Automated Program Reasoning

Laura Kovács
Automated Reasoning

In a vague sense, automated reasoning involves:

1. Representing a problem as a mathematical/logical statement

2. Computer-supported automatic check whether this statement is true
Automated Reasoning

in Program Analysis

My group @ TU Wien applies automated reasoning for:

- Software correctness
- Generating program properties
- Software synthesis
- System security
- …
Automated Reasoning

in Program Analysis

My group @ TU Wien applies automated reasoning for:

- Software correctness
- Generating program properties
- Software synthesis
- System security
- …
Automated Reasoning for Software Correctness
(ex. ~200kLoC, VAMPIRE prover)
a=0, b=0, c=0;
while (a<n) do
if A[a]>0 then B[b]=A[a]+h(b); b=b+1;
else C[c]=A[a]; c=c+1;
end if
a=a+1;
end do
a=0, b=0, c=0;
while (a<n) do
if A[a]>0 then B[b]=A[a]+h(b); b=b+1;
    else C[c]=A[a]; c=c+1;
a=a+1;
end do

Safety property:
(∀p)(0≤p<b ⇒
a=0, b=0, c=0;
while (a<n) do
if A[a]>0 then B[b]=A[a]+h(b); b=b+1;
else C[c]=A[a]; c=c+1;
end do

h

cnt=0, fib1=1, fib2=0;
while (cnt<n) do
t=fib1; fib1=fib1+fib2; fib2=t; cnt++;
end do

Automated Reasoning for Software Correctness
cnt=0, fib1=1, fib2=0;
while (cnt<n) do
  t=fib1; fib1=fib1+fib2; fib2=t; cnt++;
end do

Safety property:
fib1^4 + fib2^4 + 2*fib1*fib2^3 - 2*fib1^3*fib2 - fib1^2*fib2^2 - 1 = 0

a=0, b=0, c=0;
while (a<n) do
  if A[a]>0 then B[b]=A[a]+h(b); b=b+1;
    else C[c]=A[a]; c=c+1;
  a=a+1;
end do
Automated Reasoning for Software Correctness

Generating and Ensuring Safety Properties
Automated Reasoning for Security and Privacy

Generating and Ensuring Security and Privacy Properties
Automated Reasoning for Security and Privacy

- Array `a`: bit-wise representation of a secret key
- Hamming weight `hw`: number of 1s in the key

```
i=0, hw=0;
while (i<n) do
    hw=hw+a[i];
    i=i+1;
end do
```
Automated Reasoning for Security and Privacy

- Array \( a \): bit-wise representation of a secret key
- Hamming weight \( hw \): number of 1s in the key

\[
i=0, \ hw=0; \\
\text{while } (i<n) \text{ do} \\
\quad \ hw=hw+a[i]; \\
\quad i=i+1; \\
\text{end do}
\]

- Leaking \( hw \) ?
  (e.g. measure of side-channel leakage)
Verifying Relational Properties using Trace Logic

Gilles Barthe†, Renate Eilers†, Pamina Georgiou†, Bernhard Gleiss‡, Laura Kovács§, Matteo Maffei‡
†Max Planck Institute for Security and Privacy, Germany
‡IMDEA Software Institute, Spain
§TU Wien, Austria
§Chalmers University of Technology, Sweden

Abstract—We present a logical framework for the verification of relational properties in imperative programs. Our framework reduces verification of relational properties of imperative programs to a validity problem in trace logic, an expressive instance of first-order predicate logic. Trace logic draws its expressiveness from its syntax, which allows expressing properties over computation traces. Its axiomatization supports fine-grained reasoning about intermediate steps in program execution, notably loop iterations. We present an algorithm to encode the semantics of programs as well as their relational properties in trace logic, and then show how first-order theorem proving can be used to reason about the resulting trace logic formulas. Our work is implemented in the tool RAPID and evaluated with examples coming from the security field.

