Trace Logic for Inductive Loop Reasoning

Pamina Georgiou, Bernhard Gleiss, Laura Kovács
TU Wien, Austria

***Abstract*—We propose trace logic, an instance of many-sorted first-order logic, to automate the partial correctness verification of programs containing loops. Trace logic generalizes semantics of program locations and captures loop semantics by encoding properties at arbitrary timepoints and loop iterations. We guide and automate inductive loop reasoning in trace logic by using generic trace lemmas capturing inductive loop invariants. Our work is implemented in the RAPID framework, by extending and integrating superposition-based first-order reasoning within RAPID. We successfully used RAPID to prove correctness of many programs whose functional behavior are best summarized in the first-order theories of linear integer arithmetic, arrays and inductive data types.**

I. INTRODUCTION

One of the main challenges in automating software verification comes with handling inductive reasoning over programs containing loops. Until recently, automated reasoning in formal verification was the primary domain of satisfiability modulo theory (SMT) solvers [1], [2], yielding powerful advancements for inferring and proving loop properties with linear arithmetic and limited use of quantifiers, see e.g. [3], [4], [5]. Formal verification however also requires reasoning about unbounded data types, such as arrays, and inductively defined data types. Specifying, for example as shown in Figure 1, that every element in the array `b` is initialized by a non-negative array element of `a` requires reasoning with quantifiers and can be best expressed in many-sorted extensions of first-order logic. Yet, the recent progress in automation for quantified reasoning in first-order theorem proving has not yet been fully integrated in formal verification. In this paper we address such a use of first-order reasoning and propose trace logic $\mathcal{L}$, an instance of many-sorted first-order logic, to automate the partial correctness verification of program loops, by expressing program semantics in $\mathcal{L}$, and use $\mathcal{L}$ in combination with superposition-based first-order theorem proving.

Contributions: In our previous work [6], an initial version of trace logic $\mathcal{L}$ was introduced to formalize and prove relational properties. In this paper, we go beyond [6] and turn trace logic $\mathcal{L}$ into an efficient approach to loop (safety) verification. We propose trace logic $\mathcal{L}$ as a unifying framework to reason about both relational and safety properties expressed in full first-order logic with theories. We bring the following contributions.
(i) We generalize the semantics of program locations by treating them as functions of execution timepoints. In essence, unlike other works [7], [8], [9], [10], we formalize program properties at arbitrary timepoints of locations.
(ii) Thanks to this generalization, we provide a non-recursive axiomatization of program semantics in trace logic $\mathcal{L}$ and prove completeness of our axiomatization with respect to Hoare logic. Our semantics in trace logic $\mathcal{L}$ supports arbitrary quantification over loop iterations (Section V).
(iii) We guide and automate inductive loop reasoning in trace logic $\mathcal{L}$, by using generic trace lemmas capturing inductive loop invariants (Section VI). We prove soundness of each trace lemma we introduce.
(iv) We bring first-order theorem proving into the landscape of formal verification, by extending recent results in superposition-based reasoning [11], [12], [13] with support for trace logic properties, complementing SMT-based verification methods in the area (Section VI). As logical consequences of our trace lemmas are also loop invariants, superposition-based reasoning in trace logic $\mathcal{L}$ enables to automatically find loop invariants that are needed for proving safety assertions of program loops.
(v) We implemented our approach in the RAPID framework and combined RAPID with new extensions of the first-order theorem prover VAMPIRE. We successfully evaluated our work on more than 100 benchmarks taken from the SV-Comp repository [14], mainly consisting of safety verification challenges over programs containing arrays of arbitrary length and integers (Section VII). Our experiments show that RAPID automatically proves safety of many examples that, to the best of our knowledge, cannot be handled by other methods.

```
 1    func main() {
 2      const Int[] a;
 3
 4      Int[] b;
 5      Int i = 0;
 6      Int j = 0;
 7      while (i < a.length) {
 8        if (a[i] ≥ 0) {
 9          b[j] = a[i];
10          j = j + 1:
11        }
12        i = i + 1;
13      }
14    }
15    assert (∀k_I.∃l_I.((0 ≤ k <j ∧ a.length ≥ 0)
                → b(k) = a(l)))
16
```

Fig. 1. Program copying positive elements from array `a` to `b`.

II. RUNNING EXAMPLE

We illustrate and motivate our work with Figure 1. This program iterates over a constant integer array `a` of arbitrary length and copies positive values into a new array `b`. We are interested in proving the safety assertion given at line 15: given

https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_33

that the length `a.length` of `a` is not negative, every element in `b` is an element from `a`. Expressing such a property requires alternations of quantifiers in the first-order theories of linear integer arithmetic and arrays, as formalized in line 15. We write $k_{\mathbb{I}}$ and $l_{\mathbb{I}}$ to specify that k, l are of sort integer $\mathbb{I}$.

While the safety assertion of line 15 holds, proving correctness of Figure 1 is challenging for most state-of-the-art approaches, such as e.g. [15], [3], [4], [5]. The reason is that proving safety of Figure 1 needs inductive invariants with existential/alternating quantification and involves inductive reasoning over arbitrarily bounded loop iterations/timepoints. In this paper we address these challenges as follows.

(i) We extend the semantics of program locations to describe locations parameterized by timepoints, allowing us to express values of program variables at arbitrary program locations within arbitrary loop iterations. We write for example $i(l_{12}(it)))$ to denote the value of program variable `i` at location l_{12} in a loop iteration it, where the location l_{12} corresponds to the program line 12. We reserve the constant end for specifying the last program location l_{15}, that is line 15, corresponding to a terminating program execution of Figure 1. We then write $b(end, k)$ to capture the value of array `b` at timepoint end and position k. For simplicity, as `a` is a constant array, we simply write $a(k)$ instead $a(end, k)$.

(ii) Exploiting the semantics of program locations, we formalize the safety assertion of line 15 in trace logic $\mathcal{L}$ as follows:

$$\forall k_{\mathbb{I}}.\exists l_{\mathbb{I}}.\big((0 \leq k < j(end) \wedge a.length \geq 0) \\ \rightarrow b(end, k) \simeq a(l)\big) \tag{1}$$

(iii) We express the semantics of Figure 1 as a set $\mathcal{S}$ of first-order formulas in trace logic $\mathcal{L}$, encoding values and dependencies among program variables at arbitrary loop iterations. To this end, we extend $\mathcal{S}$ with so-called trace lemmas, to automate inductive reasoning in trace logic $\mathcal{L}$. One such trace lemma exploits the semantics of updates to `j`, allowing us to infer that *every* value of j between 0 to $j(end)$, and thus each position at which the array `b` has been updated, is given by *some* loop iteration. Moreover, updates to `j` happen at different loop iterations and thus a position `j` at which `b` is updated is visited uniquely throughout Figure 1.

(iv) We finally establish validity of (1), by deriving (1) to be a logical consequence of $\mathcal{S}$.

