Automated Program Reasoning

Laura Kovács

for(syte) TU WIEN Informatics

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
- …

erc European Research Council
Supporting top researchers from anywhere in the world

FWF
Der Wissenschaftsfonds.

W|W|T|F
VIENNA SCIENCE
AND TECHNOLOGY FUND

amazon webservices™

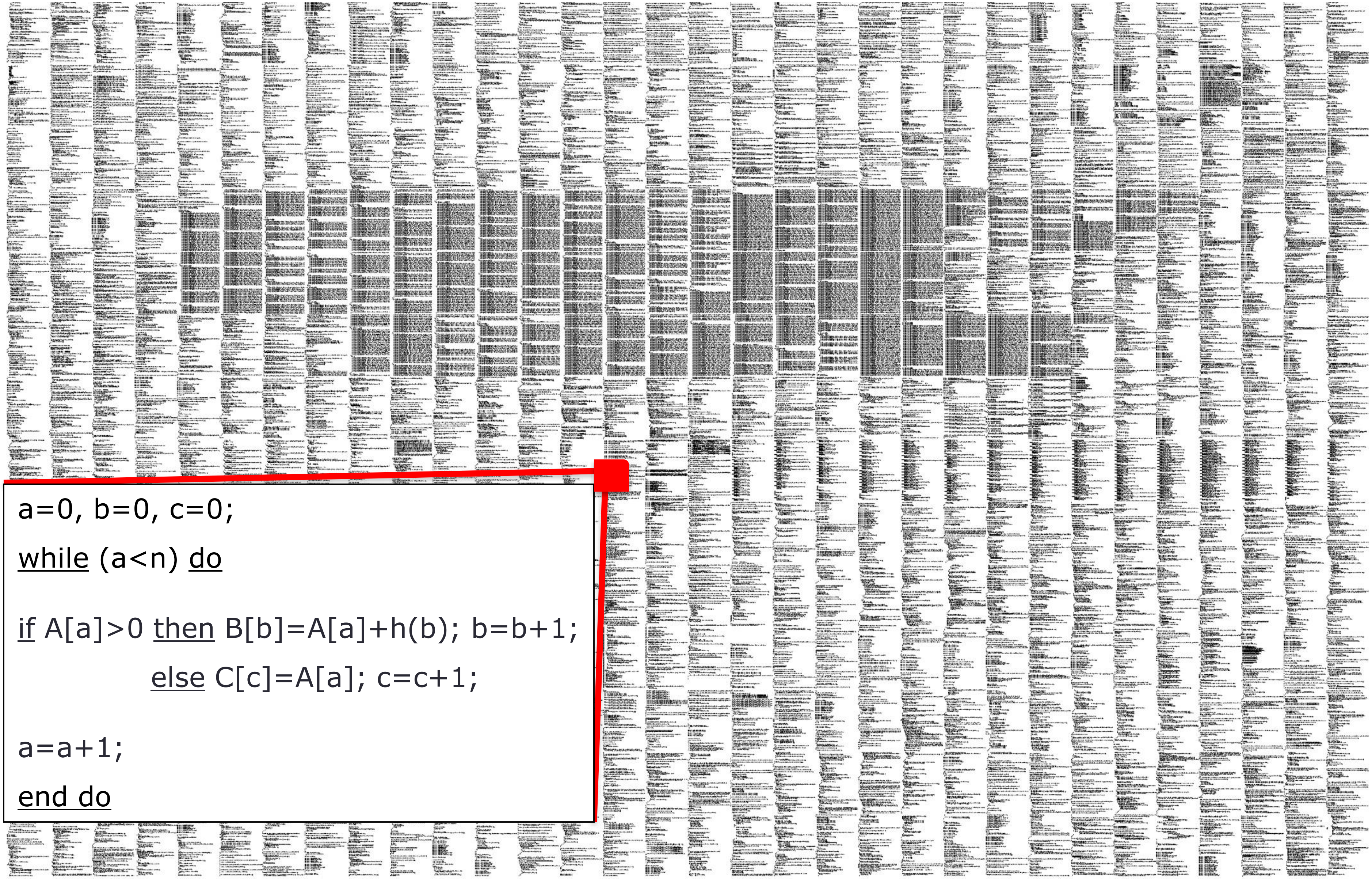
Microsoft

CERTORA

Automated Reasoning for Software Correctness

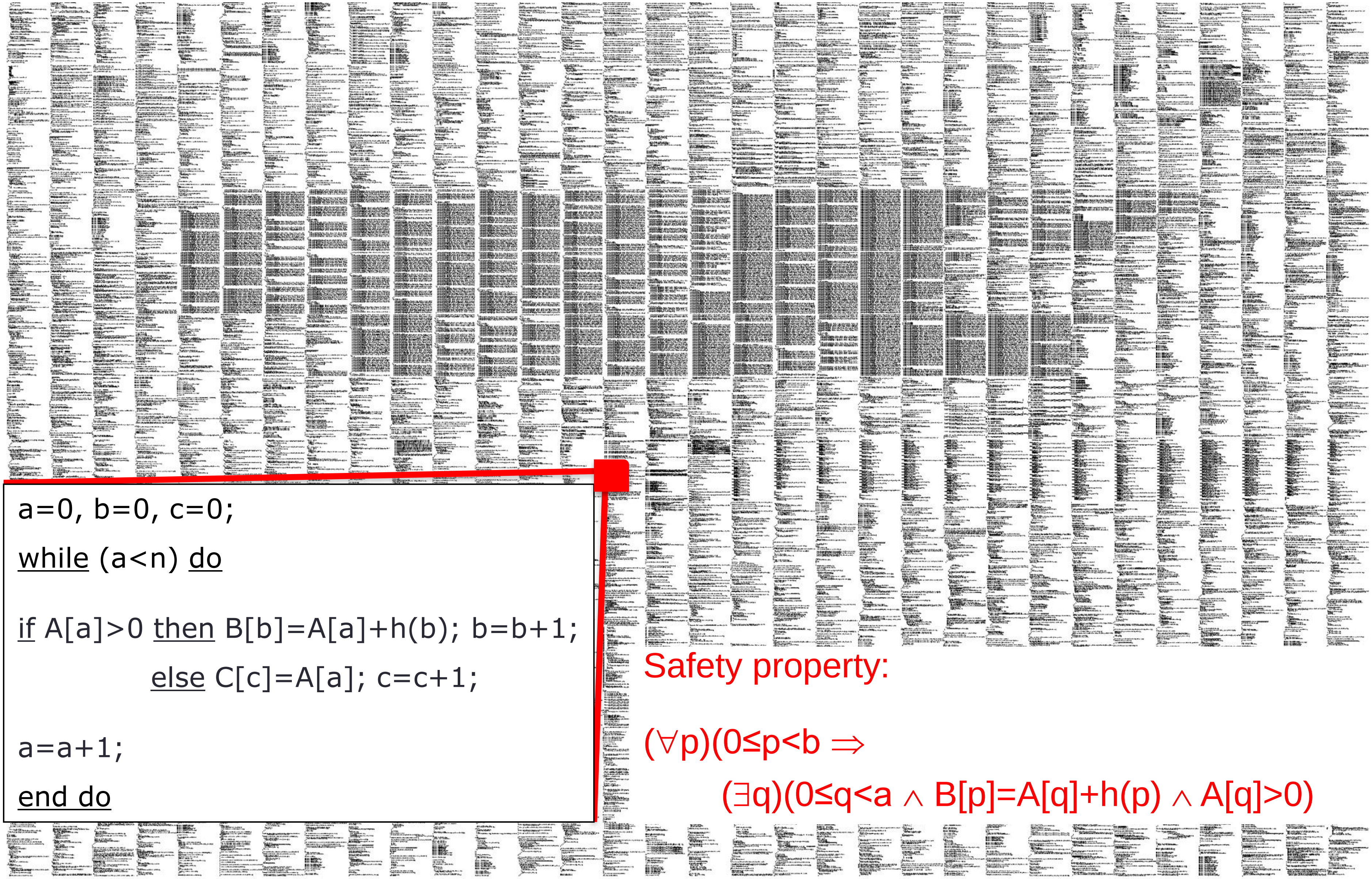
(ex. ~200kLoC, VAMPIRE prover)

Automated Reasoning for Software Correctness



```
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



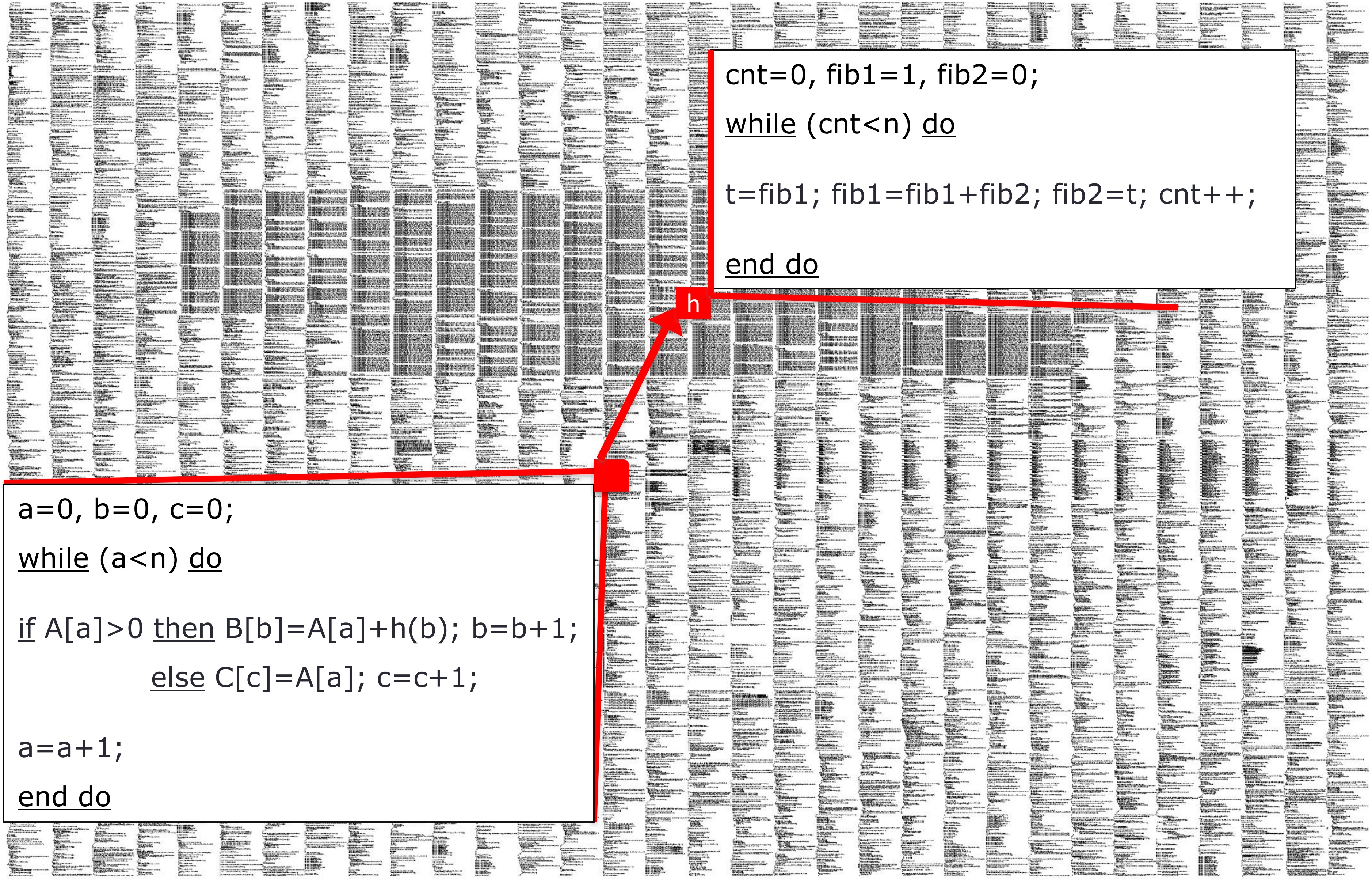
```
a=0, b=0, c=0;
while (a<n) do

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:

$(\forall p)(0 \leq p < b \Rightarrow$

$\quad (\exists q)(0 \leq q < a \wedge B[p]=A[q]+h(p) \wedge A[q]>0)$

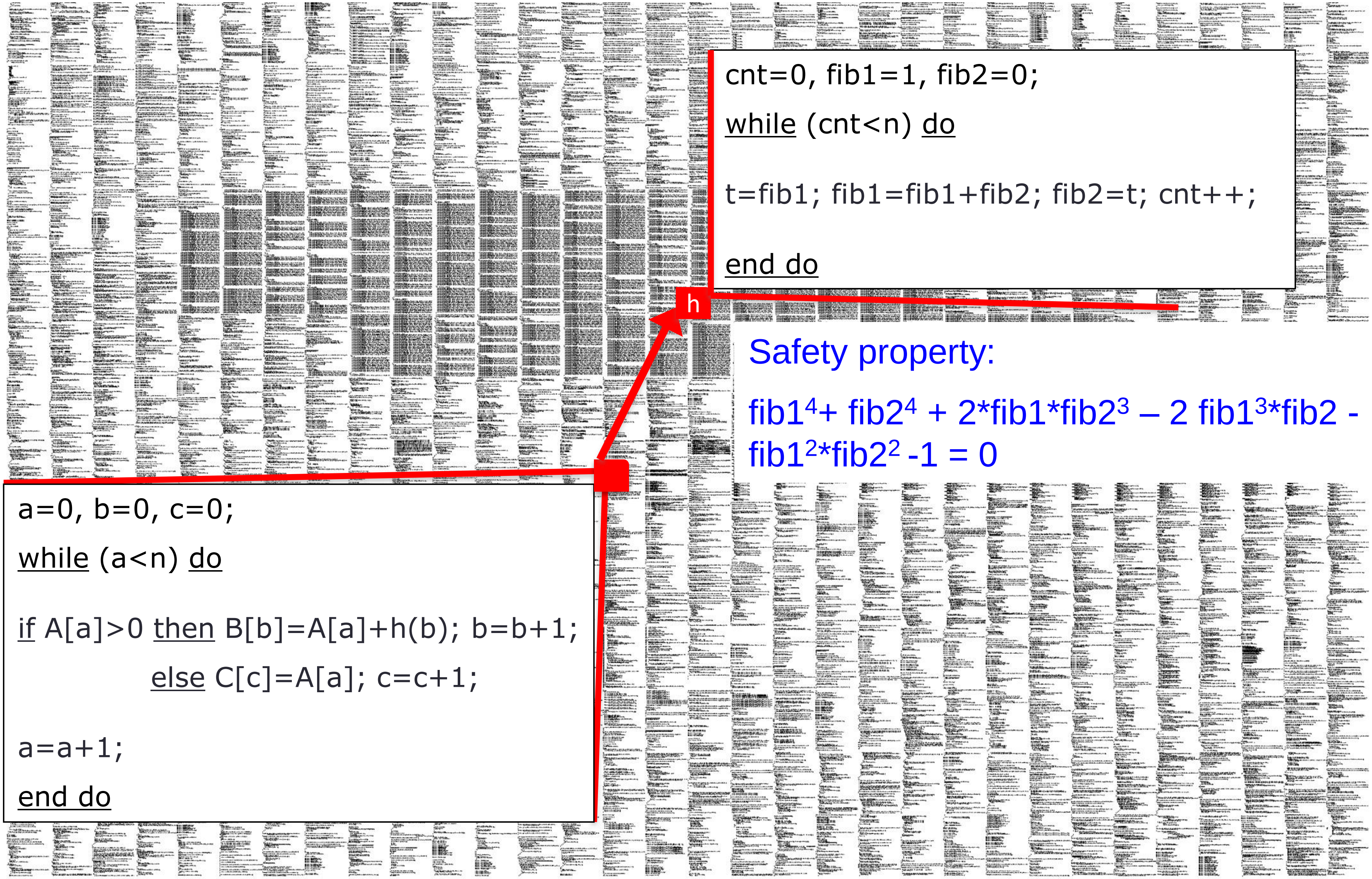
Automated Reasoning for Software Correctness



```
cnt=0, fib1=1, fib2=0;
while (cnt<n) do

t=fib1; fib1=fib1+fib2; fib2=t; cnt++;

end do
```

```
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



```
cnt=0, fib1=1, fib2=0;
while (cnt<n) do

t=fib1; fib1=fib1+fib2; fib2=t; cnt++;

end do
```

Safety property:

$fib1^4 + fib2^4 + 2*fib1*fib2^3 – 2\ fib1^3*fib2 - fib1^2*fib2^2 -1 = 0$

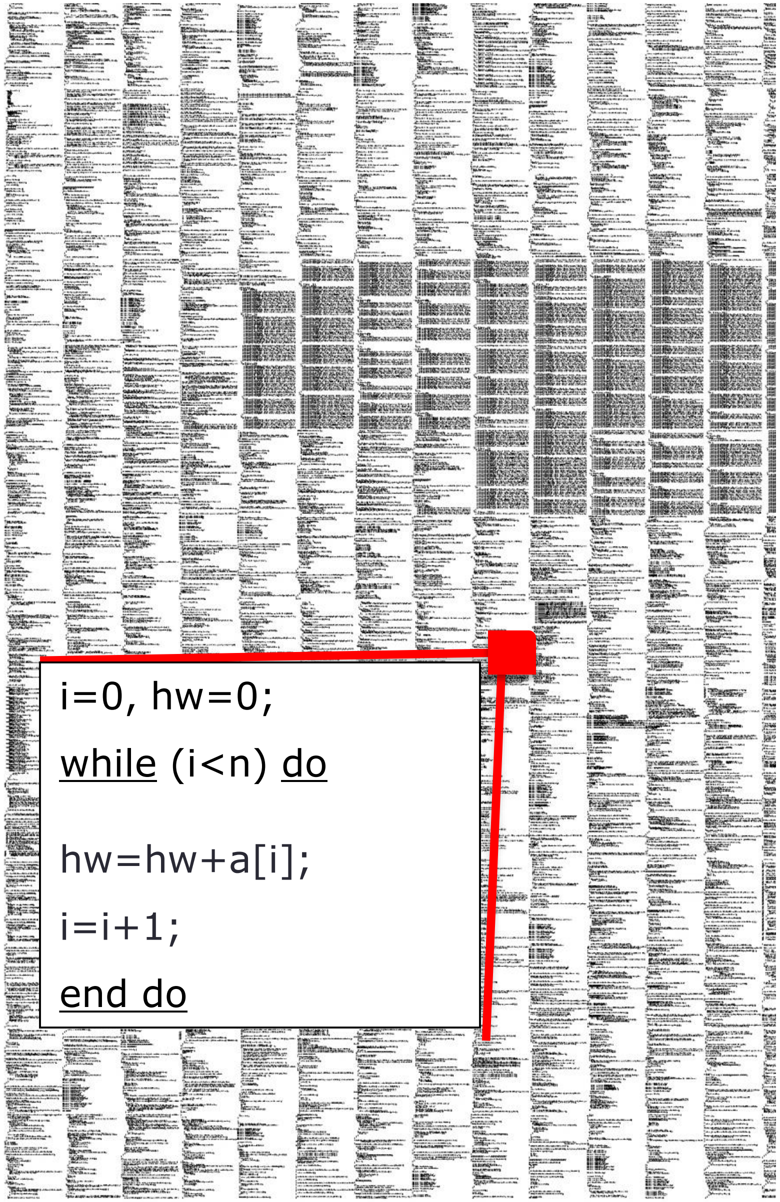
```
a=0, b=0, c=0;
while (a<n) do

