Verifying Relational Properties using Trace Logic

Bernhard Gleiss, Gilles Barthe, Renate Eilers, Pamina Georgiou, Laura Kovacs, Matteo Maffei

October 25, 2019
func main() {
    const Int[] a;

    Int sum = 0;

    for (Int i=0; i < a.length; ++i) {
        sum += a[i];
    }
}
Motivating example

```go
func main () {
  const Int [] a;

  Int sum = 0;

  for (Int i=0; i < a.length; ++i)
  {
    sum += a[i];
  }
}
```

\[ a(t_1) : \overrightarrow{\text{v w}} \quad \Rightarrow \quad sum(\text{end}, t_1) \]

\[ a(t_2) : \overrightarrow{\text{w v}} \quad \Rightarrow \quad sum(\text{end}, t_2) \]
Motivating example - Human Proof

First Iteration

Last Iteration

Induction \approx \text{Comm.}
Motivating example - Human Proof

First Iteration

iteration where $i = k$ + 2

Last Iteration
Motivating example - Human Proof

iteration where $i = k + 2$

First Iteration  iteration where $i = k$  Last Iteration

Induction $\simeq$  Comm. $^+$  Induction $\simeq$
Focus

- (Software) programs containing loops and arrays
- Proving Relational Safety Properties
- Proving Correctness (instead of finding Counterexamples)
Part 1: Language and Semantics - Trace Logic
Part 2: Verification Approach - Vampire and Trace Lemmas
Part 3: Extension - Relational Properties
Part 1: Language and Semantics - Trace Logic
Trace Logic

- full first-order logic over UFDTLIA
- explicit notion of time: able to refer to each timepoint of the execution uniquely, while preserving control flow structure
- can formulate induction directly in the language
- can denote parts of a loop and reason about those parts separately
func main() {
    const Int [] a;

    Int sum = 0;

    for (Int i = 0; i < a.length; ++i)
    {
        sum += a[i];
    }
}

\[ l_4 \quad l_8(0) \quad l_6(s(0)) \quad l_6(n_6) \quad l_6(it) \]
func main() {
    const Int[] a;

    Int sum = 0;

    for (Int i = 0; i < a.length; ++i) {
        sum += a[i];
    }
}

\[
sum(l_8(0)) \quad i(l_6(n_6)) \quad a(l_2, pos)
\]
func main() {
    const Int[] a;

    Int sum = 0;

    for (Int i=0; i < a.length; ++i) {
        sum += a[i];
    }
}

sum(l_8(0)) i(l_6(n_6)) a(pos)
func main() {
    const Int [] a;

    Int sum = 0;

    for (Int i=0; i < a.length; ++i)
    {
        sum += a[i];
    }
}

\[ i(l_6(0)) \simeq 0 \]
func main() {
    const Int[] a;
    Int sum = 0;
    for (Int i=0; i < a.length; ++i) {
        sum += a[i];
    }
}

∀it^\mathbb{N}. (it < n_6 \rightarrow i(l_6(s(it)))) \simeq i(l_8(it)) + 1)
Part 2: Verification Approach - Vampire and Trace Lemmas
Workflow - Rapid

\[ P \]

Semantics

\[[P]\]

\( FO \)-clauses

\( P \models \text{UFDTLIA Property} \)

Property

\( FO \)-clauses
Workflow - Rapid

\[
P \xrightarrow{\text{Semantics}} \left[ P \right]_{\text{FO-clauses}}
\]

\[
\text{Property}_{\text{FO-clauses}} \xrightarrow{} \text{VAMPIRE}
\]

\[
\text{Trace-Lemmas}_{\text{FO-clauses}}
\]
Trace Lemmas

- provide necessary inductive reasoning
- valid formulas, derivable from instances of the induction axiom scheme
- can’t be automatically generated by state-of-the-art tools
- manually identified set of useful Trace Lemmas
"For an arbitrary interval: if the value of $v$ stays the same in each step, then the value of $v$ at the end is the same as the value of $v$ at the beginning"

\[
\forall it_L^\mathbb{N}, \forall it_R^\mathbb{N}. ( \\
\forall it^\mathbb{N}. (it_L \leq it < it_R \rightarrow v(l_6(it)) \simeq v(l_6(s(it)))) \\
\rightarrow \\
\forall it^\mathbb{N}. v(l_6(it_L)) \simeq v(l_6(it_R)) 
\)
Intermediate Value Theorem: "If $i \leq v$ at the beginning and $i > v$ at the end and if $i$ is incremented by 1 in each iteration, then there exists an iteration, where $i = v$.