III. Preliminaries

We assume familiarity with standard first-order logic with equality and sorts. We write $\simeq$ for equality and x_S to denote that a logical variable x has sort S. We denote by $\mathbb{I}$ the set of integer numbers and by $\mathbb{B}$ the boolean sort. The term algebra of natural numbers is denoted by $\mathbb{N}$, with constructors `0` and successor `suc`. We also consider the symbols `pred` and $\leq$ as part of the signature of $\mathbb{N}$, interpreted respectively as the predecessor function and less-than-equal relation.

Let P be a first-order formula with one free variable x of sort $\mathbb{N}$. We recall the standard (step-wise) induction schema for natural numbers as being

$$\Big(P(\texttt{0}) \wedge \forall x'_{\mathbb{N}}.\big(P(x') \rightarrow P(\texttt{suc}(x'))\big)\Big) \rightarrow \forall x_{\mathbb{N}}.P(x) \tag{2}$$

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

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

In our work, we use a variation of the induction schema (2) to reason about intervals of loop iterations. Namely, we use the following schema of *bounded induction*

$$\Big(P(bl) \wedge \qquad \text{(base case)}$$
$$\forall x'_{\mathbb{N}}.\Big(\big(bl \leq x' < br \wedge P(x')\big) \rightarrow P(\texttt{suc}(x'))\Big)\Big) \qquad \text{(inductive case)}$$
$$\rightarrow \forall x_{\mathbb{N}}.\Big(bl \leq x \leq br \rightarrow P(x)\Big),$$

where $bl, br \in \mathbb{N}$ are term algebra expressions of $\mathbb{N}$, called respectively as left and right bounds of bounded induction.

IV. Programming Model $\mathcal{W}$

We consider programs written in an imperative while-like programming language $\mathcal{W}$. This section recalls terminology from [6], however adapted to our setting of safety verification. Unlike [6], we do not consider multiple program traces in $\mathcal{W}$. In Section V, we then introduce a generalized program semantics in trace logic $\mathcal{L}$, extended with reachability predicates.

Figure 2 shows the (partial) grammar of our programming model $\mathcal{W}$, emphasizing the use of contexts to capture lists of statements. An input program in $\mathcal{W}$ has a single `main`-function, with arbitrary nestings of if-then-else conditionals and while-statements. We consider *mutable and constant variables*, where variables are either integer-valued numeric variables or arrays of such numeric variables. We include standard *side-effect free expressions over booleans and integers*.

A. Locations and Timepoints

A program in $\mathcal{W}$ is considered as sets of locations, with each location corresponding to positions/lines of program statements in the program. Given a program statement `s`, we denote by l_s its (program) location. We reserve the location l_{end} to denote the end of a program. For programs with loops, some program locations might be revisited multiple times. We therefore model locations l_s corresponding to a statement `s` as functions of *iterations* when the respective location is visited. For simplicity, we write l_s also for the functional representation of the location l_s of `s`. We thus consider locations as timepoints of a program and treat them l_s as being functions l_s over iterations. The target sort of locations l_s is $\mathbb{L}$. For each enclosing loop of a statement `s`, the function symbol l_s takes arguments of sort $\mathbb{N}$, corresponding to loop iterations. Further, when `s` is a loop itself, we also

introduce a function symbol n_s with argument and target sort $\mathbb{N}$; intuitively, n_s corresponds to the last loop iteration of s. We denote the set of all function symbols l_s as S_{Tp}, whereas the set of all function symbols n_s is written as S_n.

***Example** 1:* We refer to program statements s by their (first) line number in Figure 1. Thus, l_5 encodes the timepoint corresponding to the first assignment of i in the program (line 5). We write $l_7(0)$ and $l_7(n_7)$ to denote the timepoints of the first and last loop iteration, respectively. The timepoints $l_8(\mathtt{suc}(0))$ and $l_8(it)$ correspond to the beginning of the loop body in the second and the it-th loop iterations, respectively. □

B. Expressions over Timepoints

We next introduce commonly used expressions over timepoints. For each while-statement w of $\mathcal{W}$, we introduce a function $it^{\mathtt{w}}$ that returns a unique variable of sort $\mathbb{N}$ for w, denoting loop iterations of w. Let $w_1, \ldots, w_k$ be the enclosing loops for statement s and consider an arbitrary term it of sort $\mathbb{N}$. We define $tp_{\mathtt{s}}$ to be the expressions denoting the timepoints of statements s as

$$\begin{aligned} tp_{\mathtt{s}} &:= l_s(it^{w_1}, \ldots, it^{w_k}) && \text{if s is non-while statement} \\ tp_{\mathtt{s}}(it) &:= l_s(it^{w_1}, \ldots, it^{w_k}, it) && \text{if s is while-statement} \\ lastIt_{\mathtt{s}} &:= n_s(it^{w_1}, \ldots, it^{w_k}) && \text{if s is while-statement} \end{aligned}$$

If s is a while-statement, we also introduce $lastIt_{\mathtt{s}}$ to denote the last iteration of s. Further, consider an arbitrary subprogram p, that is, p is either a statement or a context. The timepoint $start_{\mathtt{p}}$ (parameterized by an iteration of each enclosing loop) denotes the timepoint when the execution of p has started and is defined as

$$start_{\mathtt{p}} := \begin{cases} tp_{\mathtt{p}}(0) & \text{if p is while-statement} \\ tp_{\mathtt{p}} & \text{if p is non-while statement} \\ start_{\mathtt{s}_1} & \text{if p is context } \mathtt{s}_1;\ldots;\mathtt{s}_k \end{cases}$$

We also introduce the timepoint $end_{\mathtt{p}}$ to denote the timepoint upon which a subprogram p has been completely evaluated and define it as

$$end_{\mathtt{p}} := \begin{cases} start_{\mathtt{s}} & \text{if s occurs after p in a context} \\ end_{\mathtt{c}} & \text{if p is last statement in context c} \\ end_{\mathtt{s}} & \text{if p is context of if-branch or else-branch of s} \\ tp_{\mathtt{s}}(\mathtt{suc}(it^s)) & \text{if p is context of body of s} \\ l_{end} & \text{if p is top-level context} \end{cases}$$

Finally, if s is the topmost statement of the top-level context in main(), we define

$$start := start_s.$$

C. Program Variables

We express values of program variables v at various timepoints of the program execution. To this end, we model (numeric) variables v as functions $v : \mathbb{L} \mapsto \mathbb{I}$, where $v(tp)$ gives the value of v at timepoint tp. For array variables v, we add an additional argument of sort $\mathbb{I}$, corresponding to the position where the array is accessed; that is, $v : \mathbb{L} \times \mathbb{I} \mapsto \mathbb{I}$. The set of such function symbols corresponding to program variables is denoted by S_V.

Our framework for constant, non-mutable variables can be simplified by omitting the timepoint argument in the functional representation of such program variables, as illustrated below.

