Formal Methods in Computer-Aided Design 2022

The RAPID Software Verification Framework

Pamina Georgiou[a], Bernhard Gleiss[a], Ahmed Bhayat[b], Michael Rawson[a], Laura Kovács[a], Giles Reger[b]
[a] TU Wien, Vienna, Austria
[b] University of Manchester, Manchester, United Kingdom
{ pamina.georgiou, bernhard.gleiss, michael.rawson, laura.kovacs }@tuwien.ac.at, { ahmed.bhayat, giles.reger }@manchester.ac.uk

***Abstract*—We present the RAPID framework for automatic software verification by applying first-order reasoning in *trace logic*. RAPID establishes partial correctness of programs with loops and arrays by inferring invariants necessary to prove program correctness using a saturation-based automated theorem prover. RAPID can heuristically generate *trace lemmas*, common program properties that guide inductive invariant reasoning. Alternatively, RAPID can exploit nascent support for induction in modern provers to fully automate inductive reasoning without the use of trace lemmas. In addition, RAPID can be used as an invariant generation engine, supplying other verification tools with quantified loop invariants necessary for proving partial program correctness.**

I. INTRODUCTION

State-of-the-art deductive verification tools for programs containing inductive data structures ([1], [2], [3], [4], [5]) largely depend on satisfiability modulo theories (SMT) solvers to discharge verification conditions and establish software correctness. These approaches are mostly limited to reasoning over universally-quantified properties in fragments of first-order *theories*: arrays, integers, etc. In contrast, RAPID supports reasoning with arbitrary quantifiers in full first-order logic with theories [6]. Program semantics and properties are directly encoded in trace logic by quantifying over *timepoints* of program execution. This allows simultaneous reasoning about *sets* of program states, unlike model-checking approaches [2][7]. The gain in expressiveness is beneficial for reasoning about programs with unbounded arrays [6] or to prove security properties [8], for example.

This paper presents what RAPID can do, sketches its design (Section III), and describes its main components and implementation aspects (Sections IV–VII). Experimental evaluation using the SV-COMP benchmark [9] shows RAPID's efficacy in verification (Section VIII).

Given a program loop annotated with pre/post-conditions, RAPID offers two modes for proving partial program correctness. In the first, RAPID relies on so-called *trace lemmas*, apriori-identified inductive properties that are automatically instantiated for a given program. In the second, RAPID delegates inductive reasoning to the underlying first-order theorem prover [10][11], without instantiating trace lemmas. In either mode, the automated theorem prover used by RAPID is VAMPIRE [12]. RAPID can also synthesize quantified invariants from program semantics, complementing other invariant-generation methods.

```
1  func main() {
2    const Int[] a;
3    const Int alength;
4    Int[] b, c;
5    Int blength, clength, i = 0, 0, 0;
6    while(i < alength) {
7      if(a[i] >= 0) {
8        b[blength] = a[i];
9        blength = blength+1;
10     } else {
11       c[clength] = a[i];
12       clength = clength+1;
13     } i = i+1;
14   }
15 }
```

Fig. 1: Program partitioning an array `a` into two arrays `b, c` containing positive and negative elements of `a` respectively.

Related Work: Verifying programs with unbounded data structures can use model checking for invariant synthesis. Tools like Spacer/Quic3 [4][2], SEAHORN [1] or FREQHORN [7] are based on constrained horn clauses (CHC) and use either fixed-point calculation or sampling/enumerating invariants until a given safety assertion is satisfied. These approaches use SMT solvers to check validity of invariants and are limited to quantifier-free or universally-quantified invariants. Recurrence solving and data-structure-specific tactics can be used to infer and prove quantified program properties [3]. DIFFY [13] and VAJRA [5] derive relational invariants of two mutations of a program such that inductive properties can be enforced over the entire program, without invariants for each individual loop.

II. MOTIVATING EXAMPLE

We motivate RAPID using the program in Figure 1, written in a standard while-like programming language $\mathcal{W}$. Each program in $\mathcal{W}$ consists of a single top-level function `main`, with arbitrary nestings of if-then-else and while statements. $\mathcal{W}$ includes optionally-mutable integer (array) variables, and standard side-effect-free expressions over Booleans and integers.