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
```

Generating and Ensuring

Safety Properties

Generating and Ensuring Security and Privacy Properties

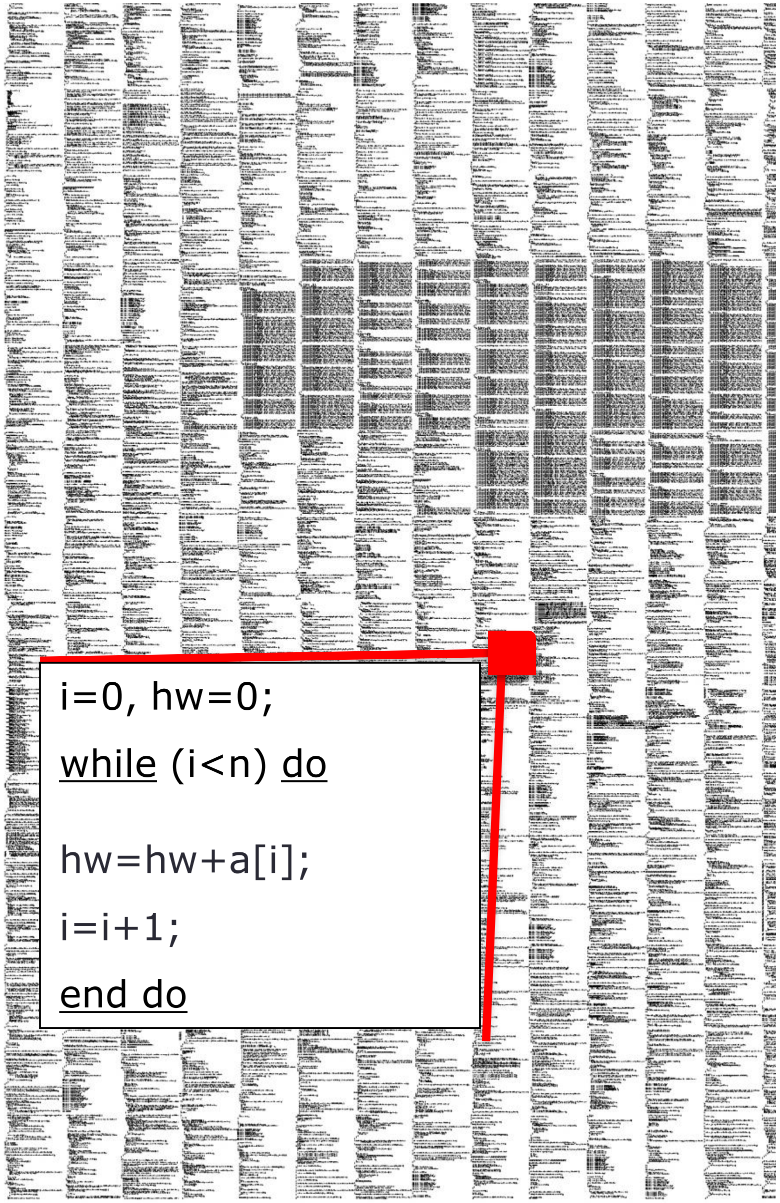
Automated Reasoning for Security and Privacy



```
i=0, hw=0;
while (i<n) do

hw=hw+a[i];
i=i+1;
end do
```

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

Automated Reasoning for Security and Privacy

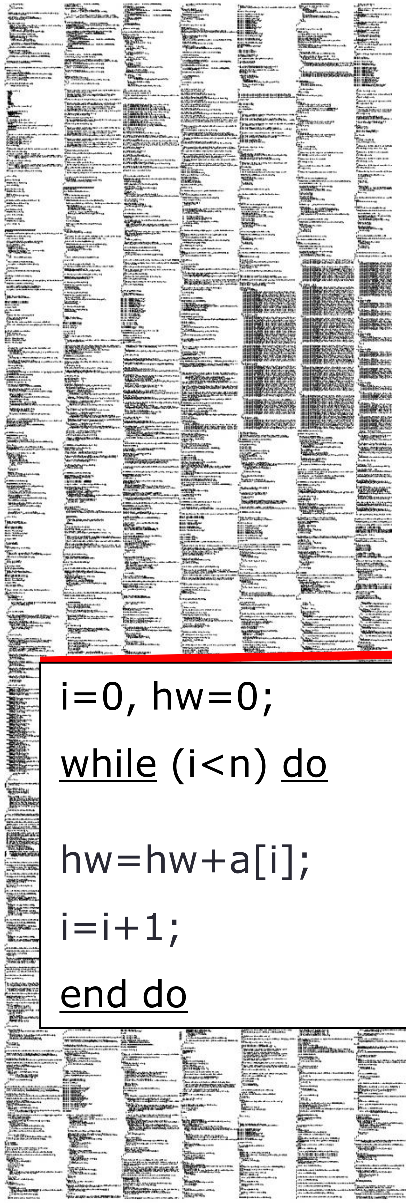


```
i=0, hw=0;
while (i<n) do

hw=hw+a[i];
i=i+1;
end do
```

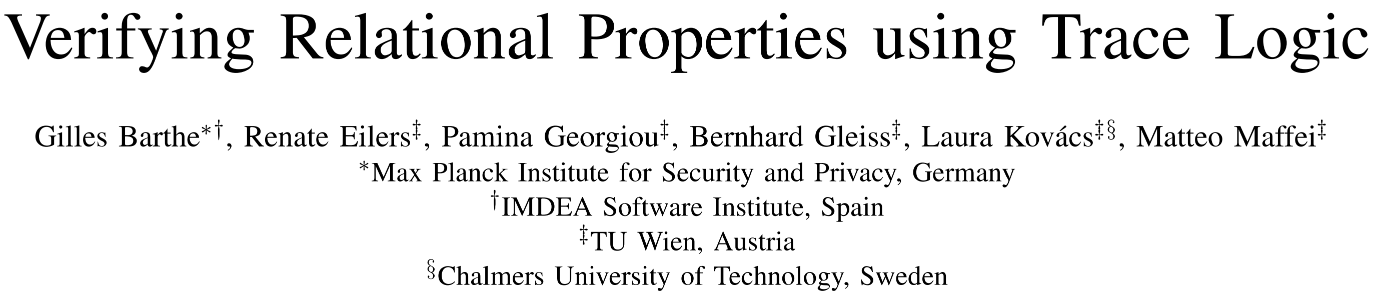
- Array **a**: bit-wise representation of a secret key
- Hamming weight **hw**: number of 1s in the key

- Leaking **hw ?**
  (e.g. measure of side-channel leakage)

Automated Reasoning for Security and Privacy

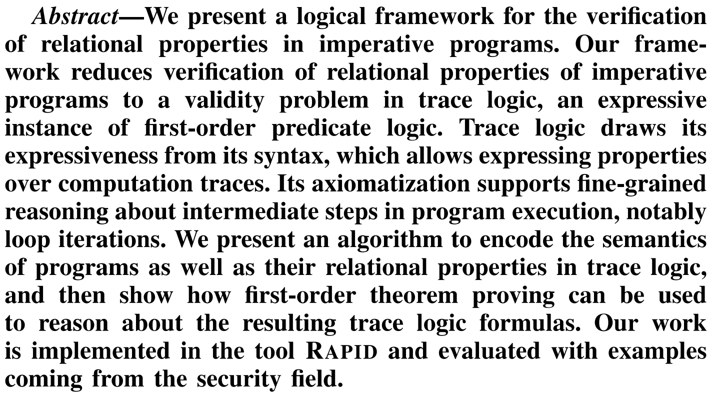


```
i=0, hw=0;
while (i<n) do

hw=hw+a[i];
i=i+1;
end do
```

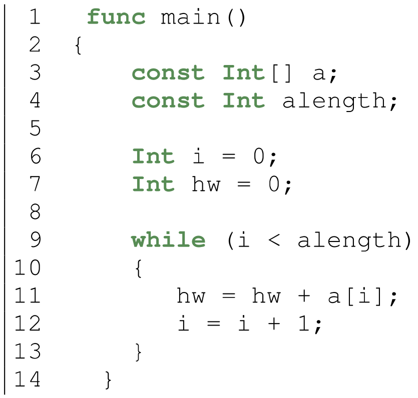


Verifying Relational Properties using Trace Logic

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



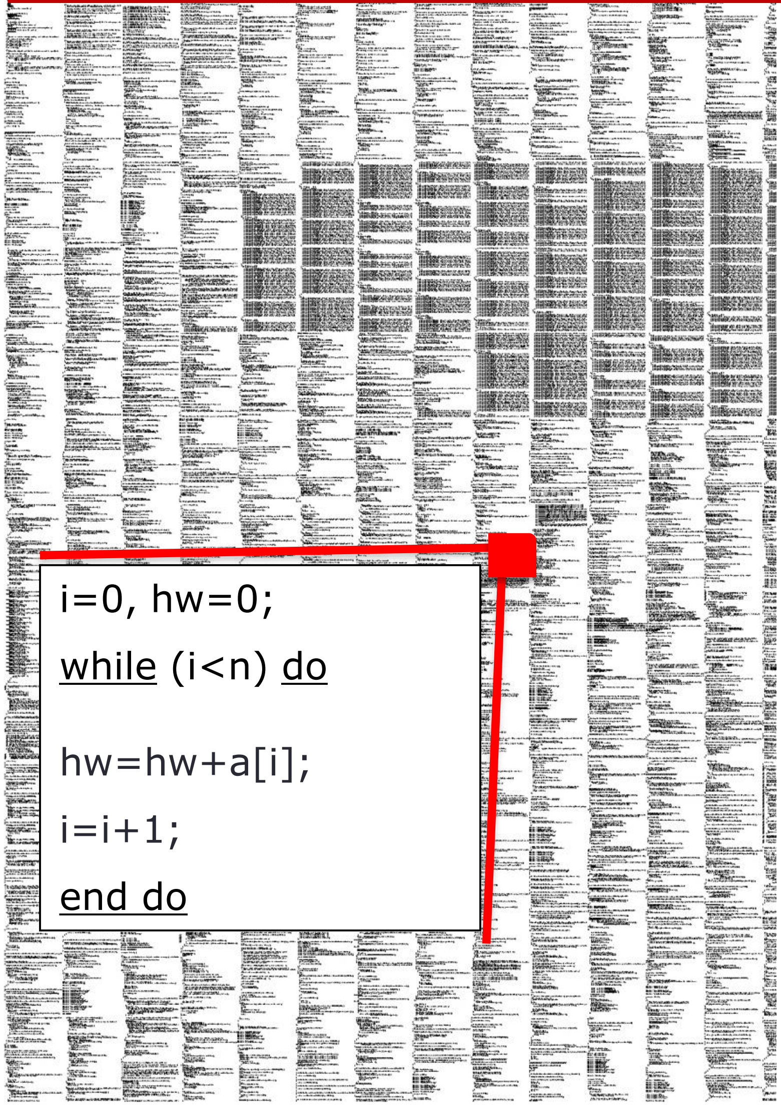
***Abstract*—We present a logical framework for the verification of relational properties in imperative programs. Our framework reduces verification of relational properties of imperative programs to a validity problem in trace logic, an expressive instance of first-order predicate logic. Trace logic 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.**



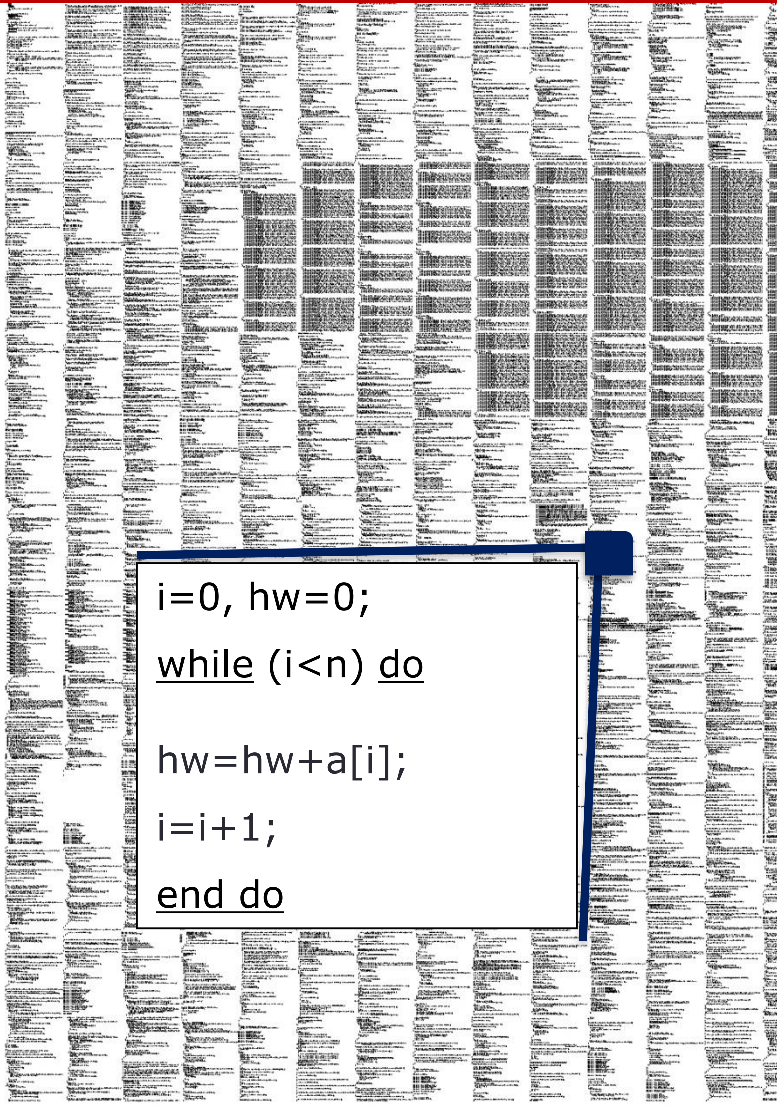
```
 func main()
{
    const Int[] a;
    const Int alength;

    Int i = 0;
    Int hw = 0;

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

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

Relational Verification

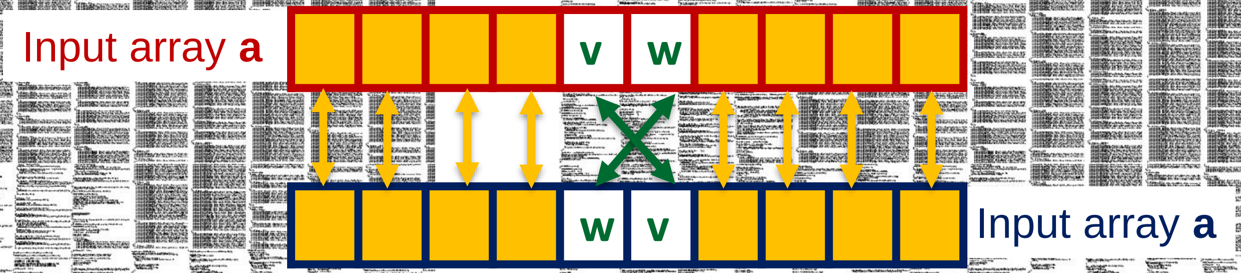


```
i=0, hw=0;
while (i<n) do

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
```

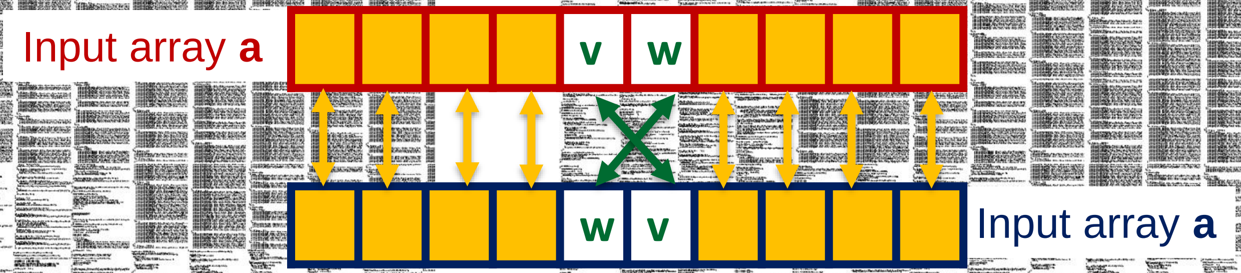
Relational Verification



```
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
```