- No matter what permutation of a, the hw is the same

```c
func main()
{
    const Int[] a;
    const Int alength;
    Int i = 0;
    Int hw = 0;
    while (i < alength)
    {
        hw = hw + a[i];
        i = i + 1;
    }
}
```
\[ i=0, \ hw=0; \]
\[ \textbf{while} \ (i<n) \ \textbf{do} \]
\[ \ hw=hw+a[i]; \]
\[ i=i+1; \]
\[ \textbf{end do} \]
\[ i=0, h_w=0; \]
\[ \text{while } (i<n) \text{ do} \]
\[ h_w=h_w+a[i]; \]
\[ i=i+1; \]
\[ \text{end do} \]
Chalmers

Laura Kovács

$i=0, \ hw=0$;

while ($i<n$) do

$hw=hw+a[i]$;

$i=i+1$;

end do

Relational Verification

Automated Reasoning for Security and Privacy
\begin{verbatim}
Chalmers

Laura Kovács

i=0, \text{hw}=0;
while (i<n) do
  \text{hw}=\text{hw}+a[i];
i=i+1;
end do
\end{verbatim}

\begin{verbatim}
Automated Reasoning for Security and Privacy

Relational Verification

Input array \textit{a}

Induction

Input array \textit{a}

\textit{hw} = \textit{hw}
\end{verbatim}
i=0, hw=0;
while (i<n) do
  hw=hw+a[i];
i=i+1;
end do

hw = hw
i=0, hw=0;
while (i<n) do
    hw=hw+a[i];
    i=i+1;
end do

i=0, hw=0;
while (i<n) do
    hw=hw+a[i];
    i=i+1;
end do

\[
\text{hw} = \text{hw}
\]
Automated Reasoning for Security and Privacy

Relational Verification

Input array $a$

Initial values:
- $i = 0$
- $hw = 0$

While loop:

1. $hw = hw + a[i]$;
2. $i = i + 1$;
3. Repeat until $i < n$.

Result:

$hw = hw$
i=0, hw=0; 
while (i<n) do 
    hw=hw+a[i]; 
    i=i+1; 
end do

\[ hw = hw \]
Automated Reasoning for Security and Privacy

We proved 11 unique problems from security and privacy.

<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>Total VAMPIRE</td>
<td>15</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>Unique VAMPIRE</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>25</td>
<td>14</td>
<td>13</td>
</tr>
</tbody>
</table>
Automated Reasoning for Security, Privacy, Safety, ...

New Reasoning Challenges:

• **Software semantics** in (extensions) of first-order logic

• **First-order theories** of data structures

• **Induction**
## What Kind of Automated Reasoners can be Used?

<table>
<thead>
<tr>
<th>Type</th>
<th>Input</th>
<th>Examples</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT Solver</td>
<td>Propositional formulae</td>
<td>MiniSat, Lingeling</td>
<td>Intel</td>
</tr>
<tr>
<td>SMT Solvers</td>
<td>(First-order) formulae + theories</td>
<td>CVC4, Z3</td>
<td>Microsoft, Amazon</td>
</tr>
<tr>
<td>Theorem Provers</td>
<td>First-order formulae (+ theories)</td>
<td>Vampire, E</td>
<td>Intel, Amazon</td>
</tr>
<tr>
<td>Proof Assistants (interactive)</td>
<td>Higher-order formulae</td>
<td>Isabelle, Coq</td>
<td>Intel, Apple</td>
</tr>
</tbody>
</table>
# Our Automated Reasoner: VAMPIRE

<table>
<thead>
<tr>
<th></th>
<th>Input</th>
<th>Examples</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SAT Solver</strong></td>
<td>Propositional formulae</td>
<td>MiniSat, Lingeling</td>
<td>Intel</td>
</tr>
<tr>
<td><strong>SMT Solvers</strong></td>
<td>(First-order) formulae + theories</td>
<td>CVC4, Z3</td>
<td>Microsoft, Amazon</td>
</tr>
<tr>
<td><strong>Theorem Provers</strong></td>
<td>First-order formulae (+ theories)</td>
<td><strong>Vampire</strong>, E</td>
<td>Intel, Amazon</td>
</tr>
<tr>
<td><strong>Proof Assistants</strong></td>
<td>Higher-order formulae</td>
<td>Isabelle, Coq</td>
<td>Intel, Apple</td>
</tr>
</tbody>
</table>
What is VAMPIRE?

