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 deployed [1]

- Graph processing
- Stream processing
- Distributed databases
- Failure detectors
- Cluster schedulers
- Version control
- ML frameworks
- Blockchains
- KV stores
- ...

• Distributed systems are widely deployed [1]

• Distributed systems are widely deployed [1]

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]


“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**
Overall: High essential complexity

Failures can be very costly

Concurrency
No central clock

Partial failures
Perf variation

We need to continue to innovate in how we build reliable distributed systems
Program analysis for distributed systems

1. Dinv
Spec miner

2. Dara
Model checker

3. PGo
Compiler

---

```
--algorithm Euclid { \var \Go \arg N }\Go
  \Go{ \var v }\Go
  \Go{ \var v_init }\Go

\var u \in \{24\}
\var v \in [1..N]
\var v_init \in \{v\}

while \!(u > 0) { 
  if \!(u < v) {
    u := v \land v := u;
  ;
  u := u \land v;
  };
}
print <<24, v_init, "have gcd", v>>, 
```

```
func main() {
  flag.Parse()
  N_ := strconv.Atoi(flag.Args()[0])
  for _v := range pgoutil.Sequence[1, N] {
    u = 24
    v_init = v
    for u != 0 {
      if u < v {
        u_new := u
        v_new := u
        u = u_new
        v = v_new
      };
      u := u \land v;
    }
    fmt.Printf("%v have gcd %v\n\", v, v_init, v)
  }
}
```
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?

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

1. Dinv
Spec miner

2. Dara
Model checker

3. PGo
Compiler

How does my system behave?

∀ nodes, InCritical ≤ 1

---

@PGo
func main() {  
flag.Parse()  
R := strconv.Atoi(flag.Args()[0])  
for u, v := range pgoutil.Sequence(1, R) {  
u := 24  
v_init := v  
for u := 0;  
if u < v {  
u_new := u  
v_new := v  
u = u_new  
v = v_new  
}  
u := u - v  
}  
fmt.Println("24 %v have gcd %v", v_init, v)  
}

--algorithm Euclid (\roc@PGo arg int N @PGo  
(\roc@PGo v init u @PGo  
@PGo v init v @PGo  
@PGo v init v init @PGo  
\roc))  
variables u = 24;  
v \in 1..N;  
v_init = v;  
{  
while (u ≠ 0) {  
if (u < v) {  
u := v || v := u;  
}  
{  
}  
print <<<24, v_init, "have gcd", v>>>  
}  

---
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: Bagpipe OOPSLA’16, IronFleet SOSP’15, Verdi PLDI’15, Chapar POPL’16; Modeling: Lamport et.al SIGOPS’02, Holtzman IEEE TSE’97]

- **Bug detection** [SAMC OSDI’14, 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** [Pensieve SOSP’17, 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: Bagpipe OOPSLA’16, IronFleet SOSP’15, Verdi PLDI’15, Chapar POPL’16; Modeling: Lamport et.al SIGOPS’02, Holtzman IEEE TSE’97]

- **Bug detection** [SAMC OSDI’14, 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** [Pensieve SOSP’17, 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: Bagpipe OOPSLA’16, IronFleet SOSP’15, Verdi PLDI’15, Chapar POPL’16; Modeling: Lamport et.al SIGOPS’02, Holtzman IEEE TSE’97]

- **Bug detection** [SAMC OSDI’14, MODIST NSDI’09, CrystalBall NSDI’09, MaceMC NSDI’07]

- **Runtime checkers** [D3S NSDI’18]

Require specifications

- Avenger SRDS’11
  - High manual effort
- CSight ICSE’14
  - Temporal model
- Udon ICSE’15
  - Multithreaded sh-state
Goal: infer 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 correctness properties

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

Key Partitioning:
\[ \forall \text{ nodes, } i, j \, \text{keys}_i \neq \text{keys}_j \]
Goal: infer correctness properties