Semantics and properties of $\mathcal{W}$-programs are expressed in *trace logic* $\mathcal{L}$, an instance of many-sorted first-order logic with theories and equality [6]. A *timepoint* in trace logic is a term of sort $\mathbb{L}$ that refers to a program location. For example, l_5 refers to `line 5` in Figure 1. If a program location occurs in a loop, a timepoint is represented by a function $l : \mathbb{N} \mapsto \mathbb{L}$, where the argument is a natural number representing a loop iteration.

https://doi.org/

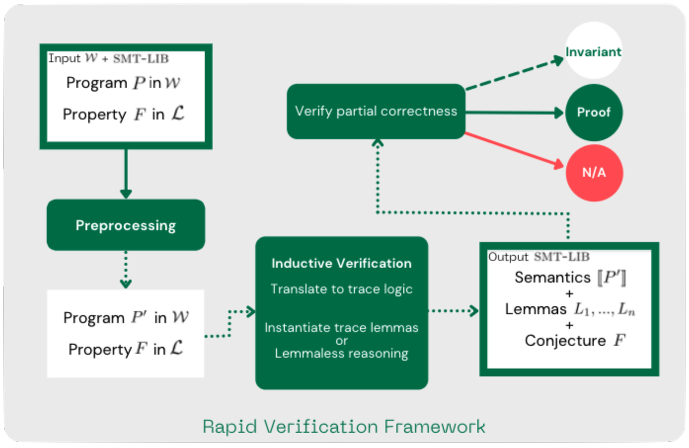
Input $\mathcal{W}$ + SMT-LIB
Program P in $\mathcal{W}$
Property F in $\mathcal{L}$
Preprocessing
Program P' in $\mathcal{W}$
Property F in $\mathcal{L}$
Inductive Verification
Translate to trace logic
Instantiate trace lemmas
or
Lemmaless reasoning
Output SMT-LIB
Semantics $[\![P']\!]$
+
Lemmas $L_1, ..., L_n$
+
Conjecture F
Verify partial correctness
Invariant
Proof
N/A
Rapid Verification Framework

Fig. 2: Overview of the RAPID verification framework.

For example, $l_6(0)$ denotes the first iteration of the loop before entering the loop body. A mutable scalar variable v is modeled as a function over time $v : \mathbb{L} \mapsto \mathbb{I}$. An array variable is modeled as a function $v : \mathbb{L} \times \mathbb{I} \mapsto \mathbb{I}$, where array indices are represented by integer arguments. For constant variables we omit the timepoint argument. We use a constant $nl_i : \mathbb{N}$ to denote the last iteration of the loop starting at l_i. When a loop is nested within other loops, the last iteration is a *function* over timepoints of all enclosing loops; l_{end} denotes the timepoint after program execution. For Figure 1, $l_6(nl_6)$ denotes the program location of the loop at its last iteration, when the loop condition no longer holds. We assume that programs terminate, and hence RAPID focuses on *partial* correctness.

Figure 1 creates two new arrays, `b` and `c`, containing positive and negative elements from the input array `a` respectively. Note that the arrays are unbounded, and we use the symbolic, non-negative constant `alength` to bound the length of the input array `a`. The constraint that `alength` be non-negative can be expressed within a conjecture (see (1) below for example). A safety property we want to check is that for any position in `b` there exists a position in `a` such that both values are equal within the respective array bounds (and similarly for `c`). This equates to the following conjecture expressed in trace logic[1]:

$$\forall pos_{\mathbb{I}}.\ \exists pos'_{\mathbb{I}}.\ 0 \leq pos < blength(l_{end}) \wedge alength \geq 0 \rightarrow \\ 0 \leq pos' < alength \wedge b(l_{end}, pos) = a(pos'), \tag{1}$$

To the best of our knowledge other verification approaches cannot automatically validate (1) due to quantifier alternation, but RAPID proves this property for Figure 1.