Relational Verification

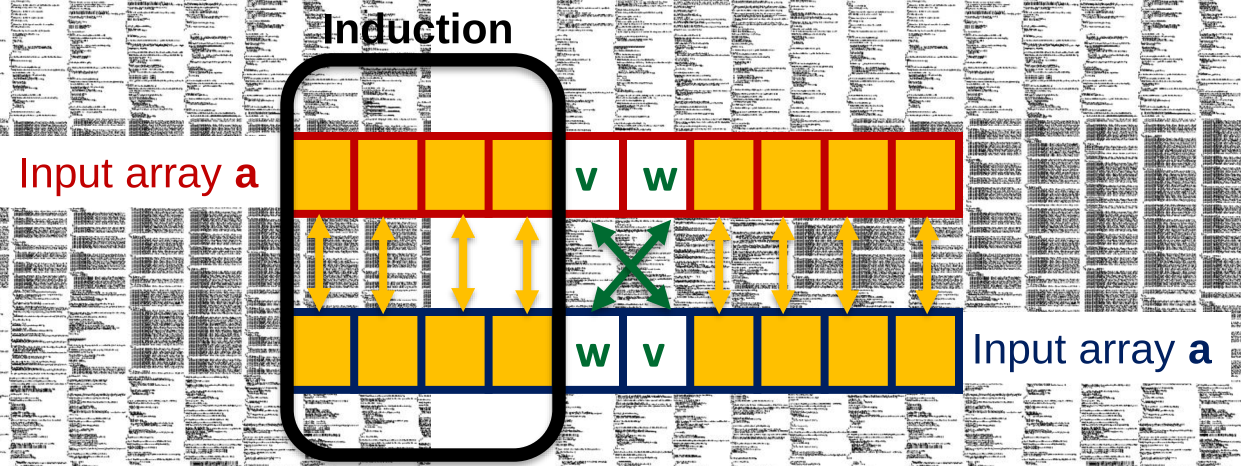


```
i=0, hw=0;
while (i<n) do

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
```

hw = hw

Relational Verification

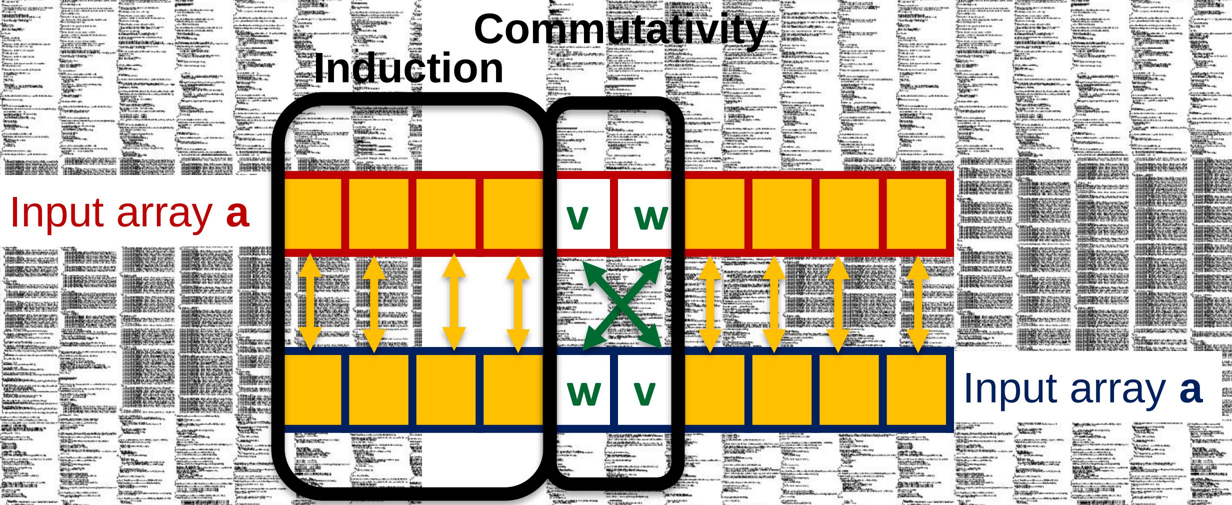


```
i=0, hw=0;
while (i<n) do

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
```

hw = hw

Relational Verification

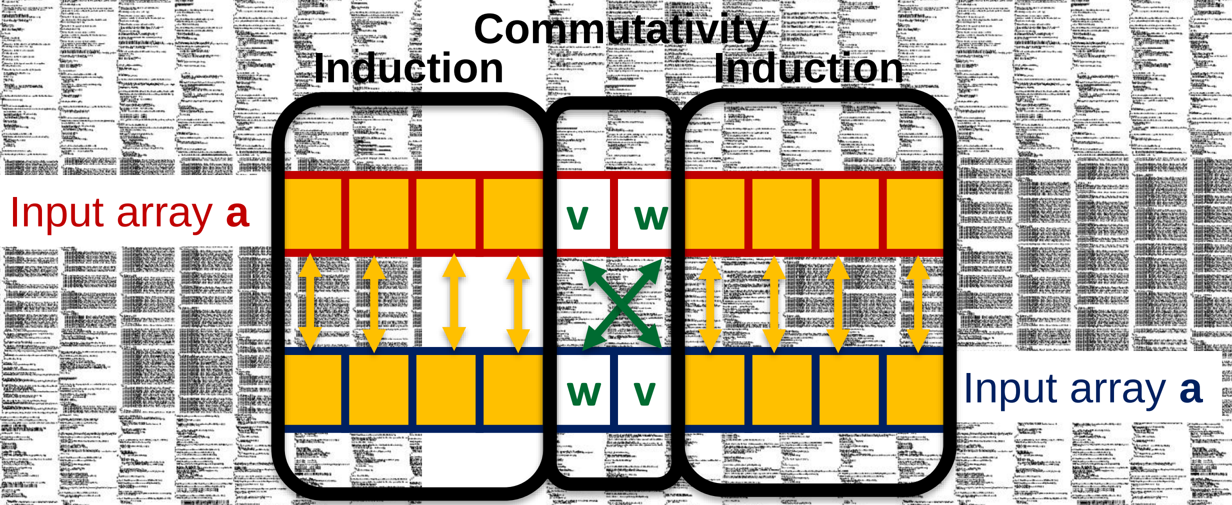


```
i=0, hw=0;
while (i<n) do

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
```

hw = hw

Relational Verification

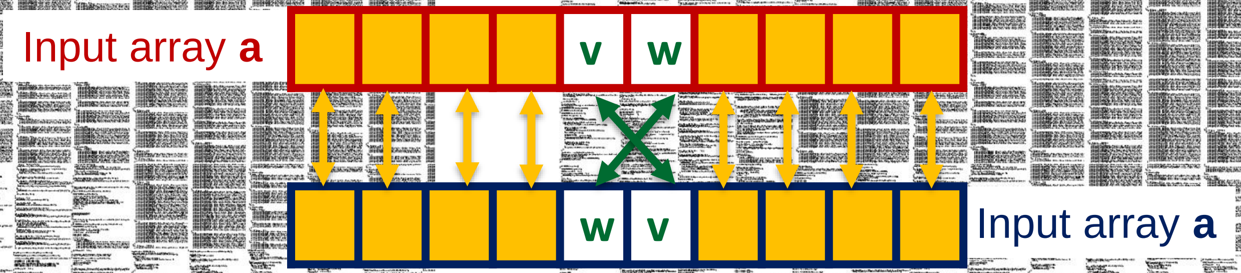


```
i=0, hw=0;
while (i<n) do

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
```

hw = hw

Relational Verification



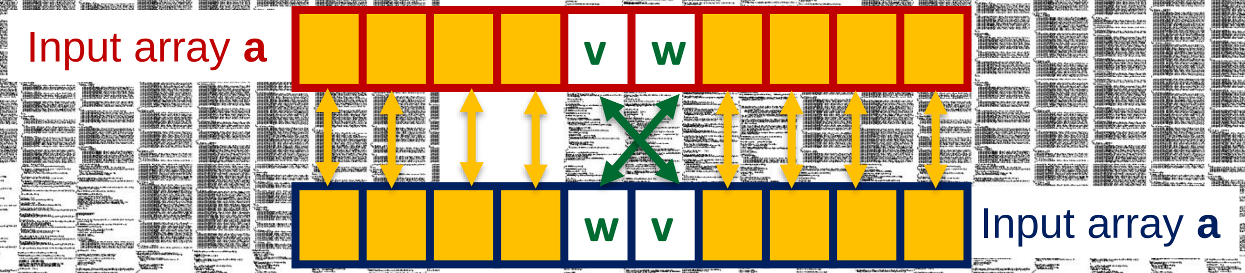
```
i=0, hw=0;
while (i<n) do

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
```

hw = hw

Automated Reasoning for Security and Privacy

Relational Verification (non-interference, sensitivity)

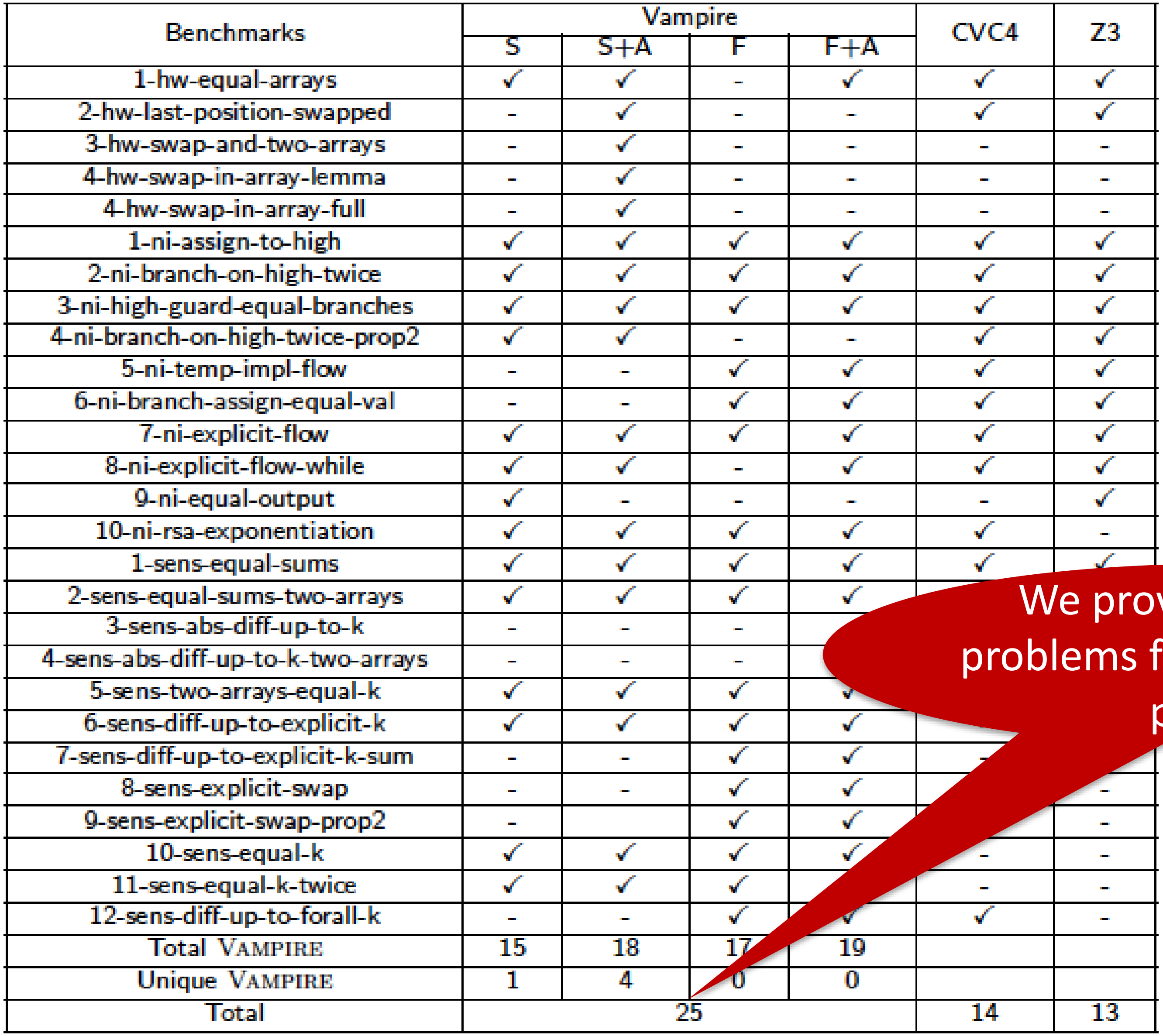


```
i=0, hw=0;
while (i<n) do

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
```

hw = hw

Automated Reasoning for Security and Privacy



Benchmarks	Vampire				CVC4	Z3
	S	S+A	F	F+A		
1-hw-equal-arrays	✓	✓	-	✓	✓	✓
2-hw-last-position-swapped	-	✓	-	-	✓	✓
3-hw-swap-and-two-arrays	-	✓	-	-	-	-
4-hw-swap-in-array-lemma	-	✓	-	-	-	-
4-hw-swap-in-array-full	-	✓	-	-	-	-
1-ni-assign-to-high	✓	✓	✓	✓	✓	✓
2-ni-branch-on-high-twice	✓	✓	✓	✓	✓	✓
3-ni-high-guard-equal-branches	✓	✓	✓	✓	✓	✓
4-ni-branch-on-high-twice-prop2	✓	✓	-	-	✓	✓
5-ni-temp-impl-flow	-	-	✓	✓	✓	✓
6-ni-branch-assign-equal-val	-	-	✓	✓	✓	✓
7-ni-explicit-flow	✓	✓	✓	✓	✓	✓
8-ni-explicit-flow-while	✓	✓	-	✓	✓	✓
9-ni-equal-output	✓	-	-	-	-	✓
10-ni-rsa-exponentiation	✓	✓	✓	✓	✓	-
1-sens-equal-sums	✓	✓	✓	✓	✓	✓
2-sens-equal-sums-two-arrays	✓	✓	✓	✓	[illegible]	[illegible]
3-sens-abs-diff-up-to-k	-	-	-	[illegible]	[illegible]	[illegible]
4-sens-abs-diff-up-to-k-two-arrays	-	-	-	[illegible]	[illegible]	[illegible]
5-sens-two-arrays-equal-k	✓	✓	✓	✓	[illegible]	[illegible]
6-sens-diff-up-to-explicit-k	✓	✓	✓	✓	-	[illegible]
7-sens-diff-up-to-explicit-k-sum	-	-	✓	✓	-	[illegible]
8-sens-explicit-swap	-	-	✓	✓	[illegible]	-
9-sens-explicit-swap-prop2	-		✓	✓	[illegible]	-
10-sens-equal-k	✓	✓	✓	✓	-	-
11-sens-equal-k-twice	✓	✓	✓	[illegible]	-	-
12-sens-diff-up-to-forall-k	-	-	✓	✓	✓	-
Total Vampire	15	18	17	19		
Unique Vampire	1	4	0	0		
Total	25				14	13

We proved 11 unique problems from security and privacy.

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?

	Input	Examples	Impact
SAT Solver	Propositional formulae	MiniSat, Lingeling	Intel
SMT Solvers	(First-order) formulae + theories	CVC4, Z3	Microsoft, Amazon
Theorem Provers	First-order formulae (+ theories)	Vampire, E	Intel, Amazon
Proof Assistants (interactive)	Higher-order formulae	Isabelle, Coq	Intel, Apple

Our Automated Reasoner: VAMPIRE

	Input	Examples	Impact
SAT Solver	Propositional formulae	MiniSat, Lingeling	Intel
SMT Solvers	(First-order) formulae + theories	CVC4, Z3	Microsoft, Amazon
Theorem Provers	First-order formulae (+ theories)	**Vampire**, E	Intel, Amazon
Proof Assistants (interactive)	Higher-order formulae	Isabelle, Coq	Intel, Apple

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.