***Example** 2:* For Figure 1, we denote by $i(l_5)$ the value of program variable i before being assigned in line 5. As the array variable a is non-mutable (specified by const in the program), we write $a(i(l_8(it)))$ for the value of array a at the position corresponding to the current value of i at timepoint $l_8(it)$. For the mutable array b, we consider timepoints where b has been updated and write $b(l_9(it), j(l_9(it)))$ for the array b at position j at the timepoint $l_9(it)$ during the loop. □

We emphasize that we consider (numeric) program variables v to be of sort $\mathbb{I}$, whereas loop iterations it are of sort $\mathbb{N}$.

D. Program Expressions

Arithmetic constants and program expressions are modeled using integer functions and predicates. Let e be an arbitrary program expression and write $[\![\mathtt{e}]\!](tp)$ to denote the value of the evaluation of e at timepoint tp. Let $v \in S_V$, that is a function v denoting a program variable v. Consider $\mathtt{e}, \mathtt{e}_1, \mathtt{e}_2$ to be program expressions and let tp_1, tp_2 denote two timepoints. We define

$$Eq(v, tp_1, tp_2) := \begin{cases} \forall pos_{\mathbb{I}}.\ v(tp_1, pos) \simeq v(tp_2, pos), & \text{if v is an array} \\ v(tp_1) \simeq v(tp_2), & \text{otherwise} \end{cases}$$

to denote that the program variable v has the same values at tp_1 and tp_2. We further introduce

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

to define that all program variables have the same values at timepoints tp_1 and tp_2. We also define

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

asserting that the numeric program variable v has been updated while all other program variables v′ remain unchanged. This definition is further extended to array updates as

$$\begin{aligned} &UpdateArr(v, e_1, e_2, tp_1, tp_2) := \\ &\quad \forall pos_{\mathbb{I}}.\ (pos \not\simeq [\![e_1]\!](tp_1) \rightarrow v(tp_2, pos) \simeq v(tp_1, pos)) \\ &\quad \wedge\ v(tp_2, [\![e_1]\!](tp_1)) \simeq [\![e_2]\!](tp_1) \\ &\quad \textstyle\bigwedge_{v' \in S_V \setminus \{v\}} Eq(v', tp_1, tp_2). \end{aligned}$$

***Example** 3:* In Figure 1, we refer to the value of i+1 at timepoint $l_{12}(it)$ as $i(l_{12}(it))+1$. Let S_V^l be the set of function symbols representing the program variables of Figure 1. For an update of j in line 10 at some iteration it, we derive

$$\begin{aligned} Update(j, \mathtt{j+1}, l_9(it), l_{10}(it)) &:= j(l_{10}(it)) \simeq (j(l_9(it)) + 1) \\ \wedge \bigwedge_{v' \in S_V^l \setminus \{j\}} & Eq(v', l_9(it), l_{10}(it)). \end{aligned}$$

□

V. Axiomatic Semantics in Trace Logic $\mathcal{L}$

Trace logic $\mathcal{L}$ has been introduced in [6], yet for the setting of relational verification. In this paper we generalize the formalization of [6] in three ways. First, (i) we define program semantics in a non-recursive manner using the $Reach$ predicate to characterize the set of reachable locations within a given program context (Section V-B). Second, and most importantly, (ii) we prove completeness of trace logic $\mathcal{L}$ with respect to Hoare Logic (Theorem 2), which could have not been achieved in the setting of [6]. Finally, (iii) we introduce the use of logic $\mathcal{L}$ for safety verification (Section VI).

A. Trace Logic $\mathcal{L}$

Trace logic $\mathcal{L}$ is an instance of many-sorted first-order logic with equality. We define the signature $\Sigma(\mathcal{L})$ of trace logic as

$$\Sigma(\mathcal{L}) := S_{\mathbb{N}} \cup S_{\mathbb{I}} \cup S_{Tp} \cup S_V \cup S_n,$$

containing the signatures of the theory of natural numbers (term algebra) $\mathbb{N}$ and integers $\mathbb{I}$, as well the respective sets of timepoints, program variables and last iteration symbols as defined in section IV.
We next define the semantics of $\mathcal{W}$ in trace logic $\mathcal{L}$.

B. Reachability and its Axiomatization

We introduce a predicate $Reach : \mathbb{L} \mapsto \mathbb{B}$ to capture the set of timepoints reachable in an execution and use $Reach$ to define the axiomatic semantics of $\mathcal{W}$ in trace logic $\mathcal{L}$. We define reachability $Reach$ as a predicate over timepoints, in contrast to defining reachability as a predicate over program configurations such as in [16], [7], [5], [10].
We axiomatize $Reach$ using trace logic formulas as follows.

***Definition** 1 (Reach-predicate):* For any context c, any statement $\mathtt{s}$, let $Cond_s$ be the expression denoting a potential branching condition in $\mathtt{s}$. We define

$$Reach(start_c) := \begin{cases} true, \\ \quad \text{if } \mathtt{c} \text{ is top-level context} \\ Reach(start_s) \wedge Cond_s(start_s), \\ \quad \text{if } \mathtt{c} \text{ is context of if-branch of } \mathtt{s} \\ Reach(start_s) \wedge \neg Cond_s(start_s), \\ \quad \text{if } \mathtt{c} \text{ is context of else-branch of } \mathtt{s} \\ Reach(start_s) \wedge it^s < lastIt_s, \\ \quad \text{if } \mathtt{c} \text{ is context of body of } \mathtt{s}. \end{cases}$$

For any non-while statement $\mathtt{s}'$ occurring in context $\mathtt{c}$, let

$$Reach(start_{s'}) := Reach(start_c),$$

and for any while-statement $\mathtt{s}'$ occurring in context $\mathtt{c}$, let

$$Reach(tp_{s'}(it^{s'})) := Reach(start_c) \wedge it^{s'} \leq lastIt_{s'}.$$

Finally let $Reach(end) := true$. □

Note that our reachability predicate $Reach$ allows specifying properties about intermediate timepoints (since those properties can only hold if the referred timepoints are reached) and supports reasoning about which locations are reached.

C. Axiomatic Semantics of $\mathcal{W}$

We axiomatize the semantics of each program statement in $\mathcal{W}$, and define the semantics of a program in $\mathcal{W}$ as the conjunction of all these axioms.

a) Main-function: Let $\mathtt{p_0}$ be an arbitrary, but fixed program in $\mathcal{W}$; we give our definitions relative to $\mathtt{p_0}$. The semantics of $\mathtt{p_0}$, denoted by $[\![\mathtt{p_0}]\!]$, consists of a conjunction of one implication per statement, where each implication has the reachability of the start-timepoint of the statement as premise and the semantics of the statement as conclusion:

$$[\![\mathtt{p_0}]\!] := \bigwedge_{\mathtt{s} \text{ statement of } \mathtt{p_0}} \forall enclIts.\big(Reach(start_s) \rightarrow [\![\mathtt{s}]\!]\big)$$

where $enclIts$ is the set of iterations $\{it^{w_1}, \ldots, it^{w_n}\}$ of all enclosing loops $w_1, \ldots, w_n$ of some statement $\mathtt{s}$ in $\mathtt{p_0}$, and the semantics $[\![\mathtt{s}]\!]$ of program statements $\mathtt{s}$ is defined as follows.

