Decidable Deductive Verification

Mooly Sagiv

Shachar Itzhaky

Oded Padon
Deductive Verification

Program P → Assertions $\phi$ → VC gen → Verification Condition $[P] \rightarrow \phi$ → SAT Solver

Counterexample → Proof
Where does undecidability occur?

• Interesting programmatic features
  • Unbounded integers
  • Floating points
  • Heaps
  • Unbounded arrays
  • Recursive procedures
  • Concurrency and distribution

• Interesting assertions
  • Quantifications
  • Numeric
  • Data types
The butterfly effect describes the phenomenon that a minor modification in one part of the program source causes changes in the outcome of the verification in other, unchanged and unrelated parts of the program. When this change in outcome causes the verifier to hit a time limit or other resource limit, previously succeeding verifications turn into spurious verification failures. The butterfly effect thus leads to verification instability, user frustration, and overall a degraded user experience.
The SAT Problem

• Given a propositional formula (Boolean function)
  • \( \varphi = (a \lor b) \land (\neg a \lor \neg b \lor c) \)

• Determine if \( \varphi \) is satisfiable
  • Find a satisfying assignment or report that such does not exist

• For \( n \) variables, there are \( 2^n \) possible truth assignments to be checked
Verification by reductions to SAT

Program P

Desired Property $\varphi$

Front-End

Formula $[[P]] \land \neg \varphi$

SAT(DPLL)

Counterexample

Proof
Verification by reduction to SAT

SAT Query:

\[
((a \land x) \lor (\lnot a \land \lnot x)) \land ((b \land y) \lor (\lnot b \land \lnot y)) \land ((x \land \lnot y) \lor (\lnot x \land y))
\]

assert x==y
Verification by reduction to SAT

SAT Query:
\[
((a \land x) \lor (\neg a \land \neg x)) \\
\land \\
((b \land y) \lor (\neg b \land \neg y)) \\
\land \\
((x \land \neg y) \lor (\neg x \land y))
\]

SAT Answer: Satisfiable by a=0, b = 1
Verification by reduction to SAT

![Diagram](image)

SAT Query:

\[((a \land x \land b) \lor (\neg a \land \neg x \land \neg b)) \land ((b \land y) \lor (\neg b \land \neg y)) \land ((x \land \neg y) \lor (\neg x \land y))\]

SAT Answer: Unsatisfiable
Challenge 1: Dealing with loops

• Bounded model checking: Verify that there are no bugs up to some loop unfolding [BMC]

• Deductive verification: user specified candidate inductive loop invariants
  • The induction hypothesis: strong enough to imply preservation in the next step

• Automatically infer loop inductive invariants
  • Abstract interpretation
Loop invariants

Reachable states

Inv

Counterexample to induction

Error states
**Simple Example: loop Invariants**

1: \(x := 1\);
2: \(y := 2\);
while * do {
   3: assert odd\([x]\);
   4: \(x := x + y\);
   5: \(y := y + 2\)
}
6:
Simple Example: loop Invariants

1: x := 1;
2: y := 2;
while * do {
   3: assert odd[x];
   4: x := x + y;
   5: y := y + 2
}
6:

Inv = odd[x] \land \neg odd[y]
Deductive verification by reductions to SAT

Program $\text{Tr}(X, X')$

Invariant $\text{Inv}$
$\text{Inv} \implies \varphi$

Desired Property $\varphi$

Front-End

$\text{Inv}(X) \land [\text{Tr}](X, X') \land \neg \text{Inv} (X')$

SAT(DPLL)

Counterexample to Induction (CTI)

Proof
Deductive Verification

1: x := 1;
2: y := 2;
while * do {
   3: assert odd[x];
   4: x := x + y;
   5: y := y + 2
}
6:

Solver

Is there a behavior of P that violates the inductiveness of I?

Inv = at(3) \Rightarrow odd[x]

at(3) \Rightarrow odd[x]

3: odd[x]=1, odd[y]=1

odd[x]' = (odd[x] \land \neg odd[y]) \lor (\neg odd[x] \land odd[y])
odd[y]' = odd[y]

odd[x]' = 0, odd[y]' = 1
Deductive Verification

1: x := 1;
2: y := 2;
while * do {
   3: assert odd[x];
   4: x := x + y;
   5: y := y + 2
}
6:

**Solver**

Is there a behavior of P that violates the inductiveness of I?