$$\forall v \in \mathbb{N}. ( $$

$$( i(l_6(0)) \leq v \land i(l_6(n_6)) > v \land \forall it \in \mathbb{N}. (it < n \rightarrow i(l_6(s(it))) \simeq i(l_6(it)) + 1)) $$

$$\rightarrow \exists it' \in \mathbb{N}. i(l_6(it')) \simeq v) $$

$$)$$
Trace Lemmas

- can be instantiated to parts of the loop
- can feature existential quantification over iterations
- can feature quantifier alternations
- can not be synthesized automatically by state-of-the-art techniques
The image contains a diagram illustrating a workflow process with the following components:

- **P**: Input
- **Semantics**: Process to derive FO-clauses from P
- **[P]**: FO-clauses derived from P
- **Property**: FO-clauses derived from [P]
- **Trace-Lemmas**: FO-clauses derived from Property
- **VAMPIRE**: Tool for:
  - synthesize split-timepoints
  - reason about loop-parts separately
  - perform interleaved reasoning
Part 3: Extension to Relational Properties
```go
func main() {
    const Int[] a;

    Int sum = 0;

    for (Int i = 0; i < a.length; ++i) {
        sum += a[i];
    }
}
```

\[
\begin{align*}
n_6 & \quad \text{sum}(l_8(0)) & \quad i(l_6(n_8))
\end{align*}
\]
func main() {
    const Int [] a;

    Int sum = 0;

    for (Int i = 0; i < a.length; ++i) {
        sum += a[i];
    }
}

\[ n_6(t_1) \quad \text{sum}(l_8(0), t_1) \quad i(l_6(n_8), t_2) \]
func main() {
    const Int [] a;

    Int sum = 0;

    for (Int i = 0; i < a.length; ++i) {
        sum += a[i];
    }
}

\( i(l_6(0)) \simeq 0 \)
func main() {
    const Int [] a;
    Int sum = 0;
    for (Int i=0; i < a.length; ++i) {
        sum += a[i];
    }
}

\forall tr^T. i(l_6(0), tr) \simeq 0
func main() {
    const Int[] a;

    Int sum = 0;

    for (Int i=0; i < a.length; ++i) {
        sum += a[i];
    }
}

∀it^\mathbb{N}. (it < n_6 \rightarrow i(l_6(s(it)))) \simeq i(l_8(it)) + 1)
func main() {
    const Int[] a;
    Int sum = 0;
    for (Int i=0; i < a.length; ++i) {
        sum += a[i];
    }
}

\forall tr^T. \forall it^N. (it < n_6(tr) \rightarrow i(l_6(s(it)), tr) \simeq i(l_8(it), tr)+1)
Extension - Trace Lemmas

*sum* has the same value in both traces in iteration *it*:

\[ Eq_{sum}(it) := \text{sum}(l_6(it), t1) \simeq \text{sum}(l_6(it), t2). \]

Bounded induction with induction hypothesis *Eq_{sum}(it)*:

\[
\forall \text{it} \in \mathbb{N}, \text{itR} \in \mathbb{N}.
\left(\left( Eq_{sum}(\text{itL}) \land \forall \text{it} \in \mathbb{N}.(\text{itL} \leq \text{it} < \text{itR} \land Eq_{sum}(\text{it})) \rightarrow Eq_{sum}(s(\text{it}))\right) \rightarrow Eq_{sum}(\text{itR})\right)
\]
Benchmarks

