e}]\!](tp) = v(tp)$. If $\mathtt{e}$ is an integer array access of the form $\mathtt{v}[\mathtt{e}_1]$, then $[\![\mathtt{e}]\!](tp) = v(tp, [\![\mathtt{e}_1]\!](tp))$. If $\mathtt{e}$ is an expression of the form $\mathtt{e}_1 + \mathtt{e}_2$, then $[\![\mathtt{e}]\!](tp) = [\![\mathtt{e}_1]\!](tp) + [\![\mathtt{e}_2]\!](tp)$.

Common Abbreviations. Let $\mathtt{e}, \mathtt{e}_1, \mathtt{e}_2$ be program expressions, tp_1, tp_2 be two timepoints and $v \in S_V$ denote the functional representation of a program variable. The trace logic formula $v(tp_1) \simeq v(tp_2)$ asserts that the variable v has the same value at timepoints tp_1 and tp_2. We introduce definitions for two formulas that are widely used in defining the axiomatic semantics of $\mathcal{W}$ in the next section. To ease the notational burden, we ignore array variables in the definitions provided. Firstly, we introduce a definition for the formula that expresses that the value of a variable v changes between timepoints tp_1 and tp_2 whilst the values of all other variables remain the same.

$$\mathit{Update}(v, e, tp_1, tp_2) \quad := \quad v(tp_2) \simeq [\![\mathtt{e}]\!](tp_1) \wedge \textstyle\bigwedge_{v' \in S_V \setminus \{v\}} v'(tp_1) \simeq v'(tp_2),$$

Secondly, we introduce a definition for the formula that expresses that the value of all variables stays the same between timepoints tp_1 and tp_2

$$\mathit{EqAll}(tp_1, tp_2) := \bigwedge_{v \in S_V} v(tp_1) \simeq v(tp_2)$$

4.4 Axiomatic Semantics of $\mathcal{W}$ in $\mathcal{L}$.

The semantics of a program in $\mathcal{W}$ is given by the conjunction of the respective axiomatic semantics of each program statement of $\mathcal{W}$ occurring in the program. In general, we define reachability of program statements over timepoints rather than program states. We briefly recall the axiomatic semantics of assignments and while-loops respectively, again ignoring the array variable case.

Assignments. Let $\mathtt{s}$ be an assignment $\mathtt{v = e}$, where $\mathtt{v}$ is an integer-valued program variable and $\mathtt{e}$ is an expression. The evaluation of $\mathtt{s}$ is performed in one step such that, after the evaluation, the variable $\mathtt{v}$ has the same value as $\mathtt{e}$ before the evaluation while all other variables remain unchanged. We obtain

$$[\![\mathtt{s}]\!] := \mathit{Update}(v, e, \mathit{start}_{\mathtt{s}}, \mathit{end}_{\mathtt{s}}) \tag{1}$$

While-Loops. Let `s` be the while-statement **`while`**`(Cond){c}` where `Cond` is the *loop condition*. The semantics of `s` is given by the conjunction of the following properties: (2a) the iteration nl_s is the first iteration where `Cond` does not hold anymore, (2b) jumping into the loop body does not change the values of the variables, (2c) the values of the variables at the end of evaluating the loop `s` are equal to the values at the loop condition location in iteration nl_s. As such, we have

$$\begin{aligned}
[\![\texttt{s}]\!] := \quad & \forall it^s_{\mathbb{N}}.\ (it^s < nl_{\texttt{s}} \rightarrow [\![\texttt{Cond}]\!](tp_{\texttt{s}}(it^s))) \\
\wedge \quad & \neg[\![\texttt{Cond}]\!](tp(nl_{\texttt{s}})) && \text{(2a)} \\
\wedge \quad & \forall it_{\mathbb{N}}.\ (it < nl_{\texttt{s}} \rightarrow \mathit{EqAll}(\mathit{start}_{\texttt{c}}, tp_{\texttt{s}}(it)) && \text{(2b)} \\
\wedge \quad & \mathit{EqAll}(\mathit{end}_{\texttt{s}}, tp_s(nl_{\texttt{s}})) && \text{(2c)}
\end{aligned}$$

4.5 Trace Lemma Reasoning

Trace logic $\mathcal{L}$ allows one to naturally express common program behavior over timepoints. Specifically, it allows us to reason about (i) all iterations of a loop, and (ii) the existence of specific timepoints. In [11], we leveraged such reasoning with the use of so-called *trace lemmas*, capturing common inductive properties of program loops. Trace lemmas are instances of the schema of bounded induction for natural numbers

$$\begin{aligned}
&\Big(P(bl) \wedge \forall x_{\mathbb{N}}.\big((bl \leq x < br \wedge P(x)) \rightarrow P(\texttt{suc}(x))\big)\Big) \rightarrow \\
&\quad \forall x_{\mathbb{N}}.\big(bl \leq x < br \wedge P(x)\big)
\end{aligned} \tag{3}$$

An example of a trace lemma would be the statement formalising that a certain program variable's value remains unchanged from a specific iteration to the end of loop execution. In this work, instead of adding instances of (3) statically to strengthen loop semantics, we move induction into the first-order prover. The advantage of adding instances of (3) dynamically is that during proof search we have more information available and can thus perform induction in a more controlled and goal oriented fashion.