b) Skip: Let $\mathtt{s}$ be a statement `skip`. Then

$$[\![\mathtt{s}]\!] := EqAll(end_{\mathtt{s}}, start_{\mathtt{s}}) \tag{3}$$

c) Integer assignments: Let $\mathtt{s}$ be an assignment `v = e`, where `v` is an integer-valued program variable and `e` is an expression. The evaluation of $\mathtt{s}$ is performed in one step such that, after the evaluation, the variable `v` has the same value as `e` before the evaluation. All other variables remain unchanged and thus

$$[\![\mathtt{s}]\!] := Update(v, e, end_s, start_s) \tag{4}$$

d) Array assignments: Consider $\mathtt{s}$ of the form `a[e1] = e2`, with `a` being an array variable and $\mathtt{e_1}, \mathtt{e_2}$ being expressions. The assignment is evaluated in one step. After the evaluation of $\mathtt{s}$, the array `a` contains the value of $\mathtt{e_2}$ before the evaluation at position pos corresponding to the value of $\mathtt{e_1}$ before the evaluation. The values at all other positions of `a` and all other program variables remain unchanged and hence

$$[\![\mathtt{s}]\!] := UpdateArr(v, e_1, e_2, end_s, start_s) \tag{5}$$

e) Conditional if-then-else Statements: Let $\mathtt{s}$ be `if(Cond){c1} else {c2}`. The semantics of $\mathtt{s}$ states that entering the if-branch and/or entering the else-branch does not change the values of the variables and we have

$$[\![\mathtt{s}]\!] := [\![\mathtt{Cond}]\!](start_s) \rightarrow EqAll(start_{\mathtt{c_1}}, start_s) \tag{6a}$$
$$\wedge \quad \neg[\![\mathtt{Cond}]\!](start_s) \rightarrow EqAll(start_{\mathtt{c_2}}, start_s) \tag{6b}$$

where the semantics $[\![\mathtt{Cond}]\!]$ of the expression `Cond` is according to Section IV-D.

f) While-Statements: Let $\mathtt{s}$ be the while-statement `while(Cond){c}`. We refer to `Cond` as the *loop condition*. The semantics of $\mathtt{s}$ is captured by conjunction of the following three properties: (7a) the iteration $lastIt_s$ is the first iteration where `Cond` does not hold, (7b) entering the loop body does not change the values of the variables, (7c) the values of the variables at the end of evaluating $\mathtt{s}$ are the same as the variable values at the loop condition location in iteration $lastIt_s$. As

such, we have

$$\begin{aligned}
[\![\mathtt{s}]\!] := \quad & \forall it^{s}_{\mathbb{N}}.\ (it^{s} < lastIt_{\mathtt{s}} \rightarrow [\![\mathtt{Cond}]\!](tp_{\mathtt{s}}(it^{s}))) \\
\wedge \quad & \neg[\![\mathtt{Cond}]\!](tp(lastIt_{\mathtt{s}})) && \text{(7a)} \\
\wedge \quad & \forall it^{s}_{\mathbb{N}}.\ (it^{s} < lastIt_{\mathtt{s}} \rightarrow EqAll(start_{\mathtt{c}}, tp_{\mathtt{s}}(it^{s})) && \text{(7b)} \\
\wedge \quad & EqAll(end_{\mathtt{s}}, tp_{s}(lastIt_{\mathtt{s}})) && \text{(7c)}
\end{aligned}$$

D. Soundness and Completeness.

The axiomatic semantics of $\mathcal{W}$ in trace logic is sound. That is, given a program $\mathtt{p}$ in $\mathcal{W}$ and a trace logic property $F \in \mathcal{L}$, we have that any interpretation in $\mathcal{L}$ is a model of F according to the small-step operational semantics of $\mathcal{W}$. We conclude the next theorem - and refer to [17] for details.

***Theorem** 1 ($\mathcal{W}$-Soundness):* Let $\mathtt{p}$ be a program. Then the axiomatic semantics $[\![\mathtt{p}]\!]$ is sound with respect to standard small-step operational semantics. □

Next, we show that the axiomatic semantics of $\mathcal{W}$ in trace logic $\mathcal{L}$ is complete with respect to Hoare logic [18], as follows. Intuitively, a Hoare Triple $\{F_1\}\mathtt{p}\{F_2\}$ corresponds to the trace logic formula

$$\forall enclIts.\big(Reach(start_p) \rightarrow ([F_1](start_p) \rightarrow [F_2](end_p))\big) \quad (8)$$

where the expressions $[F_1](start_p)$ and $[F_2](end_p)$ denote the result of adding to each program variable in F_1 and F_2 the timepoints $start_p$ respectively end_p as first arguments. We therefore define that the axiomatic semantics of $\mathcal{W}$ is *complete with respect to Hoare logic*, if for any Hoare triple $\{F_1\}\mathtt{p}\{F_2\}$ valid relative to the background theory $\mathcal{T}$, the corresponding trace logic formula (8) is derivable from the axiomatic semantics of $\mathcal{W}$ in the background theory $\mathcal{T}$. With this definition at hand, we get the following result, proved formally in [17].

***Theorem** 2 ($\mathcal{W}$-Completeness with respect to Hoare logic):* The axiomatic semantics of $\mathcal{W}$ in trace logic is complete with respect to Hoare logic. □

VI. Trace Logic for Safety Verification

We now introduce the use of trace logic $\mathcal{L}$ for verifying safety properties of $\mathcal{W}$ programs. We consider safety properties F expressed in first-order logic with theories, as illustrated in line 15 of Figure 1. Thanks to soundness and completeness of the axiomatic semantics of $\mathcal{W}$, a partially correct program $\mathtt{p}$ with regard to F can be proved to be correct using the axiomatic semantics of $\mathcal{W}$ in trace logic $\mathcal{L}$. That is, we assume termination and establish partial program correctness. Assuming the existence of an iteration violating the loop condition can be help backward reasoning and, in particular, automatic splitting of loop iteration intervals.

However, proving correctness of a program $\mathtt{p}$ annotated with a safety property F faces the reasoning challenges of the underlying logic, in our case of trace logic. Due to the presence of loops in $\mathcal{W}$, a challenging aspect in using trace logic for safety verification is to handle inductive reasoning as induction cannot be generally expressed in first-order logic. To circumvent the challenge of inductive reasoning and automate verification using trace logic, we introduce a set of first-order lemmas, called *trace lemmas*, and extend the semantics of $\mathcal{W}$ programs in trace logic with these trace lemmas. Trace lemmas describe generic inductive properties over arbitrary loop iterations and any logical consequence of trace lemmas yields a valid program loop property as well. We next summarize our approach to program verification using trace logic and then address the challenge of inductive reasoning in trace logic $\mathcal{L}$.

