Automated I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin
Manos Kapritsos, Baris Kasikci, Karem A. Sakallah

University of Michigan
Distributed Systems Are Subtle

Figure 1: Typical Figure 2 from Byzantine fault paper: Our network protocol

[Mickens 2013]
The Alternative: Formal Verification
Existing Verification Approaches

All existing approaches require the human to find an **inductive invariant**

We want to automatically find inductive invariants …

… by **combining the power of Ivy and model checking**
## Preview of Results

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Traditional approach</th>
<th>Ivy</th>
<th>I4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock server</td>
<td>500 lines (Verdi)</td>
<td>&lt;1 hour</td>
<td>Automated</td>
</tr>
<tr>
<td>Distributed lock</td>
<td>A few days (IronFleet)</td>
<td>A few hours</td>
<td>&lt; 5 min</td>
</tr>
</tbody>
</table>

Numbers come from Ivy [PLDI 2016]
Outline

Motivation

Verification of distributed systems

I4: a new approach

Design of I4

Evaluation

Conclusion
Induction on Distributed Protocol

Goal: prove that the safety property *always* holds

An execution:

Inductive proof

- Base case: prove initial state is safe
- Inductive step: if state $k$ is safe, prove state $k+1$ is safe
Safety Property vs. Inductive Invariant

Diagram showing:
- All states
- Reachable states
- Inductive invariant
- Safe states

All states encompass everything, with reachable states being a subset. Inductive invariant is another subset, and safe states are the strictest subset.
Inductive Invariants Are Complex

∀ N₁, N₂ : node, E : epoch.

locked(E, N₁) ∧ locked(E, N₂) ⇒ N₁ = N₂

∧ ∀ N₁, N₂, E. held(N₁) ∧ trans(E, N₂) ⇒ le(E, ep(N₁))
∧ ∀ N₁, N₂, E. trans(E, N₁) ∧ ¬le(E, ep(N₁)) =
∧ ∀ N₁, N₂, E₁, E₂. (trans(E₁, N₁) ∧ ¬le(E₁, ep(N₁)) ∧

Strengthening Assertion

I.e. No two nodes can hold the lock at the same time.

Existing approaches rely on manual effort and human intuition.
Outline

Motivation

Verification of distributed systems

I4: a new approach

Design of I4

Evaluation

Conclusion
I4: a new approach

Goal: Find an inductive invariant *without* relying on human intuition.

Insight: Distributed protocols exhibit regularity.

- Behavior doesn’t fundamentally change as the size increases
- E.g. distributed lock, Chord DHT ring, …

Implication: We can use inductive invariants from small instances to infer a *generalized* inductive invariant that holds for all instances.
Leveraging Model Checking

- Fully automated
- Doesn’t scale to distributed systems

I4 applies model checking to small, finite instances …
… and then generalizes the result to all instances.
Outline

Motivation

Verification of distributed systems

\(\mathcal{I}4\): a new approach

Design of \(\mathcal{I}4\)

Evaluation

Conclusion
Design of I4

Invariant generation on a finite instance

Invariant generalization

Increase Size

Invariant generation on a finite instance

Invariant generalization

Increase Size
Invariant generation on a **finite** instance

Increase Size

Invariant generalization
Making The Model Checking Problem Easier

Symmetry

Concretize (Manual)

E.g. FIRST is the node that sends the first message

FIRST = Node1

Out of memory

Invariant generalization

Counterexample

Increase Size

Create Small (Finite) Instance

Finite state machine

Model Checker

Protocol.inv

Protocol.ivy
Invariant Generation on a Finite Instance

Concretize (Manual) → Create Small (Finite) Instance → Finite state machine → Model Checker

Out of memory → Counterexample

Increase Size → Protocol.ivy

Invariant generalization

Out of memory → Concretize (Manual)

Protocol.ivy

Finite state machine

Model Checker

Protocol.finv
Invariant generation on a **finite** instance

Invariant generalization
Generalizing The Inductive Invariant

\[ \forall N_1, N_2. N_1 \neq N_2 \implies P(N_1, N_2) \]

\[ \forall N_1, N_2. (N_1 \neq N_2) \land (N_1 = first) \land (N_2 \neq first) \implies P(N_1, N_2) \]

Increase Size

Create Small

(Finite) Instance

Model

Checker

Out of memory

Invariant generation on a finite instance

Generalize
Invariant Generalization

Invariant generation on a finite instance

- Protocol.ivy
- Protocol.finv
- Increase Size

- Prune
- Protocol_inv.ivy
- Generalize

- Ivy
- Correct

- Strengthening Assertion

- Safety Property Violation
- Out of memory

Invariant generation on a finite instance.
Outline

Motivation

Verification of distributed systems

I4: a new approach

Design of I4

Evaluation

Conclusion
Evaluation

- Lock Server
- Leader Election
- Distributed lock
- Chord Ring
- Learning Switch
- Database Chain Consistency
- Two-Phase Commit

Blind Tests
## Result Summary

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Manual Effort</th>
<th>Total time (sec)</th>
<th>Minimal instance size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock server</td>
<td>None</td>
<td>0.9</td>
<td>2 clients, 1 server</td>
</tr>
<tr>
<td>Leader election in ring</td>
<td>&lt;5min</td>
<td>6.2</td>
<td>3 nodes, 3 ids</td>
</tr>
<tr>
<td>Distributed lock</td>
<td>&lt;5min</td>
<td>159.6</td>
<td>2 nodes, 4 epochs</td>
</tr>
<tr>
<td>Chord ring</td>
<td>&lt;5min</td>
<td>628.9</td>
<td>4 nodes</td>
</tr>
<tr>
<td>Learning switch</td>
<td>None</td>
<td>10.7</td>
<td>3 nodes, 1 packets</td>
</tr>
<tr>
<td>Database chain Consistency</td>
<td>None</td>
<td>12.6</td>
<td>3 transactions, 3 operations, 1 key, 2 node</td>
</tr>
<tr>
<td>Two-Phase Commit</td>
<td>None</td>
<td>4.3</td>
<td>6 nodes</td>
</tr>
</tbody>
</table>
Outline

Motivation

Verification of distributed systems

І4: a new approach

Design of І4

Evaluation

Conclusion
Conclusion

Regularity of distributed protocols makes it possible to automatically infer inductive invariants of distributed protocols from small instances.

By combining the power of **model checking** and **Ivy**, I4 can verify a number of interesting protocols with little to no manual effort.

https://github.com/GLaDOS-Michigan/I4

I'm looking for a research intern for next summer. If you're interested, just contact me.
type node
type epoch

relation le(E:epoch, E:epoch)
relation locked(E:epoch, N:node)
relation transfer(E:epoch, N:node)
relation held(N:node)

individual zero : epoch
individual e : epoch
function ep(N:node) : epoch
individual first : node

after init {
  held(X) := X:node = first;
  ep(N) := zero;
  ep(first) := e;
  transfer(E,N) := false;
  locked(E,N) := false
}