Nonetheless, due to some limitations in our first-order prover, we are unable to completely do away with additional lemmas. Specifically, we need to nudge the prover to deduce that a loop counter expression will, at the end of loop execution, have the value of the expression it is compared against in the loop condition.

(A) Equal Lengths Trace Lemma We define a common property of loop counter expressions. We call a program expression `e` *dense* at loop `w` if:

$$\mathit{Dense}_{w,e} := \forall it_{\mathbb{N}}.\Big(it < nl_{\texttt{w}} \rightarrow \begin{pmatrix} [\![\texttt{e}]\!](tp_{\texttt{w}}(\texttt{suc}(it))) \simeq [\![\texttt{e}]\!](tp_{\texttt{w}}(it)) \vee \\ [\![\texttt{e}]\!](tp_{\texttt{w}}(\texttt{suc}(it))) \simeq [\![\texttt{e}]\!](tp_{\texttt{w}}(it)) + 1 \end{pmatrix}\Big).$$

Let `w` be a while-statement, $C_{\texttt{w}} := \texttt{e} < \texttt{e'}$ be the loop condition where `e'` is a program expression that remains constant during iterations of `w`. The *equal lengths trace lemma of* `w`*,* `e` *and* `e'` is defined as

$$\begin{aligned}
&\big(\mathit{Dense}_{w,e} \wedge [\![\texttt{e}]\!](tp_{\texttt{w}}(\texttt{0})) \leq [\![\texttt{e'}]\!](tp_{\texttt{w}}(\texttt{0}))\big) \rightarrow \\
&\quad [\![\texttt{e}]\!](tp_{\texttt{w}}(nl_{\texttt{w}})) \simeq [\![\texttt{e'}]\!](tp_{\texttt{w}}(nl_{\texttt{w}})).
\end{aligned} \tag{A}$$

Trace lemma A states that a dense expression `e` smaller than or equal to some expression `e'` that does not change in the loop, will eventually, specifically in the last iteration, reach the same value as `e'`. This follows from the fact that we assume termination of a loop, hence we assume the existence of a timepoint $nl_{\mathtt{w}}$ where the loop condition does not hold anymore. As a consequence, given that the loop condition held at the beginning of the execution, we can derive that the loop counter value immediately after the loop execution $[\![\mathtt{e}]\!](tp_{\mathtt{w}}(nl_{\mathtt{w}}))$ will necessarily equate to $[\![\mathtt{e'}]\!](tp_{\mathtt{w}}(\mathsf{0})) = [\![\mathtt{e'}]\!](tp_{\mathtt{w}}(nl_{\mathtt{w}}))$. Note that a similar lemma can just as easily be added for dense but decreasing loop counters.

5 Multi-Clause Goal Induction for Lemmaless Induction

As mentioned above, the main focus of our work is moving induction into the saturation prover. We achieve this by adding inference rules that apply induction to loop counter terms. We leverage recent theorem proving effort on *bounded (integer) induction* in saturation [14, 15]. However, as illustrated in the following, these recent efforts cannot be directly used in trace logic reasoning since we need to (i) adjust bounded induction for the setting of natural numbers, and (ii) generalise to multi-clause induction. We discuss these steps using Figure 1. Verifying the safety assertion of Figure 1 requires proving the trace logic formula:

$$\begin{aligned}&\forall pos_{\mathbb{I}}.\,\exists j_{\mathbb{I}}.\,(0 \leq pos < (2 \times length)\\ &\quad \rightarrow (c(main\_end, pos) \simeq a(j) \lor c(main\_end, pos) \simeq b(j))\end{aligned} \tag{4}$$

For proving (4), it suffices to prove that the following, slightly modified statement is a loop invariant of Figure 1:

$$\begin{aligned}&\forall it_{\mathbb{N}}.\, it < nl_{\mathtt{w}} \rightarrow \forall pos_{\mathbb{I}}.\,\exists j_{\mathbb{I}}.\,(0 \leq pos < (2 \times i(tp_{\mathtt{w}}(it))))\\ &\qquad \rightarrow (c(tp_{\mathtt{w}}(it), pos) \simeq a(j) \lor c(tp_{\mathtt{w}}(it), pos) \simeq b(j))\end{aligned} \tag{5}$$

where `w` refers to the loop statement in Figure 1. As part of the program semantics in trace logic, we have formula (6) which links the value of c at the end of the loop to its value at the end of the program. Moreover, using the trace lemma A, we also derive formula (7) in trace logic:

$$\forall pos_{\mathbb{I}}.c(tp_{\mathtt{w}}(nl_{\mathtt{w}}), pos) \simeq c(main\_end, pos) \tag{6}$$

$$i(tp_{\mathtt{w}}(nl_{\mathtt{w}})) \simeq length \tag{7}$$

It is tempting to think that in the presence of these clauses (6)–(7), a saturation-based prover would rewrite the negated conjecture (4) to

$$\begin{aligned}&\neg(\forall pos_{\mathbb{I}}.\,\exists j_{\mathbb{I}}.\,(0 \leq pos < (2 \times i(tp_{\mathtt{w}}(nl_{\mathtt{w}}))))\\ &\quad \rightarrow (c(tp_{\mathtt{w}}(nl_{\mathtt{w}}), pos) \simeq a(j) \lor c(tp_{\mathtt{w}}(nl_{\mathtt{w}}), pos) \simeq b(j)))\end{aligned}$$

from which a bounded natural number induction inference (similar to the $\mathtt{IntInd}_{<}$ rule of [15]) would quickly introduce an induction hypothesis with (5) as the conclusion, by induction over nl_{w}. However, this is not the case, as most saturation provers work by first *clausifying* their input. The negated conjecture (4) would not remain a single formula, but be split into the following clauses where sk is a Skolem symbol:

$$\begin{array}{cc} a(x) \not\simeq c(main\_end, sk) & b(x) \not\simeq c(main\_end, sk) \\ \neg(sk \leq 0) & sk \leq 2 \times length \end{array}$$

These clauses can be rewritten using (6)–(7). For example, the first clause can be rewritten to $a(x) \not\simeq c(tp_{\mathrm{w}}(nl_{\mathrm{w}}, sk))$. However, attempting to prove the negation of any of the rewritten clauses individually via induction would merely result in the addition of useless induction formulas to the search space. For example, attempting to prove $\forall it_{\mathbb{N}}.\, it < nl_{\mathrm{w}} \to (\exists x_{\mathbb{I}}.\, a(x) \simeq c(tp_{\mathrm{w}}(it), sk))$, is pointless as it is clearly false. *The solution we propose in this work is to use multi-clause induction*, whereby we attempt to prove the negation of the conjunction of multiple clauses via a single induction inference. For our running example Figure 1, we can use the following rewritten versions of clauses from the negated conjecture $a(x) \not\simeq c(tp_{\mathrm{w}}(nl_{\mathrm{w}}, sk))$, $b(x) \not\simeq c(tp_{\mathrm{w}}(nl_{\mathrm{w}}, sk))$, and $sk \leq 2 \times i(tp_{\mathrm{w}}(nl_{\mathrm{w}}))$, with induction term nl_{w}, to obtain the induction formula:

$$\begin{array}{lcl} \neg\Big(\;\; \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(tp_{\mathrm{w}}(0)), sk) & & \forall it_{\mathbb{N}}.\, it < nl_{\mathrm{w}} \to \\ \quad \wedge\ \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(tp_{\mathrm{w}}(0), sk)) & \to & \quad \neg\big(\;\; \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(tp_{\mathrm{w}}(it), sk) \\ \quad \wedge\ sk \leq 2 \times i(tp_{\mathrm{w}}(0))\Big) & & \quad\quad \wedge \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(tp_{\mathrm{w}}(it), sk) \\ \wedge\ StepCase & & \quad\quad \wedge\ sk \leq 2 \times i(tp_{\mathrm{w}}(it))\big) \end{array} \tag{8}$$

where $StepCase$ is the formula:

$$\begin{array}{lcl} \forall it_{\mathbb{N}}.\, it < nl_{\mathrm{w}} \wedge & & \\ \neg\Big(\;\; \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(tp_{\mathrm{w}}(it)), sk) & & \neg\Big(\;\; \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(tp_{\mathrm{w}}(\mathtt{suc}(it)), sk) \\ \quad \wedge\ \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(tp_{\mathrm{w}}(it)), sk) & \to & \quad \wedge \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(tp_{\mathrm{w}}(\mathtt{suc}(it)), sk) \\ \quad \wedge\ sk \leq i(tp_{\mathrm{w}}(y)\Big) & & \quad \wedge\ sk \leq 2 \times i(tp_{\mathrm{w}}(\mathtt{suc}(it)))\Big) \end{array}$$

Using the induction formula (8), a contradiction can then easily be derived, establishing validity of (4). In what follows, we formalize the multi-clause induction principle we used above. To this end, we introduce a generic inference rule, called *multi-clause goal induction* and denoted as `MCGLoopInd`.

