Inferring and Asserting Distributed System Invariants

https://bitbucket.org/bestchai/dinv

Stewart Grant§, Hendrik Cech¶, Ivan Beschastnikh§
University of British Columbia§, University of Bamberg¶
Distributed Systems are pervasive

- Graph processing
- Stream processing
- Distributed databases
- Failure detectors
- Cluster schedulers
- Version control
- ML frameworks
- Blockchains
- KV stores
- ...
Distributed Systems are Notoriously Difficult to Build

- Concurrency
- No Centralized Clock
- Partial Failure
- Network Variance
Today’s state of the art (building robust dist. sys)

**Verification** - [(verification) IronFleet SOSP’15, VerdiPLDI’15, Chapar POPL’16, (modeling), Lamport et.al SIGOPS’02, Holtzman IEEE TSE’97]

**Bug Detection** - [MODIST NSDI’09, Demi NSDI’16,]

**Runtime Checkers** - [D3S NSDI’18,]

**Tracing** - [PivotTracing SOSP’15, XTrace NSDI’07, Dapper TR’10,]

**Log Analysis** - [ShiViz CACM ‘16]
Today’s state of the art (building robust dist. sys)

**Verification** - [(verification) IronFleet SOSP’15, VerdiPLDI’15, Chapar POPL’16, (modeling), Lamport et.al SIGOPS’02, Holtzman IEEE TSE’97]

**Bug Detection** - [MODIST NSDI’09, Demi NSDI’16,]

**Runtime Checkers** - [D3S NSDI’18,]

**Tracing** - [PivotTracing SOSP’15, XTrace NSDI’07, Dapper TR’10,]

**Log Analysis** - [ShiViz CACM ‘16]
Little work has been done to infer distributed specs

Some notable exceptions

- **CSight ICSE’14**
  - Communicating finite state machines
- **Avenger SRDS’11**
  - Requires enormous manual effort
- **Udon ICSE’15**
  - Requires shared state

None of these can capture stateful properties like:

- **Partitioned Key Space (Memcached):**
  - ∀ nodes i,j keys_i ≠ keys_j
- **Strong Leadership (raft):**
  - ∀ followers i length(log_leader) ≥ length(log_follower_i)
Design goal: handle real distributed systems

Wanted: distributed state invariants

Make the fewest assumptions about the system as possible.

- N nodes
- Message passing
- Lossy, reorderable channels
- Joins and failures
Goal: Infer key correctness and safety properties

Mutual exclusion:

\( \forall \text{nodes } i,j \quad \text{InCritical}_i \rightarrow \neg \text{InCritical}_j \)

Key Partitioning:

\( \forall \text{nodes } i,j \quad \text{keys}_i \neq \text{keys}_j \)
Goal: Infer key correctness and safety properties

Mutual exclusion:

\[ \forall \text{nodes i,j} \quad \text{InCritical}_i \Rightarrow \neg \text{InCritical}_j \]

Key Partitioning:

\[ \forall \text{nodes i,j} \quad \text{keys}_i \neq \text{keys}_j \]

“Distributed State”
Goal: Infer key correctness and safety properties

Mutual exclusion:

\[ \forall \text{nodes } i,j \quad \text{InCritical}_i \rightarrow \neg \text{InCritical}_j \]

Key Partitioning:

\[ \forall \text{nodes } i,j \quad \text{keys}_i \neq \text{keys}_j \]

“Distributed State”
This talk: distributed invariants and Dinv

- Automatic distributed invariant inference (techniques & challenges)
- Runtime checking: distributed assertions
- Evaluation: 4 large scale distributed systems
Capturing Distributed State Automatically

1. Interprocedural Program Slicing
2. Logging Code Injection

Developer adds dump annotations at key program points

Backward slice: code affecting the sent product variable

Variables appearing in the slice: i, n, product

Injected code to log product-affecting vars
Capturing Distributed State **Automatically**

1. Interprocedural Program Slicing
2. Logging Code Injection
Capturing Distributed State Automatically

1. Interprocedural Program Slicing
2. Logging Code Injection

Developer adds `dump` annotations at key program points

Backward slice: code affecting the sent `product` variable

Variables appearing in the slice: `i`, `n`, `product`
Capturing Distributed State **Automatically**

1. Interprocedural Program Slicing
2. Logging Code Injection

Developer adds **dump annotations** at key program points

Backward slice: code affecting the sent **product** variable

Variables appearing in the slice: **i, n, product**

Injected code to log **product-affecting vars**
Capturing Distributed State **Automatically**