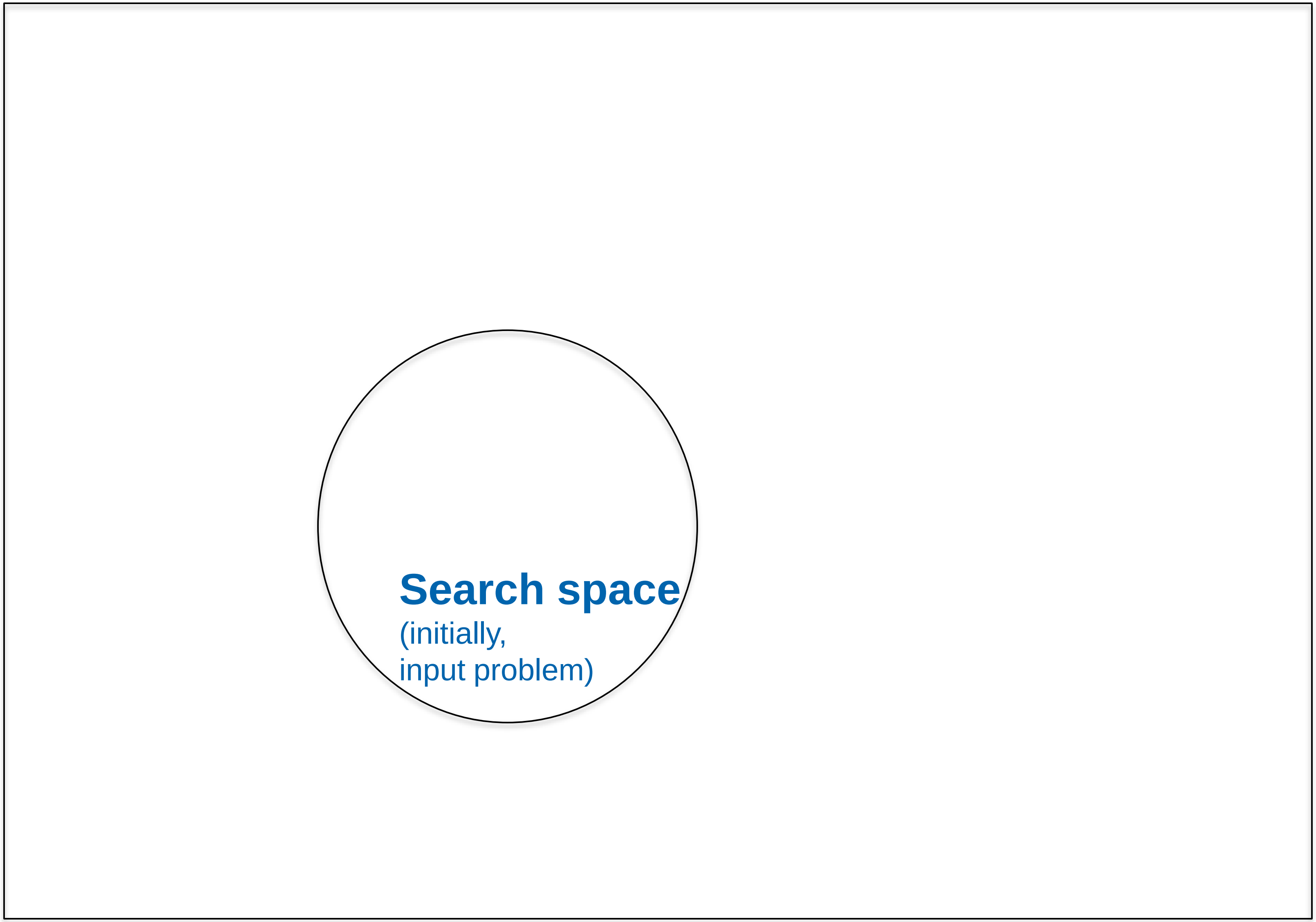
Automated Reasoning with VAMPIRE

Proof by Refutation

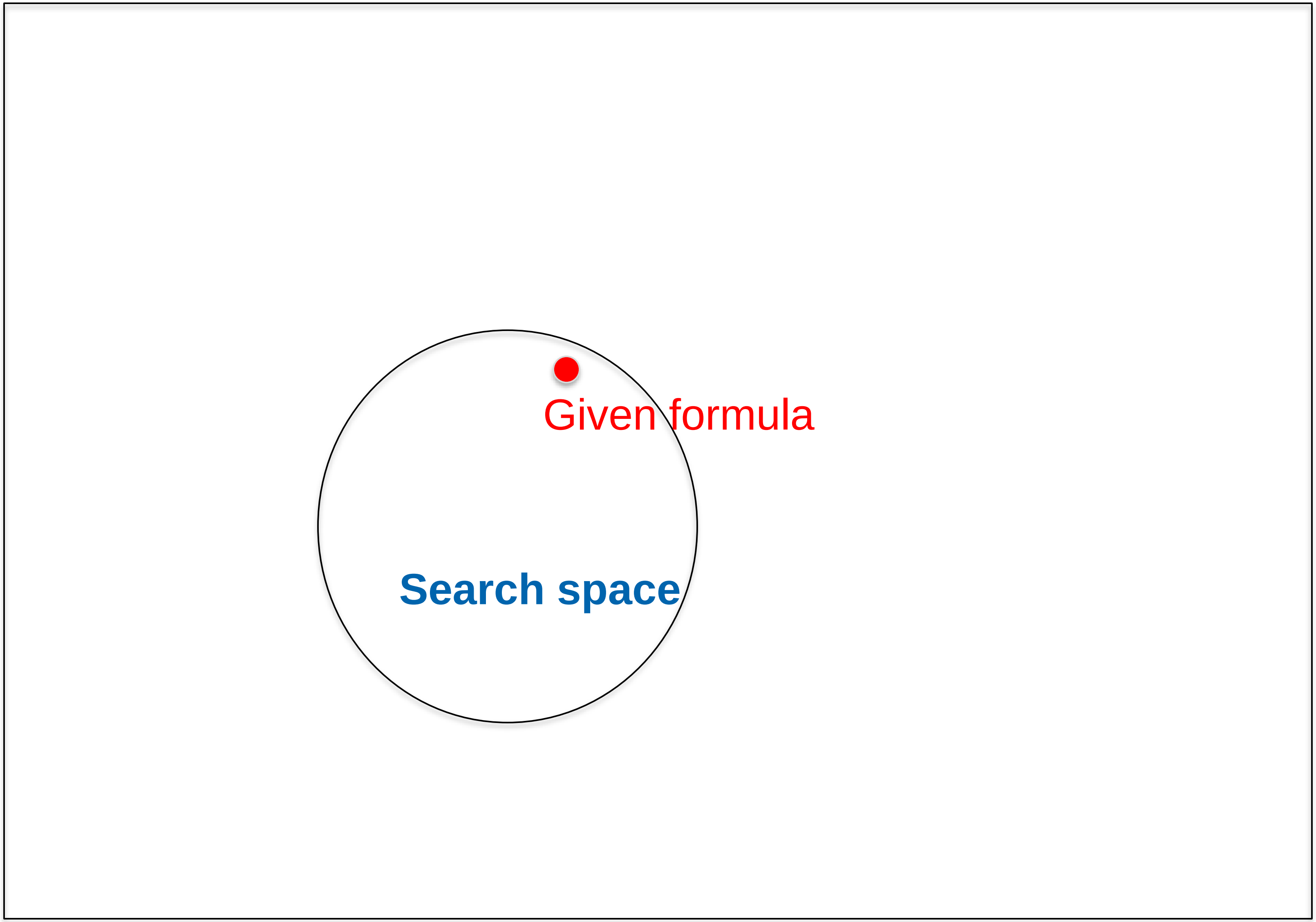
Given an input problem with assumptions $F_1, \ldots, F_n$ and goal G:

1. Negate the conjecture (¬G);
2. Establish unsatisfiability of the set of formulas $F_1, \ldots, F_n, \neg G$.

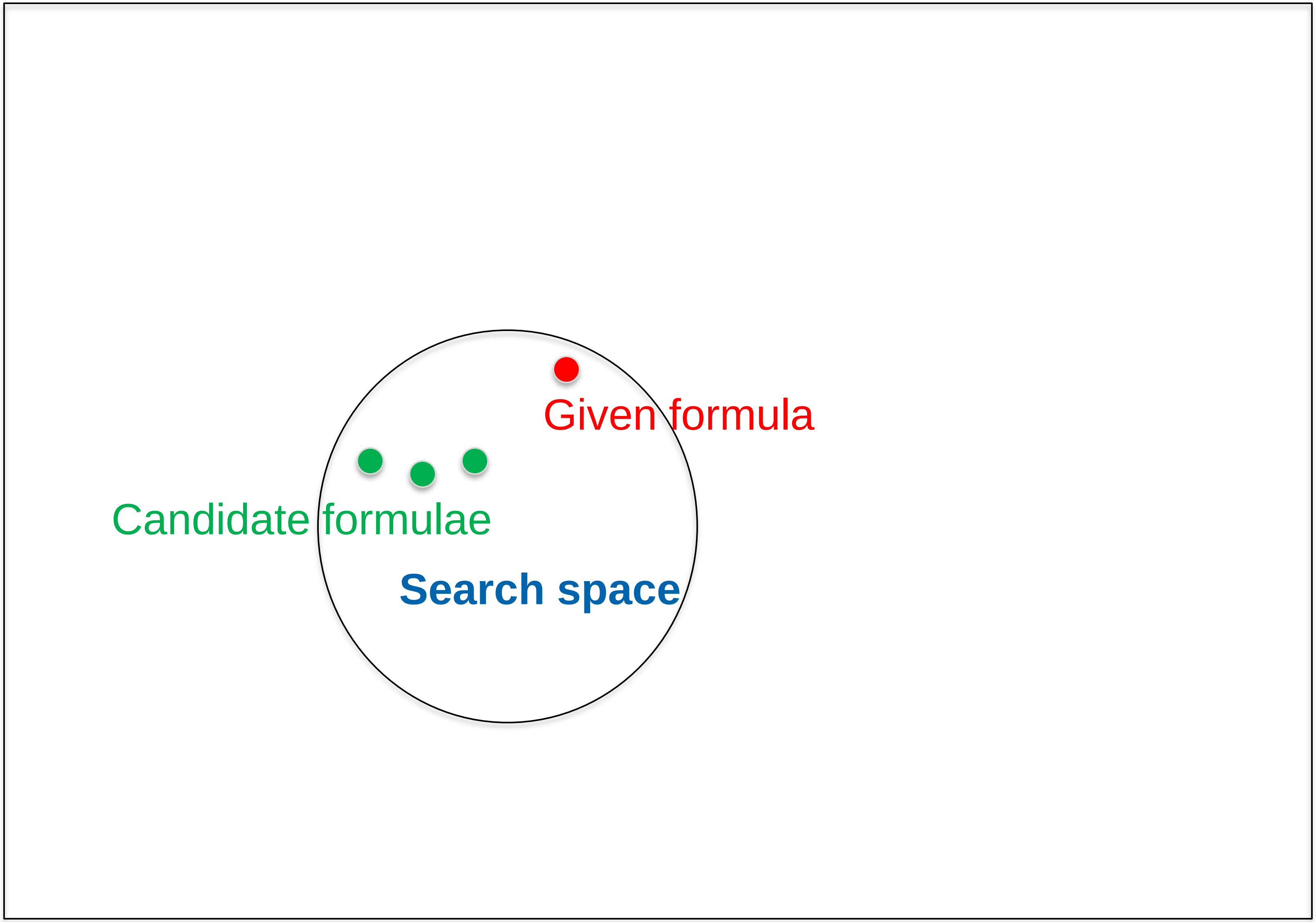
Automated Reasoning with VAMPIRE – Saturation



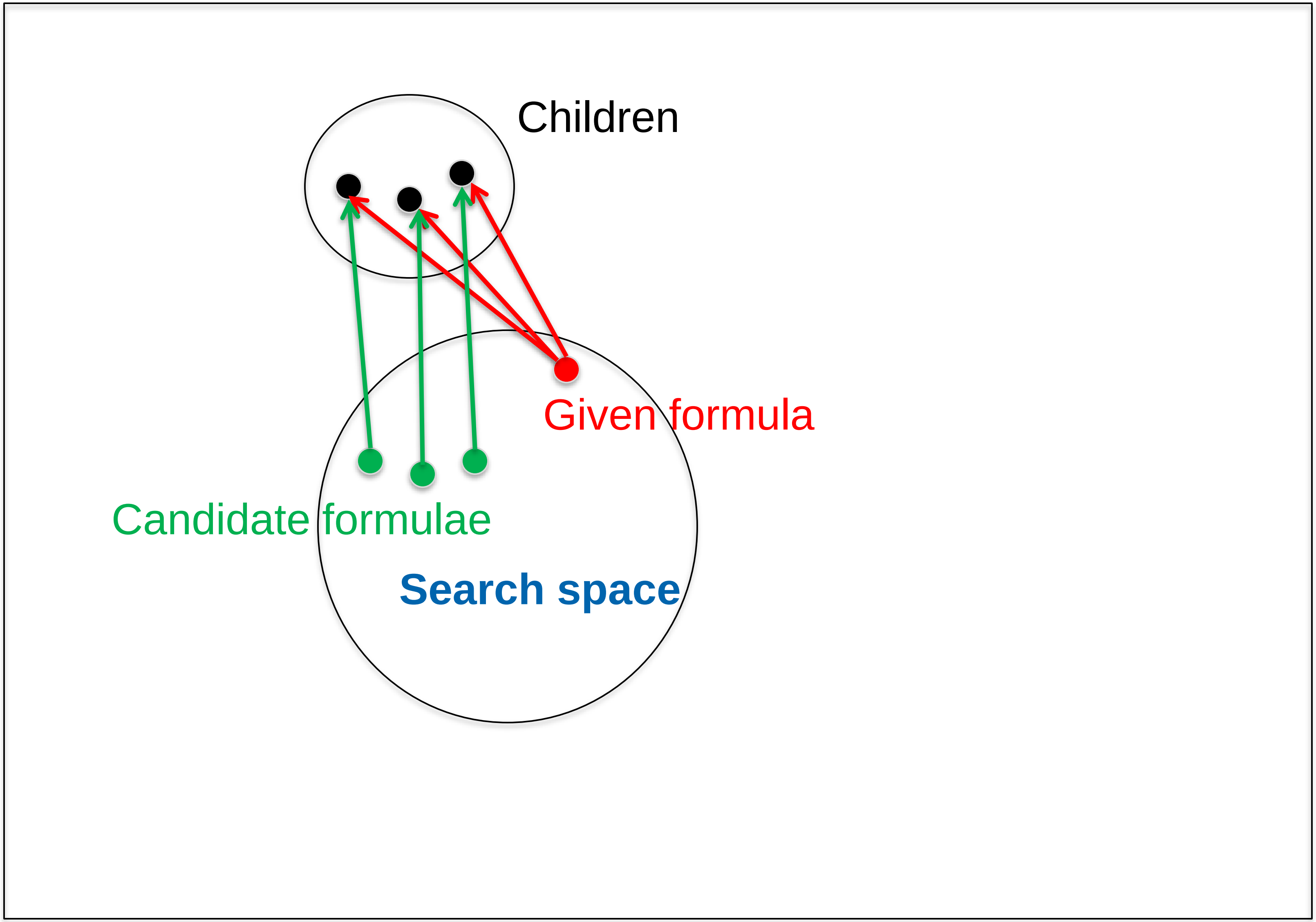
Automated Reasoning with VAMPIRE – Saturation



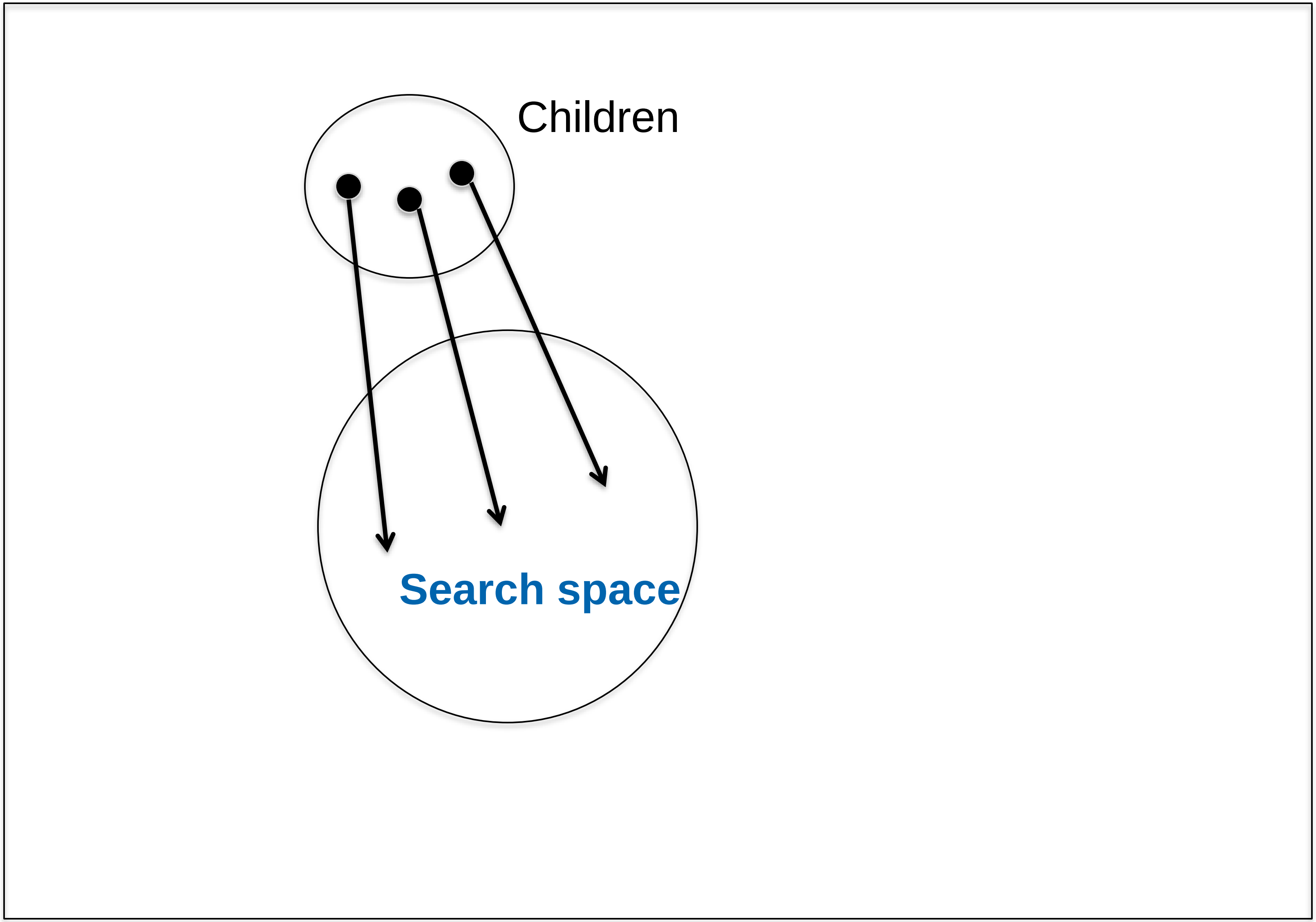
Automated Reasoning with VAMPIRE – Saturation



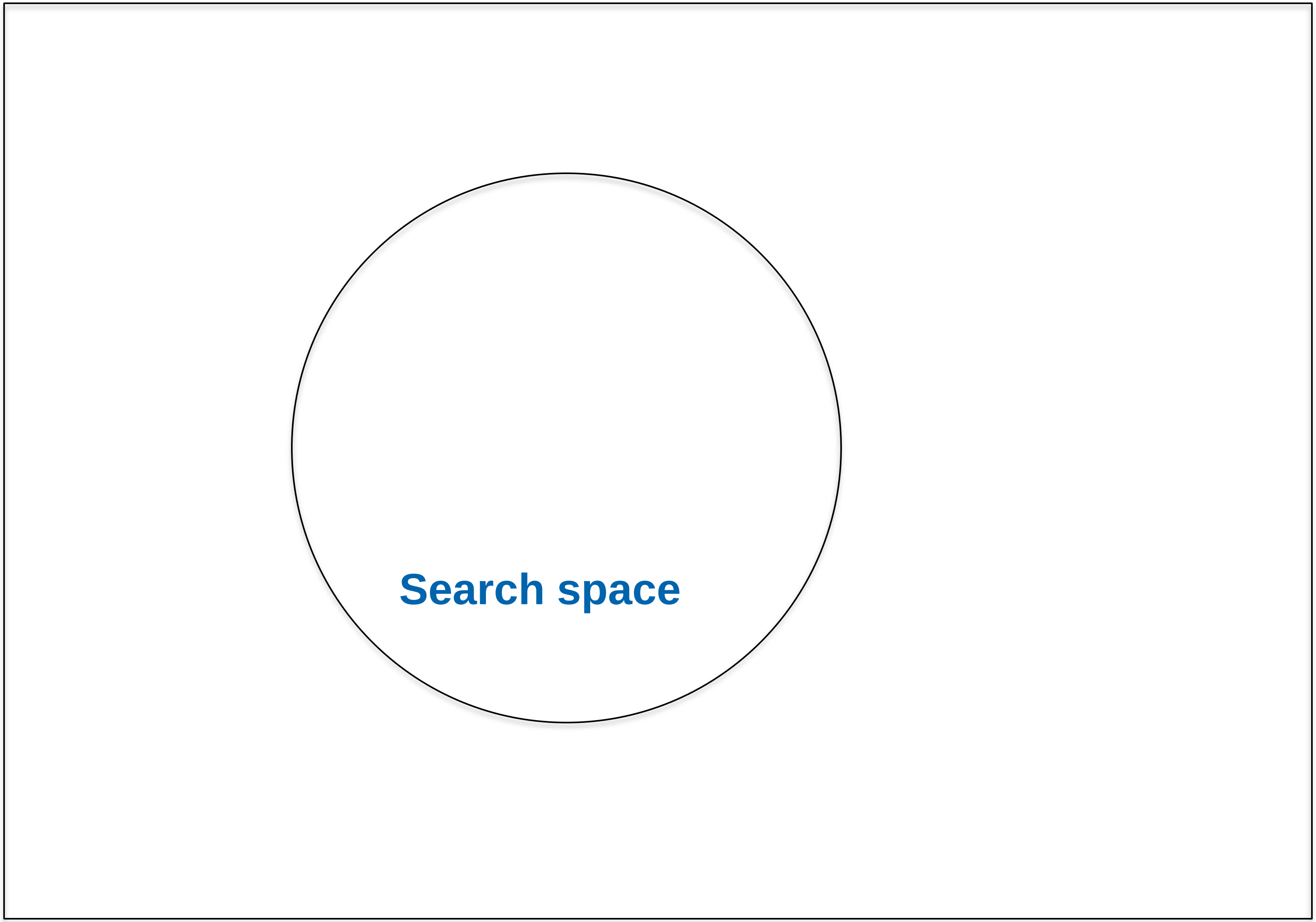
Automated Reasoning with VAMPIRE – Saturation



