Software Practices
- Gail Murphy
- Gregor Kiczales
- Ron Garcia
- William Bowman
- Reid Holmes

Networks Systems Security
- Mike Feeley
- Bill Aiello
- Margo Seltzer
- Alan Wagner
- Norm Hutchinson

Ivan Beschastnikh
Computer Science
University of British Columbia
Vancouver, Canada
Program analysis for distributed systems

(Bridging gap between design and implementation)

Dinv, Dara, PGo

Ivan Beschastnikh

Vaastav Anand, Hendrik Cech, Renato Costa, Matthew Do, Stewart Grant, Finn Hackett, Brandon Zhang

Networks, Systems and Security Lab
Software Practices Lab
Distributed systems are widely-used

- Distributed systems are widely deployed [1]
  - Graph processing
  - Stream processing
  - Distributed databases
  - Failure detectors
  - Cluster schedulers
  - Version control
  - ML frameworks
  - Blockchains
  - KV stores
  - ...

Cloud systems/apps ecosystem

• Distributed systems are widely deployed [1]

Distributed systems are widely deployed [1]

Google’s data center, Council Bluffs, IA
https://www.google.com/about/datacenters/gallery

Issue 1: Cloud creates costly fate sharing

• Distributed systems are widely deployed [1]

• Failures are very **costly**
  
  • DynamoDB’s outage in 2015 caused downtime on Netflix, Reddit, etc [2]
  
  • S3’s outage in 2017 caused loss of millions of dollars [3]

Issue 2: Distributed systems challenges

“You know you have a distributed system when the crash of a computer you’ve never heard of stops you from getting any work done.” — Leslie Lamport

• Distributed systems are hard to design and build

• Non-deterministic sequence of events

• Processes make decisions based on local state

• A variety of failures
High essential complexity

 Failures can be very costly

 We need to continue to innovate in how we build reliable distributed systems
### 1. Dinv

Spec miner

### 2. Dara

Model checker

### 3. PGo

Compiler

---

Program analysis for distributed systems

---

- **Compiler**
  - Dara
  - Spec miner

---

- **EVALUATION**
  - Implemented for Go 1.7.4 and up.
  - Libraries for Go are to TLA+ translation tool.
  - Since the TLA+ toolbox was called the PlusCal AST and parsing algorithms in the PlusCal select from a set.

**The with statement in PlusCal assigns random values to rithm for Euclid greatest common denominator finding into 8 EVALUATION implemented for Go 1.7.4 and up.**

- **Thesis 2016, 30 April, 2016, Vancouver, BC**

---

- **PGo compiled**
  - Listing 8: The Euclid algorithm in PlusCal that
    
    ```
    var N int
    var v init
    var v_new int
    var u init int
    var u_new int
    var w int
    
    A.log
    B.log
    C.log
    
    A.seq <= B.seq <= C.seq
    
    start
    
    A.log
    B.log
    C.log
    
    A.seq <= B.seq <= C.seq
    
    temp var
    
    u := temp
    
    if (u < v) {
      u := v
      v := u
    }
    
    temp var
    
    v := temp
    
    if (v < w) {
      v := w
      w := v
    }
    
    var v int
    var u int
    var v_init int
    var w int
    
    func main() {
      flag.Parse()
      N := strconv.Atoi(flag.Args()[0])
      for _, w := range pgoutil.Sequence(1, N) { u := 24
        v := w
        v_init = v
        for u := 0; u < v; u = u + v
          if u < v {
            u := v
            v := u
          }
        }
        print <<24, v_init, "have gcd", v>>
      }
    }
    ```

---

- **Verified correct**
  - Violated: counter-example

---

- **∀ nodes, InCritical ≤ 1**

---

- **ICSE 2018**

---

- **Node A**
- **Node B**
- **Node C**

---

- **Go lang**
  - **PlusCal model**
  - **System.go**
  - **Dara**
  - **property**
  - **V**

---

- **UBC**
How these tools empower developers

1. Dinv
Spec miner

How does my system behave?

A.seq ≤ B.seq ≤ C.seq

∀ nodes, InCritical ≤ 1

2. Dara
Model checker

Is my system correct?

Verified correct

3. PGo
Compiler

Can I implement my (correct) system faster?