1. Interprocedural Program Slicing
2. Logging Code Injection

Node 1

Node 2

Log Relevant Variables
Capturing Distributed State **Automatically**

1. Interprocedural Program Slicing
2. Logging Code Injection
3. Vector Clock Injection

Log Relevant Variables

Send Message (Add vector clock)
Capturing Distributed State **Automatically**

1. Interprocedural Program Slicing
2. Logging Code Injection
3. Vector Clock Injection
Capturing Distributed State **Automatically**

1. Interprocedural Program Slicing
2. Logging Code Injection
3. Vector Clock Injection
Consistent Cuts / Ground States

1. Interprocedural Program Slicing
2. Logging Code Injection
3. Vector Clock Injection

- Log Relevant Variables
- Send Message (Add vector clock)
- Receive Message (Remove vector clock)
Consistent Cuts / Ground States

- Fast Forward

Log Relevant Variables
Send Message (Add vector clock)
Receive Message (Remove vector clock)
Consistent Cuts / Ground States

- Fast Forward.

- Log Relevant Variables
- Send Message (Add vector clock)
- Receive Message (Remove vector clock)
Consistent Cuts / Ground States

- Fast Forward.

- Log Relevant Variables
- Send Message (Add vector clock)
- Receive Message (Remove vector clock)
Consistent Cuts / Ground States

- Fast Forward...

Log Relevant Variables

Send Message (Add vector clock)

Receive Message (Remove vector clock)
Consistent Cuts / Ground States

- Fast Forward....

Log Relevant Variables
Send Message (Add vector clock)
Receive Message (Remove vector clock)
Consistent Cuts / Ground States

- Fast Forward......

- Log Relevant Variables
- Send Message (Add vector clock)
- Receive Message (Remove vector clock)
Consistent Cuts / Ground States

- Fast Forward…….

- Log Relevant Variables
- Send Message (Add vector clock)
- Receive Message (Remove vector clock)
Consistent Cuts / Ground States

- Fast Forward……..

Log Relevant Variables

Send Message (Add vector clock)

Receive Message (Remove vector clock)
Consistent Cuts / Ground States

- **Green lines** mark consistent cuts
  - No messages are in flight
  - Message sent but not received
- **The red line** is not a consistent cut
  - The ping sent by Node 0 happened before the pings receipt on node 1.
Consistent Cuts / Ground States

- Huge number of consistent cuts
Consistent Cuts / Ground States

- Huge number of consistent cuts
- Require sampling heuristic
Consistent Cuts / Ground States

- Huge number of consistent cuts
- Require sampling heuristic
- Ground States: A consistent cut with no in flight messages
Consistent Cuts / Ground States

- Huge number of consistent cuts
- Require sampling heuristic
- Ground States: A consistent cut with no in flight messages
- Dramatically collapses search space
Consistent Cuts / Ground States

- Huge number of consistent cuts
- Require sampling heuristic
- Ground States: A consistent cut with no in flight messages
- Dramatically collapses search space

Ground State sampling used exclusively in evaluation
Reasoning About Global State: State Bucketing
Reasoning About Global State: State Bucketing

Node 1 → Node 2 → Node 3

Execution 1

Node 1
Ping

Node 2
Get Lock

Node 3

Execution 2

Node 1

Node 2
Ping

Get Lock

Node 3

Ack

Node.go.Line 55 :: InCritical = True
Reasoning About Global State: State Bucketing

Execution 1
Node 1 → Ping → Node 2 → Get Lock

Node.go.Line 25 :: InCritical = False

Execution 2
Node 1 → Ping → Node 2 → Get Lock
→ Ack

Node 3
Reasoning About Global State: State Bucketing

Node 1 Ping Node 2 Get Lock Node 3

Execution 1

Node 1 Ping Node 2 Get Lock Node 3

Execution 2

Node.go.Line 15 :: \texttt{InCritical = False}
Reasoning About Global State: State Bucketing
Reasoning About Global State: State Bucketing
Reasoning About Global State: State Bucketing
Reasoning About Global State: State Bucketing
Reasoning About Global State: State Bucketing
Reasoning About Global State: State Bucketing
Reasoning About Global State: State Bucketing
Reasoning About Global State: State Bucketing

Execution 1

Node 1

Node 2

Ping

Get Lock

Execution 2

Node 1

Node 2

Ping

Get Lock

Node 3

Ack
Reasoning About Global State: State Bucketing

Execution 1

Node 1 → Node 2:
- Ping → Get Lock

Execution 2

Node 1 → Node 2:
- Ping → Get Lock

Node 1 → Node 2:
- Ack
Reasoning About Global State: State Bucketing

“Likely” Invariants

Node_3_InCritical == True
Node_2_InCritical != Node_3_InCritical
Node_2_InCritical == Node_1_InCritical
Distributed Asserts