III. The Rapid Framework

The RAPID framework consists of approximately 10,000 lines of C++ [2]. Figure 2 summarizes the RAPID workflow. Inputs to RAPID are programs P written in $\mathcal{W}$ along with properties F expressed in $\mathcal{L}$. *Preprocessing* in RAPID applies program transformations for common loop-altering programming constructs, as well as timepoint inlining to obtain a simplified program P' from P (see Section IV).

Next, RAPID performs *inductive verification* (see Section V) by generating the axiomatic semantics $[\![P']\!]$ expressed in $\mathcal{L}$ and instantiating a set $L_1, ..., L_n$ of inductive properties — so-called *trace lemmas* — for the respective program variables of P'. For establishing some property F, RAPID supports two modes of inductive verification: *standard* and *lemmaless* mode. The difference in both versions relates to the underlying support for automating inductive reasoning while proving F. The *standard* verification mode equips the verification task with the trace lemmas $L_1, ..., L_n$, providing the necessary induction schemes for proving F. The *lemmaless* verification mode uses built-in inductive reasoning and relies less, or not at all, on trace lemmas. In either mode, the verification tasks of RAPID are encoded in the SMT-LIB format. Finally, a third and recent RAPID mode can be used for invariant generation (see Section VII). In this mode, RAPID "only" outputs quantified invariants using the SMT-LIB syntax; these invariants can further be used by other verification tools.

[1]we write $\forall x_S.\ F$ or $\exists x_S.\ F$ to mean that x has sort S in F
[2]available at https://github.com/vprover/rapid

```
  while(i < alength) {
    if (a[i] == x) {
      break;
    }
    i = i + 1;
  }
```

```
  Bool break = false;
  while(i < alength && !break ) {
    if (a[i] == x) {
      break = true;
    }
    if (!break) {
      i = i + 1;
    }
  }
```

Fig. 3: Loop tranformation for `break`-statement.

IV. Preprocessing in Rapid

a) Program Transformations: We use standard program transformations to translate away `break`, `continue` and `return` statements. For these, RAPID introduces fresh Boolean program variables indicating whether a statement has been executed. The program is adjusted accordingly: `return` statements end program execution; `break` statements invalidate the first enclosing loop condition; and for `continue` the remaining code of the first enclosing loop body is not executed.

***Example** 1:* Figure 3 shows a standard transformation for a `break`-statement.

b) Timepoint Inlining: RAPID uses SSA-style inlining [14], [15], [16] for timepoints to simplify axiomatic program semantics and trace lemmas of a verification task. Specifically, RAPID caches (i) for each integer variable the current program

```
a = a + 2;
b = 3;
c = a + b;

assert (a(l_end) < c(l_end))
```

(a) block assignments

```
if (x < 1) {
  x = 0;
} else {
  skip;
}
while (y > 0) {
  y = y - 1;
}

assert (x(l_end) ≥ 0)
```

(b) simple branching

Fig. 4

expression assigned to it, and (ii) for each integer-array variable the last timepoint where it was assigned. Cached values are used during traversal of the program tree to simplify later program expressions. Thus we avoid defining irrelevant equalities of program variable values over unused timepoints, and only reference timepoints relevant to the property. We illustrate this on two examples:

***Example** 2 (Inlining assigned integer expressions):* The effect of inlined semantics can be observed when we encounter block assignments to integer variables: we can skip assignments and use the last assigned expression directly in any reference to the original program variable. Consider the partial program in Figure 4a. Our axiomatic semantics in trace logic [6] would result in

$$\begin{aligned} a(l_2) &= a(l_1) + 2 &\wedge\quad b(l_2) &= b(l_1) &\wedge \\ c(l_2) &= c(l_1) &\wedge\quad a(l_3) &= a(l_2) &\wedge \\ b(l_3) &= 3 &\wedge\quad c(l_3) &= c(l_2) &\wedge \\ a(l_{end}) &= a(l_3) &\wedge\quad b(l_{end}) &= b(l_3) &\wedge \\ c(l_{end}) &= a(l_3) + b(l_3) \end{aligned}$$

whereas the inlined version of semantics is drastically shorter:

$$a(l_{end}) = a(l_1) + 2 \qquad \wedge \qquad c(l_{end}) = (a(l_1) + 2) + 3.$$

In contrast to the extended semantics that define all program variables for each timepoint, the inlined version only considers the values of referenced program variables at the timepoint of their last assignment. Thus, when c is defined, RAPID directly references the (symbolic) values assigned to a and b. While b is not defined at all, note that a *is* defined as $a(l_{end})$ is referenced in the conjecture. Furthermore, the inlined semantics only make use of two timepoints, l_1, and l_{end}, as the remaining timepoints are irrelevant to the conjecture.