Bridged gap between design and implementation
First up: distributed spec mining

1. Dinv
Spec miner

[ICSE 2018]

2. Dara
Model checker

\[ A.\text{seq} \leq B.\text{seq} \leq C.\text{seq} \]

∀ nodes, \( \text{InCritical} \leq 1 \)

3. PGo
Compiler

---

```
package main

import "fmt"

type Node int

func main() {
    for v := range pgoutil.Sequence(1, N) {
        if u := 0 {
            if u < v {
                u := u + v
                v := u
            }
        }
        print <<<24, v_init, "have god", v>>>
    }
}
```

---

How does my system behave?
Why distributed spec mining?

Dinv

Spec miner

\[ A.seq \leq B.seq \leq C.seq \]
Why distributed spec mining?

Dinv
Spec miner

Sampler of state of the art in building robust distributed systems:

- **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, CrystalBall NSDI’09, MaceMC NSDI’07]

- **Runtime checkers** [ D3S NSDI’18 ]

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

- **Log analysis** [ Demi NSDI’16, ShiViz CACM ’16 ]
Why distributed spec mining?

Dinv
Spec miner

Sampler of state of the art in building robust distributed systems:

- **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, CrystalBall NSDI’09, MaceMC NSDI’07]
- **Runtime checkers** [ D3S NSDI’18 ]
- **Tracing** [ PivotTracing SOSP’15, XTrace NSDI’07, Dapper TR’16]
- **Log analysis** [ Demi NSDI’16, ShiViz CACM ’16 ]

Require specifications
Why distributed spec mining?

Sampler of state of the art in building robust distributed systems:

- **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, CrystalBall NSDI’09, MaceMC NSDI’07]

- **Runtime checkers** [ D3S NSDI’18 ]

- **Avenger SRDS’11**
  - High manual effort
- **CSight ICSE’14**
  - Temporal model
- **Udon ICSE’15**
  - Multithreading sh-state

Require specifications
Goal: infer key correctness 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 properties

Mutual exclusion:
\[ \forall \text{ nodes, } i, j \; \text{InCritical}_i \rightarrow \neg \text{InCritical}_j \]

Key Partitioning:
\[ \forall \text{ nodes, } i, j \; \text{keys}_i \neq \text{keys}_j \]

Running example
Goal: infer (likely) correctness 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”
What is distributed state anyway?

Distributed state is *information retained in one place that describes something, or is determined by something, somewhere else in the system.*

- John Ousterhout

What is distributed state anyway?

Distributed state is information retained in one place that describes something, or is determined by something, somewhere else in the system.

- John Ousterhout

Examples:

• A table mapping files to hosts that store them

• Request id to identify the last received request

• Public key for a remote server

What is distributed state anyway?

Distributed state is information retained in one place that describes something, or is determined by something, somewhere else in the system.

- John Ousterhout

Observation: Distributed state is one key reason why distributed systems are complex

Dinv: captures distributed state and reveals distributed state runtime properties

Dinv approach: static+dynamic analysis

Static analysis

Dynamic analysis
Dinv static analysis

Instrumentation

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

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

Input Go code → Network usage detector → Vector clock injection → System execution → Consistent cut analysis → Distributed state composition → Daikon

### Instrumentation

- **Input**: Go code
- **Network usage detector**
- **Vector clock injection**
- **System execution**
- **Consistent cut analysis**
- **Distributed state composition**
- **Detected Invariants**

### Mining distributed state

#### Detected Invariants

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

#### Variables

- **product**
- **i**
- **n**

#### Code Snippet

```
1    recv(n)
2    i:= 1
3    sum := 0
4    product := 1
5    for i <= n {
6        sum := sum + 1
7        product := product * i
8        i := i + 1
9    }
10  send(sum)
11  send (product)

1    recv(n)
2    i:= 1
3    sum := 0
4    product := 1
5    for i <= n {
6        sum := sum + 1
7        product := product * i
8        i := i + 1
9    }
10  // @ dump
11  recv(n)
12  i:= 1
13  point = {[i,n,product],vcclock}
14  Log(point)
15  send (product)
```

#### Developer annotations

- **dump annotations at key program points**
- **Backward slice: code affecting the sent product variable**
- **Injected code to log product-affecting vars**
Dinv static analysis