- Distributed asserts enforce invariants at runtime
Distributed Asserts

- Distributed asserts enforce invariants at runtime
- Snapshots are constructed using approximate synchrony
Distributed Asserts

- Distributed asserts enforce invariants at runtime
- Snapshots are constructed using approximate synchrony
Distributed Asserts

- Distributed asserts enforce invariants at runtime
- Snapshots are constructed using approximate synchrony
- Asserter constructs global state by aggregating snapshots
Distributed Asserts

- Distributed asserts enforce invariants at runtime
- Snapshots are constructed using approximate synchrony
- Asserter constructs global state by aggregating snapshots
Evaluated Systems

Etcd: Key-Value store running Raft - 120K LOC

Serf: large scale gossiping failure detector - 6.3K LOC

Taipei-Torrent: Torrent engine written in Go - 5.8K LOC

Groupcache: Memcached written in Go - 1.7K LOC
<table>
<thead>
<tr>
<th>System and Targeted property</th>
<th>Dinv-inferred invariant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raft</td>
<td>∀ follower $i$, len(leader log) $\geq$ len($i$'s log)</td>
<td>All appended log entries must be propagated by the leader</td>
</tr>
<tr>
<td>Raft</td>
<td>∀ nodes $i, j$ if $i$-log$[c] = j$-log$[c] \rightarrow \forall (x \leq c), i$-log$[x] = j$-log$[x]$</td>
<td>If two logs contain an entry with the same index and term, then the logs are identical on all previous entries.</td>
</tr>
<tr>
<td>Raft</td>
<td>If $\exists$ node $i$, s.t $i$ leader, then $\forall j \neq i$, $j$ follower</td>
<td>If a leader exists, then all other nodes are followers.</td>
</tr>
</tbody>
</table>

*Raft: In search of an understandable consensus algorithm, D.Ongaro et. al*
### System and Targeted property | Dinv-inferred invariant | Description
--- | --- | ---
**Raft**  
*Strong Leader principle* | \( \forall \text{ follower } i, \text{ len(leader log)} \geq \text{ len(} i^\text{'}s \text{ log}) \) | All appended log entries must be propagated by the leader

**Raft**  
Log matching | \( \forall \text{ nodes } i, j \text{ if } i\text{-log}[c] = j\text{-log}[c] \rightarrow \forall (x \leq c), i\text{-log}[x] = j\text{-log}[x] \) | If two logs contain an entry with the same index and term, then the logs are identical on all previous entries.

**Raft**  
Leader agreement | If \( \exists \text{ node } i, \text{ s.t } i \text{ leader, than } \forall j \neq i, j \text{ follower} \) | If a leader exists, then all other nodes are followers.

---

Injected Bugs for each invariant caught with assertions

*Raft: In search of an understandable consensus algorithm, D.Ongaro et. al*
Etcd ~ 120K Lines of Code

<table>
<thead>
<tr>
<th>System and Targeted property</th>
<th>Dinv-inferred invariant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Raft</strong></td>
<td><strong>∀</strong> follower $i$, len(leader log) ≥ len($i$’s log)</td>
<td>All appended log entries must be propagated by the leader</td>
</tr>
<tr>
<td><strong>Strong Leader principle</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Raft</strong></td>
<td><strong>∀</strong> nodes $i$, $j$ if $i$-$log[c] = j$-$log[c] → ∀(x ≤ c), i$-$log[x] = j$-$log[x]</td>
<td>If two logs contain an entry with the same index and term, then the logs are identical on all previous entries.</td>
</tr>
<tr>
<td><strong>Log matching</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Raft</strong></td>
<td>If $∃$ node $i$, s.t $i$ leader, than $∀ j ≠ i$, $j$ follower</td>
<td>If a leader exists, then all other nodes are followers.</td>
</tr>
<tr>
<td><strong>Leader agreement</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Injected Bugs for each invariant caught with assertions

See the paper for full system evaluation

*Raft: In search of an understandable consensus algorithm, D.Ongaro et. al*
Limitations and future work

Limitations

- Dinv’s dynamic analysis is incomplete
- Ground state sampling is poor on loosely coupled systems
- Large number of generated invariants

Future work

- Extend analysis to temporal invariants
- Bug Isolation
- Distributed test case generation
- Mutation testing/analysis based on mined invariants
Dinv: Contributions

Analysis for distributed Go systems

- Automatic **distributed state** invariant inference
  - Static identification of distributed state
  - Automatic static instrumentation
  - Post-execution merging of distributed states
- Runtime checking: distributed assertions

Repo: [https://bitbucket.org/bestchai/dinv](https://bitbucket.org/bestchai/dinv)

Demo: [https://www.youtube.com/watch?v=n9fH9ABJ6S4](https://www.youtube.com/watch?v=n9fH9ABJ6S4)