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

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

Running example
Mutual exclusion:
\[ \forall \text{nodes, } i, j \quad \text{InCritical}_i \implies \neg \text{InCritical}_j \]

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

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

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

Input
Go code

Network usage detector → Vector clock injection

System execution → Consistent cut analysis → Distributed state composition → Daikon

Detected Invariants

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>recv(n)</td>
</tr>
<tr>
<td>2</td>
<td>i := 1</td>
</tr>
<tr>
<td>3</td>
<td>sum := 0</td>
</tr>
<tr>
<td>4</td>
<td>product := 1</td>
</tr>
</tbody>
</table>
| 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  Log(point) 14  point = \{i,n,product,vclock\} 15  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

```
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)
```

Backward slice: code affecting the sent `product` variable

```
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)
```

Variables appearing in the slice: `i`, `n`, `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)
```

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  point = {[i,n,product],vclock}
12  Log(point)
13  send (product)
```
Dinv static analysis

Instrumentation

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

Point = \{[i,n,product],vclock\}
Log(point)

Point = \{[x,y,z],vclock\}
Log(point)
Dinv static analysis

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

Instrumentation

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

Input

Go code

Detecting invariants

System execution

Minning distributed state

Detected Invariants

System

Partial order

Traces

Concrete state values
Reasoning about global state

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

- System execution
  - Consistent cut analysis
  - Distributed state composition

- Detecting invariants
  - Daikon

Mining distributed state

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

Input

Go code

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

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

Instrumentation

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

Node.go.Line 25 :: \textbf{InCritical} = False
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

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

Instrumentation

- Network usage detector
- Vector clock injection

System execution

- Consistent cut analysis
- Distributed state composition

Mining distributed state

- Consistent Cuts
- Ground States
- State Bucketing

Detected Invariants

Matching logging locations
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

Detecting invariants

Mining distributed state

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

Matching consistent state cuts
Reasoning about global state

Instrumentation

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

Detecting invariants

Mining distributed state

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

Instrumentation

System execution

Mining distributed state

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

**Instrumentation**
- Network usage detector
- Vector clock injection

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

**Mining distributed state**
- Consistent Cuts
- Ground States
- State Bucketing

**Detected Invariants**
- Daikon

Input
- Go code

Execution 1
- Node 1
- Node 2
- Node 3
- Ping
- Get Lock

Execution 2
- Node 1
- Node 2
- Node 3
- Ping
- Get Lock
- Ack
Reasoning about global state

**Instrumentation**
- Network usage detector
- Vector clock injection

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

**Mining distributed state**
- Detected Invariants

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
- Consistent cut analysis
- Distributed state composition

Mining distributed 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 | Detected Invariants |

Input
Go code

Mining distributed state

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

Instrumentation

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

Detecting invariants

Input Go code

System execution

Mining distributed state

Daikon tool

“likely” invariants

Node_3_InCritical == True
Node_2_InCritical != Node_3_InCritical
Node_2_InCritical == Node_1_InCritical

[Ernst et al. TSE 01]
• Distributed probabilistic asserts: cheap runtime enforcement of invariants

• Snapshots are constructed using approximate synchrony

• Asserter constructs global state for checking by aggregating snapshots (discards states 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
<table>
<thead>
<tr>
<th>Dinv-inferred invariant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>∀ 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>∀ nodes (i, j) if (i)-log[(c)] = (j)-log[(c)] → ∀((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>If (\exists) node (i), s.t (i) leader, than ∀ (j \neq i), (j) follower</td>
<td>If a leader exists, then all other nodes are followers.</td>
</tr>
</tbody>
</table>

- **Dinv detected all key RAFT correctness properties**
  - Just 2 annotations sufficient to detect all invs
  - Traces from YCSB-A workload generate enough diversity
### Raft invariant

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

**Constructed 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.
Probabilistic assertions

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 (requires skill to narrow down)
• Targets safety properties (cannot infer liveness properties)