Instrumentation

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

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

```
1  recv(n)
2    i:= 1
3    sum := 0
4    product := 1
5    for i <= n {
6        sum := sum + 1
7        product := product * i
8        i := i + 1
9    }
10   send(sum)
11  // @ dump
12  send (product)
```

```
1  recv(n)
2    i:= 1
3
4    product := 1
5    for i <= n {
6        product := product * i
7        i := i + 1
8    }
9    // @ dump
10
11  send (product)
```

```
1  recv(n)
2    i:= 1
3    sum := 0
4    product := 1
5    for i <= n {
6        sum := sum + 1
7        product := product * i
8        i := i + 1
9    }
10   send(sum)
11  point = {i,n,product},vclock
12  Log(point)
13  send (product)
```
Dinv static analysis

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

```
1    recv(n)
2    i:= 1
3    sum := 0
4    product := 1
5    for i <= n {
6        sum := sum + 1
7        product := product * i
8        i := i + 1
9    }
10  send(sum)
11  // @ dump
12  send (product)
```

```
1    recv(n)
2    i:= 1
3    sum := 0
4    product := 1
5    for i <= n {
6        sum := sum + 1
7        product := product * i
8        i := i + 1
9    }
10  send(sum)
11  // @ dump
12  send (product)
```

```
1    recv(n)
2    i:= 1
3    sum := 0
4    product := 1
5    for i <= n {
6        sum := sum + 1
7        product := product * i
8        i := i + 1
9    }
10  send(sum)
11  // @ dump
12  send (product)
13  point = {
14      [i,n,product],vclock
15  }
16  Log(point)
17  send (product)
```
Dinv static analysis

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

\[
\text{point} = \{[i,n,product],v\text{clock}\}
\]

Log(point)

\[
\text{point} = \{[x,y,z],v\text{clock}\}
\]

Log(point)
Dinv static analysis

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

Input
Go code

Network usage detector → Vector clock injection → System execution → Consistent cut analysis → Distributed state composition → Daikon

Instrumentation

Mining distributed state

Detecting invariants

Time

Log Relevant Variables

Send Message (Add vector clock)
Dinv static analysis

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

1. Interprocedural Program Slicing
2. Logging Code Injection
3. Vector Clock Injection
Run the system + collect traces

- Instrumentation
  - Network usage detector
  - Vector clock injection

- System execution
  - System execution

- Consistent cut analysis
  - Distributed state composition

- Daikon

- Detected Invariants

- Input
  - Go code

- System
  - Traces
    - Partial order
    - Concrete state values
Reasoning about global state

1. Consistent Cuts
2. Ground States
3. State Bucketing
Reasoning about global state

1. Consistent Cuts
2. Ground States
3. State Bucketing

Node.go.Line 55 :: InCritical = True
Reasoning about global state

Instrumentation
- Network usage detector
  - Vector clock injection
  - System execution

System execution
- Consistent cut analysis
- Distributed state composition

Mining distributed state
1. Consistent Cuts
2. Ground States
3. State Bucketing

Node.go.Line 25 :: InCritical = False
Reasoning about global state

1. Consistent Cuts
2. Ground States
3. State Bucketing

Node.go.Line 15 :: InCritical = False
Reasoning about global state

Instrumentation

- Network usage detector
- Vector clock injection

System execution

- System execution
- Consistent cut analysis
- Distributed state composition

Detected Invariants

1. Consistent Cuts
2. Ground States
3. State Bucketing

Matching logging locations

Node.go.Line 55 :: InCritical = True

Node.go.Line 55 :: InCritical = True
Reasoning about global state

Input Go code
Network usage detector
Vector clock injection
System execution
Consistent cut analysis
Distributed state composition
Daikon

Instrumentation
System execution
Mining distributed state
Detecting invariants

1. Consistent Cuts
2. Ground States
3. State Bucketing
Reasoning about global state

1. Consistent Cuts
2. Ground States
3. State Bucketing

Matching consistent state cuts
Reasoning about global state

Mining distributed state

1. Consistent Cuts
2. Ground States
3. State Bucketing
Reasoning about global state

1. Consistent Cuts
2. Ground States
3. State Bucketing
Reasoning about global state

1. Consistent Cuts
2. Ground States
3. State Bucketing
Reasoning about global state

Instrumentation

Network usage detector → Vector clock injection → System execution → Consistent cut analysis → Distributed state composition → Daikon