$$\frac{C_1[nl_{\mathrm{w}}] \qquad C_2[nl_{\mathrm{w}}] \qquad \ldots \qquad C_n[nl_{\mathrm{w}}]}{\mathrm{CNF}\left(\left(\begin{array}{c} \neg(C_1[0] \wedge C_2[0] \wedge \ldots \wedge C_n[0]) \wedge \\ \forall it_{\mathbb{N}}.\, \left(\begin{array}{l} ((it < nl_{\mathrm{w}}) \wedge \neg(C_1[it] \wedge C_2[it] \wedge \ldots \wedge C_n[it])) \to \\ \neg(C_1[\mathtt{suc}(it)] \wedge C_2[\mathtt{suc}(it)] \wedge \ldots \wedge C_n[\mathtt{suc}(it)])) \end{array} \right) \end{array} \right) \\ \to (\forall it_{\mathbb{N}}.\, (it < nl_{\mathrm{w}}) \to \neg(C_1[it] \wedge C_2[it] \wedge \ldots \wedge C_n[it])) \right)}$$

For performance reasons, we mandate that the premises $C_1 \ldots C_n$ be derived from trace logic formulas expressing safety assertions and not from formulas encoding the program semantics. The `MCGLoopInd` rule is formalised only as an induction inference over last loop iteration symbols. While restricting to nl_{w} terms is of purely heuristic nature, our experiments justify the necessity and usefulness of this condition (Section 8).

```
    func main(){
      const Int alength;
      Int[] a;
      Int i = 0;
      const Int n;

      while(i < alength){
        a[i] = a[i] + n;
        i = i + 1;
      }

      Int j = 0;
      while(j < alength){
        a[j] = a[j] - n;
        j = j + 1;
      }
    }
  assert (∀pos𝕀.((0 ≤ pos < alength)
         → a(main_end, pos) = a(main_start, pos)))
```

Fig. 3: Adding and subtracting `n` to every element of array `a`.

6 Array Mapping Induction for Lemmaless Induction

Multi-clause goal induction neatly captures goal-oriented application of induction. Nevertheless, there are verification challenges where `MCGLoopInd` fails to prove inductive loop properties. This is particularly the case for benchmarks containing multiple loops, such as in Figure 3. We first discuss the limitations of `MCGLoopInd` using Figure 3, after which we present our solution, the *array mapping induction* inference.

Let $\mathtt{w}_1$ be the first loop statement of Figure 3 and $\mathtt{w}_2$ be the second loop. Using `MCGLoopInd`, we would attempt to prove

$$\begin{aligned}&\forall it_{\mathbb{N}}.\, it \leq nl_{\mathtt{w}_2} \rightarrow\\ &\quad \forall pos_{\mathbb{I}}.\,(0 \leq pos < j(tp_{\mathtt{w}_2}(it))) \rightarrow (a(tp_{\mathtt{w}_2}(it), pos) \simeq a(main\_start, pos)\end{aligned} \tag{9}$$

However, formula (9) is not a useful invariant for proving the assertion. Rather, for $\mathtt{w}_2$ we need a loop invariant similar to

$$\begin{aligned}\forall it_{\mathbb{N}}.\, it \leq nl_{\mathtt{w}_2} \rightarrow \forall pos_{\mathbb{I}}.\,(0 \leq pos < j(tp_{\mathtt{w}_2}(it)))&\\ \rightarrow (a(tp_{\mathtt{w}_2}(it), pos) \simeq a(tp_{\mathtt{w}_2}(0), pos) - n&\end{aligned} \tag{10}$$

and a similar loop invariant for loop $\mathtt{w}_1$. The loop invariant (10) is however not linked to the safety assertion of Figure 3, and thus multi-clause goal induction is unable to infer and prove with it. To aid with the verification of benchmarks such as Figure 3, we introduce another induction inference which we call *array mapping induction*. In this case, we trigger induction not on clauses and terms coming from the goal, but on clauses and terms appearing in the program semantics.

The *array mapping induction* inference rule, denoted as `AMLoopInd` is given below. Essentially, `AMLoopInd` involves analysing a clause set to heuristically devise a suitable loop invariant. Guessing a candidate loop invariant is a difficult problem. The `AMLoopInd` inference is triggered if clauses of the shapes of C_1 and C_2 defined below are present in the clause set. Intuitively, C_2 can be read as saying that on each round of some loop $\mathtt{w}$, some array a at position i is set to some function F of its previous value at that position. Clause C_1 states that i increases by m in each round of the loop. Together the two clauses suggest that the loop is mapping the function F to each mth location of the array starting from the array cell located at $i(tp_{\mathtt{w}}(0))$. This is precisely what the induction formula attempts to prove. Note that for ease of notation, we present the inference for the case where the indexing variable is *increasing*. It is straightforward to generalise to the decreasing case. The `AMLoopInd` rule is[3]