A. Safety Verification in Trace Logic

Given a program $\mathtt{p}$ in $\mathcal{W}$ and a safety property F,

(i) we express program semantics $[\![\mathtt{p}]\!]$ in trace logic $\mathcal{L}$, as given in Section V;

(ii) we formalize the safety property in trace logic $\mathcal{L}$, that is we express F by using program variables as functions of locations and timepoints (similarly as in (1)). For simplicity, let us denote the trace logic formalization of F also by F;

(iii) we introduce instances $\mathcal{T}^{\mathtt{p}}_{\mathcal{L}}$ of a set $\mathcal{T}_{\mathcal{L}}$ of trace lemmas, by instantiating trace lemmas with program variables, locations and timepoints of $\mathtt{p}$;

(iv) to verify F, we then show that F is a logical consequence of $[\![\mathtt{p}]\!] \wedge \mathcal{T}^{\mathtt{p}}_{\mathcal{L}}$;

(v) however to conclude that $\mathtt{p}$ is partially correct with regard to F, two more challenges need to be addressed. First, in addition to Theorem 1, soundness of our trace lemmas $\mathcal{T}_{\mathcal{L}}$ needs to be established, implying that our trace lemma instances $\mathcal{T}^{\mathtt{p}}_{\mathcal{L}}$ are also sound. Soundness of $\mathcal{T}^{\mathtt{p}}_{\mathcal{L}}$ implies then validity of F, whenever F is proven to be a logical consequence of sound formulas $[\![\mathtt{p}]\!] \wedge \mathcal{T}^{\mathtt{p}}_{\mathcal{L}}$. However, to ensure that F is provable in trace logic, as a second challenge we need to ensure that our trace lemmas $\mathcal{T}_{\mathcal{L}}$, and thus their instances $\mathcal{T}^{\mathtt{p}}_{\mathcal{L}}$, are strong enough to prove $[\![\mathtt{p}]\!] \wedge \mathcal{T}^{\mathtt{p}}_{\mathcal{L}} \implies F$. That is, proving that F is a safety assertion of $\mathtt{p}$ in our setting requires finding a suitable set $\mathcal{T}_{\mathcal{L}}$ of trace lemmas.

In the remaining of this section, we address (v) and show that our trace lemmas $\mathcal{T}_{\mathcal{L}}$ are sound consequences of bounded induction (Section VI-B). Practical evidence for using our trace lemmas are further given in Section VII-B.

B. Trace Lemmas $\mathcal{T}_{\mathcal{L}}$ for Verification

Trace logic properties support arbitrary quantification over timepoints and describe values of program variables at arbitrary loop iterations and timepoints. We therefore can relate timepoints with values of program variables in trace logic $\mathcal{L}$, allowing us to describe the value distributions of program variables as functions of timepoints throughout program executions. As such, trace logic $\mathcal{L}$ supports

(1) reasoning about the *existence* of a specific loop iteration, allowing us to split the range of loop iterations at a particular timepoint, based on the safety property we want to prove. For example, we can express and derive loop iterations corresponding to timepoints where one program variable takes a specific value for *the first time during loop execution*;

(2) universal quantification over the array content and range of loop iterations bounded by two arbitrary left and right bounds, allowing us to apply instances of the induction scheme (3) within a range of loop iterations bounded, for example, by it and $lastIt_s$ for some while-statement $\mathtt{s}$.

Addressing these benefits of trace logic, we express generic patterns of inductive program properties as *trace lemmas*. Identifying a suitable set $\mathcal{T}_\mathcal{L}$ of trace lemmas to automate inductive reasoning in trace logic $\mathcal{L}$ is however challenging and domain-specific. We propose three trace lemmas for inductive reasoning over arrays and integers, by considering

(A1) one trace lemma describing how values of program variables change during an interval of loop iterations;

(B1-B2) two trace lemmas to describe the behavior of loop counters.

We prove soundness of our trace lemmas - below we include only one proof and refer to [17] for further details.

(A1) Value Evolution Trace Lemma: Let $\mathtt{w}$ be a while-statement, let $\mathtt{v}$ be a mutable program variable and let $\circ$ be a reflexive and transitive relation - that is $\simeq$ or $\leq$ in the setting of trace logic. The *value evolution trace lemma of* $\mathtt{w}$*,* $\mathtt{v}$*, and* $\circ$ is defined as

$$\begin{aligned}&\forall bl_{\mathbb{N}}, br_{\mathbb{N}}.\\ &\Big(\forall it_{\mathbb{N}}.\Big((bl \leq it < br \wedge v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it)))\\ &\qquad \rightarrow v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(\mathtt{suc}(it)))\Big)\\ &\quad \rightarrow \big(bl \leq br \rightarrow v(tp_{\mathtt{w}}(br)) \circ v(tp_{\mathtt{w}}(br))\big)\Big)\end{aligned} \tag{A1}$$

In our work, the value evolution trace lemma is mainly instantiated with the equality predicate $\simeq$ to conclude that the value of a variable does not change during a range of loop iterations, provided that the variable value does not change at any of the considered loop iterations.

Example 4: For Figure 1, the value evaluation trace lemma (A1) yields the property

$$\begin{aligned}&\forall j_{\mathbb{I}}.\ \forall bl_{\mathbb{N}}.\ \forall br_{\mathbb{N}}.\\ &\Big(\forall it_{\mathbb{N}}.\Big((bl \leq it < br\ \wedge\ b(l_8(bl), j) = b(l_8(it), j))\\ &\qquad \rightarrow b(l_8(bl), j) = b(l_8(s(it)), j)\Big)\\ &\rightarrow \big(bl \leq br \rightarrow b(l_8(bl), j) = b(l_8(br), j)\big)\Big),\end{aligned}$$

which allows to prove that the value of $\mathtt{b}$ at some position $\mathtt{j}$ remains the same from the timepoint it the value was first set until the end of program execution. That is, we derive $b(l_9(end), j(l_9(it))) = a(i(l_8(it)))$. □

We next prove soundness of our trace lemma (A1).

Proof *(Soundness Proof of Value Evolution Trace Lemma* (A1)*)* Let bl and br be arbitrary but fixed and assume that the premise of the outermost implication of (A1) holds. That is,

$$\begin{aligned}&\forall it_{\mathbb{N}}.\big((bl \leq it < br \wedge v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it)))\\ &\qquad \rightarrow v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(\mathtt{suc}(it)))\big)\end{aligned} \tag{9}$$

We use the induction axiom scheme (3) and consider its instance with $P(it) := v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it))$, yielding the following instance of (3):

$$\Big(v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it)) \quad \wedge \tag{10a}$$

$$\forall it_{\mathbb{N}}.\big((bl \leq it < br \wedge v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it))) \rightarrow v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(\mathtt{suc}(it)))\big)\Big) \tag{10b}$$

$$\rightarrow \forall it_{\mathbb{N}}.\Big(bl \leq it \leq br \rightarrow v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it))\Big) \tag{10c}$$

Note that the base case property (10a) holds since $\circ$ is reflexive. Further, the inductive case (10b) holds also since it is implied by (9). We thus derive property (10c), and in particular $bl \leq br \leq br \rightarrow v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(br))$. Since $\leq$ is reflexive, we conclude $bl \leq br \rightarrow v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(br))$, proving thus our trace lemma (A1). □