**Mining distributed state**

1. Consistent Cuts
2. Ground States
3. State Bucketing

**Scalability:** only process "ground states" (no msgs in flight)
Reasoning about global state

Instrumentation
- Network usage detector
- Vector clock injection

System execution
- System execution
- Consistent cut analysis
- Distributed state composition

Mining distributed state
1. Consistent Cuts
2. Ground States
3. State Bucketing

Detected Invariants
- Daikon
Reasoning about global state

Instrumentation

Network usage detector → Vector clock injection → System execution

System execution

Consistent cut analysis → Distributed state composition

Mining distributed state

Detected Invariants

Input
Go code

1. Consistent Cuts
2. Ground States
3. State Bucketing
From concrete values to abstract relations

Instrumentation
- Network usage detector
- Vector clock injection

System execution
- System execution
- Consistent cut analysis

Mining distributed state
- Distributed state composition
- Daikon

Detecting invariants

"likely" invariants

Node_3_InCritical == True
Node_2_InCritical != Node_3_InCritical
Node_2_InCritical == Node_1_InCritical
Enforcement: distributed assertions

- Distributed **probabilistic** asserts: cheap runtime enforcement of invariants
- Snapshots are constructed using approximate synchrony
- Asserter constructs global state by aggregating snapshots (discards if inconsistent)
Dinv evaluation

- **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
Dinv evaluation

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
**System and Targeted property** | **Dinv-inferred invariant** | **Description**
--- | --- | ---
Raft  
Strong Leader principle | ∀ follower \(i\), len(leader log) ≥ len(i’s log) | All appended log entries must be propagated by the leader
Raft  
Log matching | ∀ nodes \(i, j\) if \(i\)-log[\(c\)] = \(j\)-log[\(c\)]  
→ ∀(\(x ≤ c\)), \(i\)-log[\(x\)] = \(j\)-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 \(∃\) node \(i\), s.t \(i\) leader, than ∀ \(j ≠ i, j\) follower | If a leader exists, then all other nodes are followers.

- **Dinv** Detected all of RAFT core correctness properties
### Raft invariant

<table>
<thead>
<tr>
<th>Strong leadership</th>
<th>Leadership agreement</th>
<th>Log matching</th>
</tr>
</thead>
</table>

**Constructive and injected silent bugs for each invariant into a running etcd system**
Probabilistic assertions

<table>
<thead>
<tr>
<th>Raft invariant</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong leadership</td>
<td>11</td>
</tr>
<tr>
<td>Leadership agreement</td>
<td>13</td>
</tr>
<tr>
<td>Log matching</td>
<td>72</td>
</tr>
</tbody>
</table>

LOC in assertion
(developer must write)
### Probabilistic assertions

<table>
<thead>
<tr>
<th>Raft invariant</th>
<th>LOC</th>
<th>P=1.0</th>
<th>P=0.1</th>
<th>P=0.01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong leadership</td>
<td>11</td>
<td>0.07</td>
<td>0.05</td>
<td>2.96</td>
</tr>
<tr>
<td>Leadership agreement</td>
<td>13</td>
<td>0.36</td>
<td>0.34</td>
<td>6.75</td>
</tr>
<tr>
<td>Log matching</td>
<td>72</td>
<td>2.22</td>
<td>4.35</td>
<td>6.07</td>
</tr>
</tbody>
</table>

Time (seconds) to catch an injected silent bug for different assert probabilities

See our ICSE 2018 paper for more evaluation details

**Inferring and Asserting Distributed System Invariants**
Stewart Grant, Hendrik Cech, Ivan Beschastnikh.
Dinv limitations and future work

Limitations

• Dinv’s dynamic analysis is incomplete
• Ground state sampling is poor on loosely coupled systems
• Large number of output invariants (many uninteresting ones!)
• Targets safety properties (cannot infer liveness properties)

Future work

• Root cause analysis\impact analysis\etc
• Distributed test case generation
• Extend analysis to temporal invariants
Ongoing: distributed model checking

1. Dinv
Spec miner

2. Dara
Model checker

3. PGo
Compiler

---

Thesis 2016, 30 April, 2016, Vancouver, BC

---

PGo is a simplified Java version of the built-in Go AST in Go. To TLA+ translation tool. Since the TLA+ toolbox was called the PlusCal AST and parsing algorithms in the PlusCal.
PGo is built using existing code for TLA+ toolbox, specifically the clause