Future work

• Root cause analysis\impact analysis\etc
• Distributed test case generation
• Extend analysis to temporal invariants
3. PGo
Compiler

2. Dara
Model checker

∀ nodes, InCritical ≤ 1
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
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

[ SAMC OSDI'14, MODIST NSDI'09, Demi NSDI'16 ]
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
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 1: use implementation to bootstrap the abstract model/MC

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**: abstract false positive
Implementation is the model oracle

**Idea 1: use implementation to bootstrap the abstract model/MC**

**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: abstract false positive

**Idea 3: refine the abstract model with counter-examples**

- False positive are counter-examples: use them to improve model
- Update the abstract model to exclude the non-buggy path
Implementation is the model oracle

Idea 1: use implementation to bootstrap the abstract model/MC

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

**Key: use the (faster) abstract model for the bulk of the checking**

Idea 3: refine the abstract model with counter-examples

- False positive are counter-examples: use them to improve model
- Update the abstract model to exclude the non-buggy path
Concrete traces → Abstract model
High-level view of the approach

Feasible system behaviors
Generate traces using the concrete MC: exhaustive.. but bounded/incomplete
High-level view of the approach

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

Feasible system behaviors

Inferred model

Traces

Infer abstract model that generalizes

insufficient generalization

good generalization

incorrect generalization
High-level view of the approach

Feasible system behaviors

Inferred model

Traces

Update mode to remove infeasible behavior

incorrect generalization

Lots of RW in formal methods, e.g., CEGAR, Abstract Interpretation
Key challenge: concrete model checker

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

```
<table>
<thead>
<tr>
<th>Unmodified Program</th>
<th>Unmodified Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified Go Runtime</td>
<td>Modified Go Runtime</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Communication Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS (Linux)</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Global Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure Simulation</td>
</tr>
<tr>
<td>Virtual Clock</td>
</tr>
<tr>
<td>Global Assertions</td>
</tr>
<tr>
<td>GoRoutine State</td>
</tr>
<tr>
<td>Abstract Schedule</td>
</tr>
</tbody>
</table>
```
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)

Current prototype is ~6K LOC
1. Dinv
Spec miner

2. Dara
Model checker

∀ nodes, InCritical ≤ 1

3. PGo
Compiler

Ongoing: compiling distributed systems

18
17
11
10
written in Java8, PGo is implemented in Java8. The Go AST will replace with statements with a call to TLA+ translation tool. Since the TLA+ toolbox was built using existing code for TLA+ toolbox, specifically the PlusCal AST and parsing algorithms in the PlusCal model.

PGo is able to convert to an PlusCal algorithm for Euclid greatest common denominator finding into a loop, and filled the rest of the body like the loop body. In the above Euclidean algorithm, PGo has successfully compiled a single threaded PlusCal algorithm: $\text{Euclid}(u, v) = \text{Euclid}(\text{if } u < v \text{ then } v \text{ else } u, u - v)$.

Through unit tests, and sample compilations, PGo is able to verify correct semantics for variable assignment. PGo correctly handled assignments of the type $u := v$ or $v := u$ in the loop body.

8.2 Can PGo maintain PlusCal syntax?

8.3 Can PGo handle constants and variables?

8.4 Can PGo infer types from PlusCal?

32
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10 

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

Developer writes specification

PGo compiles it to a matching implementation

Source is compiled

Verified Distributed System!

Making writing of verified distributed systems easier

Transition from design (specification) to implementation is automated

PGo Workflow: (1) Example System

Round-Robin Resource Sharing

Developer writes specification

Shared
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;

        }

    }

}
PGo Workflow: (1)
Properties of our System

Invariants

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

Properties

Counter Converges Termination \( \Rightarrow \) (counter = procs * iters)
Processes Get the Token \( \forall p \in \text{ProcSet} : \langle\rangle(\text{token} = p) \)
PGo Workflow: (1) Verifying

Model Checked with TLC!