(B1) Intermediate Value Trace Lemma: Let $\mathtt{w}$ be a while-statement and let $\mathtt{v}$ be a mutable program variable. We call $\mathtt{v}$ to be *dense* if the following holds:

$$\begin{aligned}Dense_{w,v} := \forall it_{\mathbb{N}}.\Big(&it < lastIt_{\mathtt{w}} \rightarrow\\ &\big(v(tp_{\mathtt{w}}(\mathtt{suc}(it))) = v(tp_{\mathtt{w}}(it)) \vee\\ &v(tp_{\mathtt{w}}(\mathtt{suc}(it))) = v(tp_{\mathtt{w}}(it)) + 1\big)\Big)\end{aligned}$$

The *intermediate value trace lemma of* $\mathtt{w}$ *and* $\mathtt{v}$ is defined as

$$\begin{aligned}&\forall x_{\mathbb{I}}.\Big(\big(Dense_{w,v} \wedge v(tp_{\mathtt{w}}(\mathtt{0})) \leq x < v(tp_{\mathtt{w}}(lastIt_{\mathtt{w}}))\big) \rightarrow\\ &\quad \exists it_{\mathbb{N}}.\big(it < lastIt_{\mathtt{w}} \wedge\ v(tp_{\mathtt{w}}(it)) \simeq x\ \wedge\\ &\qquad v(tp_{\mathtt{w}}(\mathtt{suc}(it))) \simeq v(tp_{\mathtt{w}}(it)) + 1\big)\Big)\end{aligned} \tag{B1}$$

The intermediate value trace lemma (B1) allows us conclude that if the variable $\mathtt{v}$ is dense, and if the value x is between the value of $\mathtt{v}$ at the beginning of the loop and the value of $\mathtt{v}$ at the end of the loop, then there is an iteration in the loop, where $\mathtt{v}$ has exactly the value x and is incremented. This trace lemma is mostly used to find specific iterations corresponding to positions x in an array.

Example 5: In Figure 1, using trace lemma (B1) we synthesize the iteration it such that $b(l_9(it), j(l_9(it))) = a(i(l_8(it)))$. □

(B2) Iteration Injectivity Trace Lemma: Let $\mathtt{w}$ be a while-statement and let $\mathtt{v}$ be a mutable program variable. The *iteration injectivity trace lemma of* $\mathtt{w}$ *and* $\mathtt{v}$ is

$$\begin{aligned}&\forall it^1_{\mathbb{N}}, it^2_{\mathbb{N}}.\Big(\big(Dense_{w,v} \wedge v(tp_{\mathtt{w}}(\mathtt{suc}(it^1))) = v(tp_{\mathtt{w}}(it^1)) + 1\\ &\qquad \wedge it^1 < it^2 \leq lastIt_{\mathtt{w}}\big)\\ &\qquad \rightarrow v(tp_{\mathtt{w}}(it^1)) \not\simeq v(tp_{\mathtt{w}}(it^2))\Big)\end{aligned} \tag{B2}$$

The trace lemma (B2) states that a strongly-dense variable visits each array-position at most once. As a consequence, if each array position is visited only once in a loop, we know that its value has not changed after the first visit, and in particular the value at the end of the loop is the value after the first visit.

Example 6: Trace lemma (B2) is necessary in Figure 1 to apply the value evolution trace lemma (A1) for $\mathtt{b}$, as we need to make sure we will never reach the same position of $\mathtt{j}$ twice. □

Based on the soundness of our trace lemmas, we conclude the next result.

***Theorem** 3 (Trace Lemmas and Induction):* Let p be a program. Let L be a trace lemma for some while-statement w of p and some variable v of p. Then L is a consequence of the bounded induction scheme (3) and of the axiomatic semantics of $[\![\mathtt{p}]\!]$ in trace logic $\mathcal{L}$. □

VII. Implementation and Experiments

A. Implementation

We implemented our approach in the RAPID tool, written in C++ and available at https://github.com/gleiss/rapid. RAPID takes as input a program in the while-language $\mathcal{W}$ together with a property expressed in trace logic $\mathcal{L}$ using the SMT-LIB syntax [19]. RAPID outputs (i) the program semantics as in Section V, (ii) instantiations of trace lemmas for each mutable variable and for each loop of the program, as discussed in Section VI-B, and (iii) the safety property, expressed in trace logic $\mathcal{L}$ and encoded in the SMT-LIB syntax.

For establishing safety, we pass the generated reasoning task to the first-order theorem prover VAMPIRE [20] to prove the safety property from the program semantics and the instantiated trace lemmas[1], as discussed in Section VI-A. VAMPIRE searches for a proof by refuting the negation of the property based on saturation of a set of clauses with respect to a set of inference rules such as resolution and superposition.

In our experiments, we use a custom version[2] of VAMPIRE with a timeout of 60 seconds, in two different configurations. On the one hand, we use a configuration RAPID$^-$, where we tune VAMPIRE to the trace logic domain using (i) existing options and (ii) domain-specific implementation to guide the high-level proof search. On the other hand, we use a configuration RAPID*, which extends RAPID$^-$ with recent techniques from [11], [12] improving theory reasoning in equational theories. As such, RAPID* represents the result of a fundamental effort to improve VAMPIRE's reasoning for software verification. In particular, theory split queues [12] present a partial solution to the prevalent challenge of combining quantification and *light-weight* theory reasoning, drastically improving first-order reasoning in applications of software verification, as shown next.

B. Experimental Results

We considered challenging Java- and C-like verification benchmarks from the SV-Comp repository [14], containing the combination of loops and arrays. We omitted those examples for which the task is to find bugs in form of counterexample traces, as well as those examples that cannot be expressed in our programming model $\mathcal{W}$, such as examples with explicit memory management. In order to improve the set of benchmarks, we also included additional challenging programs and functional properties. As a result, we obtained benchmarks ranging over 45 unique programs with a total of 103 tested properties. Our benchmarks are available in the RAPID repository[3].

We manually transformed those benchmarks into our input format. SV-Comp benchmarks encode properties featuring universal quantification by extending the corresponding program with an additional loop containing a standard C-like assertion. For instance, the property

$$\forall i_{\mathbb{I}}.\ 0 \leq i < a.length \rightarrow P(a(i, end))$$

would be encoded by extending the program with a loop

```
for(int i = 0; i < a.length; i++)
assert(P(a[i]))
```

While this encoding loses explicit structure and results in a harder reasoning task, it is necessary as other tools do not support explicit universal quantification in their input language. In contrast, our approach can handle arbitrarily quantified properties over unbounded data structures. We, thus, directly formulate universally quantified properties, without using any program transformations.

The results of our experiments are presented in Table 1. We divided the results in four segments in the following order: the first eleven problems are quantifier-free, the largest part of 62 problems are universally quantified, seven problems are existentially quantified, while the last 23 problems contain quantifier alternations. First, we are interested in the overall number of problems we are able to prove correct. In the configuration RAPID*, which represents our main configuration, VAMPIRE is able to prove 78 out of 103 encodings. In particular, we verify Figure 1, corresponding to benchmark copy_positive_1, as well as other challenging properties that involve quantifier alternations, such as partition_5.