Listing 7: Corresponding Go to an PlusCal

PGo also uses an external library for sets for Go from

Listing 8: The Euclid algorithm in PlusCal that

PGo compiled

Listing 9: The Go code that PGo compiled

∀ nodes, InCritical ≤ 1

Is my system correct?
Model checking (MC)

- “Exhaustive testing”

- Explore the state space of a system w.r.t some **model**

- **Check** predicate at each state (safety property) for violation

- Violation is a path = bug in the model: output to developer

- Main challenge: *state space explosion*
Concrete (implementation-level) MC

- The implementation is the model
- No false positives: all found bugs are real
- Huge (concrete) state space
- Engineering complexity
Trade-offs in model checking (MC)

**Concrete** (implementation-level) MC

- The implementation is the model
- No false positives: all found bugs are real
- Huge (concrete) state space
- Engineering complexity

**Abstract** (model-based) MC

- Limited state space
- Several available checkers (e.g., SPIN, TLC)
- Must develop a separate model of your system
- Opens the door for false positives

[ MODIST NSDI'09, Demi NSDI'16 ]

[ IronFleet SOSP'15, VerdiPLDI'15, Chapar POPL'16, (modeling), Lamport et.al SIGOPS'02, Holtzman IEEE TSE'97 ]
Trade-offs in model checking (MC)

**Concrete** (implementation-level) MC
- The implementation is the model
- No false positives: all found bugs are real
- Huge (concrete) state space
- Engineering complexity

**Abstract** (model-based) MC
- Limited state space
- Several available checkers (e.g., SPIN, TLC)
- Must develop a separate model of your system
- Opens the door for false positives

Can we get the best of both worlds?
Idea 1: use implementation to bootstrap the abstract model/MC

- Use concrete MC to generate traces of the system
- Use traces to infer an abstract model of the system
- Model check abstract model for violations
Implementation is the model oracle

Idea 2: use implementation to check for abstract false positives

- Map each abstract violation into a concrete violation (replay)
  - Attempt to reproduce the abstract execution by replaying it on the actual system

  ➡️ **Bug reproduced**: bug found, show trace to user

  ➡️ **Bug not reproduced**: update the abstract model to exclude the non-buggy path

Idea 3: refine the abstract model with counter-examples

- False positive are counter-examples: use them to improve model
Implementation is the model oracle

Idea 2: use implementation to check for abstract false positives

- Map each abstract violation into a concrete violation (replay)
  - Attempt to reproduce the abstract execution by replaying it on the actual system

  ➡ Bug reproduced: bug found, show trace to user

  ➡ Bug not reproduced: update the abstract model to exclude the non-buggy path

Idea 3: refine the abstract model with counter-examples

- False positive are counter-examples: use them to improve model

Use the (faster) abstract model for the bulk of the checking
High-level view of the approach

Feasible system behaviors
Generate traces using the concrete MC: exhaustive.. but bounded/incomplete
Feasible system behaviors

Inferred model

Traces

Infer abstract model that generalizes
High-level view of the approach

Feasible system behaviors

Inferred model

Traces

Infer abstract model that generalizes

good generalization

insufficient generalization

incorrect generalization
High-level view of the approach

Update mode to remove infeasible behavior

incorrect generalization
Key challenge: concrete model checker

- Demonstrated by MODIST [NSDI’09]
- Trap all non-determinism across all nodes in the distributed system
- Handle unmodified, complex, code

![Diagram showing the structure of the model checker system]

- Instrumented Program
- Modified Go Runtime
- Communication Layer
- OS (Linux)
• Built up the theory linking concrete and abstract model checkers (abstract checker is SPIN)

• Developing the blackbox MC for Go-based systems based on MODIST [NSDI’09]

• Concrete-abstract loop works on simple apps (dining philosophers)
Ongoing: compiling distributed systems

1. Dinv
Spec miner

2. Dara
Model checker

3. PGo
Compiler

The code outputed is also formatted for readability.

PGo is built using existing code for TLA+ toolbox, specifying from a set.