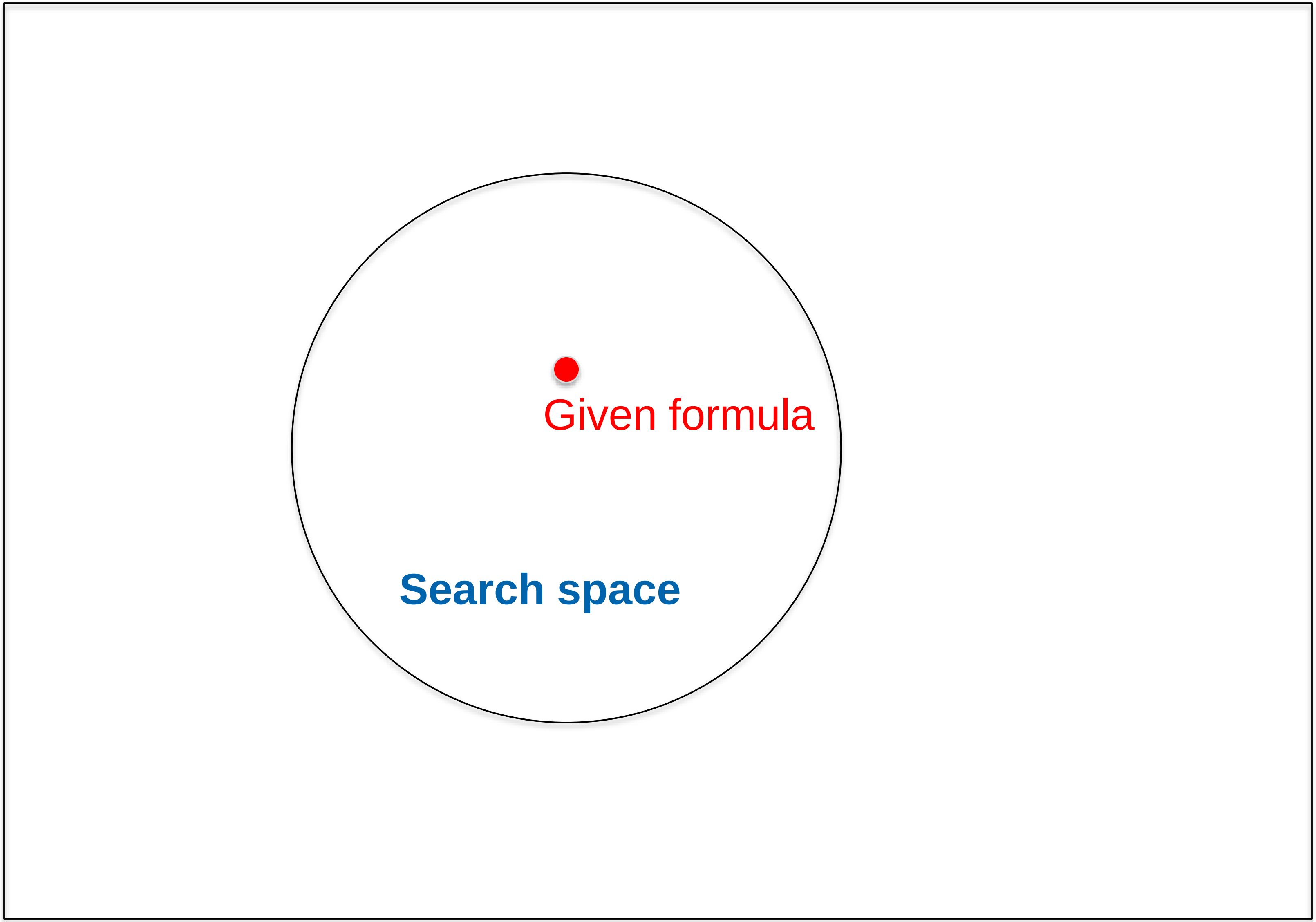
Automated Reasoning with VAMPIRE – Saturation



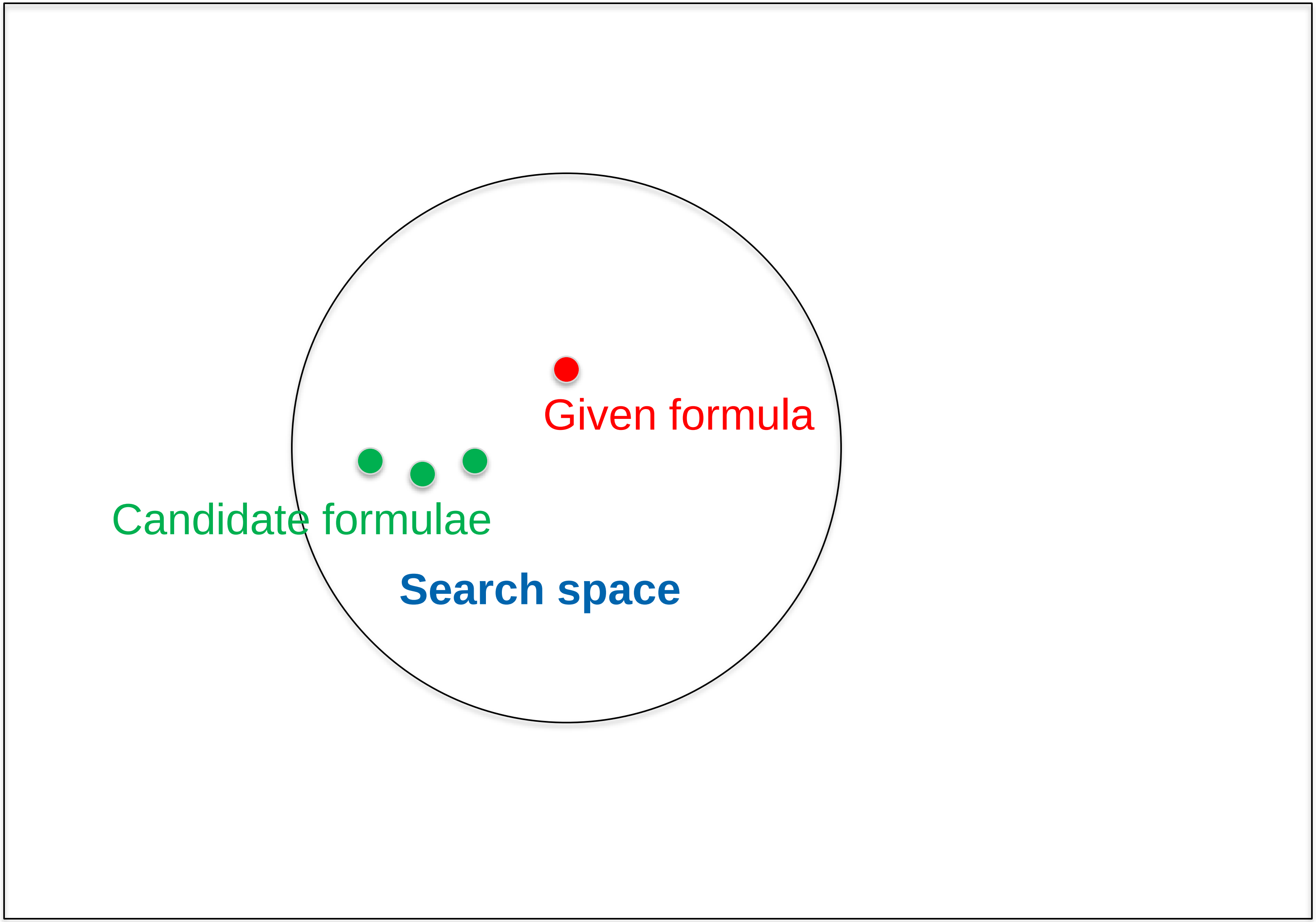
Automated Reasoning with VAMPIRE – Saturation



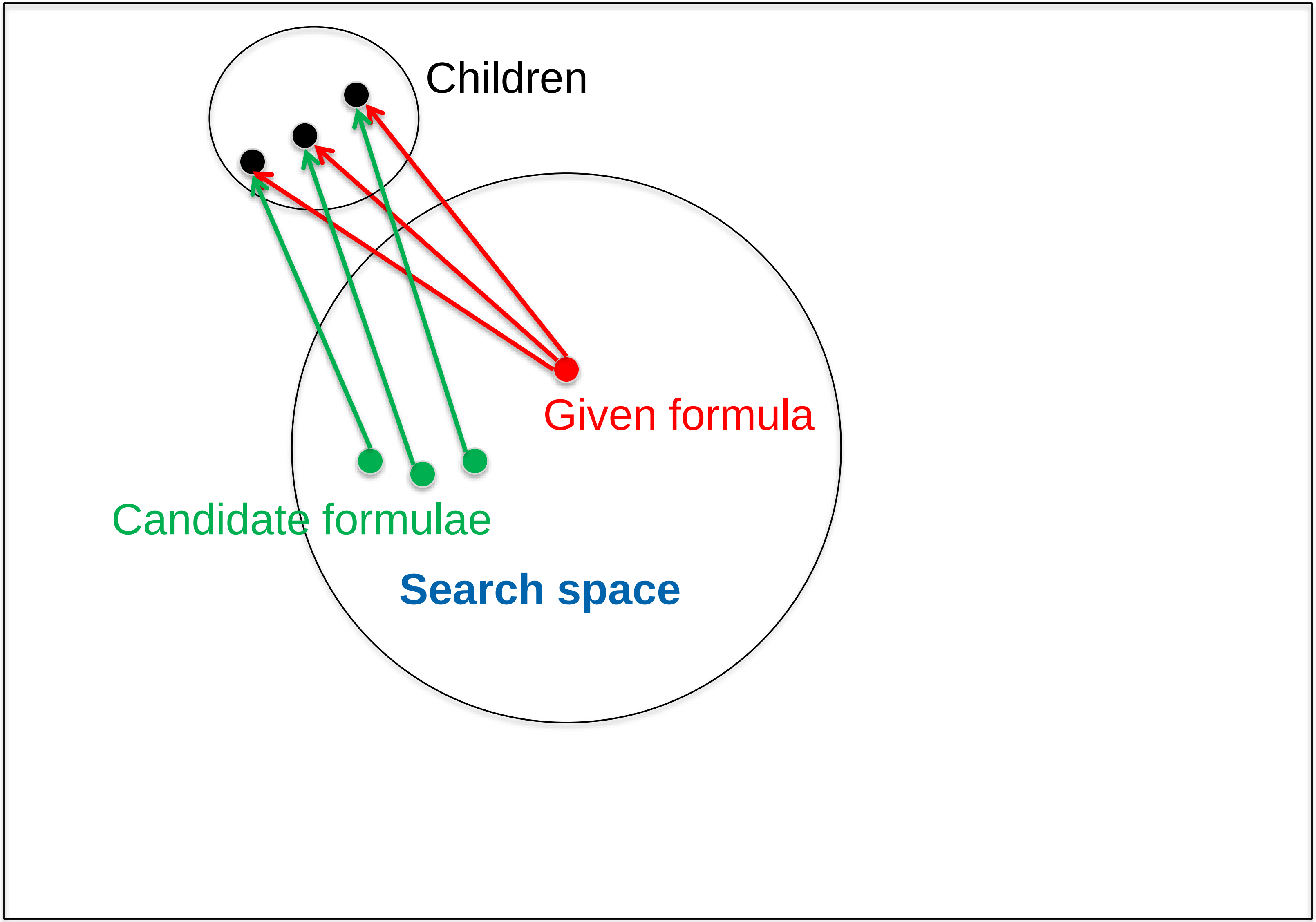
Automated Reasoning with VAMPIRE – Saturation



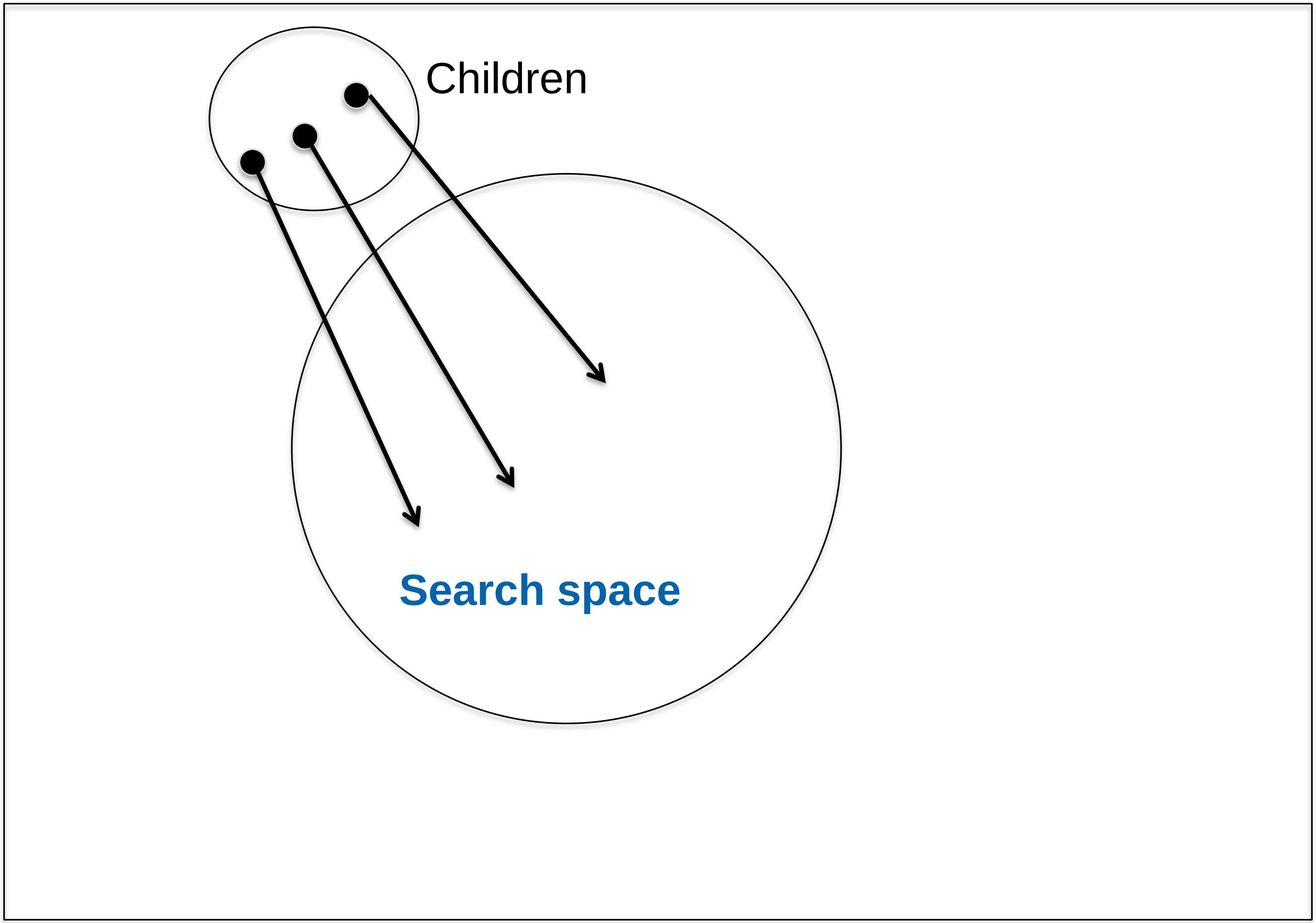
Automated Reasoning with VAMPIRE – Saturation