***Example** 3 (Inlining equalities with branching.):* Figure 4b shows another program that benefits from inlining equalities, as well as only considering timepoints relevant to the conjecture. The original semantics defines program variables x and y for all program locations: l_1, l_2, l_3, l_4, $l_6(it)$, $l_6(nl_6)$, l_{end}, for some iteration it and final iteration nl_6. While the program contains two variables x and y, only x is used in the property we want to prove. Since no assignments to x contain references to y, the loop semantics do not interfere with x, so we have

$$\begin{aligned} x(l_3) < 1 \rightarrow x(l_6(0)) &= 0 &\wedge \\ x(l_3) \geq 1 \rightarrow x(l_6(0)) &= x(l_3) &\wedge \\ x(l_{end}) &= x(l_6(0)) \end{aligned}$$

where the semantics of the loop defining y are omitted. Note that all timepoints of the if-then-else statements are flattened into the timepoint at the beginning of the loop at l_6 in iteration 0. The axiomatic semantics thus reduce to three conjuncts defining the value of x throughout the execution. However, x is not defined in any loop iteration other than the first as they are irrelevant to the property.

c) User-defined input: RAPID is fully automated. However, it may still benefit from manually-defined invariants to support the prover. Users can therefore extend the input to RAPID with first-order axioms written in the SMT-LIB format.

V. Inductive Verification in Rapid

As mentioned above, RAPID implements two verification modes; in the default *standard* mode, RAPID uses trace lemmas to prove inductive properties of programs. In its *lemmaless* mode RAPID relies on built-in induction support in saturation-based first-order theorem proving. In this section we elaborate on both modes further.

A. Standard Verification Mode: Reasoning with Trace Lemmas

RAPID's *standard* mode relies on trace lemma reasoning to automate inductive reasoning. Trace lemmas are sound formulas that are: (i) derived from bounded induction over loop iterations; (ii) represent common inductive program properties for a set of similar input programs; and (iii) are automatically instantiated for all relevant program variables of a specific input program during its translation to trace logic; see [6].

In all of our experiments from Section VIII, including the example from Figure 1, we only instantiate three generic inductive trace lemmas to establish partial correctness. One such trace lemma asserts, for example, that a program variable is not mutated after a certain execution timepoint.

***Example** 4:* Consider the safety assertion (1) of our running example from Figure 1. In its standard verification mode, RAPID proves correctness of (1) by using, among others, the following trace lemma instance

$$\begin{aligned} &\forall j_{\mathbb{I}}.\ \forall b_{L\mathbb{N}}.\ \forall b_{R\mathbb{N}}.\Big(\\ &\qquad \forall it_{\mathbb{N}}.\big((b_L \leq it < b_R \ \wedge\ b(l_9(b_L), j) = b(l_9(it), j)) \\ &\qquad \rightarrow b(l_9(b_L), j) = b(l_9(s(it)), j)\big) \\ &\rightarrow \big(b_L \leq b_R \rightarrow b(l_9(b_L), j) = b(l_9(b_R), j)\big)\Big), \end{aligned}$$

stating that the value of `b` at some position `j` is unchanged between two bounds b_L and b_R if, for any iteration it and its successor $s(it)$, values of b are unchanged.

Multitrace Generalization: RAPID can also be used to prove k-safety properties over k traces, useful for security-related hyperproperties such as non-interference and sensitivity [8]. For such problems it is sufficient to extend program variables to functions over time and trace, such that program variables are represented as $(\mathbb{L} \times \mathbb{T} \mapsto \mathbb{I})$. Program locations, and hence timepoints, are similarly parameterized by an argument of sort $\mathbb{T}$ to denote the same timepoint in different executions.