Second, we are interested in comparing the results for configurations RAPID$^-$ and RAPID*, in order to understand the importance of recently developed techniques from [11] and [12] for reasoning in the trace logic domain. While RAPID$^-$ is only able to prove 15 out of 103 properties, RAPID* is able to prove 78 properties, that is, RAPID* improves over RAPID$^-$ by 63 examples. Moreover, only RAPID* is able to prove advanced properties involving quantifier alternations. We therefore see that RAPID* drastically outperforms RAPID$^-$, suggesting that the recently developed techniques are essential for efficient reasoning in trace logic.

Third, we are interested in what kinds of properties RAPID can prove. It comes with no surprise that all quantifier-free instances could be proved. Out of 62 universally quantified properties, RAPID could establish correctness of 53 such properties. More interestingly, RAPID proves 14 out of 30 benchmarks containing either existentially quantified properties or such with quantifier alternations. The benchmarks that could not be solved by RAPID are primarily universally and alternatingly quantified properties that need additional trace lemmas relating values of multiple program variables.

[1] We also established the soundness of each trace lemma instance separately by running additional validity queries with VAMPIRE.

[2] https://github.com/vprover/vampire/tree/gleiss-rapid

[3] https://github.com/gleiss/rapid/tree/master/examples/arrays

TABLE I
EXPERIMENTAL RESULTS

Benchmark	RAPID$^-$	RAPID*	Benchmark	RAPID$^-$	RAPID*	Benchmark	RAPID$^-$	RAPID*
atleast_one_iteration_0	✓	✓	in_place_max	-	✓	swap_1	-	✓
atleast_one_iteration_1	✓	✓	inc_by_one_0	-	✓	vector_addition	-	✓
find_sentinel	✓	✓	inc_by_one_1	-	✓	vector_subtraction	-	✓
find1_0	-	✓	inc_by_one_harder_0	-	✓	check_equal_set_flag_0	✓	✓
find1_1	-	✓	inc_by_one_harder_1	-	✓	find_max_1	-	-
find2_0	-	✓	init	-	✓	find_max_from_second_1	-	-
find2_1	✓	✓	init_conditionally_0	-	✓	find1_2	✓	✓
indexn_is_arraylength_0	✓	✓	init_conditionally_1	-	✓	find1_3	✓	✓
indexn_is_arraylength_1	-	✓	init_non_constant_0	-	✓	find2_2	✓	✓
set_to_one	✓	✓	init_non_constant_1	-	✓	find2_3	✓	✓
str_cpy_3	✓	✓	init_non_constant_2	-	✓	collect_indices_eq_val_2	-	✓
both_or_none	-	✓	init_non_constant_3	-	✓	collect_indices_eq_val_3	-	-
check_equal_set_flag_1	-	✓	init_non_constant_easy_0	-	✓	copy_nonzero_1	-	✓
collect_indices_eq_val_0	-	✓	init_non_constant_easy_1	-	✓	copy_positive_1	-	✓
collect_indices_eq_val_1	-	✓	init_non_constant_easy_2	-	✓	find_max_local_0	-	-
copy	-	✓	init_non_constant_easy_3	-	✓	find_max_local_1	-	-
copy_absolute_0	-	✓	init_partial	-	✓	find_max_up_to_1	-	-
copy_absolute_1	-	✓	init_prev_plus_one_0	-	✓	find_min_1	-	-
copy_nonzero_0	-	✓	init_prev_plus_one_1	-	✓	find_min_local_0	-	-
copy_partial	-	✓	init_prev_plus_one_alt_0	-	✓	find_min_local_1	-	-
copy_positive_0	-	✓	init_prev_plus_one_alt_1	-	✓	find_min_up_to_1	-	-
copy_two_indices	-	✓	max_prop_0	-	✓	merge_interleave_2	-	-
find_max_0	-	✓	max_prop_1	-	✓	partition_2	-	✓
find_max_2	-	✓	merge_interleave_0	-	-	partition_3	-	✓
find_max_from_second_0	-	-	merge_interleave_1	-	-	partition_4	-	-
find_max_local_2	-	-	min_prop_0	-	✓	partition_5	-	✓
find_max_up_to_0	-	-	min_prop_1	-	✓	partition_6	-	-
find_max_up_to_2	-	-	partition_0	-	✓	partition-harder_0	-	✓
find_min_0	-	✓	partition_1	-	✓	partition-harder_1	-	✓
find_min_2	-	✓	push_back	-	✓	partition-harder_2	-	-
find_min_local_2	-	-	reverse	-	✓	partition-harder_3	-	-
find_min_up_to_0	-	-	str_cpy_0	-	✓	partition-harder_4	-	-
find_min_up_to_2	-	-	str_cpy_1	-	✓	str_len	✓	✓
find1_4	-	✓	str_cpy_2	✓	✓			
find2_4	✓	✓	swap_0	-	✓	**Total solved**	**15**	**78**

Comparing with other tools. We compare our work against other approaches in VIII. Here, we omit a direct comparison of RAPID with other tools for the following reasons:
(1) Our benchmark suite includes 62 universally quantified and 11 non-quantified properties that could technically be supported by state-of-the-art tools such as SPACER/SEAHORN and FREQHORN. Our benchmarks, however, also include 30 benchmarks with existential (7 examples) and alternating quantification (23 examples) that these tools cannot handle. As these examples depend on invariants that are alternatingly or at least existentially quantified, we believe these other tools cannot solve these benchmarks, while RAPID* could solve 14 examples in this domain.
(2) In our preliminary work [6], we already compared our reasoning within RAPID against Z3 and CVC4. These experiments showed that due to the fundamental difference in handling variables as functions over timepoints in our semantics, RAPID outperformed SMT-based reasoning approaches.
(3) Our program semantics is different than the one used in Horn clause verification techniques.

Concerning previous approaches with first-order reasoners, the benchmarks of [21] represent a subset of 55 examples from our current benchmark suite: only 21 examples from our benchmark suite could be proved by [21]. For instance, our example in Figure 1 could not be proven in [21]. We believe that our work can be combined with approaches from [22], [21] to non-trivial invariants and loop bounds from saturation-based proof search. Our work can, thus, complement existing tools in proving complex quantified properties.

VIII. RELATED WORK

Our work is closely related to recent efforts in using first-order theorem provers for proving software properties [22], [21]. While [21] captures programs semantics in the first-order language of extended expressions over loop iterations, in our work we further generalize the semantics of program locations and consider program expressions over loop iterations and arbitrary timepoints. Further, we introduce and prove trace lemmas to automate inductive reasoning based on bounded induction over loop iterations. Our generalizations in trace logic proved to be necessary to automate the verification of properties with arbitrary quantification, which could not be effectively achieved in [21]. Our work is not restricted to reasoning about single loops as in [21].
Compared to [6], we provide a non-recursive generalization of the axiomatic semantics of programs in trace logic, prove completeness of our axiomatization in trace logic, ensure soundness of our trace lemmas and use trace logic for safety verification.