- An automated theorem prover for first-order logic and theories.

  https://vprover.github.io/download.html

- Completely automatic: once you started a proof attempt, it can only be interrupted by terminating the process.

- Champion of the CASC world-cup in first-order theorem proving: won CASC > 55 times.
VAMPIRE

- It produces detailed proofs
- It competes with SMT solvers on their problems
- In normal operation, it is saturation-based
- It is portfolio-based - works best when uses lots of strategies
- It supports lots of extra features and options helpful, for example, system security, including induction and theory reasoning.
Proof by Refutation

Given an input problem with assumptions \( F_1, \ldots, F_n \) and goal \( G \):

1. Negate the conjecture (\( \neg G \));

2. Establish unsatisfiability of the set of formulas \( F_1, \ldots, F_n, \neg G \).
Automated Reasoning with VAMPIRE – Saturation

Search space
(initially, input problem)
Automated Reasoning with VAMPIRE – Saturation

Search space

Given formula
Automated Reasoning with VAMPIRE – Saturation

- Search space
- Given formula
- Candidate formulae
Automated Reasoning with VAMPIRE – Saturation

Search space

Given formula

Candidate formulae

Children
Automated Reasoning with VAMPIRE – Saturation

- Search space
- Children
Automated Reasoning with VAMPIRE – Saturation

Search space
Automated Reasoning with VAMPIRE – Saturation

Search space

Given formula
Automated Reasoning with VAMPIRE – Saturation

Search space

Given formula

Candidate formulae
Automated Reasoning with VAMPIRE – Saturation

Search space

Given formula

Candidate formulae

Children
Automated Reasoning with VAMPIRE – Saturation

Search space

Children
Automated Reasoning with VAMPIRE – Saturation

Search space
Automated Reasoning with VAMPIRE – Saturation

MEMORY

Search space
Automated Reasoning with VAMPIRE – In practice

In practice there are three possible scenarios:

1. At some moment proof is found; in this case, the input is valid/true.

2. Saturation will terminate without ever finding a proof, in this case the input is satisfiable.

3. Saturation will run until we run out of resources, but without ever finding a proof. In this case it is unknown whether the input is valid.
In practice there are three possible scenarios:

1. At some moment proof is found; in this case, the input is valid/true.

2. Saturation will terminate without ever finding a proof, in this case the input is satisfiable.

3. Saturation will run until we run out of resources, but without ever finding a proof. In this case it is unknown whether the input is valid.

CHALLENGE: How to solve unknown?
In practice there are three possible scenarios:

1. At some moment proof is found; in this case, the input is valid/true.

2. Saturation will terminate without ever finding a proof, in this case the input is satisfiable.

3. Saturation will run until we run out of resources, but without ever finding a proof. In this case it is unknown whether the input is valid.

CHALLENGE: How to solve unknown? How to improve performance?
Automated Program Reasoning – Our recipe

First-Order Theorem Proving in Software Development
Automated Program Reasoning – Our recipe

I can’t get no satisfaction:
- and I try …
- and I try …
- and I try …
- and I try

[The Rolling Stones]
Automated Reasoning for Security, Privacy, Safety, ...

**New Reasoning Challenges in Systems Engineering**

- **Software semantics** in (extensions) of first-order logic

- **First-order theories** of data structures

- **Induction**
Automated Reasoning for Security, Privacy, Safety, ...

New Reasoning Challenges in Systems Engineering

- **Software semantics** in (extensions) of first-order logic

---

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.

1. 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.