$$\frac{\begin{array}{c} C_1 = i(tp_{\mathtt{w}}(\mathtt{suc}(x))) \simeq i(tp_{\mathtt{w}}(x)) + m \ \vee \ \neg(x < nl_{\mathtt{w}}) \\ C_2 = a\big(tp_{\mathtt{w}}(\mathtt{suc}(x)), i(tp_{\mathtt{w}}(x))\big) \simeq F[a\big(tp_{\mathtt{w}}(x), i(tp_{\mathtt{w}}(x))\big)] \ \vee \ \neg(x < nl_{\mathtt{w}}) \end{array}}{\text{CNF}(StepCase \rightarrow Conclusion)}$$

where $\mathtt{w}$ is some loop and F an arbitrary non-empty context. Let i_0 be an abbreviation for $i(tp_{\mathtt{w}}(0))$. Then:

$$\begin{aligned} StepCase: \quad & \forall it_{\mathbb{N}}. \big(\forall y_{\mathbb{I}}. it < nl_{\mathtt{w}} \wedge \\ & \qquad y < i(tp_{\mathtt{w}}(it)) - i_0 \wedge y \geq 0 \wedge y \ \mathtt{mod}\ m = 0 \\ & \qquad\quad \rightarrow a(tp_{\mathtt{w}}(it), i_0 + y) \simeq F[a(tp_{\mathtt{w}}(0), i_0 + y)]\big) \rightarrow \\ & \quad (\forall y_{\mathbb{I}}. y < i(tp_{\mathtt{w}}(\mathtt{suc}(it))) - i_0 \wedge y \geq 0 \wedge y \ \mathtt{mod}\ m = 0 \\ & \qquad\quad \rightarrow a(tp_{\mathtt{w}}(\mathtt{suc}(it)), i_0 + y) \simeq F[a(tp_{\mathtt{w}}(0), i_0 + y)]) \\ Conclusion: \ & \forall x_{\mathbb{I}}. x < i(tp_{\mathtt{w}}(nl_{\mathtt{w}})) - i_0 \wedge x \geq 0 \wedge x \ \mathtt{mod}\ m = 0 \\ & \qquad\quad \rightarrow a(tp_{\mathtt{w}}(nl_{\mathtt{w}}), i_0 + x) \simeq F[a(tp_{\mathtt{w}}(0), i_0 + x)] \end{aligned}$$

To prove $StepCase$, it is necessary to be able to reason that positions in the array a remain unchanged until visited by the indexing variable. This can be achieved via the addition of another induction to the conclusion of the inference. We do not provide details of this induction formula here, but it is added to the conclusion by our implementation which we present in Section 7. The `AMLoopInd` inference is thus sufficient to prove the assertion of Figure 3. While `AMLoopInd` is a limited approach for guessing inductive loop invariants, we believe it can be extended towards further, more generic methods to guess invariants, as discussed in Section 9. We conclude this section by noting that our induction rules are sound, based on trace logic semantics. Since both rules merely add instances of the bounded induction schema for natural numbers (3) to the search space, soundness is trivial and we do not provide a proof.

Theorem 1 (Soundness of Lemmaless Induction). *The inference rules* `MCGLoopInd` *and* `AMLoopInd` *are sound.*

7 Implementation

Our approach is implemented as an extension of the RAPID framework, using the first-order theorem prover VAMPIRE.

[3] In the conclusion we ignore the base case of the induction formula as it is trivially true.

Extensions to **RAPID.** RAPID takes as an input a $\mathcal{W}$ program along with a property expressed in $\mathcal{L}$. It outputs the semantics of the program expressed in $\mathcal{L}$ using SMT-LIB syntax along with the property to be proven. For our "lemmaless induction" framework, we have extended RAPID as follows. Firstly, we prevent the output of all trace lemmas other than trace lemma A (Section 4.5). We added custom extensions to the SMT-LIB language to identify trace logic symbols, such as loop iteration symbols, program variables, within the RAPID encodings. This way, trace logic symbols to be used for induction inferences are easily identified and can also be used for various proving heuristics. We refer to this version (available online[4]) as RAPID^{l-}.

Extensions to **VAMPIRE.** We implemented the `MCGLoopInd` inference rule and a slightly simplified version of the `AMLoopInd` rule in a new branch of VAMPIRE[5]. The main issue with the induction inferences `MCGLoopInd` and `AMLoopInd` is their explosiveness which can cause proof search to diverge. We have, therefore, introduced various heuristics in the implementation to try and control them. For `MCGLoopInd` we not only necessitate that the premises are derived from the conjecture, but that their derivation length from the conjecture is below a certain distance controlled by an option. The premises must be unit clauses unless another option `multi_literal_ clauses` is toggled on. The option `induct_all_loop_counts` allows `MCGLoopInd` induction to take place on all loop counter terms, not just final loop iterators. In order for the `MCGLoopInd` and `AMLoopInd` inferences to be applicable, we need to rewrite terms not containing final loop counters to terms that do. However, rewriting in VAMPIRE is based on superposition, which is parameterised by a term order preventing smaller terms to be rewritten into larger ones. In this case, the term order may work against us and prevent such rewrites from happening. We implemented a number of heuristics to handle this problem. One such heuristic is to give terms representing constant program variables a large weight in the ordering. Then, equations such as $alength \simeq i(tp_{\mathrm{w}}(nl_{\mathrm{w}}))$ will be oriented left to right as desired. We combined these options with others to form a portfolio of strategies[6] that contains 13 strategies each of which runs in under 10s.