In comparison to verification approaches based on program transformations [8], [9], [23], we do not require user-provided functions to transform program states to smaller-sized states [10], nor are we restricted to universal properties generated by symbolic executions [9]. Rather, we use only three trace lemmas that we prove sound and automate the verification of first-order properties, possibly with alternations of quantifiers.

The works [24], [25] consider expressive abstract domains and limit the generation of universal invariants to these domains, while supporting potentially more generic program grammars than our $\mathcal{W}$ language. Our work however can verify universal and/or existential first-order properties with theories, which is not the case in [8], [9], [24], [25]. Verifying universal loop properties with arrays by implicitly finding invariants is addressed in [4], [5], [26], [27], [28], [29], and by using constraint horn clause reasoning within property-driven reachability analysis in [16], [30].

Another line of research proposes abstraction and lazy interpolation [31], [32], as well as recurrence solving with SMT-based reasoning [33]. Synthesis-based approaches, such as [5], are shown to be successful when it comes to inferring universally quantified invariants and proving program correctness from these invariants. Synthesis-based term enumeration is used also in [23] in combination with user-provided invariant templates. Compared to these works, we do not consider programs only as a sequence of states, but model program values as functions of loop iterations and timepoints. We synthesize bounds on loop iterations and infer first-order loop invariants as logical consequences of our trace lemmas and program semantics in trace logic.

IX. Conclusion

We introduced trace logic to reason about safety loop properties over arrays. Trace logic supports explicit timepoint reasoning to allow arbitrary quantification over loop iterations. We use trace lemmas as consequences of bounded induction to automated inductive loop reasoning in trace logic. We formalize the axiomatic semantics of programs in trace logic and prove it to be both sound and complete. We report on our implementation in the RAPID framework, allowing us to use superposition-based reasoning in trace logic for verifying challenging verification examples. Generalizing our work to termination analysis and extending our programming language, and its semantics in trace logic, with more complex constructs are interesting tasks for future work.

Acknowledgements. This work was funded by the ERC Starting Grant 2014 SYMCAR 639270, the ERC Proof of Concept Grant 2018 SYMELS 842066, the Wallenberg Academy Fellowship 2014 TheProSE, and the Austrian FWF research project W1255-N23.

References

[1] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, 2008, pp. 337–340.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*, 2011, pp. 171–177.

[3] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham, "Property-directed inference of universal invariants or proving their absence," pp. 583–602, 2015.

[4] A. Gurfinkel, S. Shoham, and Y. Vizel, "Quantifiers on demand," in *ATVA*, 2018, pp. 248–266.

[5] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, "Quantified invariants via syntax-guided synthesis," in *CAV*, 2019, pp. 259–277.

[6] G. Barthe, R. Eilers, P. Georgiou, B. Gleiss, L. Kovács, and M. Maffei, "Verifying relational properties using trace logic," in *FMCAD*, 2019, pp. 170–178.

[7] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, "Horn Clause Solvers for Program Verification," in *Fields of Logic and Computation II*, 2015, pp. 24–51.

[8] N. Kobayashi, G. Fedyukovich, and A. Gupta, "Fold/unfold transformations for fixpoint logic," in *TACAS*, 2020, pp. 195–214.

[9] S. Chakraborty, A. Gupta, and D. Unadkat, "Verifying array manipulating programs with full-program induction," in *TACAS*, 2020, pp. 22–39.

[10] O. Ish-Shalom, S. Itzhaky, N. Rinetzky, and S. Shoham, "Putting the squeeze on array programs: Loop verification via inductive rank reduction," in *VMAI*, 2020, pp. 112–135.

[11] B. Gleiss, L. Kovács, and J. Rath, "Subsumption demodulation in first-order theorem proving," in *IJCAR*, 2020.

[12] B. Gleiss and M. Suda, "Layered clause selection for theory reasoning," in *IJCAR*, 2020.

[13] L. Kovács, S. Robillard, and A. Voronkov, "Coming to Terms with Quantified Reasoning," in *POPL*, 2017, pp. 260–270.

[14] D. Beyer, "Automatic verification of c and java programs: Sv-comp 2019," in *TACAS*, 2019, pp. 133–155.

[15] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *CAV*, 2015, pp. 343–361.

[16] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, 2012, pp. 157–171.

[17] P. Georgiou, B. Gleiss, and L. Kovács, "Trace Logic for Inductive Loop Reasoning," 2020, arXiv:2008.01387.

[18] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[19] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, available at `www.SMT-LIB.org`.

[20] L. Kovács and A. Voronkov, "First-Order Theorem Proving and Vampire," in *CAV*, 2013, pp. 1–35.

[21] B. Gleiss, L. Kovács, and S. Robillard, "Loop Analysis by Quantification over Iterations," in *LPAR*, 2018, pp. 381–399.

[22] L. Kovács and A. Voronkov, "Finding loop invariants for programs over arrays using a theorem prover," in *FASE*, 2009, pp. 470–485.

[23] W. Yang, G. Fedyukovich, and A. Gupta, "Lemma synthesis for automating induction over algebraic data types," in *CP*, 2019, pp. 600–617.

[24] I. Dillig, T. Dillig, and A. Aiken, "Fluid Updates: Beyond Strong vs. Weak Updates," in *ESOP*, 2010, pp. 246–266.

[25] P. Cousot, R. Cousot, and F. Logozzo, "A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis," in *POPL*, 2011, pp. 105–118.

[26] A. Komuravelli, N. Bjorner, A. Gurfinkel, and K. L. McMillan, "Compositional verification of procedural programs using horn clauses over integers and arrays," in *FMCAD*, 2015, pp. 89–96.

[27] G. Fedyukovich, S. J. Kaufman, and R. Bodík, "Sampling invariants from frequency distributions," in *FMCAD*, 2017, pp. 100–107.

[28] G. Fedyukovich and R. Bodík, "Accelerating syntax-guided invariant synthesis," in *TACAS*, 2018, pp. 251–269.

[29] Y. Matsushita, T. Tsukada, and N. Kobayashi, "Rusthorn: Chc-based verification for rust programs," in *ESOP*, 2020, pp. 484–514.

[30] A. Cimatti and A. Griggio, "Software model checking via ic3," in *CAV*, 2012, pp. 277–293.

[31] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina, "Lazy abstraction with interpolants for arrays," in *LPAR*, 2012, pp. 46–61.

[32] M. Afzal, S. Chakraborty, A. Chauhan, B. Chimdyalwar, P. Darke, A. Gupta, S. Kumar, C. Babu, D. Unadkat, and R. Venkatesh, "Veriabs: Verification by abstraction and test generation (competition contribution)," in *TACAS*, 2020, pp. 383–387.

[33] P. Rajkhowa and F. Lin, "Extending viap to handle array programs," in *VSTTE*, 2018, pp. 38–49.