- Inv= at(3) ⇒ odd[x] ∧ ¬odd[y]
- at(3) ⇒ odd[x]

N

Proof
Challenge 2: Dealing with unbounded states

• Unbounded memory allocation
  • New objects
• Unbounded number of nodes in a distributed system
• Unbounded number of messages in the queue
• ...

17
Common Solution: SMT (Sat Modulo Theory)

• Support rich logic
  • \( \exists a, b, c, n: a, b, c > 0 \land n > 2 \land a^n + b^n = c^n \)
  • Theories
  • Numeric
   • Non linear arithmetic
   • Strings
  • Quantifier alternation
  • Separation logic
  • Transitive closure
  • ...

• Divergence

• Inconclusive & Infinite counterexamples

"The difficulty lies, not in new ideas, but in escaping from the old ones, which ramify, for those brought up as most of have been, into the corners of our minds." — John Maynard Keynes
Decidable Deductive Verification

• Express the assertions in a decidable logic
• For every assertion $a$:
  • There exists an algorithm to determine if $a$ holds for some states (SAT)
Useful Decidable Logics

• Propositional logic
  • “odd[x] \land odd[y] \Rightarrow \text{even}[x’]”

• Linear Arithmetic
  • \( 2 \times x + 3 \times y \geq 0 \land x \leq 2 \times y \)

• Effectively Propositional Logic
Simple Example

• $\exists a, b. \ r(a, b) \land \ \forall z. \ \neg r(z, z)$
Effectively PRopositional Logic – EPR
a.k.a. Bernays-Schönfinkel-Ramsey class

- Limited fragment of first-order logic
  - Restricted quantifier prefix: $\exists^* \forall^* \phi_{Q.F.}$
  - No $\forall^* \exists^*$
    - No recursive function symbols
    - No arithmetic

- Small model property
  - $\exists x_1, \ldots, x_n. \forall y_1, \ldots, y_m. \phi_{Q.F.}$ has a model iff it has a model of at most $n+k$ elements ($k$ - number of constant symbols)

- Satisfiability is decidable
  - $\text{NEXPTIME complete}/\Sigma_2$

- Support from Z3, Iprover, Vampire

Simple Example

\[ \exists a, b: \forall x, y: \ r(a, b) \land \neg r(b, a) \land (r(x, y) \implies r(y, x)) \]

\[ \models_{\text{SAT}} \]

\[ \forall x, y: \ r(c_a, c_b) \land \neg r(c_b, c_a) \land (r(x, y) \implies r(y, x)) \]

\[ \models_{\text{SAT}} \]

\[ \forall x, y \in \{c_a, c_b\}: \ r(c_a, c_b) \land \neg r(c_b, c_a) \land (r(x, y) \implies r(y, x)) \]

\[ \models_{\text{SAT}} \]

\[ ( r(c_a, c_b) \land \neg r(c_b, c_a) \land (r(c_a, c_a) \implies r(c_a, c_a) ) ) \land \\
( r(c_a, c_b) \land \neg r(c_b, c_a) \land (r(c_a, c_b) \implies r(c_b, c_a) ) ) \land \\
( r(c_a, c_b) \land \neg r(c_b, c_a) \land (r(c_b, c_a) \implies r(c_a, c_b) ) ) \land \\
( r(c_a, c_b) \land \neg r(c_b, c_a) \land (r(c_b, c_b) \implies r(c_b, c_b) ) ) \]

\[ \models_{\text{SAT}} \]

\[ (p_{ab} \land \neg p_{ba} \land (p_{aa} \implies p_{aa})) \land (p_{ab} \land \neg p_{ba} \land (p_{ab} \implies p_{ba})) \land \\
(p_{ab} \land \neg p_{ba} \land (p_{ba} \implies p_{ab})) \land (p_{ab} \land \neg p_{ba} \land (p_{bb} \implies p_{bb})) \]

\[ \models_{\text{UNSAT}} \]
Why EPR?

• $\forall x. \text{le}(x, x)$ Reflexive
• $\forall x, y, z. \text{le}(x, y) \land \text{le}(y, z) \Rightarrow \text{le}(x, z)$ Transitive
• $\forall x, y. \text{le}(x, y) \land \text{le}(y, x) \Rightarrow x = y$ Antisymmetric
• $\forall x, y. \text{le}(x, y) \lor \text{le}(y, x)$ Total
• $\exists \text{zero}. \forall x. \text{le}(\text{zero}, x)$ Non-empty
• $\forall x. \exists y. \text{le}(x, y) \land x \neq y$
Deductive verification by reductions to EPR

1. Program Tr $\exists^* \forall^*$
2. Universal $\forall^*$
   Invariant Inv
   $\text{Inv} \Rightarrow \phi$
3. Universal Desired Property $\forall^* \exists^*$

Diagram:
- Front-End
- Formula
  $\text{Inv}(X) \land [\text{Tr}] \land \neg \text{Inv}(X')$
- EPR Solver
  - Y: Counterexample to Induction (CTI)
  - N: Proof
Common Wisdom

• EPR is too weak to express properties of software systems
• Cannot express theories like numerics
  • The number of nodes in a distributed systems
  • Quorums in consensus algorithms
• Reason about paths in a dynamically changing graphs
  • Not even first order expressible
  • What about modularity
Invariant List Reversal

assume ac [h] \land \forall x: h < n^* > x

Node reverse(Node h) {
    Node c = h;  Node d = null;
    while {I} (c != null) {
        Node t = c.next;
        c.next =  d;
        d = c;
        c = t;
    }
    return d
}

I = \forall x, y:
{x < n^* > y \leftrightarrow y < n^* > x} \quad d < n^* > x
\quad c < n^* > x \quad \neg d < n^* > x
\quad (x < n^* > y \leftrightarrow x < n^* > y)
void foo() {
    Node l1, l2, t;
    l1 = null;
    l2 = null;
    while (*) {
        new(t);
        t.next = l1; l1 = t
        new(t);
        t.next = l1; l2 = t;
    }
    while (l1 != null) {
        t = l1.next;
        assert l2 != null;
        t = l2.next;
    }
}
Sound & Complete Reasoning about Deterministic Paths [CAV’13, POPL’14]

- $\forall x: n^*(x, x)$  
  Reflexivity

- $\forall x, y: n^*(x, y) \land n^*(y, z) \Rightarrow n^*(x, z)$  
  Transitivity

- $\forall x, y: n^*(x, y) \land n^*(y, x) \Rightarrow x = y$  
  Acyclicity

- $\forall x, y, z: n^*(x, y) \land n^*(x, z) \Rightarrow n^*(y, z) \lor n^*(z, y)$  
  Linearity

---


Inverting $n^* \Rightarrow n$

$n^* \leftrightarrow n^*(\alpha, \beta) \land \alpha \neq \beta$
Inverting \( n^* \Rightarrow n \)

\[
n(\alpha) = \beta \iff n+(\alpha, \beta) \land \forall \gamma: n+(\alpha, \gamma) \rightarrow n^*(\beta, \gamma)
\]
Key idea: representing deterministic paths

Alternative 1: maintain \( n \)
- \( n^* \) defined by transitive closure of \( n \)
- not definable in first-order logic

Alternative 2: maintain \( n^* \)
- \( n \) defined by transitive reduction of \( n^* \)
- Unique due acyclicity and outdegree at most 1
- Definable in first order logic (for roots)
  - \( n^+(a, b) \equiv n^*(a, b) \land a \neq b \)
  - \( n(a, b) \equiv n^+(a, b) \land \forall z: n^+(a, z) \rightarrow n^*(b, z) \)
Exploring Locality

• The program updates edge relations
• The compiler generates formulas to update paths
• This can always be done
  • Even in distributed systems
  • Uniform updates
• EPR formulas are closed under WP
Incremental
Simple updates

\[ X.n := \text{null} \]
Updating directed reachability in general graph is hard
Removing an edge in deterministic graphs (destructive update)

\[ C \rightarrow n = \text{NULL} \]

\[ n'(x,y) \leftrightarrow n^*(x,y) \land \neg n^*(x,C) \land n^+(C,y) \]
Mutating Single Linked Lists

• \( \text{wp}(\text{C.n} := \text{null}, Q) = Q[(\neg n^*(\alpha, \beta) \land (\neg n^*(\alpha, C) \lor n^*(\beta, C)) \lor n^*(\alpha, \beta)] \)

• Can also enforce absence of null dereferences
  \( C \neq \text{null} \)

• Can deal with nested cyclic lists

• But not arbitrary DAGs
  • Requires quantifier alternations [Dong&Su 95]
Single Mutation C.n := D (assuming C.n == null)

- Simple for general graphs
- $AF^R$ for arbitrary data structures
- $wp(C.n := D, Q) = Q[(n^*(\alpha, C) \land n^*(D, \beta))/ n^*(\alpha, \beta)]$
- Can also enforce acyclicity $\neg n^*(y, C)$
Adding an edge
C.n= D

assert \neg n^*(D, C)
n'^*(x,y) \iff n^*(x,y) \lor (n^*(x, C) \land n^*(D,y))
Traversing an edge
C = D.n (C is fresh)

\[ n^+(D, C) \land \forall x: n^+(D, x) \Rightarrow n^*(C, x) \]
WP Compound statements

- $\text{wp}(\text{skip}, Q) = Q$
- $\text{wp}(X := Y, Q) = Q[Y/ X]$
- $\text{wp}(S_1; S_2, Q) = \text{wp}(S_1, (\text{wp}(S_2, Q))$
- $\text{wp}($if $B$ then $S_1$ else $S_2$, Q) =$
  (B \land \text{wp}(S_1, Q)) \lor (\neg B \land \text{wp}(S_2, Q))$
- $\text{wp}(\text{while } B \text{ do } \{I\} S, Q) = I$
VC rules

• \( VC_{\text{gen}}({\{P}\ \{S\ \{Q\}\}}) = P \rightarrow \text{wp}(S,Q) \land \bigwedge VC_{\text{aux}}(S, Q) \)

• \( VC_{\text{aux}}(S, Q) = \{} \) (for any atomic statement)

• \( VC_{\text{aux}}(S_1; S_2, Q) = \)
  \[ VC_{\text{aux}}(S_1, \text{wp}(S_2, Q)) \cup VC_{\text{aux}}(S_2, Q) \]

• \( VC_{\text{aux}}(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) = \)
  \[ VC_{\text{aux}}(S_1, Q) \cup VC_{\text{aux}}(S_2, Q) \]

• \( VC_{\text{aux}}(\text{while } B \text{ do } S, Q) = VC_{\text{aux}}(S, I) \cup \)
  \[ \{I \land \lbrack B \rbrack \rightarrow \text{wp}(S,I)\} \cup \]
  \[ \{I \land \lnot \lbrack B \rbrack \rightarrow Q\} \]
But what about pointer traversals?

• Hoare assignment rule goes outside EPR

• $\text{wp}(X := Y.n,Q) = Q[n(y) / x]$
  • Outside EPR

• Reason about list segments

• Coincides with complications in pointer and shape analysis
Pointer Traversals

• Observe that \( wp \) is only used positively in VCs (unlike invariants and preconditions)
• Allows EA formulas with reachability
• \( wp (C := D.n, Q) = \forall \alpha: \{n(D)=\alpha\} \rightarrow Q[\alpha/C] \)
  • Replace \( n \) with \( n^* \) using reachability inversions
• Universal quantifications are also used for allocation \( C := \text{new()} \)
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$P, Q$</th>
<th>$\mathcal{V}_C$</th>
<th>Solving time (Z3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SLL$: reverse</td>
<td>2</td>
<td>11</td>
<td>57ms</td>
</tr>
<tr>
<td>$SLL$: filter</td>
<td>5</td>
<td>14</td>
<td>39ms</td>
</tr>
<tr>
<td>$SLL$: create</td>
<td>1</td>
<td>1</td>
<td>13ms</td>
</tr>
<tr>
<td>$SLL$: delete</td>
<td>5</td>
<td>12</td>
<td>23ms</td>
</tr>
<tr>
<td>$SLL$: deleteAll</td>
<td>3</td>
<td>7</td>
<td>32ms</td>
</tr>
<tr>
<td>$SLL$: insert</td>
<td>8</td>
<td>6</td>
<td>17ms</td>
</tr>
<tr>
<td>$SLL$: find</td>
<td>7</td>
<td>7</td>
<td>15ms</td>
</tr>
<tr>
<td>$SLL$: last</td>
<td>3</td>
<td>5</td>
<td>15ms</td>
</tr>
<tr>
<td>$SLL$: merge</td>
<td>14</td>
<td>31</td>
<td>226ms</td>
</tr>
<tr>
<td>$SLL$: rotate</td>
<td>6</td>
<td>-</td>
<td>22ms</td>
</tr>
<tr>
<td>$SLL$: swap</td>
<td>14</td>
<td>-</td>
<td>26ms</td>
</tr>
<tr>
<td>$DLL$: fix</td>
<td>5</td>
<td>11</td>
<td>32ms</td>
</tr>
<tr>
<td>$DLL$: splice</td>
<td>10</td>
<td>-</td>
<td>27ms</td>
</tr>
</tbody>
</table>
## Disproving with SAT

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Nature of defect</th>
<th>Formula Size</th>
<th>Solving time (Z3)</th>
<th>C.e. Size (vertices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL: find</td>
<td>null pointer dereference</td>
<td>P,Q # ( \forall )</td>
<td>I # ( \forall )</td>
<td>VC # ( \forall )</td>
</tr>
<tr>
<td>SLL: deleteAll</td>
<td>Loop invariant in annotation is too weak to prove the desired property</td>
<td>3 # ( \forall )</td>
<td>5 # ( \forall )</td>
<td>68 # ( \forall )</td>
</tr>
<tr>
<td>SLL: rotate</td>
<td>Transient cycle introduced during execution</td>
<td>6 # ( \forall )</td>
<td>- -</td>
<td>109 # ( \forall )</td>
</tr>
<tr>
<td>SLL: insert</td>
<td>Unhandled corner case when an element with the same value already exists in the list --- ordering violated</td>
<td>8 # ( \forall )</td>
<td>6 # ( \forall )</td>
<td>178 # ( \forall )</td>
</tr>
</tbody>
</table>
Open Questions

• What kind of data structure manipulations cannot be expressed in EPR?
  • Correlated list length

• Can we use a more limited logic?
From linked lists to distributed protocols

• Represent the global state of the system using labeled directed hypergraph
• The user specifies global invariants
• Each of the actions can be executed non-deterministically
Example: Leader Election in a Ring

- Nodes are organized in a ring
- Each node has a unique numeric id
- Protocol:
  - Each node sends its id to the next
  - A node that receives a message passes it (to the next) if the id in the message is higher than the node’s own id
  - A node that receives its own id becomes the leader
- Theorem:
  - The protocol selects at most one leader

Inductive Invariant for Leader Election

• \( \leq (\text{ID, ID}) \) – total order on node id’s
• \( \text{btw} (\text{Node, Node, Node}) \) – the ring topology
• \( \text{id}: \text{Node} \rightarrow \text{ID} \) – relate a node to its id
• \( \text{pending}(\text{ID, Node}) \) – pending messages
• \( \text{leader}(\text{Node}) \) – leader(n) means n is the leader

Safety property: \( I_0 \)

\[ I_0 = \forall x, y: \text{Node. leader}(x) \land \text{leader}(y) \Rightarrow x = y \]

Inductive invariant: \( \text{Inv} = I_0 \land I_1 \land I_2 \land I_3 \)

\[ I_1 = \forall n_1, n_2: \text{Node. leader}(n_2) \land n_1 \neq n_2 \Rightarrow id[n_1] \leq id[n_2] \]

\[ I_2 = \forall n_1, n_2: \text{Node. pending}(id[n_2], n_2) \land n_1 \neq n_2 \Rightarrow id[n_1] \leq id[n_2] \]

\[ I_3 = \forall n_1, n_2, n_3: \text{Node. btw}(n_1, n_2, n_3) \land \]
\[ \text{pending}(id[n_2], n_1) \Rightarrow id[n_3] \leq id[n_2] \]

How can we find an inductive invariant without knowing it?
Simple invariants for proving the safety of distributed protocols

Oded Padon, Kenneth McMillan, Aurojit Panda, Sharon Shoham

http://microsoft.github.io/ivy/
State of the art in formal verification

“the proofs consisted of about 5000 lines and assumed several nontrivial invariants of the Raft protocol. This paper discusses the verification of Raft as a whole, including all the invariants from the original Raft paper [32]. These new proofs consist of about 45000 additional lines” [Verdi, CPP’16]
State of the art in formal verification

“but our input language cannot compete in generality with mechanized proof methods that rely heavily on human expertise, e.g., IVY [55], Verdi [68], IronFleet [38], TLAPS [16]” [Konnov et al, POPL’17]
IVY’s Principles

• Express the program transition system and the safety property using limited first order logic
  • Logic is mostly hidden
  • Deduction is decidable
  • Display Counter Examples to Inductions graphically

• Assist the user in writing inductive invariants
Effectively Propositional Logic – EPR
a.k.a. Bernays-Schönfinkel-Ramsey class

• Limited fragment of first-order logic
  • Restricted quantifier prefix: $\exists^* \forall^* \phi_{Q,F}$.
  • No $\forall^* \exists^*$
• No function symbols
  • Possible to add stratified function symbols
• No arithmetic
• Small model property
  • $\exists x_1, \ldots, x_n. \forall y_1, \ldots, y_m. \phi_{Q,F}$ has a model iff it has a model of at most $n+k$ elements ($k$ - number of constant symbols)
• Satisfiability is decidable
  • NEXPTIME / $\Sigma_2$
• Supported by theorem provers (e.g., Z3, iProver, Vampire)

Relational Modeling Language (RML)

- Designed to make verification tasks decidable (EPR)
  - Yet expressive enough to model systems
- Turing-Complete
- Universally quantified inductive invariants are decidable to check
- System state described by finite (unbounded) relations
- No numerics
- Simple (quantifier-free) updates
- Universally quantified axioms (domain specific)
  - Total orders, partial orders, lists, trees, rings, quorums, ...
## Languages and verification

<table>
<thead>
<tr>
<th>Language</th>
<th>Executable</th>
<th>Expressiveness</th>
<th>Inductiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, Java, Python...</td>
<td>✓</td>
<td>Turing-Complete</td>
<td>Undecidable</td>
</tr>
<tr>
<td>SMV</td>
<td>✗</td>
<td>Finite-state</td>
<td>Temporal Properties</td>
</tr>
<tr>
<td>Ivy</td>
<td>C++</td>
<td>Turing-Complete</td>
<td>Decidable(EPR)</td>
</tr>
<tr>
<td>Dafny</td>
<td>✓</td>
<td>Turing-Complete</td>
<td>Undecidable with lemmas</td>
</tr>
<tr>
<td>Coq, Isabelle/HOL</td>
<td>✓</td>
<td>Turing-Complete</td>
<td>Manual with tactics</td>
</tr>
<tr>
<td>TLA+</td>
<td>✗</td>
<td>Turing-Complete</td>
<td>Manual</td>
</tr>
</tbody>
</table>
Leader Election Protocol (Ivy)

- \( \leq (ID, ID) \) – total order on node id’s
- \( \text{btw} \) (Node, Node, Node) – the ring topology
- \( \text{id} \) : Node \( \rightarrow \) ID – relate a node to its unique id
- \( \text{pending} \) (ID, Node) – pending messages
- \( \text{leader} \) (Node) – leader(n) means n is the leader

protocol = (send | receive)*

assert I0 = \( \forall \ x, y : \text{Node.} \ \text{leader}(x) \land \text{leader}(y) \rightarrow x = y \)
Inductive Invariant for Leader Election

- \( \preceq (ID, ID) \) – total order on node id’s
- \( \text{btw} (\text{Node}, \text{Node}, \text{Node}) \) – the ring topology
- \( \text{id}: \text{Node} \rightarrow \text{ID} \) – relate a node to its id
- \( \text{pending}(\text{ID}, \text{Node}) \) – pending messages
- \( \text{leader}(\text{Node}) \) – leader\( (n) \) means \( n \) is the leader

**Safety property:** \( I_0 \)

\[ I_0 = \forall x, y: \text{Node. leader}(x) \land \text{leader}(y) \Rightarrow x = y \]

**Inductive invariant:** \( \text{Inv} = I_0 \land I_1 \land I_2 \land I_3 \)

\[ I_1 = \forall n_1, n_2: \text{Node. leader}(n_2) \land n_1 \neq n_2 \Rightarrow \text{id}[n_1] \leq \text{id}[n_2] \]

\[ I_2 = \forall n_1, n_2: \text{Node. pending}(\text{id}[n_2], n_2) \land n_1 \neq n_2 \Rightarrow \text{id}[n_1] \leq \text{id}[n_2] \]

\[ I_3 = \forall n_1, n_2, n_3: \text{Node. btw}(n_1, n_2, n_3) \land \text{pending}(\text{id}[n_2], n_1) \Rightarrow \text{id}[n_3] \leq \text{id}[n_2] \]

*How can we find an inductive invariant without knowing it?*
## Verified Protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Model (# LOC)</th>
<th>Property (# Literals)</th>
<th>Invariant (# Literals)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leader in Ring</td>
<td>59</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Learning Switch</td>
<td>50</td>
<td>11</td>
<td>18</td>
</tr>
<tr>
<td>DB Chain Replication</td>
<td>143</td>
<td>11</td>
<td>35</td>
</tr>
<tr>
<td>Chord</td>
<td>155</td>
<td>35</td>
<td>46</td>
</tr>
<tr>
<td>Lock Server (500 Coq lines [Verdi])</td>
<td>122</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>Distributed Lock (1 week [IronFleet])</td>
<td>41</td>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td>Single Decree Paxos</td>
<td>85</td>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td>Multi Paxos</td>
<td>102</td>
<td>3</td>
<td>38</td>
</tr>
<tr>
<td>Vertical Paxos</td>
<td>123</td>
<td>3</td>
<td>65</td>
</tr>
<tr>
<td>Fast Paxos</td>
<td>117</td>
<td>3</td>
<td>59</td>
</tr>
<tr>
<td>Flexible Paxos</td>
<td>88</td>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td>Virtually Synchronous Paxos</td>
<td></td>
<td></td>
<td>Work in progress</td>
</tr>
<tr>
<td>Practical Byzantine Fault Tolerance</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Consensus [OOPSLA’17]

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Model [LOC]</th>
<th>Invariant [Conjectures]</th>
<th>EPR [sec] $\mu$</th>
<th>$\sigma$</th>
<th>RW [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paxos</td>
<td>85</td>
<td>11</td>
<td>1.0</td>
<td>0.1</td>
<td>1.2</td>
</tr>
<tr>
<td>Multi-Paxos</td>
<td>98</td>
<td>12</td>
<td>1.2</td>
<td>0.1</td>
<td>1.4</td>
</tr>
<tr>
<td>Vertical Paxos*</td>
<td>123</td>
<td>18</td>
<td>2.2</td>
<td>0.2</td>
<td>-</td>
</tr>
<tr>
<td>Fast Paxos*</td>
<td>117</td>
<td>17</td>
<td>4.7</td>
<td>1.6</td>
<td>1.5</td>
</tr>
<tr>
<td>Flexible Paxos</td>
<td>88</td>
<td>11</td>
<td>1.0</td>
<td>0</td>
<td>1.2</td>
</tr>
<tr>
<td>Stoppable Paxos*</td>
<td>132</td>
<td>16</td>
<td>3.8</td>
<td>0.9</td>
<td>1.6</td>
</tr>
</tbody>
</table>

*first mechanized verification
Transformation to EPR reusable across all variants!
Why do people hate First Order Logic?

- Hard to understand and error prone
  - Nested quantifiers and negations
- Too weak: Cannot express
  - Parity
  - Numeric
  - Quorums
  - Finiteness
  - Paths in a graph
- Hard for automation
  - Satisfiability is undecidable
  - NP-complete for fixed size

Our Solution

- Limited EPR formulas $\exists^* \forall^*$
  - Finite model guaranteed to exit
  - Display graphs graphically
- Define axioms per of domain of programs
  - Total orders
  - Paths in deterministic graphs
- Use a Turing complete programming language for updates
  - Satisfiability is $\text{NEXPTIME complete}/\Sigma_2$
    - Support from Z3, Iprover, Vampire
- Rely on user provided loop invariants or sound inferred invariants
Main contributions

• Showing the linked lists can be verified in a sound and complete way using EPR solver [CAV’13, POPL’14]

• Characterizing the absence of universal invariants relaxed traces [CAV’15, JACM]

• Characterizing the decidability of inferring universal invariants [POPL’16]

• A system for interactively inferring universal invariants [PLDI’16]