8 Experimental Results

Benchmarks. For our experiments, we use a total of 111 examples whose verification involved proving safety assertions of different logical complexity (quantifier-free, only universally/existentially quantified, and with quantifier alternations). Our benchmarks are divided into four groups, as indicated in Table 1: (i) the first 13 problems have quantifier-free proof obligations; (ii) the majority of benchmarks, in total 68 examples, contain universally quantified safety assertions; (iii) 7 problems come with the task of verifying existentially quantified assertions; (iv) and the last 23 programs contain assertions with alternation of quantifiers. The examples from (i)-(ii), a total of 81 programs,

[4] See commit `285e54b7e` of `https://github.com/vprover/rapid/tree/ahmed-induction-support`.
[5] See commit `4a0f319f` of `https://github.com/vprover/vampire/tree/ahmed-rapid`.
[6] `--mode portfolio --schedule rapid_induction`..

Table 1: Experimental results.

Benchmark	(1)	(2)	(3)	(4)
atleast_one_iteration_0	✓	✓	✓	✓
atleast_one_iteration_1	✓	✓	✓	✓
count_down	✓	-	-	-
eq	✓	-	✓	-
find_sentinel	✓	✓	-	-
find1_0	✓	✓	✓	-
find1_1	✓	✓	✓	-
find2_0	✓	✓	✓	-
find2_1	✓	✓	✓	-
indexn_is_arraylength_0	✓	✓	✓	-
indexn_is_arraylength_1	✓	✓	✓	-
set_to_one	✓	✓	✓	✓
str_cpy_3	✓	✓	✓	-
add_and_subtract	✓	-	-	✓
both_or_none	✓	✓	-	✓
check_equal_set_flag_1	✓	✓	-	✓
collect_indices_eq_val_0	✓	✓	-	✓
collect_indices_eq_val_1	✓	✓	-	-
copy	✓	✓	-	✓
copy_absolute_0	✓	✓	-	✓
copy_absolute_1	✓	✓	-	✓
copy_and_add	✓	-	-	✓
copy_nonzero_0	✓	✓	-	✓
copy_partial	✓	✓	-	✓
copy_positive_0	✓	✓	-	✓
copy_two_indices	✓	✓	-	-
find_max_0	✓	✓	-	✓
find_max_2	✓	✓	-	✓
find_max_from_second_0	✓	-	-	✓
find_max_local_2	-	-	-	-
find_max_up_to_0	-	-	-	-
find_max_up_to_2	-	-	-	-
find_min_0	✓	✓	-	✓
find_min_2	✓	✓	-	-
find_min_local_2	-	-	-	-
find_min_up_to_0	-	-	-	-
find_min_up_to_2	-	-	-	-
find1_4	-	✓	-	-
find2_4	✓	✓	-	-
in_place_max	✓	✓	-	✓
inc_by_one_0	✓	✓	-	✓
inc_by_one_1	✓	✓	-	✓
inc_by_one_harder_0	✓	✓	-	✓
inc_by_one_harder_1	✓	✓	-	✓
init	✓	✓	-	-
init_conditionally_0	✓	✓	-	-
init_conditionally_1	✓	✓	-	✓
init_non_constant_0	✓	✓	-	-
init_non_constant_1	✓	✓	-	✓
init_non_constant_2	✓	✓	-	✓
init_non_constant_3	✓	✓	-	✓
init_non_constant_easy_0	✓	✓	-	-
init_non_constant_easy_1	✓	✓	-	✓
init_non_constant_easy_2	✓	✓	-	✓
init_non_constant_easy_3	✓	✓	-	✓
init_partial	✓	✓	-	✓
init_prev_plus_one_0	✓	✓	-	-
init_prev_plus_one_1	✓	✓	-	-
init_prev_plus_one_alt_0	✓	✓	-	-
init_prev_plus_one_alt_1	✓	✓	-	-
insertion_sort	-	-	-	-
max_prop_0	✓	✓	-	✓
max_prop_1	✓	✓	-	✓
merge_interleave_0	✓	-	-	✓
merge_interleave_1	✓	-	-	✓
min_prop_0	✓	✓	-	✓
min_prop_1	✓	✓	-	✓
partition_0	✓	✓	-	✓
partition_1	✓	✓	-	✓
push_back	✓	✓	-	✓
reverse	✓	✓	-	-
rewnifrev	✓	-	-	✓
rewrev	✓	-	-	✓
skipped	✓	-	-	✓
str_cpy_0	✓	✓	-	-
str_cpy_1	✓	✓	-	-
str_cpy_2	✓	✓	-	-
swap_0	-	✓	✓	✓
swap_1	-	✓	✓	✓
vector_addition	✓	✓	-	✓
vector_subtraction	✓	✓	-	✓
check_equal_set_flag_0	✓	✓	-	-
find_max_1	-	-	-	-
find_max_from_second_1	✓	-	-	-
find1_2	✓	✓	-	-
find1_3	✓	✓	-	-
find2_2	✓	✓	-	-
find2_3	✓	✓	-	-
collect_indices_eq_val_2	-	✓	-	-
collect_indices_eq_val_3	✓	-	-	-
copy_nonzero_1	✓	✓	-	-
copy_positive_1	✓	✓	-	-
find_max_local_0	-	-	-	-
find_max_local_1	✓	-	-	-
find_max_up_to_1	-	-	-	-
find_min_1	-	-	-	-
find_min_local_0	-	-	-	-
find_min_local_1	✓	-	-	-
find_min_up_to_1	-	-	-	-
merge_interleave_2	✓	-	-	-
partition_2	✓	✓	-	-
partition_3	✓	✓	-	-
partition_4	-	-	-	-
partition_5	-	✓	-	-
partition_6	-	-	-	-
partition-harder_0	✓	✓	-	-
partition-harder_1	✓	✓	-	-
partition-harder_2	✓	-	-	-
partition-harder_3	✓	-	-	-
partition-harder_4	✓	-	-	-
str_len	✓	✓	-	-
Total solved	**93**	**78**	**13**	**47**