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

Fig. 1. Program copying positive elements from array a to b.
Automated Reasoning for Security, Privacy, Safety, …

New Reasoning Challenges in Systems Engineering

- **Software semantics** in (extensions) of first-order logic

---

Trace Logic for Inductive Loop Reasoning

Pamina Georgiou, Bernhard Glaß, 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.

1. 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.

```c
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             i = i + 1;  
12         }  
13             i = i + 1;  
14         }  
15     assert (\forall k. 3li((0 <= k < j ∧ a.length >= 0) → b(k) = a(l)));  
```
Trace Logic for Automated Loop Reasoning

Loop Language $\rightarrow$ First-Order Language
**Trace Logic** for Automated Loop Reasoning

Extended Loop Language → First-Order Language Extended with Extra Symbols
Trace Logic for Automated Loop Reasoning

Extended Loop Language $\rightarrow$ First-Order Language Extended with Extra Symbols

Diagram:
- **Loop**
  - Extend language with extra symbols: loop cnt, array update predicates
- **Loop Properties**
- **Loop Requirements**
Trace Logic for Automated Loop Reasoning

Extended Loop Language $\rightarrow$ First-Order Language Extended with Extra Symbols

Extended Loop Language with extra symbols:
- loop cnt
- array update predicates

Derive consequences

Loop Properties

Loop Requirements
Trace Logic for Automated Loop Reasoning

Extended Loop Language → Trace Logic

Extended Loop Language

Loop

Extend language with extra symbols:
- loop cnt,
- array update predicates

Loop Properties

Derived consequences

Loop Requirements

Static Analysis of Loop Updates

Saturation-Based Theorem Proving

Trace Lemmas
Trace Logic for Automated Loop Reasoning

\begin{align*}
\text{assume } (\forall x \ h(x)=0) \land (0<n<a.\text{length})
\end{align*}

\begin{verbatim}
a=0, \ b=0, \ c=0;
\textbf{while} (a<n) \textbf{do}

\textbf{if} A[a]>0 \textbf{then} B[b]=A[a]+h(b);
\quad \quad b=b+1;
\quad \textbf{else} \ C[c]=A[a];
\quad \quad c=c+1;

a=a+1;
\textbf{end do}

\textbf{assert} (\forall p) \ (0\leq p< b \Rightarrow (\exists i)(0\leq i<n \land A[i]=B[p]))
\end{verbatim}
Trace Logic for Automated Loop Reasoning

```
assume ((∀x) h(x)=0) ∧ (0<n<a.length)

a=0, b=0, c=0;
while (a<n) do
  if A[a]>0 then B[b]=A[a]+h(b);
    b=b+1;
  else C[c]=A[a];
    c=c+1;
  a=a+1;
end do

assert (∀p) (0≤p<b ⇒ (∃i)(0≤i<n ∧ A[i]=B[p]))
```

Extra Symbols of Trace Logic

- loop counter cnt
- loop iteration predicate iter
- loop variable v as functions v\(^{(i)}\) of iteration i
- array update properties \(\text{upd}_V(i,p), \text{upd}_V(i,p,x)\) for array variable V, loop iteration i, array position p, array element value x
Trace Logic for Automated Loop Reasoning

assume \((\forall x) h(x)=0\) \& \((0<n<a.length)\)

\[
\begin{align*}
a &= 0, \quad b = 0, \quad c = 0; \\
\text{while } (a<n) \text{ do} \\
\text{if } A[a]>0 \text{ then } B[b] &= A[a]+h(b); \\
\quad b &= b+1; \\
\quad \text{else } C[c] &= A[a]; \\
\quad c &= c+1; \\
a &= a+1; \\
\text{end do}
\end{align*}
\]

assert \((\forall p) (0\leq p<b \Rightarrow (\exists i)(0\leq i<n \& A[i]=B[p]))\)