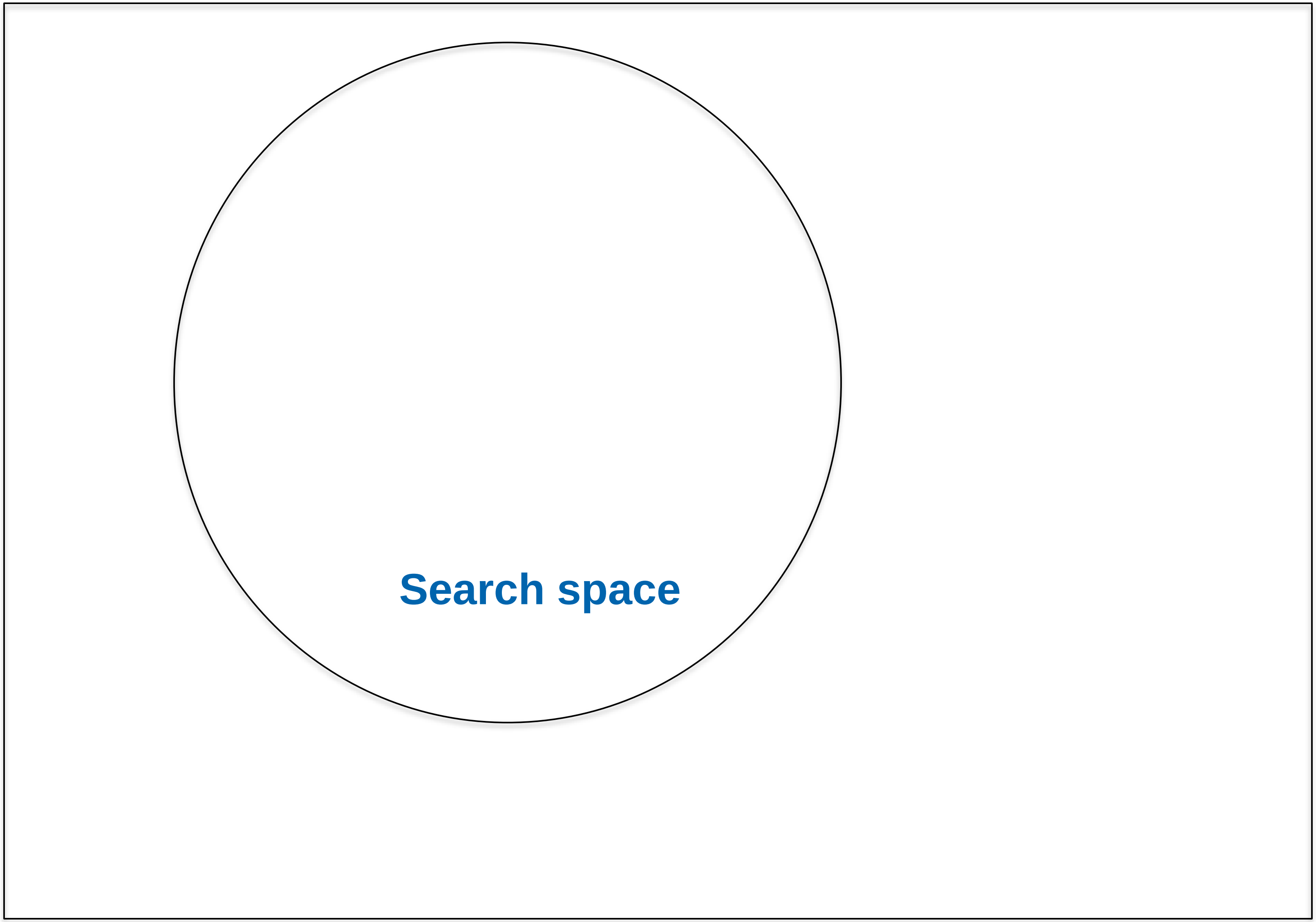
Automated Reasoning with VAMPIRE – Saturation



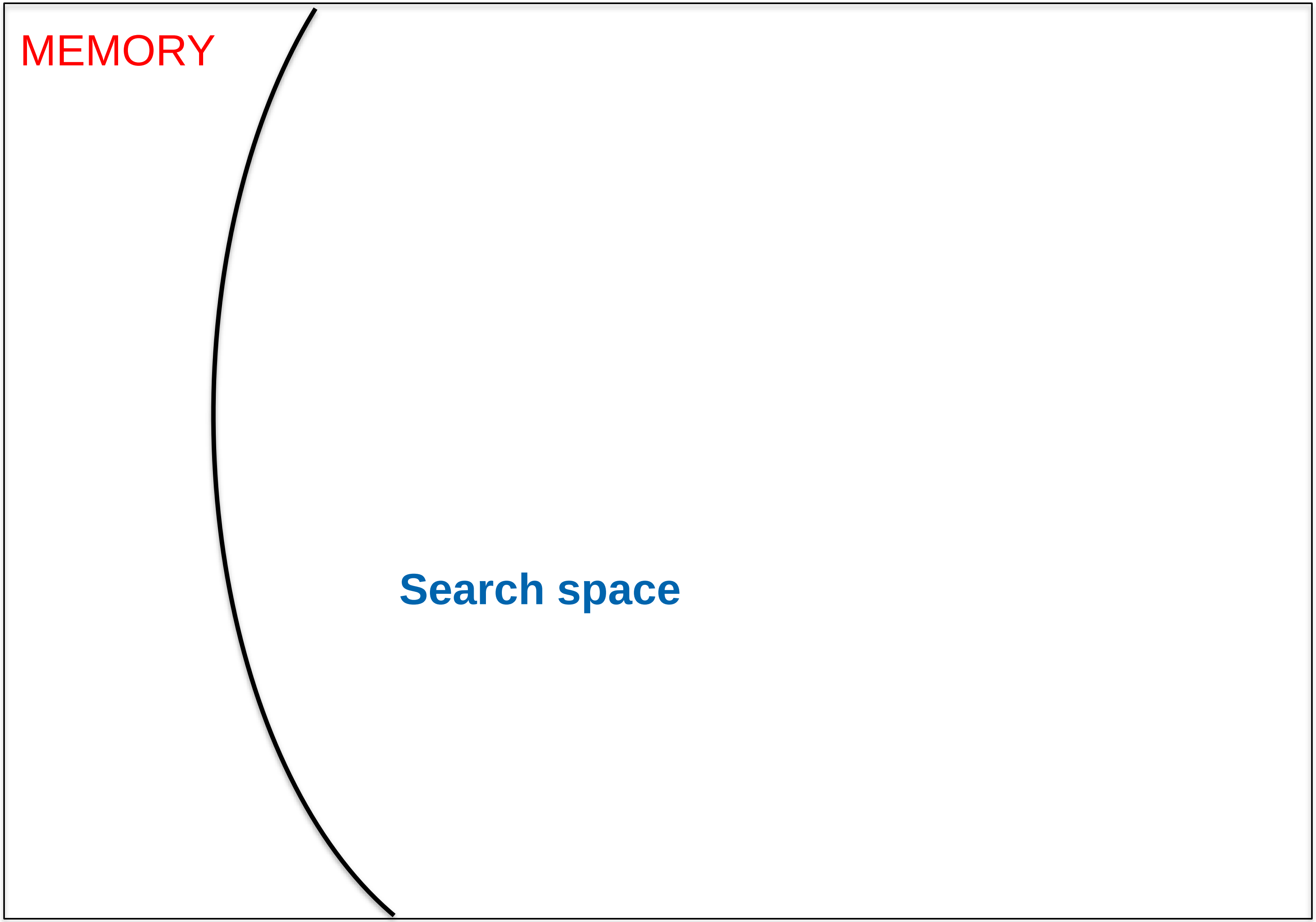
Automated Reasoning with VAMPIRE – Saturation



Automated Reasoning with VAMPIRE – Saturation



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.

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.

CHALLENGE: How to solve unknown?

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.

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

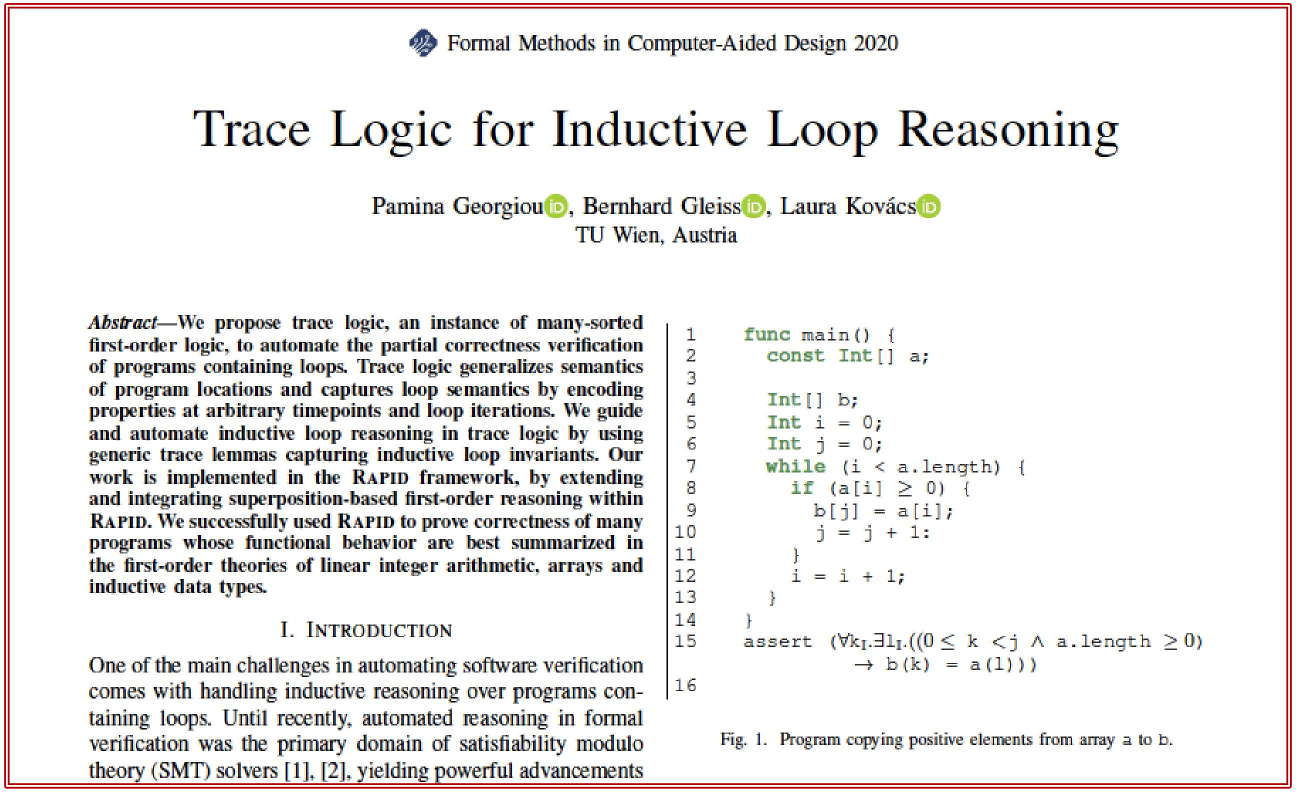
- **First-order theories** of data structures

- **Induction**

Automated Reasoning for Security, Privacy, Safety, …

New Reasoning Challenges in Systems Engineering

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



Formal Methods in Computer-Aided Design 2020

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

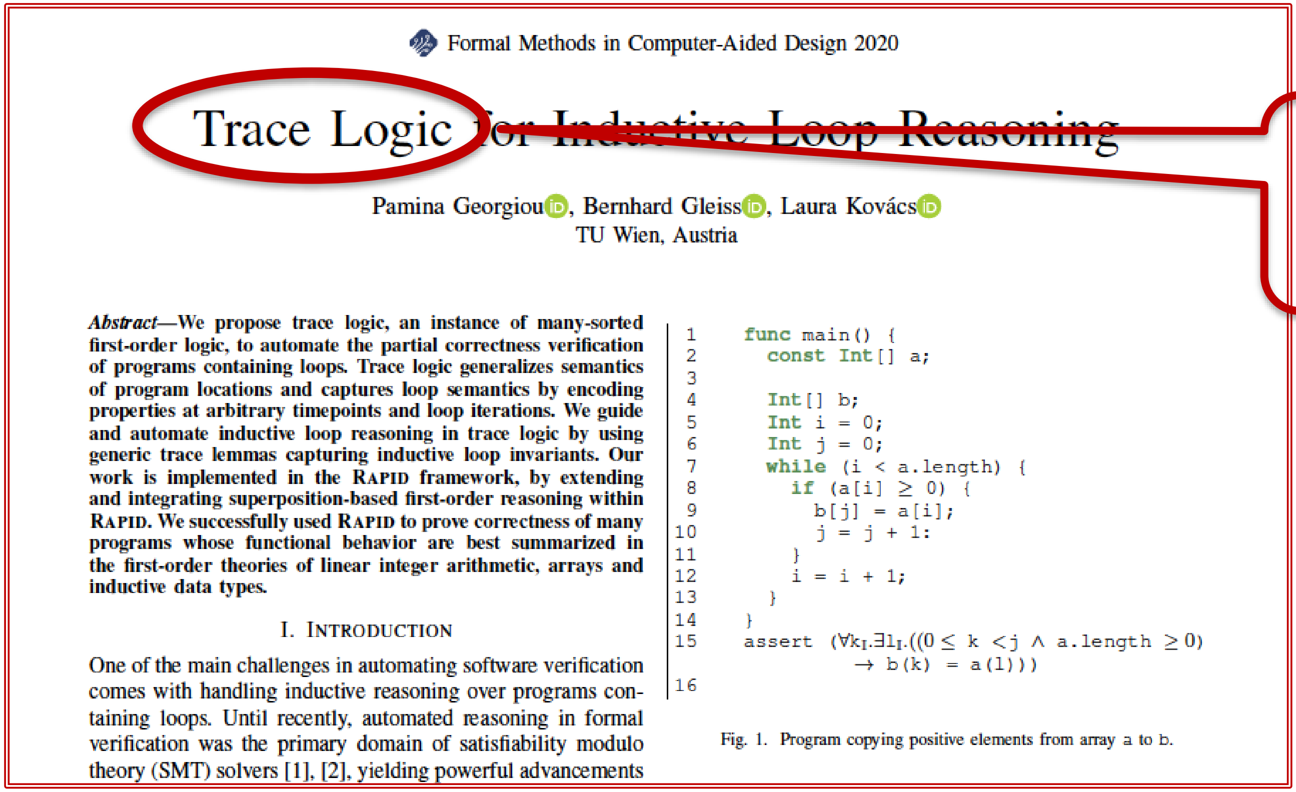
```
func main() {
  const Int[] a;

  Int[] b;
  Int i = 0;
  Int j = 0;
  while (i < a.length) {
    if (a[i] ≥ 0) {
      b[j] = a[i];
      j = j + 1;
    }
    i = i + 1;
  }
}
assert (∀k_I.∃l_I.((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



Formal Methods in Computer-Aided Design 2020

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

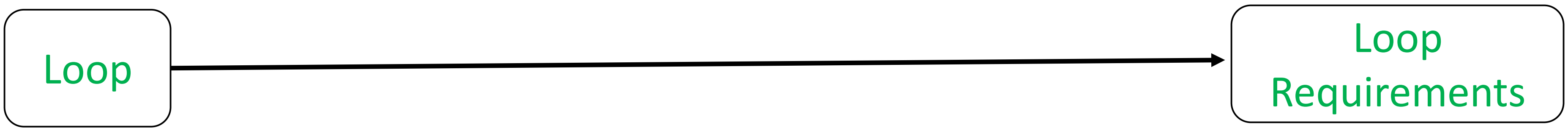
```
func main() {
  const Int[] a;

  Int[] b;
  Int i = 0;
  Int j = 0;
  while (i < a.length) {
    if (a[i] ≥ 0) {
      b[j] = a[i];
      j = j + 1;
    }
    i = i + 1;
  }
}
assert (∀k_I.∃l_I.((0 ≤ k <j ∧ a.length ≥0)
          → b(k) = a(l)))

```

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

Program variables as functions capturing all variables values throughout the loop

Trace Logic for Automated Loop Reasoning

Loop Language → First-Order Language

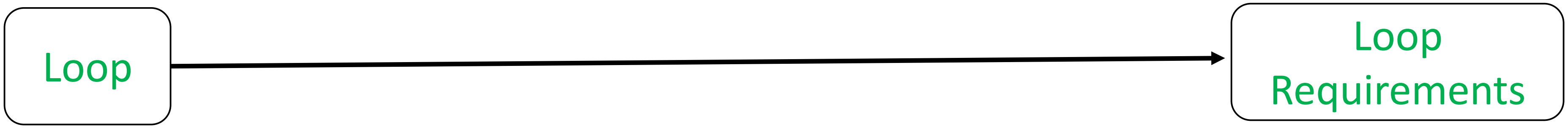


Loop

Loop Requirements

Trace Logic for Automated Loop Reasoning

Extended Loop Language → First-Order Language Extended with Extra Symbols

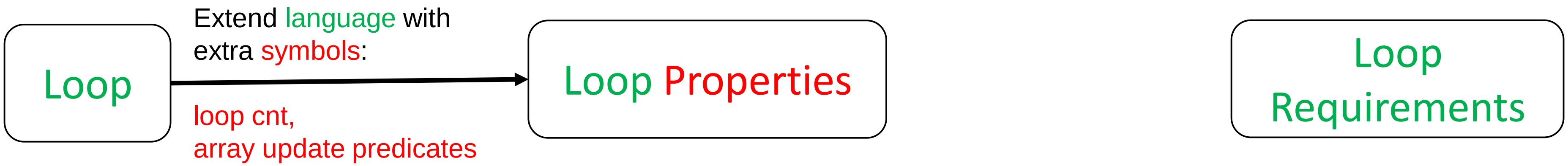


Loop

Loop Requirements

Trace Logic for Automated Loop Reasoning

Extended Loop Language → First-Order Language Extended with Extra Symbols



Loop

Extend language with extra symbols:

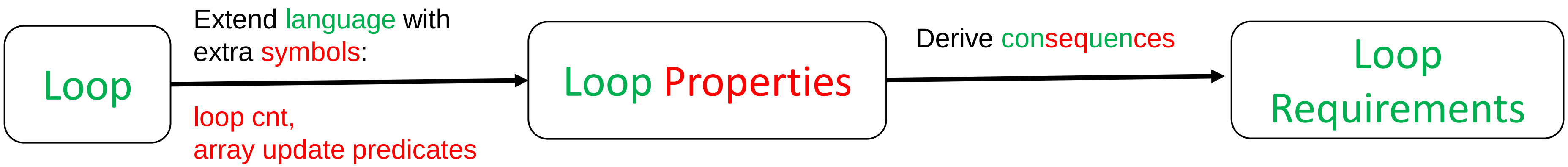
loop cnt,
array update predicates

Loop Properties

Loop Requirements

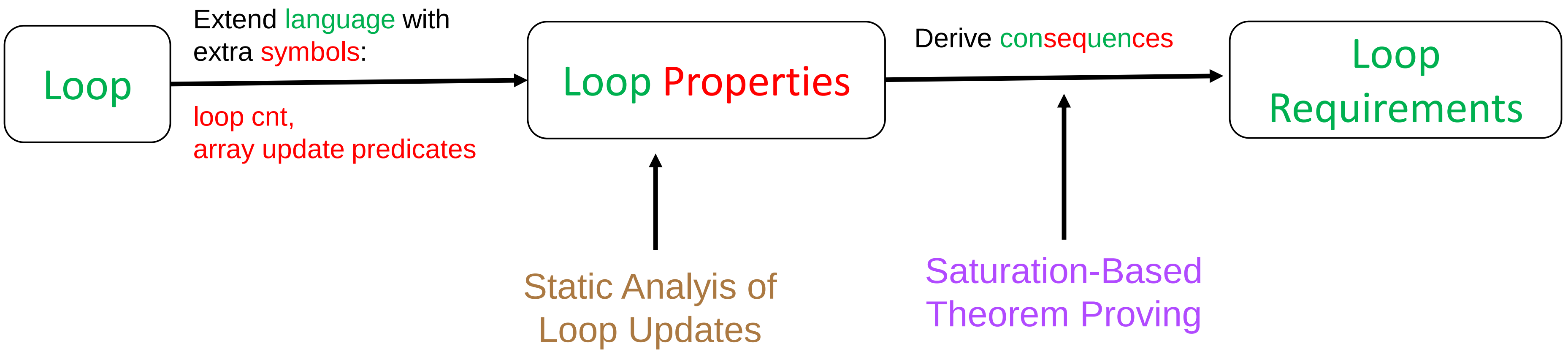
Trace Logic for Automated Loop Reasoning

Extended Loop Language → First-Order Language Extended with Extra Symbols



Trace Logic for Automated Loop Reasoning

Extended Loop Language → **Trace Logic**



Loop

Extend language with extra symbols:

loop cnt,
array update predicates

Loop Properties

Derive consequences

Loop Requirements

Static Analyis of
Loop Updates

Saturation-Based
Theorem Proving

Trace Lemmas

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]))
```