B. Lemmaless Verification Mode

When in *lemmaless* mode RAPID does not add any trace lemma to its verification task but relies on first-order theorem proving to derive inductive loop properties. An extended version of SMT-LIB (see Section VI) is used to provide the underlying prover with additional information to guide the search for necessary inductive schemes, such as likely symbols for induction. We further equip saturation-based theorem proving with two new inference rules that enable induction on such terms; see [17] for details. *Multi-clause goal induction* takes a formula derived from a safety assertion that contains a final loop counter, that is a symbol denoting last loop iterations, and inserts an instance of the induction schema for natural numbers with the negation of this formula as its conclusion into the proof search space. For example, consider the formula $x(l_5(nl_5)) < 0$. Multi-clause goal induction introduces the induction hypothesis $x(l_5(0)) \geq 0 \quad \wedge \quad \forall it_{\mathbb{N}}.\ (it < nl_5 \ \wedge\ x(l_5(it)) \geq 0) \rightarrow x(l_5(s(it))) \geq 0 \quad \rightarrow \quad x(l_5(nl_5)) \geq 0$. If the base and step cases can be discharged, a contradiction can be easily produced from the conclusion and original clause.

Array mapping induction also introduces an instance of the induction schema to the search space, but is not based on formulas derived from the goal. Instead, this rule uses clauses derived from program semantics to generate a suitable conclusion for the induction hypothesis.

VI. Verifying Partial Correctness in RAPID

For proving the verification tasks of Section V, and thus verifying partial program correctness, RAPID relies on saturation-based first-order theorem proving. To this end, each verification mode of RAPID uses the VAMPIRE prover, for which we implemented the following, RAPID-specific adjustments.

a) Extending SMT-LIB*:* Each verification task of RAPID is expressed in extensions of SMT-LIB, allowing us to treat some terms and definitions in a special way during proof search:

(i) `declare-nat`: The VAMPIRE prover has been extended with an axiomatization of the natural numbers as a term algebra, especially for RAPID-style verification purposes. We use the command `(declare-nat Nat zero s p Sub)` to declare the sort `Nat`, with constructors `zero` and successor `s`, predecessor `p` and ordering relation `Sub`.

(ii) `declare-lemma-predicate`: Our trace lemmas are usually of the form $(P_1 \wedge ... \wedge P_n) \rightarrow Conclusion_L$ for some trace lemma L with premises $P_1 \wedge ... \wedge P_n$. In terms of reasoning, it makes sense for the prover to derive the premises of such a lemma before using its conclusion to derive more facts, as we have many automatically instantiated lemmas of which we can only prove the premises of some from the semantics. To enforce this, we adapt literal selection such that inferences from premises are preferred over inferences from conclusions. Lemmas are split into two clauses $(P_1 \wedge ... \wedge P_n) \rightarrow Premise_L$ and $Premise_L \rightarrow Conclusion_L$, where $Premise_L$ is declared as a *lemma literal*. We ensure our literal selection function selects either a negative lemma literal[3] if available, or a positive lemma literal only in combination with another literal, requiring the prover to resolve premises before using the conclusion.

The *lemmaless* mode of RAPID introduces the following additional declarations to SMT-LIB:

(i) `declare-const-var`: assign symbols representing constant program variables a large weight in the prover's term ordering, allowing constant variables to be rewritten to non-constant expressions.

(ii) `declare-timepoint`: distinguish a symbol representing a timepoint from program variables, guiding VAMPIRE to apply induction upon timepoints.

(iii) `declare-final-loop-count`: declare a symbol as a final loop count symbol, eligible for induction.

b) Portfolio Modes: We further developed a collection of RAPID-specific proof options in VAMPIRE, using for example extensions of theory split queues [18] and equality-based rewritings [19]. Such options have been distilled into a RAPID portfolio schedule that can be run with `--mode portfolio -sched rapid`. Moreover, the multi-clause goal induction rule and the array mapping induction inference of RAPID have been compiled to a separate portfolio mode, accessed via `--mode portfolio -sched induction_rapid`.