- 27 challenging benchmarks from security applications
- 2-safety properties: non-interference and sensitivity
- 60 second timeout
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Vampire</th>
<th>CVC4</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
<td>S+A</td>
<td>F</td>
</tr>
<tr>
<td>1-hw-equal-arrays</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>2-hw-last-position-swapped</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>3-hw-swap-and-two-arrays</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>4-hw-swap-in-array-lemma</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>4-hw-swap-in-array-full</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>1-ni-assign-to-high</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2-ni-branch-on-high-twice</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3-ni-high-guard-equal-branches</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4-ni-branch-on-high-twice-prop2</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>5-ni-temp-impl-flow</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>6-ni-branch-assign-equal-val</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>7-ni-explicit-flow</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>8-ni-explicit-flow-while</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>9-ni-equal-output</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10-ni-rsa-exponentiation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1-sens-equal-sums</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2-sens-equal-sums-two-arrays</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3-sens-abs-diff-up-to-k</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4-sens-abs-diff-up-to-k-two-arrays</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5-sens-two-arrays-equal-k</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>6-sens-diff-up-to-explicit-k</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>7-sens-diff-up-to-explicit-k-sum</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>8-sens-explicit-swap</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>9-sens-explicit-swap-prop2</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>10-sens-equal-k</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>11-sens-equal-k-twice</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>12-sens-diff-up-to-forall-k</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Total VAMPIRE</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Unique VAMPIRE</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>25</td>
<td>14</td>
</tr>
</tbody>
</table>
Conclusion

- Trace Logic Language and Semantics
- Verification Approach - Vampire and Trace Lemmas
- Application to Relational Verification
Extra slides
Background Theory

- Full first-order logic with equality and uninterpreted functions
- Iterations - Datatype \((0, s, p, <)\) (no arithmetic!)
- Timepoints - Uninterpreted Sort
- Values of program variables - Integers
Motivating example - Property in Trace Logic

\[ \text{Swapped}_a(k) := \]
\[ 0 \leq k < k + 1 < a.\text{length} \]
\[ \land \quad \forall pos^\parallel.((pos \not\equiv k \land pos \not\equiv k + 1) \rightarrow \]
\[ a(pos, t_1) \simeq a(pos, t_2)) \]
\[ \land \quad a(k, t_1) \simeq a(k + 1, t_2) \]
\[ \land \quad a(k, t_2) \simeq a(k + 1, t_1) \]

\[ \forall k^\parallel.(\text{Swapped}_a(k) \rightarrow \text{sum}(\text{end}, t_1) \simeq \text{sum}(\text{end}, t_2)) \]
Rapid Tool

Available at:

https://github.com/gleiss/rapid
new: 40588. less(-4,0) — less(-6,4)
new: 40589. less(-4,-1) — less(-4,4)
new: 40590. less(-4,-1) — less(-3,4)
new: 40591. less(-4,-1) — less(-2,4)
new: 40592. less(-4,-1) — 0 = 4 — less(0,4)
new: 40593. less(-4,-1) — less(4,0)
new: 40594. less(-4,2) — less(-2,4)
new: 40595. less(-4,2) — less(-1,4)
new: 40596. less(-4,2) — less(0,4)
new: 40597. less(-4,2) — less(4,3)
new: 40598. less(-4,2) — less(1,4)
new: 40599. less(-4,-2) — less(-4,4)
new: 40600. less(-4,-2) — less(-3,4)
new: 40601. less(-4,-3) — less(-4,4)
new: 40602. less(-4,3) — less(-3,4)
new: 40603. less(-4,3) — less(-1,4)
new: 40604. less(-4,3) — less(-2,4)
new: 40605. less(-4,3) — less(0,4)
new: 40606. less(-4,3) — less(1,4)
new: 40607. less(-4,3) — less(2,4)
new: 40608. less(-4,4) — less(-3,4)
new: 40609. less(-4,4) — less(-2,4)
new: 40610. less(-4,4) — less(2,4)
new: 40611. less(-4,4) — less(1,4)
new: 40612. less(-4,4) — less(0,4)
new: 40613. less(-4,4) — less(-1,4)
new: 40614. less(-4,5) — less(-1,4)
new: 40615. less(-4,5) — less(1,4)
new: 40616. less(-4,5) — less(0,4)