6.5 Additional Pluscal syntax semantics

Listing 7: Corresponding Go to an PlusCal

Listing 8: The Euclid algorithm in PlusCal that

Can I implement my (correct) system faster?
Existing verification approaches

- **Verdi** reduces **proof** burden by automatically handling failures [PLDI’15]

- **IronFleet** provides a framework to write specifications and implementations [SOSP’15]

- **MODIST** checks the **implementation** rather than a specification [NSDI’09]
Existing verification approaches

• **Verdi** reduces proof burden by automatically handling failures [PLDI’15]

• **IronFleet** provides a framework to write specifications and implementations [SOSP’15]

• **MODIST** checks the implementation rather than a specification [NSDI’09]

Takes a long time to prove/check, or require a lot of work from developers
PGo: Compiling Distributed Systems

Making writing of verified distributed systems easier

Developer writes specification
PGo compiles it to a matching implementation
Source is compiled
Verified Distributed System!

PGo: Compiling Distributed Systems

Making writing of verified distributed systems easier

Developer writes specification

PGo compiles it to a matching implementation

Source is compiled

Verified Distributed System!

PGo: Compiling Distributed Systems

Making writing of *verified* distributed systems easier

Developer writes **specification**

PGo compiles it to a **matching implementation**

Source is compiled

Verified Distributed System!

Transition from *design* (specification) to *implementation* is **automated**

---

PGo Workflow: (1) Example System

Round-Robin Resource Sharing

Developer writes specification

Shared
PGo Workflow: (1) PlusCal Spec

Developer writes specification

1

2

token

N

counter
PGo Workflow: (1) PlusCal Spec

**CONSTANTS** procs, iters
CONSTANTS procs, iters

(*
-- algorithm RoundRobin {
  variables counter = 0,
  token = 0;
}
PGo Workflow: (1) PlusCal Spec

```
CONSTANTS procs, iters
(*
   -- algorithm RoundRobin {
      variables counter = 0,
      token = 0;
   fair process (P \in 0..procs-1)
   variable i = 0;
```
CONSTRANTS procs, iters
(*
-- algorithm RoundRobin {
    variables counter = 0,
    token = 0;

    fair process (P \in 0..procs-1)
    variable i = 0;

    {
        w: while ( i < iters) {
            inc: await token = self;
            counter := counter + 1;
            token := (self + 1) % procs;
            i := i + 1;
        }
    }
}
CONSTANTS procs, iters
(*
-- algorithm RoundRobin {
    variables counter = 0,
    token = 0;

    fair process (P \in 0..procs-1)
    variable i = 0;

    {
        w: while (i < iters) {
            inc: await token = self;
            counter := counter + 1;
            token := (self + 1) % procs;
            i := i + 1;
        }
    }
}
CONSTANTS procs, iters
(*
-- algorithm RoundRobin {
    variables counter = 0,
    token = 0;

    fair process (P \in 0..procs-1)

    variable i = 0;

    {
    w: while (i < iters) {
        inc: await token = self;
        counter := counter + 1;
        token := (self + 1) % procs;
        i := i + 1;
    }
}}
CONSTANTS procs, iters

(*
-- algorithm RoundRobin {
  variables counter = 0,
  token = 0;

  fair process (P \in 0..procs-1)
  variable i = 0;

  {
    w: while ( i < iters) {
      inc: await token = self;
      counter := counter + 1;
      token := (self + 1) % procs;
      i := i + 1;
    }
  }
}
CONSTANTS procs, iters

(*

-- algorithm RoundRobin {
    variables counter = 0,
    token = 0;

    fair process (P \in 0..procs-1)
    variable i = 0;
    {
        w: while (i < iters) {
            inc: await token = self;
            counter := counter + 1;
            token := (self + 1) % procs;
            i := i + 1;
        }
    }
}
**Properties of our System**

**Invariants**

- Token is within bounds: \( \text{token} \in 0..\text{procs-1} \)

**Properties**

- Counter Converges: (counter = procs * iters)
- Termination \( \Rightarrow \)
- Processes: \( \forall p \in \text{ProcSet} : \langle (\text{token} = p) \)
Model Checked with TLC!