VII. Invariant Generation with RAPID

RAPID can also be used as an invariant generation engine, synthesizing first-order invariants using the VAMPIRE theorem prover. To do so, we use a special mode of VAMPIRE to derive logical consequences of the semantics produced by RAPID. Some of these consequences may be loop invariants. The *symbol elimination* approach of [20] defined some set of program symbols undesirable, and only reports consequences that have *eliminated* such symbols from their predecessors. In RAPID, we adjust symbol elimination for deriving invariants in trace logic using VAMPIRE. These invariants may contain quantifier alternations, and some conjunction of them may well be enough to help other verification tools show some property. When RAPID is in *invariant generation* mode, the encoding of the problem is optimized for invariant generation. We limit trace lemmas to more specific versions of the bounded induction scheme. We also remove RAPID-specific symbols such as lemma literals so that they do not appear in consequences.

[3] Note that lemma literals become negative in the premise definition after CNF-transformation.

Symbol Elimination: Loop invariants should only contain symbols from the input loop language, with no timepoints. To remove such constructs, we apply symbol elimination: any symbol representing a variable `v` used on the left-hand side of an assignment is eliminated. However, we still want to generate invariants containing otherwise-eliminated variables at specific locations, so for each eliminated variable `v` we define `v_init` $= v(l_1)$ and `v_final` $= v(l_2)$ for appropriate locations l_1, l_2: these new symbols need not be eliminated.

We further adjusted symbol elimination in RAPID to output fully-simplified consequences during proof search in VAMPIRE (the so-called *active set* [12]) at the end of a user-specified time limit. Consequences that contain undesirable symbols or are pure consequences of theories are removed at this stage.

Reasoning with Integers vs. Naturals: In the standard setting, RAPID uses natural numbers (internally `Nat`) to describe loop iterations. However, in some situations it is advantageous to use the theory of integers: loop counter variable `i` of sort $\mathbb{I}$ will have the same numerical value as nl of sort $\mathbb{N}$ at the end of a loop. Integer-based timepoints allow deriving $i(l(nl)) = nl$. Such a clause can be very helpful for invariant generation, as shown in Example 5.

***Example** 5:* Consider the property $\forall x_{\mathbb{I}}.0 \le x \le alength \rightarrow a(x) = b(x)$. The property essentially requires us to prove that two arrays `a, b` are equal in all positions between 0 and `alength`. Such a property might for example be useful to prove when we copy from an array `b` into array `a` in a loop with loop condition $i < alength$ where `i` is the loop counter variable incremented by one in each iteration. Now when we run RAPID in the invariant generation mode, we might be able to derive a property $\forall x.0 \le x \le nl \rightarrow a(x) = b(x)$, essentially stating that the property holds for all iterations of the loop. The prover can further easily deduce that $i(l(nl)) \ge alength$ thanks to our semantics.

However, in case of natural numbers we cannot deduce that $i(l(nl)) = nl$ since the sorts of `i` and `nl` differ. In order to derive an invariant strong enough to prove the postcondition we depend upon the prover to find the invariant $\forall x.0 \le x \le i(l(nl)) \rightarrow a(x) = b(x)$ directly which cannot be deduced by the prover as our loop semantics are bounded by loop iterations rather than the loop counter values.

When using `-integerIterations on` we can circumvent this problem as the prover can then simply deduce the equality $i(l(nl)) = nl$ which makes the conjunction of clauses strong enough to prove the desired postcondition.

VIII. EXPERIMENTAL EVALUATION

We evaluated the two verification modes of RAPID and compare against the state-of-the-art solvers DIFFY and SEAHORN, as summarized below.

Benchmark Selection: Our benchmarks[4] are based on the `c/ReachSafety-Array` category of the SV-COMP repository [21], specifically from the `array-examples/*` subcategory[5] as it contains problems suitable for our input language.

