The Rapid Software Verification Framework

Pamina Georgiou, Bernhard Gleiss, Ahmed Bhayat, Michael Rawson, Laura Kovács, Giles Reger

Automated Program Reasoning Group, TU Wien

TU WIEN Informatics

MANCHESTER 1824
The University of Manchester

What are we solving?

```
func main() {
 const Int[] a;
 Int[] b, c;
 Int i, j, k = 0;
 while (i < a.length) {
  if (a[i] ≥ 0) {
   b[j] = a[i];
   j++;
  } else {
   c[k] = a[i];
   k++;
  }
  i++;
 }
}
```

Partial correctness:

b is initialized by elements of a

What are we solving?

```
func main() {
 const Int[] a;
 Int[] b, c;
 Int i, j, k = 0;
 while (i < a.length) {
  if (a[i] ≥ 0) {
   b[j] = a[i];
   j++;
  } else {
   c[k] = a[i];
   k++;
  }
  i++;
 }
}
```

Partial correctness:

b is initialized by elements of a

$$\forall pos_{\mathbb{I}}.\exists pos'_{\mathbb{I}}.(\\ 0 \leq pos < j \land a.length \geq 0 \\ \rightarrow \\ 0 \leq pos' < a.length \\ \land b(pos) = a(pos'))$$

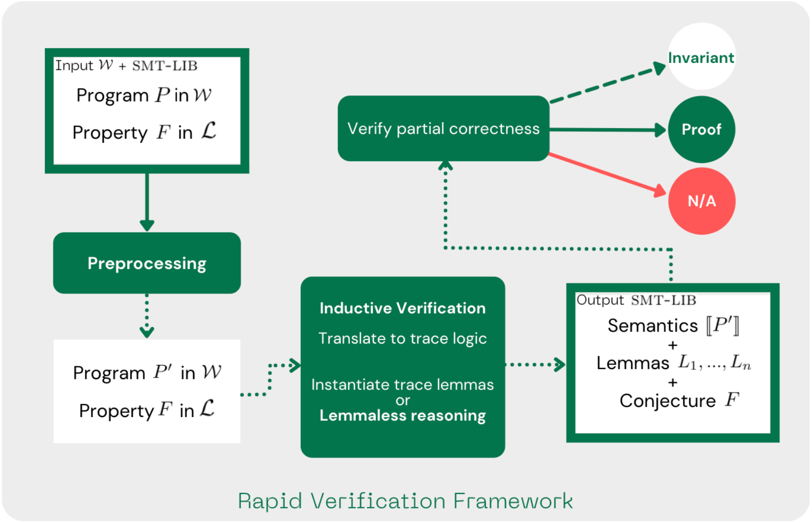
What are we solving?

```
func main() {
 const Int[] a;
 Int[] b, c;
 Int i, j, k = 0;
 while (i < a.length) {
  if (a[i] ≥ 0) {
   b[j] = a[i];
   j++;
  } else {
   c[k] = a[i];
   k++;
  }
  i++;
 }
}

```

- Partial correctness
- Programs containing integer and integer-array variables
- Arbitrary amount of loops
- Arbitrarily quantified program properties
- Automated first-order theorem proving

The Rapid Verification Framework



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

Trace Logic in a Nutshell

- **Full first-order logic** with equality (over UFDTLIA)
- Program values: standard theory of integers
- Loop iterations: theory of natural numbers $(0, s, p, <)$ (no arithmetic!)
- Reasoning over **timepoints**
  - allows to express induction directly in the language
  - reason about properties of *all loop iterations*
  - reason about the *existence of certain loop iterations*

Semantics based on Trace Logic

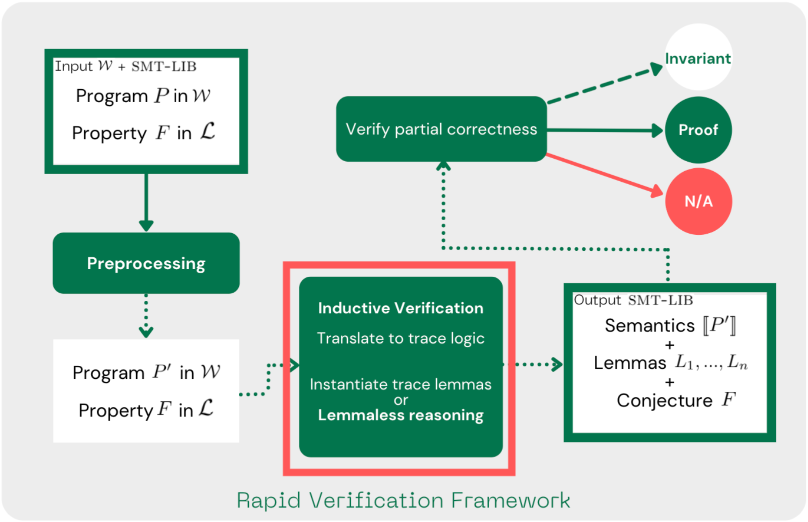
```
1    func main() {
2     const Int[] a;
3     Int[] b, c;
4     Int i, j, k = 0;
5     while (i < a.length) {
6      if (a[i] ≥ 0) {
7       b[j] = a[i];
8       j++;
9      } else {
10      c[k] = a[i];
11      k++;
12     }
13     i++;
14    }
15   }
16
```

Conjunction of all statements

$$i(l_4) \simeq 0$$

$$\forall it_{\mathbb{N}}. \Big(it < n_5 \rightarrow$$
$$i(l_5(s(it))) \simeq i(l_{13}(it)) + 1$$
$$\wedge$$
$$\big(a(i(l_5(it)) \geq 0) \rightarrow$$
$$b(l_9(it), j(l_5(it))) \simeq a(i(l_5(it)))\big)\Big)$$

What about induction?



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

What about induction?

- Trace Lemma Reasoning
- Lemmaless Induction

Trace Lemma Reasoning

- valid formulas, derivable from *instances of the induction scheme*
- *manually identified* set of useful lemmas
- can include *quantifier alternations*
- can include *quantification over loop iterations and variable values*
- can’t be automatically generated by state-of-the-art techniques

Trace Lemma Example

```
1    func main() {
2     const Int[] a;
3     Int[] b, c;
4     Int i, j, k = 0;
5     while (i < a.length) {
6      if (a[i] ≥ 0) {
7      b[j] = a[i];
8      j++;
9     } else {
10     c[k] = a[i];
11     k++;
12    }
13    i++;
14    }
15   }
16
```

Value Evolution Theorem

Apply bounded induction over loop iterations to derive equality of variable values throughout time

$$\forall p^{\mathbb{I}}.\ \forall it_L^{\mathbb{N}}.\ \forall it_R^{\mathbb{N}}.\ \Big(\big(\forall it^{\mathbb{N}}.(it_L \le it < it_R \wedge b(l_7(it_L),p) = b(l_7(it),p)) \rightarrow b(l_7(it_L),p) = b(l_7(s(it)),p)\big) \rightarrow \big(it_L \le it_R \rightarrow b(l_7(it_L),p) = b(l_7(it_R),p)\big)\Big)$$

Trace Lemma Reasoning

- automatically instantiated for all (relevant) program variables
- many unnecessary lemmas generated that don’t help to find a proof
- search space *blow up*
- can’t be automatically generated by state-of-the-art techniques

Lemmaless Induction

- inbuilt *inductive inference rules* in first-order theorem proving
- specialized for trace logic
- uses clauses that contain interesting timepoints
- goal-directed reasoning: *multi-clause goal induction*
- program semantics reasoning: *array-mapping induction*

Multi-clause Goal Induction

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

- clauses are *derived from safety assertion*
- works well for *properties that are structurally close to the required invariant*

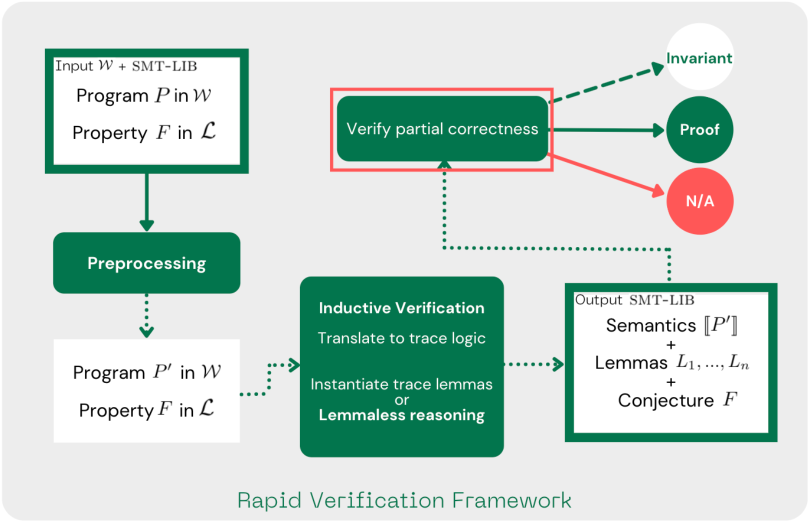
Array-mapping Induction

```
func main(){
 Int[] a;
 Int i, j = 0;
 const Int n;
 while(i < a.length) {
  a[i] = a[i] + n;
  i = i + 1;
 }
 while(j < a.length) {
  a[j] = a[j] - n;
  j = j + 1;
 }
}

```

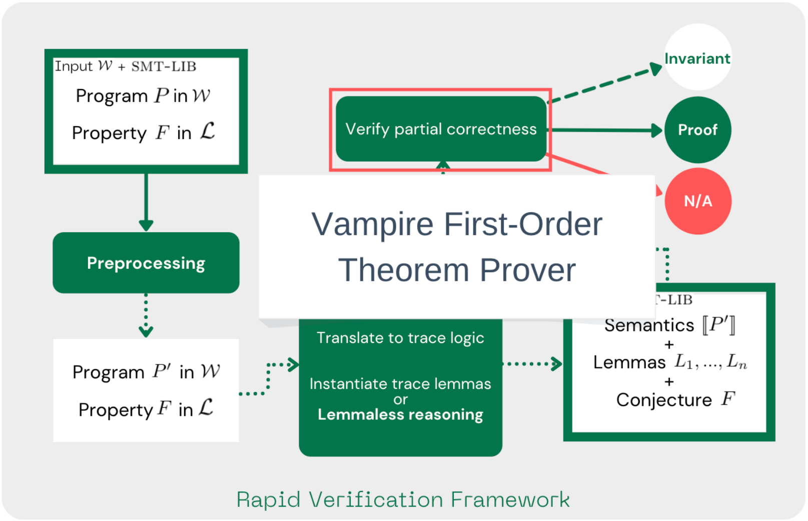
- clauses are *derived from program semantics*
- necessary when required invariant(s) depend on program behavior rather than safety assertion
- *consecutive loops!*

How to discharge verification conditions?



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

How to discharge verification conditions?



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

Vampire First-Order Theorem Prover

Results

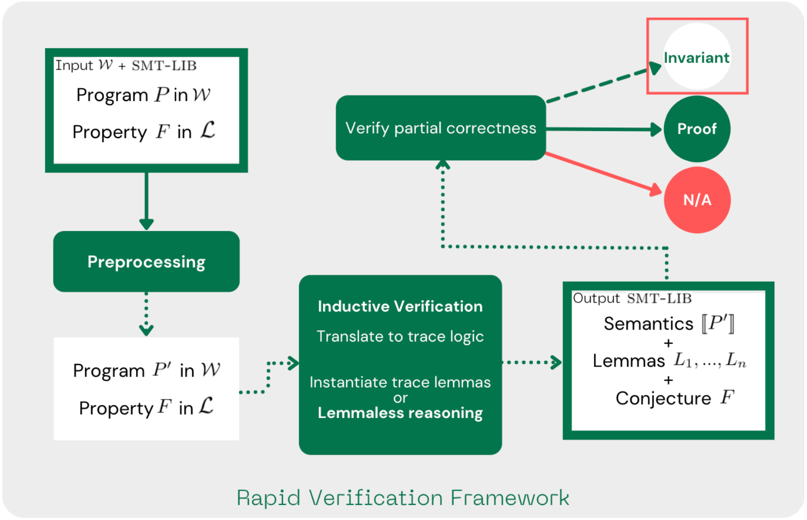
TABLE I: Experimental Results

Total	Rapid$_{\mathrm{std}}$	Rapid$_{\mathrm{lemmaless}}$	Diffy	SeaHorn
140	91 (5)	103 (10)	61 (1)	17 (0)

Fin

Thank you for your attention!

What about invariants?



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

Invariant Generation

- *consequence finding* from semantics and trace lemmas until a conjunction of clauses proves a postcondition
- allows integration with other tools
- works (at most) for already found proofs

Semantics based on Trace Logic

```
 1   func main() {
 2    const Int[] a;
 3    Int[] b, c;
 4    Int i, j, k = 0;
 5    while (i < a.length) {
 6     if (a[i] ≥ 0) {
 7      b[j] = a[i];
 8      j++;
 9     } else {
10      c[k] = a[i];
11      k++;
12     }
13     i++;
14    }
15   }
16
```

Timepoint reasoning

$$l_4 : Timepoint$$

$$l_5(0) : Nat \mapsto Timepoint$$

$$l_5(s(0))$$

$$l_5(n_5)$$

$$l_5(it)$$

Semantics based on Trace Logic

```
func main() {
 const Int[] a;
 Int[] b, c;
 Int i, j, k = 0;
 while (i < a.length) {
  if (a[i] ≥ 0) {
   b[j] = a[i];
   j++;
  } else {
   c[k] = a[i];
   k++;
  }
  i++;
 }
}
```

Program variables

$i(l_5(0))$ $j(l_5(n_5))$

$a(i(l_5(0)))$ $b(l_5(it), j(l_5(it)))$

$$i : Timepoint \mapsto Int$$

$$a : Int \mapsto Int$$

$$b : Timepoint \times Int \mapsto Int$$