Trace Logic for Automated Loop Reasoning

Extra Symbols of Trace Logic

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

- loop counter cnt
- loop iteration predicate iter
- loop variable v as functions $v^{(i)}$ of iteration i
- array update properties $upd_V(i,p)$, $upd_V(i,p,x)$ for array variable V, loop iteration i, array position p, array element value x

Trace Logic for Automated Loop Reasoning

Loop Properties in Trace Logic

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

```
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 $(\forall p)\ (0 \le p < b \Rightarrow (\exists i)(0 \le i < n \wedge A[i]=B[p]))$

$(\forall i)\ (i \in iter \Leftrightarrow 0 \le i < cnt)$

$(\forall i)\ (i \in iter \Rightarrow a^{(i)} < a^{(i+1)})$

$(\forall i)\ (i \in iter \Rightarrow (b^{(i)} = b^{(i+1)} \vee b^{(i+1)} = b^{(i)}+1)$

$(\forall p)\ (0 \le p < b^{(cnt)} \Rightarrow ((\exists i) i \in iter \wedge p = b^{(i)} \wedge A[a^{(i)}]>0))$

$upd_B(i,p,x) \wedge (\forall j>i)(\neg upd_B(i,p,x) \Rightarrow B^{(cnt)}[p]=x)$

$$
\begin{aligned}
(\forall i)\ (i \in iter \wedge A[a^{(i)}]>0 \Rightarrow (&B^{(i+1)}[b^{(i)}]=A[a^{(i)}] \wedge \\
&b^{(i+1)}=b^{(i)}+1 \wedge \\
&c^{(i+1)}=c^{(i)}))
\end{aligned}
$$

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

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

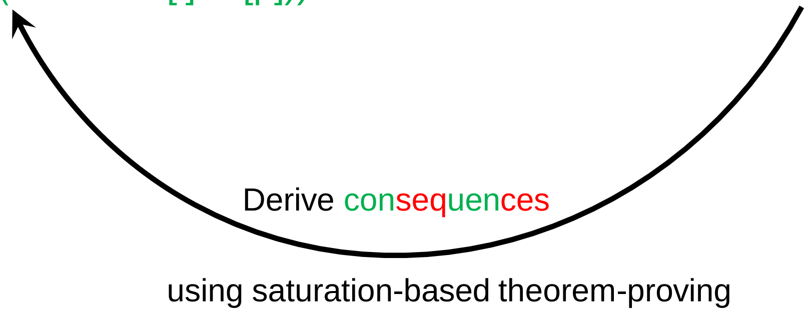
$(\forall i)\ (i\in iter \Rightarrow a^{(i)}<a^{(i+1)})$

$(\forall i)\ (i\in iter \Rightarrow (b^{(i)}=b^{(i+1)} \lor b^{(i+1)}=b^{(i)}+1)$

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

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

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



Derive consequences

using saturation-based theorem-proving

Trace Logic for Automated Loop Reasoning

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

```
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 $(\forall p)\ (0 \le p < b \Rightarrow (\exists i)(0 \le i < n \wedge A[i]=B[p]))$

Loop Properties in Trace Logic

$$(\forall i)\ (i \in iter \Leftrightarrow 0 \le i < cnt)$$

$$(\forall i)\ (i \in iter \Rightarrow a^{(i)} < a^{(i+1)})$$

$$(\forall i)\ (i \in iter \Rightarrow (b^{(i)} = b^{(i+1)} \vee b^{(i+1)} = b^{(i)}+1)$$

$$(\forall p)\ (0 \le p < b^{(cnt)} \Rightarrow ((\exists i) i \in iter \wedge p = b^{(i)} \wedge A[a^{(i)}] > 0))$$

$$upd_B(i,p,x) \wedge (\forall j>i)(\neg upd_B(i,p,x) \Rightarrow B^{(cnt)}[p]=x)$$

$$\begin{aligned}(\forall i)\ (i \in iter \wedge A[a^{(i)}]>0 \Rightarrow (&B^{(i+1)}[b^{(i)}] = A[a^{(i)}] \wedge \\ &b^{(i+1)} = b^{(i)}+1 \wedge \\ &c^{(i+1)} = c^{(i)}))\end{aligned}$$

Derive consequences

using saturation-based theorem-proving

$$\boxed{(\forall p)\ (0 \le p < b \Rightarrow (\exists i)(0 \le i < a \wedge A[i]=B[p]))}$$

Invariant

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]))
```

Loop Properties in Trace Logic

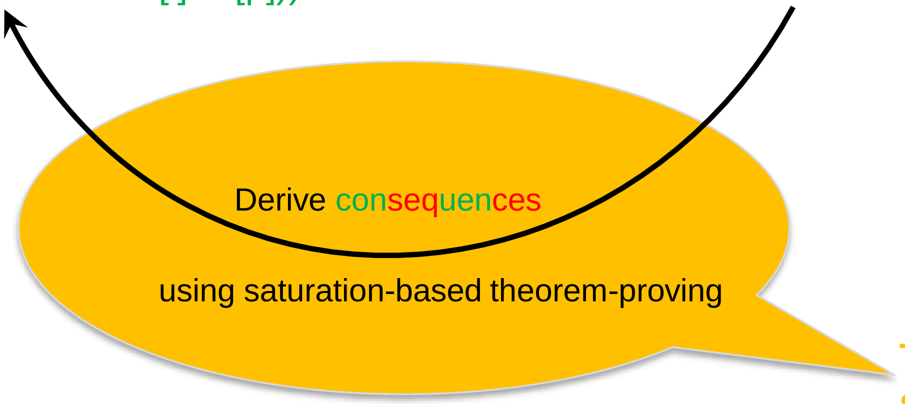
$$(\forall i)\ (i \in iter \Leftrightarrow 0 \leq i < cnt)$$

$$(\forall i)\ (i \in iter \Rightarrow a^{(i)} < a^{(i+1)})$$
$$(\forall i)\ (i \in iter \Rightarrow (b^{(i)} = b^{(i+1)} \vee b^{(i+1)} = b^{(i)} + 1)$$

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

$$upd_B(i,p,x) \wedge (\forall j > i)(\neg upd_B(i,p,x) \Rightarrow B^{(cnt)}[p] = x)$$

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



Derive consequences

using saturation-based theorem-proving

Tailored changes in saturation

Trace Logic for Automated Loop Reasoning

Deriving useful loop properties in saturation

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

$v^{(0)}=v_0$ and $v^{(cnt)}=v$

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

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

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

x+y=y+x is not useful

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

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

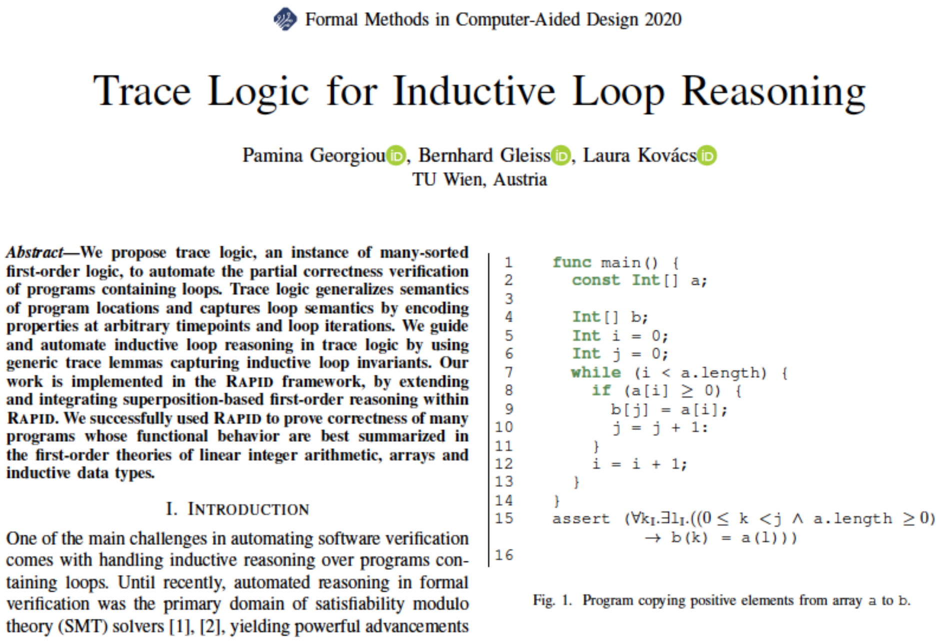
- useful clauses:
  - contain only useable symbols
  - contains at least a target symbols or a skolem functions

- 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*



Formal Methods in Computer-Aided Design 2020

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

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

  Int[] b;
  Int i = 0;
  Int j = 0;
  while (i < a.length) {
    if (a[i] ≥ 0) {
      b[j] = a[i];
      j = j + 1:
    }
    i = i + 1;
  }
}
assert (∀k_I.∃l_I.((0 ≤ k <j ∧ a.length ≥ 0)
           → b(k) = a(l)))

```

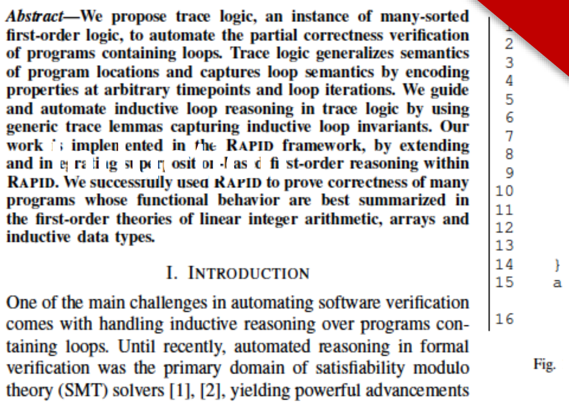
Fig. 1. Program copying positive elements from array a to b.

Automated Reaso

New Reasoning Ch

- Software semantics

Proved 24 unique problems

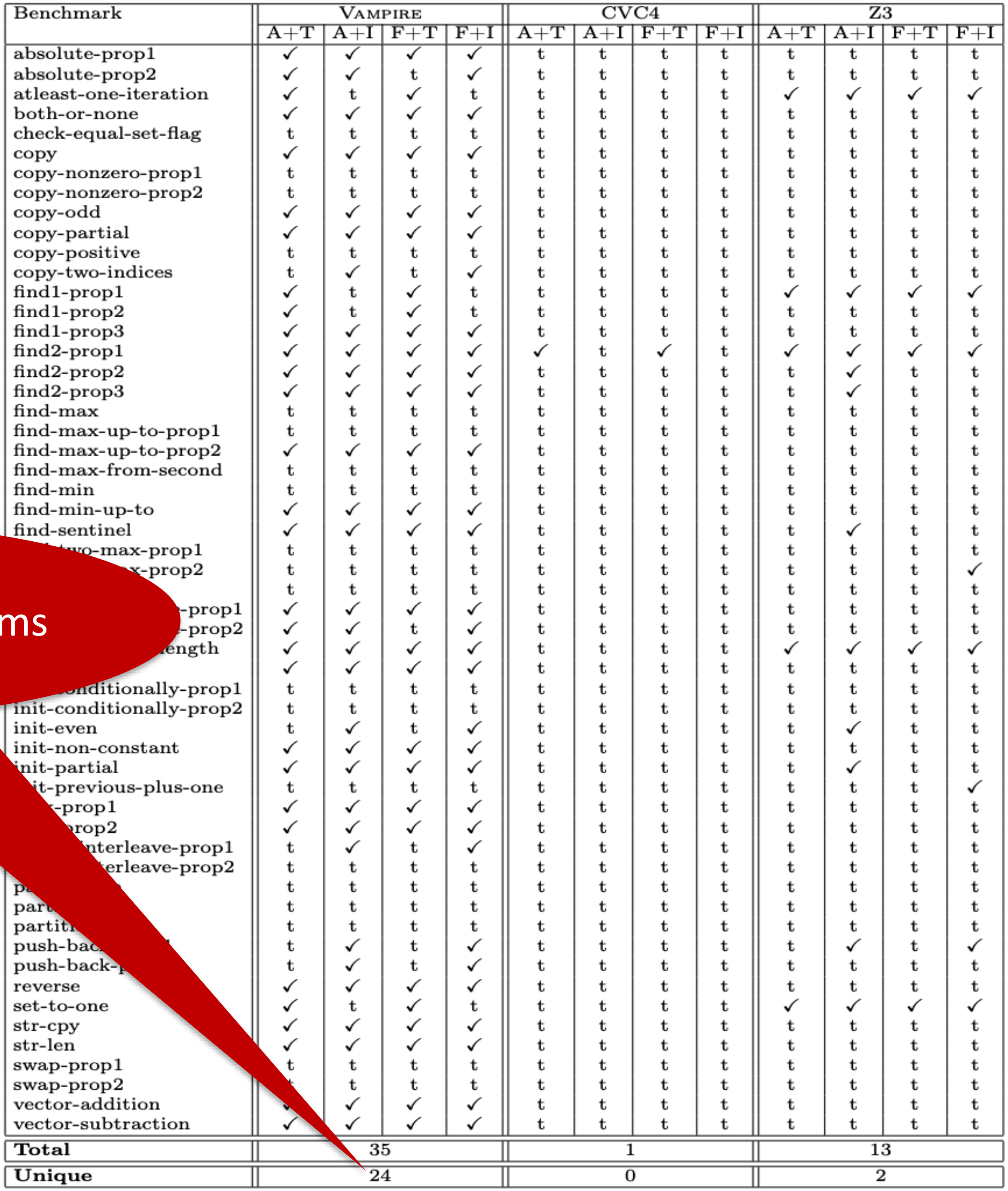


Abstract—We propose trace logic, an instance of many-sorted first-order logic, to automate the partial correctness verification of programs containing loops. Trace logic generalizes semantics of program locations and captures loop semantics by encoding properties at arbitrary timepoints and loop iterations. 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

Fig.



Benchmark	VAMPIRE				CVC4				Z3			
	A+T	A+I	F+T	F+I	A+T	A+I	F+T	F+I	A+T	A+I	F+T	F+I
absolute-prop1	✓	✓	✓	✓	t	t	t	t	t	t	t	t
absolute-prop2	✓	✓	t	✓	t	t	t	t	t	t	t	t
atleast-one-iteration	✓	t	✓	t	t	t	t	t	✓	✓	✓	✓
both-or-none	✓	✓	✓	✓	t	t	t	t	t	t	t	t
check-equal-set-flag	t	t	t	t	t	t	t	t	t	t	t	t
copy	✓	✓	✓	✓	t	t	t	t	t	t	t	t
copy-nonzero-prop1	t	t	t	t	t	t	t	t	t	t	t	t
copy-nonzero-prop2	t	t	t	t	t	t	t	t	t	t	t	t
copy-odd	✓	✓	✓	✓	t	t	t	t	t	t	t	t
copy-partial	✓	✓	✓	✓	t	t	t	t	t	t	t	t
copy-positive	t	t	t	t	t	t	t	t	t	t	t	t
copy-two-indices	t	✓	t	✓	t	t	t	t	t	t	t	t
find1-prop1	✓	t	✓	t	t	t	t	t	✓	✓	✓	✓
find1-prop2	✓	t	✓	t	t	t	t	t	t	t	t	t
find1-prop3	✓	✓	✓	✓	t	t	t	t	t	t	t	t
find2-prop1	✓	✓	✓	✓	✓	t	✓	t	✓	✓	✓	✓
find2-prop2	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
find2-prop3	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
find-max	t	t	t	t	t	t	t	t	t	t	t	t
find-max-up-to-prop1	t	t	t	t	t	t	t	t	t	t	t	t
find-max-up-to-prop2	✓	✓	✓	✓	t	t	t	t	t	t	t	t
find-max-from-second	t	t	t	t	t	t	t	t	t	t	t	t
find-min	t	t	t	t	t	t	t	t	t	t	t	t
find-min-up-to	✓	✓	✓	✓	t	t	t	t	t	t	t	t
find-sentinel	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
wo-max-prop1	t	t	t	t	t	t	t	t	t	t	t	t
x-prop2	t	t	t	t	t	t	t	t	t	t	t	✓
	t	t	t	t	t	t	t	t	t	t	t	t
-prop1	✓	✓	✓	✓	t	t	t	t	t	t	t	t
-prop2	✓	✓	t	✓	t	t	t	t	t	t	t	t
ength	✓	✓	✓	✓	t	t	t	t	✓	✓	✓	✓
	✓	✓	✓	✓	t	t	t	t	t	t	t	t
nditionally-prop1	t	t	t	t	t	t	t	t	t	t	t	t
init-conditionally-prop2	t	t	t	t	t	t	t	t	t	t	t	t
init-even	t	✓	t	✓	t	t	t	t	t	✓	t	t
init-non-constant	✓	✓	✓	✓	t	t	t	t	t	t	t	t
init-partial	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
it-previous-plus-one	t	t	t	t	t	t	t	t	t	t	t	✓
-prop1	✓	✓	✓	✓	t	t	t	t	t	t	t	t
rop2	✓	✓	✓	✓	t	t	t	t	t	t	t	t
nterleave-prop1	t	✓	t	✓	t	t	t	t	t	t	t	t
erleave-prop2	t	t	t	t	t	t	t	t	t	t	t	t
p	t	t	t	t	t	t	t	t	t	t	t	t
part	t	t	t	t	t	t	t	t	t	t	t	t
partiti	t	t	t	t	t	t	t	t	t	t	t	t
push-bac	t	✓	t	✓	t	t	t	t	t	✓	t	✓
push-back-	t	✓	t	✓	t	t	t	t	t	t	t	t
reverse	✓	✓	✓	✓	t	t	t	t	t	t	t	t
set-to-one	✓	t	✓	t	t	t	t	t	✓	✓	✓	✓
str-cpy	✓	✓	✓	✓	t	t	t	t	t	t	t	t
str-len	✓	✓	✓	✓	t	t	t	t	t	t	t	t
swap-prop1	t	t	t	t	t	t	t	t	t	t	t	t
swap-prop2	t	t	t	t	t	t	t	t	t	t	t	t
vector-addition	✓	✓	✓	✓	t	t	t	t	t	t	t	t
vector-subtraction	✓	✓	✓	✓	t	t	t	t	t	t	t	t
Total	35				1				13			
Unique	24				0				2			