Model Checking Results

- General
  - Start time: Fri May 04 01:45:30 PDT 2018
  - End time: Fri May 04 01:45:37 PDT 2018
  - TLC mode: Breadth-first search
  - Last checkpoint time: 
  - Current status: Not running
  - Errors detected: No errors

Developer writes specification
• **counter is global**: semantics need to be maintained
  - Runtime manages state across processes

• **Labels are atomic**
  - Processes coordinate access to atomic blocks

• **High-level concepts such as** `await`
  - Lock and check predicate

```go
fair process (P \ in 0..procs-1)
variable i = 0;
{
  w: while (i < iters) {
    inc: await token = self;
    counter := counter + 1;
    token := (self + 1) % procs;
    i := i + 1;
  }
}
```
PGo Workflow: (2) Compilation

- **counter is global:** semantics need to be maintained
  - Runtime manages state across processes
- **Labels are atomic**
  - Processes coordinate access to atomic blocks
- **High-level concepts such as await**
  - Lock and check predicate

fair process (P \in 0..procs-1)
variable i = 0;
{
  w: while (i < iters) {
    inc: await token = self;
    counter := counter + 1;
    token := (self + 1) % procs;
    i := i + 1;
  }
}
counter is global: semantics need to be maintained
- Runtime manages state across processes

Labels are atomic
- Processes coordinate access to atomic blocks

High-level concepts such as await
- Lock and check predicate

```
fair process (P \in 0..procs-1) variable i = 0;
{
    w: while ( i < iters) {
        inc: await token = self;
        counter := counter + 1;
        token := (self + 1) % procs;
        i := i + 1;
    }
}
```
counter is global: semantics need to be maintained
- Runtime manages state across processes

Labels are atomic
- Processes coordinate access to atomic blocks

High-level concepts such as await
- Lock and check predicate

```
fair process (P \in 0..procs-1)
variable i = 0;
{
    w: while (i < iters) {
        inc: await token = self;
        counter := counter + 1;
        token := (self + 1) % procs;
        i := i + 1;
    }
}
```
Generates Go code can run as any of the processes defined in PlusCal

$ ./counter
Usage: ./counter process(argument) ip:port

$ ./counter 'P(1)' 192.168.1.80:2222
• PGo is currently able to compile **concurrent** and **distributed** systems

• Support for different strategies to deal with **global state** in a distributed system

• Compiles simple distributed applications
  
  • Example: ~30 lines of PlusCal generates ~80 lines of Go source code
- Support a larger subset of PlusCal/TLA+

- Generating distributed systems that are fault tolerant

- Make it easy for developers to change generated code (without compromising safety)
• Specifications are very high level: not everything can be compiled efficiently

• May require developers to insert annotations when PGo cannot infer required information (e.g., types)

• Both the PGo compiler and the associated runtime need to be trusted in order to claim correctness
Program analysis for distributed systems

1. Dinv
Spec miner
https://bitbucket.org/bestchai/dinv

2. Dara
Model checker

∀ nodes, InCritical ≤ 1

3. PGo
Compiler
https://github.com/UBC-NSS/pgo

Bridging gap between design and implementation
Backup slides
Dinv runtime overhead

<table>
<thead>
<tr>
<th>Number of annotations</th>
<th>Executed annotations</th>
<th>Log size (MB)</th>
<th>Runtime (s)</th>
<th>Runtime overhead %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.66</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2.8K</td>
<td>3.2</td>
<td>2.70</td>
<td>1.5</td>
</tr>
<tr>
<td>2</td>
<td>5.6K</td>
<td>4.3</td>
<td>2.77</td>
<td>4.0</td>
</tr>
<tr>
<td>5</td>
<td>14K</td>
<td>9.7</td>
<td>3.01</td>
<td>12.9</td>
</tr>
<tr>
<td>10</td>
<td>28K</td>
<td>18.0</td>
<td>3.31</td>
<td>24.3</td>
</tr>
<tr>
<td>30</td>
<td>85K</td>
<td>51.7</td>
<td>4.48</td>
<td>68.0</td>
</tr>
<tr>
<td>100</td>
<td>261K</td>
<td>167.9</td>
<td>7.66</td>
<td>187.5</td>
</tr>
</tbody>
</table>

- YCSB-A workload, 3 nodes
- 1 logging statement runtime ~ 20μs
- Static instrumentation negligible
Static instrumentation negligible logging statement runtime ~ 20 µs with a round trip time of 0.05ms while running etcd with 1 second nodes in a 4 node cluster. At 5 nodes and above the bandwidth heartbeats and caused a reduction that adding vector clocks to Raft slowed down the broadcast of nodes was aggregated together for these measurements. We found above while varying the number of nodes. The bandwidths of all overhead in a real system we executed etcd Raft using the setup of messages:

- The number of interacting nodes in an execution and the number clock timestamp. The overhead of vector clocks is a product of integers: one to identify the node, and the other is the node's logi-head. Each entry in Dinv's vector clocks timestamp has two 32 bit integers per node before perturbing the system.

