Reasoning about Program Data Structure Shape: from the Heap to Distributed Systems

Mooly Sagiv
Credits

A. Benerjee  N. Immerman  S. Itzhaky  A. Karbyhev  O. Lahav

K. McMillan  A. Nanevski  O. Padon  A. Panda  S. Shoham
Is there a behavior of P that violates $\varphi$?

Counterexample

Proof
Challenges

1. Specifying safety properties
2. Undecidability of checking interesting properties
   1. The halting problem
   2. Rice theorem
   3. Simple programs can do complicated things
Programs $\approx$ Infinite Transition Systems

1: $x := 1$
2: $y := 2$
while * do {
  3: assert $x \geq 1$
  4: $x = x + y$
  5: $y := y + 1$
}
A safety property $\varphi$ holds in a transition system $\tau$ if and only if there exists an inductive invariant $I$ such that

$$I \Rightarrow \varphi \text{ (Safety)}$$

$$\text{Init} \Rightarrow I \text{ (Initiation)}$$

if $\sigma \models I$ and $\sigma \tau \sigma'$ then $\sigma' \models I$

(Consecution)
Semi-Automatic Program Verification

Program $P$

Candidate Invariant $I$

Safety Property $\varphi$

Solver

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

Counterexample

Proof
Semi-Automatic Program Verification

1: \( x := 1; \)
2: \( y := 2; \)
while * do {
  3: \textbf{assert} \( x \geq 1; \)
  4: \( x := x + y; \)
  5: \( y := y + 1 \)
}
6: 

\[ \text{at}(3) \implies x \geq 1 \]

Solver

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

3: \( <1, -2> \)
Semi-Automatic Program Verification

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

at(3) ⇒ x ≥ 1 ∧ y ≥ 0

at(3) ⇒ x ≥ 1

 Solver

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

Proof
Challenges

1. Specifying safety properties
2. Inductive Invariants for Floyd/Hoare style verification
   - Hard to express
   - Hard to change
   - Hard to infer
3. Deduction
   - Reasoning about inductive invariants
     - Undecidability of implication checking
Semi-Automatic Program Verification

1: x := 1;
2: y := 2;
while * do {
   3: assert x ≥ 1;
   4: x = x + y;
   5: y := y + 1
}
6: at(3) ⇒ x ≥ 1 ∧ y ≥ 0

Solver

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

Proof
1: x := 1;
2: y := 2;
while *do {
  3: assert x ≥ 1;
  4: x = (x*x-y*y) / (x-y);
  5: y := y + 1
}
6: at(3) ⇒ x ≥ 1 ∧ y ≥ 0

Solver

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

at(3) ⇒ x ≥ 1

Proof
Challenge 3: Deductive Verification about Reachability

Sound and complete Dafny w/o matching loops

Reasoning about directed reachability in dynamically evolving graphs (relations)

- No garbage
- Preservation of data structure invariants
- Termination
- Reachability properties in distributed protocols
- Even sortedness
traverse(Node a, Node b) {
    for (t = a; t != b; t = t->n) {
        ...
    }
}
Directed Reachability

- Directed reachability suffices to describe many properties of data structures
  - Absence of garbage
    - $\forall x: r^*(\text{root}, x)$
  - Acyclicity
    - $\forall x: \neg r^+(x, x)$
  - Data Structure Invariants
    - $\forall x: f^*(\text{root}, x) \iff b^*(\text{root}, x)$

$r^*(x, y)$ denotes a finite directed path of relation of $r$ from $x$ to $y$
rotate(List first, List last) {
    assert acyclic first
    if (first != NULL) {
        last -> next = first;
        first = first -> next;
        last = last -> next;
        last -> next = NULL;
    }
    assert acyclic first;
}
Reachability in Distributed Protocols

- The topology evolves over time
- Reason about evolving relations
- Prove safety
  - Absence of paths
    - Isolation
  - Absence of cycles
Learning Switch

\[ \alpha \rightarrow \beta \]

Input Port | Packet | Output Port
---|---|---

Routing Table

<table>
<thead>
<tr>
<th>Dst</th>
<th>Prt</th>
</tr>
</thead>
</table>
### Learning Switch

![Diagram of a switch with input and output ports and a routing table.