come from the array verification benchmarks of SV-COMP repository [1], with most of these examples originating from [9, 13].[7] These examples correspond to the set of those SV-COMP benchmarks which use the C fragment supported by RAPID; specifically, when selecting examples (i)-(ii) from SV-COMP, we omitted examples containing pointers or memory management. All SV-COMP examples from (i)-(ii) are adapted to our input format, as for example arrays in trace logic are treated as unbounded data structures. Further, the examples (iii)-(iv) are new examples crafted by us, in total 30 new examples. They contain existential and alternating quantification in safety assertions. We intend to submit these 30 examples from (iii)-(iv) to SV-COMP.

Experimental Setting. We used two versions of RAPID in our experiments. First, **(1)** RAPID^{l-} denotes our RAPID approach, using lemmaless induction `MCGLoopInd` and `AMLoopInd` in VAMPIRE. Further, **(2)** RAPID^{l+} uses trace lemmas for inductive reasoning, as described in [11]. We also compared RAPID^{l-} with other verification tools. In particular, we considered **(3)** SEAHORN and **(4)** VAJRA (and its extension DIFFY that produced for us exactly the same results as VAJRA). SEAHORN converts the program into a constrained horn clause (CHC) problem and uses the SMT solver Z3 for solving. VAJRA and DIFFY implement inductive reasoning and recurrence solving over loop counters; in the background, they also use Z3.

RAPID ***Experiments.*** Table 1 shows that RAPID^{l-} is superior to RAPID^{l+}, as it solves a total of 93 problems, while RAPID^{l+} only proved 78 assertions correct. Particularly, RAPID^{l-} can solve benchmark `merge_interleave_2` corresponding to our motivating example 1, and other challenging problems such as `find_max_local_1` also containing quantifier alternations.

While RAPID^{l-} can solve a total of ten problems more than RAPID^{l+}, it is interesting to look into which problems can now be solved. Many of the newly solved problems are structurally very close to the loop invariants needed to prove them. This is where multi-clause goal-oriented induction `MCGoalInd` makes the biggest impact. For instance, this allows RAPID^{l-} to prove the partial correctness of `find_max_from_second_0` and `find_max_from_second_1`.

On the other hand, RAPID^{l-} also lost two challenging benchmarks that were previously solved by RAPID^{l+}, namely `swap_0` and `partition_5`. This could be for two reasons: (1) the strategies in the induction schedule of RAPID^{l-} are too restrictive for such benchmarks, or (2) the step case of the induction axiom introduced by our two rules are too difficult for VAMPIRE to prove. Strengthening lemmaless induction with additional trace lemmas from RAPID^{l+} is an interesting line of further work.