Loop Properties in Trace Logic

\[
\begin{align*}
(\forall i) (i\in iter \Leftrightarrow 0\leq i<cnt) \\
(\forall i) (i\in iter \Rightarrow a(i)<a(i+1)) \\
(\forall i) (i\in iter \Rightarrow (b(i)=b(i+1) \lor b(i+1)=b(i)+1)) \\
(\forall p) (0\leq p<b(cnt) \Rightarrow ((\exists i)i\in iter \& p=b(i) \& A(a(i))>0)) \\
\text{upd}_B(i,p,x) \& (\forall j>i)(\neg \text{upd}_B(i,p,x) \Rightarrow B(cnt)[p]=x) \\
(\forall i) (i\in iter \& A(a(i))>0 \Rightarrow (B(i+1)[b(i)]=A(a(i)) \& b(i+1)=b(i)+1 \& c(i+1)=c(i)))
\end{align*}
\]
Trace Logic for Automated Loop Reasoning

\[ \text{assume } ((\forall x) \, h(x)=0) \land (0<n<a.\text{length}) \]

\[ a=0, \, b=0, \, c=0; \]

\[ \textbf{while } (a<n) \textbf{ do} \]

\[ \text{if } A[a]>0 \text{ then } B[b]=A[a]+h(b); \]

\[ b=b+1; \]

\[ \text{else } C[c]=A[a]; \]

\[ c=c+1; \]

\[ a=a+1; \]

\[ \textbf{end do} \]

\[ \text{assert } (\forall p) \, (0\leq p<b \Rightarrow (\exists i)(0\leq i<n \land A[i]=B[p])) \]

Loop Properties in Trace Logic

\[ (\forall i) \, (i\in \text{iter} \iff 0\leq i<c) \]

\[ (\forall i) \, (i\in \text{iter} \Rightarrow a(i)<a(i+1)) \]

\[ (\forall i) \, (i\in \text{iter} \Rightarrow (b(i)=b(i+1) \vee b(i+1)=b(i)+1)) \]

\[ (\forall p) \, (0 \leq p<b^{(\text{cnt})} \Rightarrow ((\exists i)\, i\in \text{iter} \land p=b(i) \land A[a(i)]>0)) \]

\[ \text{upd}_{B}(i,p,x) \land (\forall j>i)(\neg \text{upd}_{B}(i,p,x) \Rightarrow B^{(\text{cnt})}[p]=x) \]

\[ (\forall i) \, (i\in \text{iter} \land A[a(i)]>0 \Rightarrow (B^{(i+1)}[b(i)]=A[a(i)] \land b^{(i+1)}=b(i)+1 \land c^{(i+1)}=c(i))) \]

Derive consequences using saturation-based theorem-proving
**Trace Logic** for Automated Loop Reasoning

```
assume ((\forall x) h(x)=0) \land (0<n<a.length)

a=0, b=0, c=0;
while (a<n) do
    if A[a]>0 then B[b]=A[a]+h(b);
        b=b+1;
    else C[c]=A[a];
        c=c+1;

    a=a+1;
end do
```

assert (\forall p) (0\leq p<b \Rightarrow (\exists i)(0\leq i<n \land A[i]=B[p]))

**Loop Properties in Trace Logic**

`(\forall i) (i\in iter \Leftrightarrow 0\leq i<cnt)`

`(\forall i) (i\in iter \Rightarrow a(i)<a(i+1))`  
`(\forall i) (i\in iter \Rightarrow (b(i)=b(i+1) \lor b(i+1)=b(i)+1))`

`(\forall p) (0 \leq p<b^{(cnt)} \Rightarrow ((\exists i)i\in iter \land p=b(i) \land A[a(i)]>0))`

`upd_{B}(i,p,x) \land (\forall j>i)(\neg upd_{B}(i,p,x) \Rightarrow B^{(cnt)}[p]=x)`

`(\forall i) (i\in iter \land A[a(i)]>0 \Rightarrow (B^{(i+1)}[b(i)]=A[a(i)] \land b^{(i+1)}=b^{(i)}+1 \land c^{(i+1)}=c^{(i)}))`