Automated Reasoning **for Security, Privacy, Safety, …**

New Reasoning Challenges ***in Systems Engineering***

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

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

New Reasoning Challenges in Systems Engineering

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

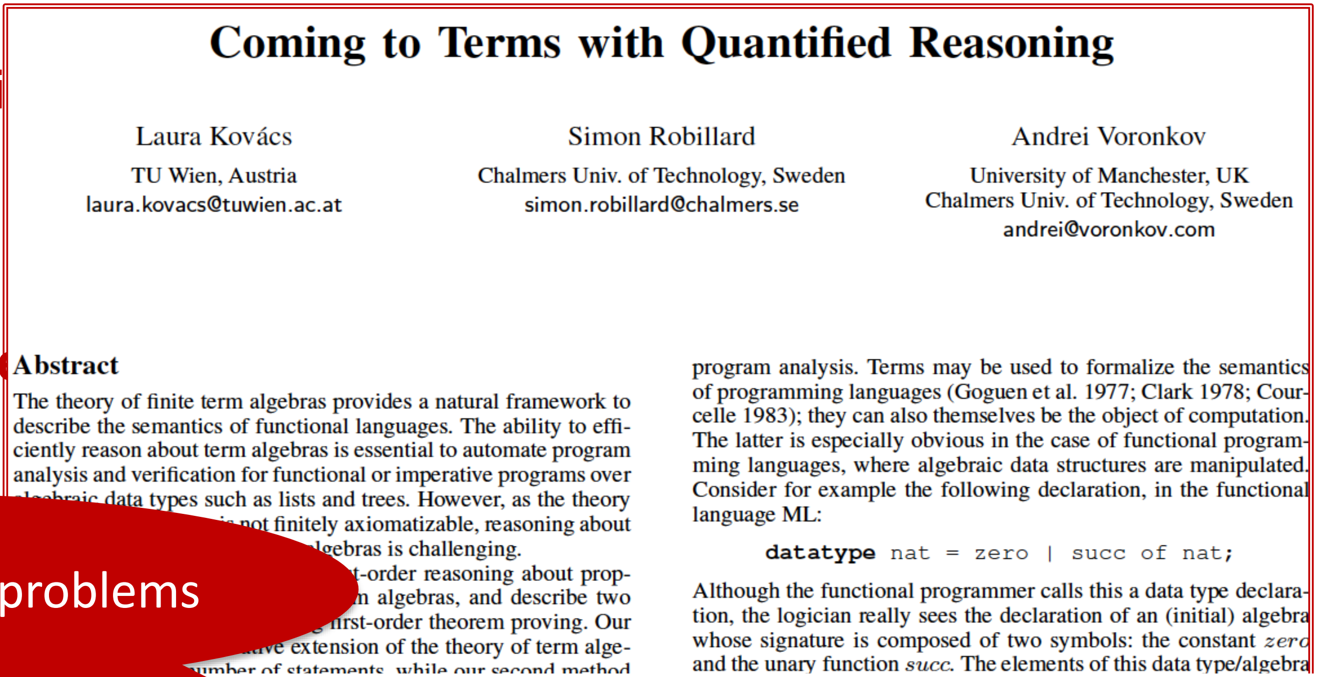
- **First-order theories** of data structures
  - Term algebras: subterm predicate for finite axiomatisations

Automated Reasoning for Security, Privacy, Safety, …

New Reasoning Challenges in Systems Engineering

- Software semanti

- First-order theori
  - Term algebras:



Coming to Terms with Quantified Reasoning

Laura Kovács
TU Wien, Austria
laura.kovacs@tuwien.ac.at

Simon Robillard
Chalmers Univ. of Technology, Sweden
simon.robillard@chalmers.se

Andrei Voronkov
University of Manchester, UK
Chalmers Univ. of Technology, Sweden
andrei@voronkov.com

Abstract

The theory of finite term algebras provides a natural framework to describe the semantics of functional languages. The ability to efficiently reason about term algebras is essential to automate program analysis and verification for functional or imperative programs over algebraic data types such as lists and trees. However, as the theory [illegible] is not finitely axiomatizable, reasoning about [illegible] algebras is challenging.

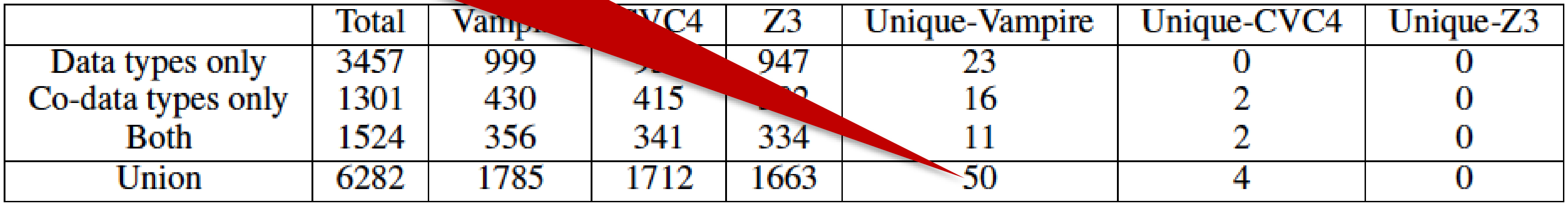
[illegible] -order reasoning about prop- [illegible] m algebras, and describe two [illegible] first-order theorem proving. Our [illegible] ive extension of the theory of term alge- [illegible] number of statements, while our second method

program analysis. Terms may be used to formalize the semantics of programming languages (Goguen et al. 1977; Clark 1978; Courcelle 1983); they can also themselves be the object of computation. The latter is especially obvious in the case of functional programming languages, where algebraic data structures are manipulated. Consider for example the following declaration, in the functional language ML:

```
datatype nat = zero | succ of nat;
```

Although the functional programmer calls this a data type declaration, the logician really sees the declaration of an (initial) algebra whose signature is composed of two symbols: the constant *zero* and the unary function *succ*. The elements of this data type/algebra

Proved 50 unique problems



	Total	Vampire	CVC4	Z3	Unique-Vampire	Unique-CVC4	Unique-Z3
Data types only	3457	999	[illegible]	947	23	0	0
Co-data types only	1301	430	415	[illegible]	16	2	0
Both	1524	356	341	334	11	2	0
Union	6282	1785	1712	1663	50	4	0

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

Bridiging 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*

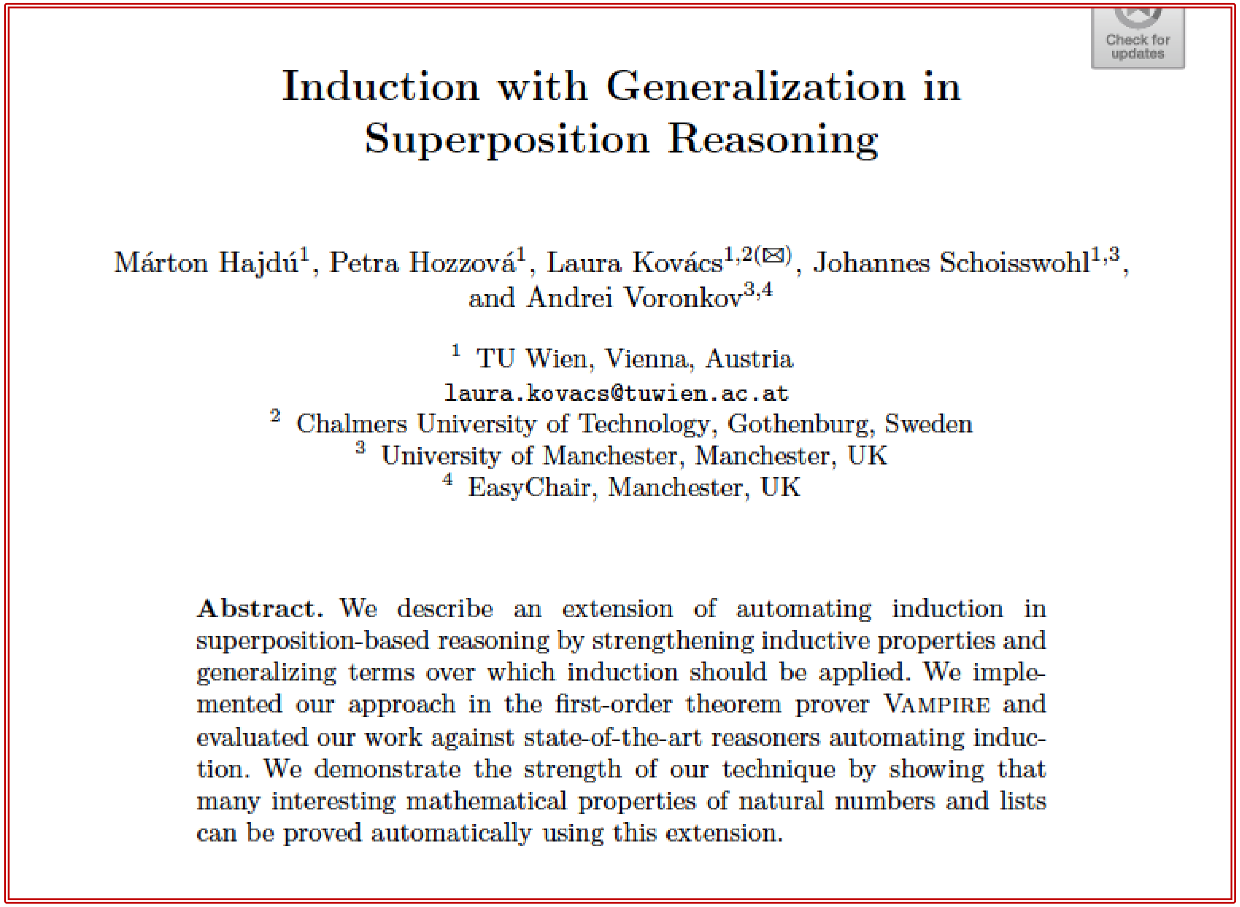
- **First-order theories** of data structures
  reasoning with quantifiers + int/real, naturals/term algebras, arrays, …

- **Induction**

Automated Reasoning for Security, Privacy, Safety, …

New Reasoning Challenges in Systems Engineering

- **Induction**



Check for updates

Induction with Generalization in Superposition Reasoning

Márton Hajdú[1], Petra Hozzová[1], Laura Kovács[1,2](✉), Johannes Schoisswohl[1,3], and Andrei Voronkov[3,4]

[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

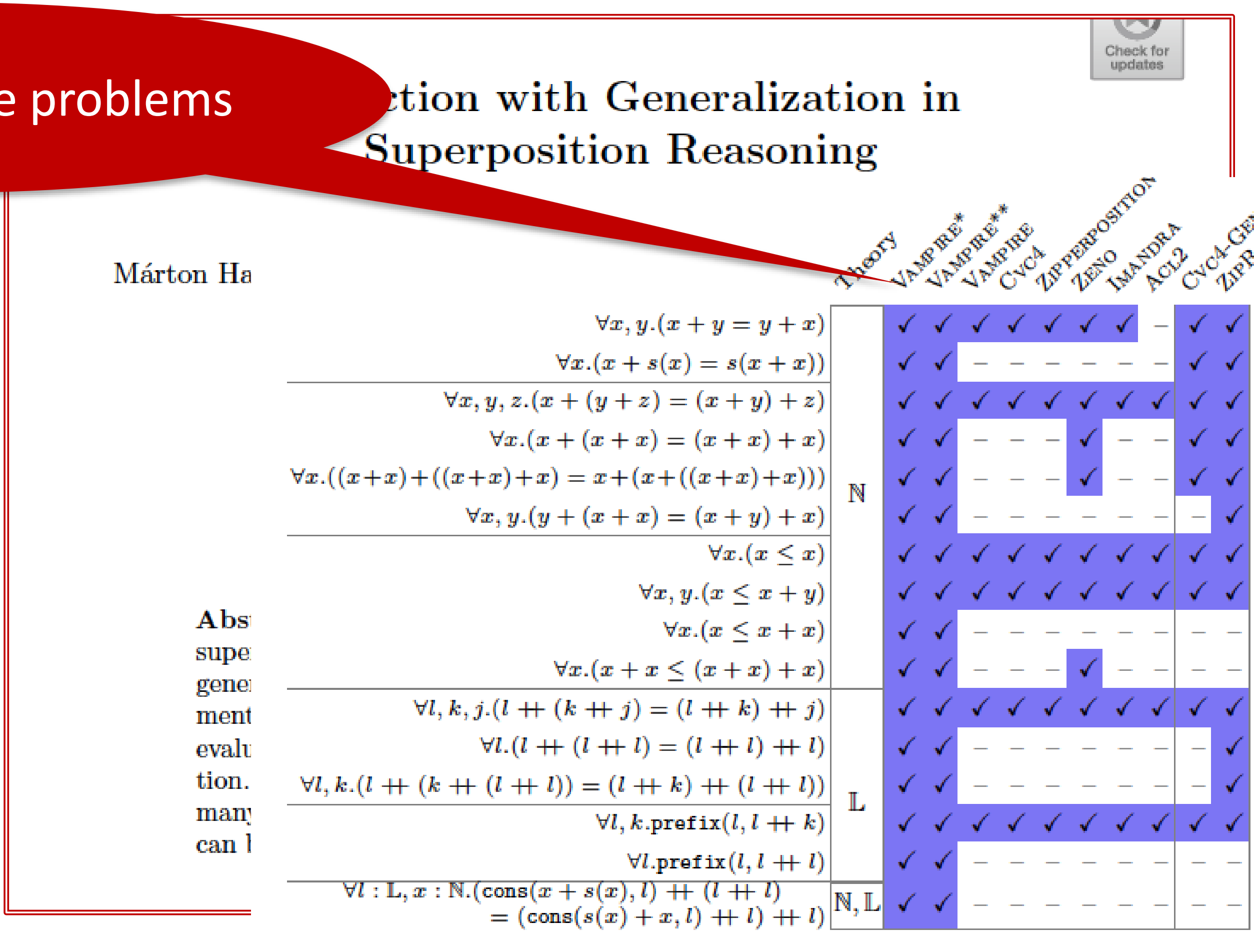
Abstract. We describe an extension of automating induction in superposition-based reasoning by strengthening inductive properties and generalizing terms over which induction should be applied. We implemented our approach in the first-order theorem prover Vampire and evaluated our work against state-of-the-art 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



Check for updates

…ction with Generalization in Superposition Reasoning

Márton Ha

Abs supe gene ment evalu tion. many can l

	Theory	VAMPIRE*	VAMPIRE**	VAMPIRE	CVC4	ZIPPERPOSITION	ZENO	IMANDRA	ACL2	CVC4-GEN	ZIPREWRITE
$\forall x, y.(x + y = y + x)$	$\mathbb{N}$	✓	✓	✓	✓	✓	✓	✓	–	✓	✓
$\forall x.(x + s(x) = s(x + x))$		✓	✓	–	–	–	–	–	–	✓	✓
$\forall x, y, z.(x + (y + z) = (x + y) + z)$		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall x.(x + (x + x) = (x + x) + x)$		✓	✓	–	–	–	✓	–	–	✓	✓
$\forall x.((x+x)+((x+x)+x) = x+(x+((x+x)+x)))$		✓	✓	–	–	–	✓	–	–	✓	✓
$\forall x, y.(y + (x + x) = (x + y) + x)$		✓	✓	–	–	–	–	–	–	–	✓
$\forall x.(x \leq x)$		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall x, y.(x \leq x + y)$		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall x.(x \leq x + x)$		✓	✓	–	–	–	–	–	–	–	–
$\forall x.(x + x \leq (x + x) + x)$		✓	✓	–	–	–	✓	–	–	–	–
$\forall l, k, j.(l ++ (k ++ j) = (l ++ k) ++ j)$	$\mathbb{L}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall l.(l ++ (l ++ l) = (l ++ l) ++ l)$		✓	✓	–	–	–	–	–	–	–	✓
$\forall l, k.(l ++ (k ++ (l ++ l)) = (l ++ k) ++ (l ++ l))$		✓	✓	–	–	–	–	–	–	–	✓
$\forall l, k.\mathtt{prefix}(l, l ++ k)$		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall l.\mathtt{prefix}(l, l ++ l)$		✓	✓	–	–	–	–	–	–	–	–
$\forall l : \mathbb{L}, x : \mathbb{N}.(\mathtt{cons}(x + s(x), l) ++ (l ++ l)$ $= (\mathtt{cons}(s(x) + x, l) ++ l) ++ l)$	$\mathbb{N}, \mathbb{L}$	✓	✓	–	–	–	–	–	–	–	–

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.

amazon web services™ Microsoft CERTORA

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.

erc European Research Council
Supporting top researchers
from anywhere in the world

FWF
Der Wissenschaftsfonds.

W|W|T|F
VIENNA SCIENCE
AND TECHNOLOGY FUND