Times we can introduce approximately 50K logging statements that all asserts found the bugs, but they took longer with lower timeouts we can introduce approximately 50K logging statements. In our local area network logging statement runtime was 4.7s. In practice just two annotations were su-Inferring and Asserting Distributed System Invariants ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

tial to analyzing distributed systems with a small number of nodes. The greatest slowdown was incurred when asserts were placed at intervals in increments of 30s. Results in Table 5 show that Dinv's analysis performs with regard to the length of execution, we could perform 51200 IOPS. Below we measure the end-to-end latency of client requests to the etcd cluster.

Table 5: Generated Dinv log size and Dinv's dynamic analy-

<table>
<thead>
<tr>
<th>Number of annotations</th>
<th>Executed annotations</th>
<th>Log size (MB)</th>
<th>Runtime (s)</th>
<th>Runtime overhead %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.66</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2.8K</td>
<td>3.2</td>
<td>2.70</td>
<td>1.5</td>
</tr>
<tr>
<td>2</td>
<td>5.6K</td>
<td>4.3</td>
<td>2.77</td>
<td>4.0</td>
</tr>
<tr>
<td>5</td>
<td>14K</td>
<td>9.7</td>
<td>3.01</td>
<td>12.9</td>
</tr>
<tr>
<td>10</td>
<td>28K</td>
<td>18.0</td>
<td>3.31</td>
<td>24.3</td>
</tr>
<tr>
<td>30</td>
<td>85K</td>
<td>51.7</td>
<td>4.48</td>
<td>68.0</td>
</tr>
<tr>
<td>100</td>
<td>261K</td>
<td>167.9</td>
<td>7.66</td>
<td>187.5</td>
</tr>
</tbody>
</table>

- YCSB-A workload, 3 nodes
- 1 logging statement runtime ~ 20 µs
- Static instrumentation negligible

All Raft invariants can be detected with just two annotations
with a round trip time of 0.05ms while running etcd with 1 second nodes in a 4 node cluster. At 5 nodes and above the bandwidth heartbeats and caused a nodes was aggregated together for these measurements. We found above while varying the number of nodes. The bandwidths of all overhead in a real system we executed etcd Raft using the setup of messages:

- the number of interacting nodes in an execution and the number
cal clock timestamp. The overhead of vector clocks is a product of
head. Each entry in Dinv's vector clocks timestamp has two 32 bit
per node before perturbing the system.

- timeouts we can introduce approximately 50K logging statements
detect the Raft invariants. The average execution time of a single
and runtime. In practice just two annotations were su
a linear relationship between the number of logging statements
run 3 times and we averaged the total running time. Table 4 shows
 ging statements, each one logging 7 variables. We benchmarked
system. We instrumented etcd Raft with increasing number of log-
use) runtime was 4.7s.

- increased slightly to 3.2s. At 64K annotations (far beyond practical
time remained constant at 3s until 4K annotations at which point it
ing counts of randomly located
sands of variables. We measured instrumentation time with increas-
matic analysis time exponential in the number of nodes. We measured analysis
due to the exponential growth of partial orderings our analysis
grew linearly with an overhead of 1KB/s for 5 nodes and

greater slowdown was incurred when asserts were placed at
2.5x, and P=0.01 reduced it further to 1.02x. Leader agreement and
bottleneck program points. For example, asserting strong leadership
response times.

- experiments with probabilistic asserts with two probabilities: P=0.1
was exercised at 3 load levels: 100, 150, and 200 client request per

- to wait for multiple asserts to execute. Using P=0.1 reduced this to
exponentially with the number of nodes. We measured analysis
grows
cient, we analyzed etcd Raft and Groupcache. We exercised them by
tion, we analyzed etcd Raft and GroupCache (GCache).

- a linear relationship between the number of system run times, for two sys-

- to measure how analysis time is a

- Log size + analysis time linear in sys runtime

- Can be done offline + parallelized