Comparing with other tools. Both, SEAHORN and VAJRA/DIFFY require C code as input, whereas RAPID uses its own syntax. We translated our benchmarks to C code expressing the same problem. However, a direct comparison of RAPID, and in particular RAPID^{l-}, with most other verifiers requiring standard C code as an input is

[7] Artifact evaluation: in order to reproduce the results reported in this section, please follow the instructions at `https://github.com/vprover/vampire_publications/tree/master/experimental_data/CICM-2022-RAPID-INDUCTION`

not possible as we consider slightly different semantics. In contrast to SEAHORN and VAJRA/DIFFY, we assume that integers and arrays are unbounded and that all array positions are initialized by arbitrary data. Further, we can read/write at any array position without allocating the accessed memory beforehand. Apart from semantic differences, RAPID can directly express assertions and assumptions containing quantifiers and put variable contents from different points in time into relation. In order to deal with the latter, we introduced history variables in the code provided to SEAHORN and VAJRA/DIFFY. Quantification was simulated by non-deterministically assigned variables and by loops. As a result, SEAHORN verified 13 examples, whereas VAJRA/DIFFY 47 of our benchmarks. As VAJRA/DIFFY restrict their input programs to contain only loops having very specific loop-conditions, several of our benchmarks failed. For example, $i < length$ is permitted, whereas $a[i] \neq 0$ is not. VAJRA/DIFFY could prove correctness for nearly all the programs satisfying these restrictions. SEAHORN, on the other hand, has problems with the complexity introduced by the arrays. It could solve especially those benchmarks whose correctness do not depend on the arrays' content.

9 Future Directions and Conclusion

We introduced lemmaless induction to fully automate the verification of inductive properties of program loops with unbounded arrays and integers. We introduced goal-oriented and array mapping induction inferences, triggered by loop counters, in superposition-based theorem proving. Our results show that lemmaless induction in trace logic outperforms other state-of-the-art approaches in the area. There are various ways to further develop lemmaless induction in trace logic. On larger benchmarks, particularly those containing multiple loops, our approach struggles. For loops where the required invariant is not connected to the conjecture, we introduced array mapping induction. However, the array mapping induction inference is limited in the form of invariants it can generate. We would like to investigate other methods, such as machine learning for synthesising loop invariants that are not too prolific. A completely different line of research that we are currently working on, is updating the trace logic syntax and semantics of $\mathcal{W}$ to deal with memory and memory allocation, aiming to efficiently reason about loop operations over the memory.

As shown in [19], the validity problem for first-order formulas of linear arithmetic extended with non-theory function symbols is Π_1^1-complete. Therefore, we do not expect any completeness result for inductive theorem proving. Proving relative completeness results for our verification framework is an interesting question.

Acknowledgements. This research was partially supported by the ERC consolidator grant ARTIST 101002685, the FWF research project LogiCS W1255-N23, the TU Wien SecInt doctoral program, and the EUProofNet Cost Action CA20111. Our research was partially funded by the Digital Security by Design (DSbD) Programme delivered by UKRI to support the DSbD ecosystem.

References

1. "sv-comp repository". `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks`.

2. Vampire website. `https://vprover.github.io/`.
3. L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.
4. Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.
5. Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Verifying array manipulating programs with full-program induction. In *TACAS*, pages 22–39, 2020.
6. Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Diffy: Inductive Reasoning of Array Programs Using Difference Invariants. In *CAV*, pages 911–935, 2021.
7. Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *IJCAR*, pages 392–406. Springer, 2013.
8. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
9. Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266, 2010.
10. Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In *CAV*, pages 259–277, 2019.
11. Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace Logic for Inductive Loop Reasoning. In *FMCAD*, pages 255–263, 2020.
12. Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The SeaHorn verification framework. In *CAV*, pages 343–361, 2015.
13. Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Quantifiers on demand. In *ATVA*, pages 248–266, 2018.
14. Márton Hajdu, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Induction with generalization in superposition reasoning. In *CICM*, pages 123–137, 2020.
15. Petra Hozzová, Laura Kovács, and Andrei Voronkov. Integer induction in saturation. In *CADE*, pages 361–377. Springer, 2021.
16. Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *APLAS*, pages 304–311. Springer, 2010.
17. E. G. Karpenkov and D. Monniaux. Formula slicing: Inductive invariants from preconditions. In *HVC*, pages 169–185, 2016.
18. Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, pages 203–213, 1997.
19. Konstantin Korovin and Andrei Voronkov. Integrating linear arithmetic into superposition calculus. In *CSL*, pages 223–237, 2007.
20. Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to terms with quantified reasoning. In *POPL*, pages 260–270, 2017.
21. Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *CAV*, pages 1–35, 2013.
22. D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In *VMCAI*, pages 169–188, 2013.
23. K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, pages 348–370, 2010.
24. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.
25. Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 335–367. Elsevier Science, 2001.

26. Pritom Rajkhowa and Fangzhen Lin. Extending VIAP to handle array programs. In *VSTTE*, pages 38–49, 2018.
27. Giles Reger, Nikolaj Bjorner, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In *GCAI*, pages 39–52, 2016.
28. Giles Reger, Johannes Schoisswohl, and Andrei Voronkov. Making theory reasoning simpler. In *TACAS*, pages 164–180, 2021.
29. S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *PLDI*, pages 223–234, 2009.