Derive consequences using saturation-based theorem-proving

`(\forall p) (0\leq p<b \Rightarrow (\exists i)(0\leq i<n \land A[i]=B[p]))`
Trace Logic for Automated Loop Reasoning

Loop Properties in Trace Logic

\[
\begin{align*}
& \forall i \ (i \in \text{iter} \iff 0 \leq i < \text{cnt}) \\
& \forall i \ (i \in \text{iter} \implies a(i) < a(i+1)) \\
& \forall i \ (i \in \text{iter} \implies (b(i) = b(i+1) \lor b(i+1) = b(i) + 1)) \\
& \forall p \ (0 \leq p < b(\text{cnt}) \implies ((\exists i) i \in \text{iter} \land p = b(i) \land A(a(i)) > 0)) \\
& \text{upd}_B(i, p, x) \land (\forall j > i) (\neg \text{upd}_B(i, p, x) \implies B(\text{cnt})[p] = x) \\
& \forall i \ (i \in \text{iter} \land A(a(i)) > 0 \implies (B(i+1)[b(i)] = A(a(i)) \land b(i+1) = b(i) + 1 \land c(i+1) = c(i))}
\end{align*}
\]

Derive consequences using saturation-based theorem-proving
Trace Logic for Automated Loop Reasoning

Deriving useful loop properties in saturation

- for every loop variable \( v \) → target symbols \( v_0 \) and \( v \)
  \[ v^{(0)} = v_0 \quad \text{and} \quad v^{(\text{cnt})} = v \]
Trace Logic for Automated Loop Reasoning

Deriving useful loop properties in saturation

- for every loop variable $v \rightarrow$ target symbols $v_0$ and $v$

- useable symbols:
  - target or interpreted symbols
  - skolem functions introduced while preprocessing
**Trace Logic** for Automated Loop Reasoning

### Deriving useful loop properties in saturation

- for every loop variable $v$ → target symbols $v_0$ and $v$

- **useable** symbols:
  - target or interpreted symbols
  - skolem functions introduced while preprocessing

- **useful** clauses:
  - contain only **useable** symbols
Deriving useful loop properties in saturation

- for every loop variable $v \rightarrow$ target symbols $v_0$ and $v$

- useable symbols:
  - target or interpreted symbols
  - skolem functions introduced while preprocessing

- useful clauses:
  - contain only useable symbols
  - contains at least a target symbols or a skolem functions

$x+y=y+x$ is not useful
Trace Logic for Automated Loop Reasoning

Deriving useful loop properties in saturation

- for every loop variable $v \rightarrow$ target symbols $v_0$ and $v$

- useable symbols:
  - target or interpreted symbols
  - skolem functions introduced while preprocessing

- useful clauses:
  - contain only useable symbols
  - contains at least a target symbols or a skolem functions

- simplication/derivation ordering $\succ$ in saturation:
  - useless symbols $\succ$ useable symbols
Automated Reasoning for Security, Privacy, Safety, ...

New Reasoning Challenges in Systems Engineering

• Software semantics in trace logic

Trace Logic for Inductive Loop Reasoning

Pamela 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...

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.∃l.(0 ≤ k ≤ j ∧ a.length ≥ 0) → b(k) = a(l)))

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

New Reasoning Challenges

- Software semantics

Proved 24 unique problems
Automated Reasoning for Security, Privacy, Safety, …

New Reasoning Challenges in Systems Engineering

➢ Software semantics in trace logic

➢ First-order theories of data structures
Automated Reasoning for Security, Privacy, Safety, ...

New Reasoning Challenges in Systems Engineering

➢ Software semantics *in trace logic*

➢ First-order theories of data structures
  - Term algebras: subterm predicate for finite axiomatisations
Automated Reasoning for Security, Privacy, Safety, ...