#### Input Port | Packet | Output Port
--- | --- | ---
1 | $\alpha \rightarrow \beta$ | 2, 3

#### Routing Table

<table>
<thead>
<tr>
<th>Dst</th>
<th>Prt</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>1</td>
</tr>
</tbody>
</table>
# Learning Switch

![Learning Switch Diagram]

\[ \beta \rightarrow \alpha \]

<table>
<thead>
<tr>
<th>Input Port</th>
<th>Packet</th>
<th>Output Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \alpha \rightarrow \beta )</td>
<td>2, 3</td>
</tr>
<tr>
<td>2</td>
<td>( \beta \rightarrow \alpha )</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Routing Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dst</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>( \alpha )</td>
</tr>
</tbody>
</table>
event receive =
    \langle p: \text{packet}, m: \text{node} \rangle \in \text{pending} \Rightarrow
    \text{pending.remove} \langle p, m \rangle
    \text{route}[p.\text{src}] := \{p.\text{ingress}\}; \quad \text{// learn}
    \exists \text{pr} : \text{route}[p.\text{dst}] = \{\text{pr}\} \Rightarrow
    \text{forward } p \text{ to } \text{pr} \quad \text{// adds new tuple to pending}
    \text{route}[p.\text{dst}] = \{\} \Rightarrow
    \text{flood } p \quad \text{// adds new tuples to pending}
    \text{assert acyclic for all } Dst: \text{route}[Dst];

Verification can identify a topology in which a forwarding loop in the routing table occur
A Forwarding Loop

Routing Table

<table>
<thead>
<tr>
<th>Dst</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
</tr>
</tbody>
</table>

α → β

α → β → a → b → a → c → b → α
A Forwarding Loop

Routing Table

<table>
<thead>
<tr>
<th>Dst</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
</tr>
</tbody>
</table>

Routing Table

<table>
<thead>
<tr>
<th>Dst</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>4</td>
</tr>
</tbody>
</table>
A Forwarding Loop

Routing Table

<table>
<thead>
<tr>
<th>Dst</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>1</td>
</tr>
</tbody>
</table>

Routing Table

<table>
<thead>
<tr>
<th>Dst</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>4</td>
</tr>
</tbody>
</table>

Routing Table

<table>
<thead>
<tr>
<th>Dst</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>7</td>
</tr>
</tbody>
</table>
Loop-Free Learning Switch Code

event receive =
  ⟨p: packet, m: node⟩ ∈ pending 🔄
  pending.remove ⟨p, m⟩
  route[p.src] = {} 🔄
  route[p.src] := {p.ingress} // learn
  exists pr : route[p.dst] = {pr} 🔄
    forward p to pr // adds new tuple to pending
  route[p.dst] = {} 🔄 // flood
  flood p // adds new tuples to pending
assert acyclic for all Dst: route[Dst];

Verification proves the absence of forwarding loops for arbitrary topologies 🤘
Challenges

• Complexity of reasoning about reachability assertions
  – Not first order expressible
  – Undecidability of reachability (not even RE)

"there is a mismatch between the simple intuitions about the way pointer operations work and the complexity of their axiomatic treatments"

O'Hearn, Reynolds, Yang [CSL 2001]

• [Inferring reachability properties from the code]
Do I have to Solve Hilbert’s 10\textsuperscript{th} problem?

count {
List a = NULL, b= NULL, t;
int c = 0; read(c);
while (c > 0) {
    t = malloc(); t→next = a; a = t;
    t = malloc(); t→next = b; b = t;
    c--;}
while (a != null) {
    assert a!=null; print(a→d);
    assert b!=null; print(b→d);}
}
Jackson’s Thesis

• If a program has a bug \( \Rightarrow \) it also occurs on small input \( k \)
  – True in many cases
  – But
    \( \frowny \) What if not?
    \( \frowny \) Hard to find \( k \)
    \( rowny \) Hard to scale checking to \( k \)
Itzhaky’s thesis: Linked list manipulations are simple

• Simple to reason about correctness
  – Small counterexamples
• Deterministic paths
• Even for doubly/circular/nested lists/distributed protocols
  – Sortedness
  – Size
• “Simple” inductive invariants suffice to show safety
  – Alternation Free + Reachability “⊆” ∃∀
Do I have to Solve Hilbert’s 10\textsuperscript{th} problem?

count {
    List a = NULL, b = NULL, t;
    int c = 0; read(c);
    while (c > 0) {
        t = malloc(); t$\rightarrow$next = a; a = t;
        t = malloc(); t$\rightarrow$next = b; b = t;
        c--;
    }
    while (a != null) {
        assert a!=null; print(a$\rightarrow$d);
        assert b!=null; print(b$\rightarrow$d);
    }
}
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 valid
• 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
• But many practical tools exist
SAT made some progress...
Semi-Automatic Verification Process

Program

Candidate Inductive Invariant I

Property $\varphi$

VC gen

Verification Conditions:
1) $\text{Init} \land \neg I$
2) $\llbracket P \rrbracket(V, V') \land I(V) \land \neg I(V')$
3) $I(V) \land \neg \varphi(V)$

SAT Solver

Counterexample

Proof

Unbounded systems
(Uninterpreted Relational) First Order Logic w/o functions

$$t ::= c$$  \hspace{1cm} \text{Constant symbol}

$$| \hspace{1cm} x$$  \hspace{1cm} \text{Logical variable}

$$\varphi ::= r(t_1, t_2, ... , t_n)$$  \hspace{1cm} \text{Relation}

$$| \hspace{1cm} t_1 = t_2$$  \hspace{1cm} \text{Equality}

$$| \hspace{1cm} \exists x. \varphi$$  \hspace{1cm} \text{Existential Quantification}

$$| \hspace{1cm} \forall x. \varphi$$  \hspace{1cm} \text{Universal Quantification}

$$| \hspace{1cm} \varphi_1 \lor \varphi_2$$  \hspace{1cm} \text{Disjunction}

$$| \hspace{1cm} \varphi_1 \land \varphi_2$$  \hspace{1cm} \text{Conjunction}

$$| \hspace{1cm} \neg \varphi$$  \hspace{1cm} \text{Negation}
SAT becomes undecidable

- $\forall x. \text{le}(x, x)$  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
- $\forall x. \exists y. \text{le}(x, y) \land x \neq y$
SAT becomes undecidable

- $\forall x. \text{le}(x, x)$  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
- $\forall x. \exists y. \text{le}(x, y) \land x \neq y$
Effectively Propositional Logic – EPR
a.k.a. Bernays-Schönfinkel-Ramsey class

- Fragment of first-order logic
  - Restricted quantifier prefix: $\exists^*\forall^* \varphi_\text{Q.F.}$
  - No function symbols

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

- Satisfiability is decidable
  - NEXPTIME

- Support from Z3

Can we reason about interesting properties with EPR?

Some parts have to be provided by domain experts for a class of programs

Axioms provided by domain experts
Semi-Automatic Program Verification

Is there a behavior of P in which c=e?

\[ \forall x. \neg (n^*(a,x) \land n^*(b,x)) \land n(a, c) \land n(b, d) \land n(d, e) \land c=e \]

\[ n = \{(a,c), (b,d), (d,c)\} \]
\[ n^* = \{\} \]

Counterexample
Complete Reasoning about Deterministic Paths

- \(n^*(x, x)\)                                              Reflexivity
- \(n^*(x, y) \land n^*(y, z) \Rightarrow n^*(x, z)\)      Transitivity
- \(n^*(x, y) \land n^*(y, x) \Rightarrow x = y\)         Acyclicity
- \(n^*(x, y) \land n^*(x, z) \Rightarrow n^*(y, z) \lor n^*(z, y)\) Linearity
- \(n^+(x, y) \equiv n^*(x, y) \land x \neq y\)
- \(n(a, b) \equiv n^+(a, b) \land \forall x: n^+(a, x) \Rightarrow n^*(b, x)\)


Semi-Automatic Program Verification

Is there a behavior of P in which $c=e$?

**axioms $\land$
\[ \forall x. \neg (n^*(a,x) \land n^*(b,x)) \land 
\text{“}n(a, c)\text{“} \land \text{“}n(b, d)\text{“} \land \text{“}n(d, e)\text{“} \land c=e \]

**SAT Solver (Z3)**

Proof

**assume $\forall x. \neg (n^*(a,x) \land n^*(b,x))$
$c := a\rightarrow n$;
$d := b\rightarrow n$;
$e := d\rightarrow n$;
assert $c \neq e$;**
But how can we model the program in EPR?

- The program updates edge relations
- The compiler generates EPR formulas to update paths
- This can always be done
Incremental
Simple updates

\[ n \xrightarrow{x \mapsto n := \text{NULL}} n' \]

\[ n^* \xrightarrow{\text{FO}^T_C} n' \]

\[ n'^* \xrightarrow{\text{EPR}} n' \xrightarrow{\text{FO}^T_C} n'^* \]
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>P,Q</th>
<th>Formula Size</th>
<th>Solving time (Z3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#</td>
<td>#</td>
<td></td>
</tr>
<tr>
<td></td>
<td>∀</td>
<td>∀</td>
<td></td>
</tr>
<tr>
<td>SLL: reverse</td>
<td>2</td>
<td>11</td>
<td>133</td>
</tr>
<tr>
<td>SLL: filter</td>
<td>5</td>
<td>14</td>
<td>280</td>
</tr>
<tr>
<td>SLL: create</td>
<td>1</td>
<td>1</td>
<td>36</td>
</tr>
<tr>
<td>SLL: delete</td>
<td>5</td>
<td>12</td>
<td>152</td>
</tr>
<tr>
<td>SLL: deleteAll</td>
<td>3</td>
<td>7</td>
<td>106</td>
</tr>
<tr>
<td>SLL: insert</td>
<td>8</td>
<td>6</td>
<td>178</td>
</tr>
<tr>
<td>SLL: find</td>
<td>7</td>
<td>7</td>
<td>64</td>
</tr>
<tr>
<td>SLL: last</td>
<td>3</td>
<td>5</td>
<td>74</td>
</tr>
<tr>
<td>SLL: merge</td>
<td>14</td>
<td>31</td>
<td>2255</td>
</tr>
<tr>
<td>SLL: rotate</td>
<td>6</td>
<td>-</td>
<td>73</td>
</tr>
<tr>
<td>SLL: swap</td>
<td>14</td>
<td>-</td>
<td>965</td>
</tr>
<tr>
<td>DLL: fix</td>
<td>5</td>
<td>11</td>
<td>121</td>
</tr>
<tr>
<td>DLL: splice</td>
<td>10</td>
<td>-</td>
<td>167</td>
</tr>
</tbody>
</table>
## Disproving with SAT

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Nature of defect</th>
<th>\textbf{Formula Size}</th>
<th>Solving time (Z3)</th>
<th>C.e. Size (vertices)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P,Q</strong></td>
<td><strong>I</strong></td>
<td><strong>VC</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLL: find</td>
<td>null pointer dereference</td>
<td>7 1 7 1 64 3</td>
<td>18ms</td>
<td>2</td>
</tr>
<tr>
<td>SLL: deleteAll</td>
<td>Loop invariant in annotation is too weak to prove the desired property</td>
<td>3 2 5 2 68 3</td>
<td>58ms</td>
<td>5</td>
</tr>
<tr>
<td>SLL: rotate</td>
<td>Transient cycle introduced during execution</td>
<td>6 1 - - 109 3</td>
<td>25ms</td>
<td>3</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 1 6 1 178 3</td>
<td>33ms</td>
<td>4</td>
</tr>
</tbody>
</table>
Summary thus far

• Reduced the undecidable problem of checking inductiveness to the NEXPTIME problem of checking EPR satisfiability
  – Efficient in practice
  – Useful for bounded model checking
  – Useful for synthesis

• But what about inferring EPR invariants?
Automatically Inferring EPR Invariants

- PDR/IC3 procedure for inferring universal invariants [CAV’15]
- Inferring universal invariants for linked-lists is decidable [POPL’16]
- Systematic extensions for decidability of some distributed protocols [POPL’16]
- Inferring general universal invariants is undecidable [POPL’16]
- Inferring alternation-free invariants for linked-lists is undecidable [POPL’16]

Ivy: Interactive Verification via EPR

Goal: Engage the user in automated verification

- Use powerful invariant generation heuristics interactively
- Bidirectional feedback between user and heuristics

Questions:

- What *decidable problem* should we let the machine solve?
- What is a useful *interaction mode* between the user and the machine heuristics?

CTI Mode

- M
- Inv
- Ind?
- Modify Inv
- “minimal” CTI
- Diagnose CTI
- User
- Heuristics

BMC

- M
- Spec
- BMC
- Fix model / spec

Abstract Reachability & Concept Graphs

- User
- Heuristics

???
Heuristics for User Interaction

- Carefully select CTI
  - Minimize certain “metrics”
- Interactive Generalization
  - Select visible relations
  - Gather facts from user selection
  - BMC
    - Check conjecture
    - Minimize conjecture
  - Sufficiency for current failure
  - Relative inductiveness
Summary

• EPR is useful to reason about infinite state systems
  – BMC
  – Inductive invariants
  – Effective reasoning about TC
• Exploit simplicity of quantifier free updates in distributed systems
• The next challenge is invariant inference
BACKUP SLIDES
Some Related Work

• Monadic second order logic [CIAA’00] [SAS’11]
• Decidable separation logic
• Sound first order axioms

Updating Reachability
Adding an edge $c \rightarrow n = d$

assert $\neg n^*(\beta, \alpha)$

$n'^*(\alpha, \beta) \iff n^*(\alpha, \beta) \lor (n^*(\alpha, c) \land n^*(d, \beta))$
Updating Directed Reachability in General Graph is Hard
Removing an edge
(destructive update)

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

\[ \alpha \quad c \quad d \quad \beta \]

\[ n'*(\alpha, \beta) \leftrightarrow n^*(\alpha, \beta) \land \neg(n^*(\alpha, c) \land n^+(c, \beta)) \]
Traversing an edge

\[ c = d \rightarrow n \ (c \text{ is fresh}) \]

d

\[ n^+(d,c) \land \]
\[ \forall x: \ n^+(d,x) \Rightarrow n^*(c,x) \]
Reasoning about Distributed Protocols

• The correctness of very simple distributed protocol can be tricky
  – Safety, Consensus, Serializability, Liveness
  – Widely used

• Examples: Raft, Paxos, Chord

• Unlimited resources

• Counterintuitive reasoning

• Topology affects correctness
Beyond EPR

- EPR cannot force the existence of unbounded sets
- Non-emptyness of the routing relations
- Hole-punching firewall
The Instrumentation Principle

- Users define extra derived relations
- Expressible outside EPR
- The system generates update formulas
- Guaranteed soundness
- Completeness no longer guaranteed
  - But concrete states are precise

[TOPLAS’10] T.W. Reps, M. Sagiv, A. Loginov:
Finite differencing of logical formulas for static analysis
The Static Analysis Tradeoff

Precision:
Rich Properties
Few False Alarms

Applications
- Bug finding
- Memory Safety
- Education
- Program Synthesis
- Comparing Programs
- Security
- Networks
- Distributed Protocols
- Cloud

Efficient Algorithms
- SAT solving
- Consequence Finding
- Constraint Solving
- Context Free Reachability
- Property Directed Reachability
- Decision Procedures
- Theory Solvers
  - Linear Programming

User Interaction

Domain Specialization

Scalability
Summary

• Domain specific verification/static analysis
• Symbolic reasoning on directed reachability can be useful for verification and bug finding in
  – Linked data structures
  – Distributed systems
• Much more need to be done
  – Invariant Inference
  – Efficient decision procedures
\( \exists \gamma: \alpha<n^*>\gamma \land \gamma<n^*>c \land \\
\text{ } n(\gamma)=\delta \land \delta<n^*>\beta \land \neg\delta<n^*>c \)
Loop-Free Learning Switch Code

event receive =
  <p: packet, m: node> ∈ pending ⇝
  pending.remove <p, m>
  route[p.src] = {} ⇝
  route[p.src] := {p.ingress} // learn
  exists pr : route[p.dst] = {pr} ⇝
  forward p to pr // adds new tuple to pending
  route[p.dst] = {} ⇝ // flood
  flood p // adds new tuples to pending
  assert acyclic forall Dst: route[Dst];

∀dst, node1, node2:
route[node2, dst] ≠ {} → ¬path[dst](node1, node2)

Expressible in a weak decidable logic ∃*∀*