Developer writes specification

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: Not running
  - Current status: Not running
  - Errors detected: No errors
**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;
    }
}
```
• Generated 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 25K LOC (compiler) and 3K (runtime)

• Able to compile concurrent and distributed systems

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

• Designing ModularPlusCal: extending PlusCal with more modularity features for large systems, and more separation of design + implementation

• Collecting and developing system specs for demo/evaluation:
  
  • Load balancer, dist. queue, dist. counter, two phase commit, dist. mutex, Euclid’s algorithm, n-queens,…

  • ~30 lines of PCal generates ~80 lines of Go; compiled n-queens perf within 5% of a native Go implementation

  • [https://github.com/UBC-NSS/pgo/tree/master/examples](https://github.com/UBC-NSS/pgo/tree/master/examples)
Example specs/properties

- **N-Queens** (not written by us): computes all solutions to N-Queens
  - *Property*: at every step, the set of solutions found is a subset of all existing solutions

- **DijkstraMutex** (not written by us): Dijkstra’s mutual exclusion algorithm
  - *Property*: deadlock freedom

- **Counter**: N processes increment a shared, global counter a fixed number of times
  - *Property*: when all processes are done, counter is equal to \((N \times \# \text{ of iterations})\)

- **dqueue**: Distributed queue, with one producer and multiple consumers
  - *Property*: mutual exclusion (consumer and producer are not mutating shared queue at the same time)
• Support a larger subset of PlusCal/TLA+

• Generating distributed systems that are fault tolerant

• Use modularity to make it easy for developers to change generated code (without compromising safety)
PGo Limitations

- Specifications are very high level: not everything can be compiled efficiently.
- Requires developers to also specify environment during compilation (e.g., number of processes, transport protocol, etc).
- Both the PGo compiler and the associated runtime need to be trusted to claim correctness.
1. **Dinv** [ICSE 2018]
Spec miner
https://bitbucket.org/bestchaei/dinv

2. **Dara**
Model checker
https://github.com/DARA-Project

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

---

**Bridging gap between design and implementation**
1 logging statement runtime ~ 20 µs (given no other alternative). We discuss the associated decrease in nodes in a 4 node cluster. At 5 nodes and above the bandwidth heartbeats and caused a reduction in bandwidth of 10KB/s for all nodes while varying the number of nodes. The bandwidths of all clusters with 3 nodes, and a YCSB-A workload. Each cluster was run 3 times and we averaged the total running time. Table 4 shows the overhead in a real system we executed etcd Raft using the setup of 1.0, 0.1, and 0.01. We measured the average time delay between events we can introduce approximately 50K logging statements per node before perturbing the system.

Dinv's static analysis (detecting networking calls, adding logging counts of randomly located vectors) we used etcd Raft, which contains 144K LOC and thousands of variables. We measured instrumentation time with increasing code, etc) we used etcd Raft, which contains 144K LOC and thousands of variables. We measured instrumentation time with increasing code, etc).

### 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 ~ 20 µs
- Static instrumentation negligible
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 $\sim 20\mu s$
- Static instrumentation negligible

All Raft invariants can be detected with just two annotations.
Dinv analysis time

<table>
<thead>
<tr>
<th>System runtime (s)</th>
<th>Raft log (MB)</th>
<th>Raft analysis (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>5.1</td>
<td>12.7</td>
</tr>
<tr>
<td>60</td>
<td>10.5</td>
<td>28.1</td>
</tr>
<tr>
<td>90</td>
<td>13.7</td>
<td>35.9</td>
</tr>
<tr>
<td>120</td>
<td>17.4</td>
<td>48.7</td>
</tr>
<tr>
<td>150</td>
<td>22.5</td>
<td>68.8</td>
</tr>
<tr>
<td>180</td>
<td>27.7</td>
<td>99.1</td>
</tr>
</tbody>
</table>

- Log size + analysis time linear in sys runtime
- Can be done offline + parallelized