New Reasoning Challenges in Systems Engineering

➢ Software semantics

➢ First-order theories
  - Term algebras:
    - Subterm predicate for finite axiomatisations
    - Arrays: polymorphic theory with extensionality
    - Integers/reals: incomplete but sound set of axioms
    - Natural numbers: integer vs term algebra encoding

Proved 50 unique problems
Automated Reasoning for Security, Privacy, Safety, ...

New Reasoning Challenges in Systems Engineering

- **Software semantics** *in trace logic*

- **First-order theories** of data structures
  - Term algebras: subterm predicate for finite axiomatisations
  - Arrays: polymorphic theory with extensionality
  - Integers/reals: incomplete but sound set of axioms
  - Natural numbers: integer vs term algebra *encoding*
Automated Reasoning for Security, Privacy, Safety, ...

New Reasoning Challenges in Systems Engineering

➢ Software semantics in trace logic

➢ First-order theories of data structures

Bridging the gap between SMT solving and first-order theorem proving
Automated Reasoning for Security, Privacy, Safety, …

New Reasoning Challenges in Systems Engineering

- Software semantics in trace logic

- First-order theories of data structures
  reasoning with quantifiers + int/real, naturals/term algebras, arrays, …

- Induction
New Reasoning Challenges in Systems Engineering

Induction with Generalization in Superposition Reasoning

Márton Hajdú¹, Petra Hozzová¹, Laura Kovács¹,²[E], Johannes Schoisswohl¹,³, and Andrei Voronkov³,⁴

¹ TU Wien, Vienna, Austria
laura.kovacs@tuwien.ac.at
² Chalmers University of Technology, Gothenburg, Sweden
³ University of Manchester, Manchester, UK
⁴ EasyChair, Manchester, UK

Abstract. We describe an extension of automating induction in superposition-based reasoning by strengthening inductive properties and generalizing terms over which induction should be applied. We implemented our approach in the first-order theorem prover VAMPIRE and evaluated our work against state-of-the-art reasoners automating induction. We demonstrate the strength of our technique by showing that many interesting mathematical properties of natural numbers and lists can be proved automatically using this extension.
Automated Reasoning for Security, Privacy, Safety, …

New Reasoning Challenges in Systems Engineering

- Induction

Proved 9+ unique problems
Automated Reasoning for Security, Privacy, Safety, ...

New Reasoning Challenges in Systems Engineering

- **Software semantics** in trace logic

- **First-order theories** of data structures
  reasoning with quantifiers + int/real, naturals/term algebras, arrays, ...

- **Induction**
  not a first-order property
Conclusion and Outlook

1. Automated reasoning will remain central in rigorous systems engineering.

   The role of automated reasoning in these areas is and will be growing.
Conclusion and Outlook

1. **Automated reasoning** will remain central in rigorous systems engineering.

   The role of automated reasoning in these areas is and will be growing.

2. **Automated reasoners** will be used by a large number of users who do not understand automated reasoning and by users with very elementary knowledge of logic.
Conclusion and Outlook

1. Automated reasoning will remain central in rigorous systems engineering.

   The role of automated reasoning in these areas is and will be growing.

2. Automated reasoners will be used by a large number of users who do not understand automated reasoning and by users with very elementary knowledge of logic.

   *Formal reasoning cannot be handled by engineers alone.*
Conclusion and Outlook

1. **Automated reasoning will remain central in rigorous systems engineering.**
   
   The role of automated reasoning in these areas is and will be growing.

2. **Automated reasoners will be used by a large number of users who do not understand automated reasoning and by users with very elementary knowledge of logic.**

   *Formal reasoning cannot be handled by engineers alone.*

   *Formal reasoning cannot be handled by experts alone.*
Conclusion and Outlook

1. **Automated reasoning** will remain central in rigorous systems engineering. The role of automated reasoning in these areas is and will be growing.

2. **Automated reasoners** will be used by a large number of users who do not understand automated reasoning and by users with very elementary knowledge of logic.

3. **Automated reasoning with theories** will remain the main challenge in ensuring system reliability (at least) for the next decade.