[4]https://github.com/vprover/rapid/tree/main/examples/arrays

[5]https://github.com/sosy-lab/sv-benchmarks/tree/master/c/array-examples

TABLE I: Experimental Results

Total	RAPID$_{\text{std}}$	RAPID$_{\text{lemmaless}}$	DIFFY	SEAHORN
140	91 (5)	103 (10)	61 (1)	17 (0)

Other examples are not yet expressible in $\mathcal{W}$ due to the presence of function calls and/or unsupported memory access constructs. We manually translate all programs to $\mathcal{W}$ and express pre/post-conditions as trace logic properties. Additionally, we extend some SV-COMP examples with new conjectures containing existential and alternating quantification.

In general SV-COMP benchmarks are bounded to a certain array size N. By contrast, we treat arrays as unbounded in RAPID and reason using symbolic array lengths. Some benchmarks in the original SV-COMP repository are minor variations of each other that differ only in one concrete integer value, e.g to increment a program variable by some integer. Instead of copying each such variation for different digits, we abstract such constant values to a single symbolic integer constant such that just one of our benchmark covers numerous cases in the original SV-COMP setup.

Results: We compare our two RAPID verification modes, indicated by RAPID$_{\text{std}}$ and RAPID$_{\text{lemmaless}}$ respectively, against SEAHORN and DIFFY. All experiments were run on a cluster with two 2.5GHz 32-core CPUs with a 60-seconds timeout. Note that DIFFY produced the same results as its precursor VAJRA in this experiment. Table I summarizes our results, parentheticals indicating uniquely solved problems. Of a total of 140 benchmarks, RAPID$_{\text{std}}$ solves 91 problems, while RAPID$_{\text{lemmaless}}$ surpasses this by 12 problems. Particularly, RAPID$_{\text{lemmaless}}$ could solve more variations with quantifier alternations of our running example 1, as property-driven induction works well for such problems. A small number of instances, however, was solved by RAPID$_{\text{std}}$ but not by RAPID$_{\text{lemmaless}}$ within the time limit, indicating that trace lemma reasoning can help to fast-forward proof search. In total, RAPID solves 112 benchmarks, whereas SEAHORN and DIFFY could respectively prove 17 and 61 problems (with mostly universally quantified properties). For more detailed experimental data on subsets of these benchmarks we refer to [6], [17].

IX. CONCLUSION

We described the RAPID verification framework for proving partial correctness of programs containing loops and arrays, and its applications towards efficient inductive reasoning and invariant generation. Extending RAPID with function calls, and automation thereof, is an interesting task for future work.

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

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

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

[4] H. G. V. Krishnan, Y. Chen, S. Shoham, and A. Gurfinkel, "Global guidance for local generalization in model checking," in *CAV*. Springer, 2020, pp. 101–125.

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

[6] P. Georgiou, B. Gleiss, and L. Kovács, "Trace logic for inductive loop reasoning," in *FMCAD*. IEEE, 2020, pp. 255–263.

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

[8] 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.

[9] D. Beyer, "Software verification: 10th comparative evaluation (SV-COMP 2021)," in *TACAS*, 2021, pp. 401–422.

[10] P. Hozzová, L. Kovács, and A. Voronkov, "Integer induction in saturation," in *CADE*, 2021, pp. 361–377.

[11] G. Reger and A. Voronkov, "Induction in saturation-based proof search," in *CADE*, 2019, pp. 477–494.

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

[13] S. Chakraborty, A. Gupta, and D. Unadkat, "Diffy: Inductive reasoning of array programs using difference invariants," in *CAV*, 2021.

[14] P. Briggs and K. D. Cooper, "Effective partial redundancy elimination," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 159–170, 1994.

[15] A. W. Appel, "SSA is functional programming," *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, 1998.

[16] ——, *Modern compiler implementation in C*. Cambridge university press, 2004.

[17] A. Bhayat, P. Georgiou, C. Eisenhofer, L. Kovács, and G. Reger, "Lemmaless induction in trace logic," Preprint, https://github.com/vprover/vampire_publications/blob/master/paper_drafts/rapid_induction.pdf.

[18] B. Gleiss and M. Suda, "Layered clause selection for theory reasoning," in *IJCAR*, 2020, pp. 297–315.

[19] B. Gleiss, L. Kovács, and J. Rath, "Subsumption demodulation in first-order theorem proving," in *IJCAR*, 2020, pp. 297–315.

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

[21] SV-COMP. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks.