CS COMPUTER SCIENCE

Ivan Beschastnikh

Computer Science
University of British Columbia
Vancouver, Canada

UBC

Ivan Beschastnikh

Computer Science
University of British Columbia
Vancouver, Canada

Ivan Beschastnikh

Computer Science
University of British Columbia
Vancouver, Canada

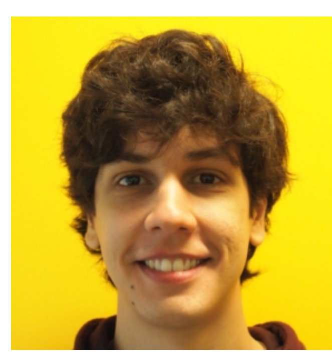
Program analysis for distributed systems

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

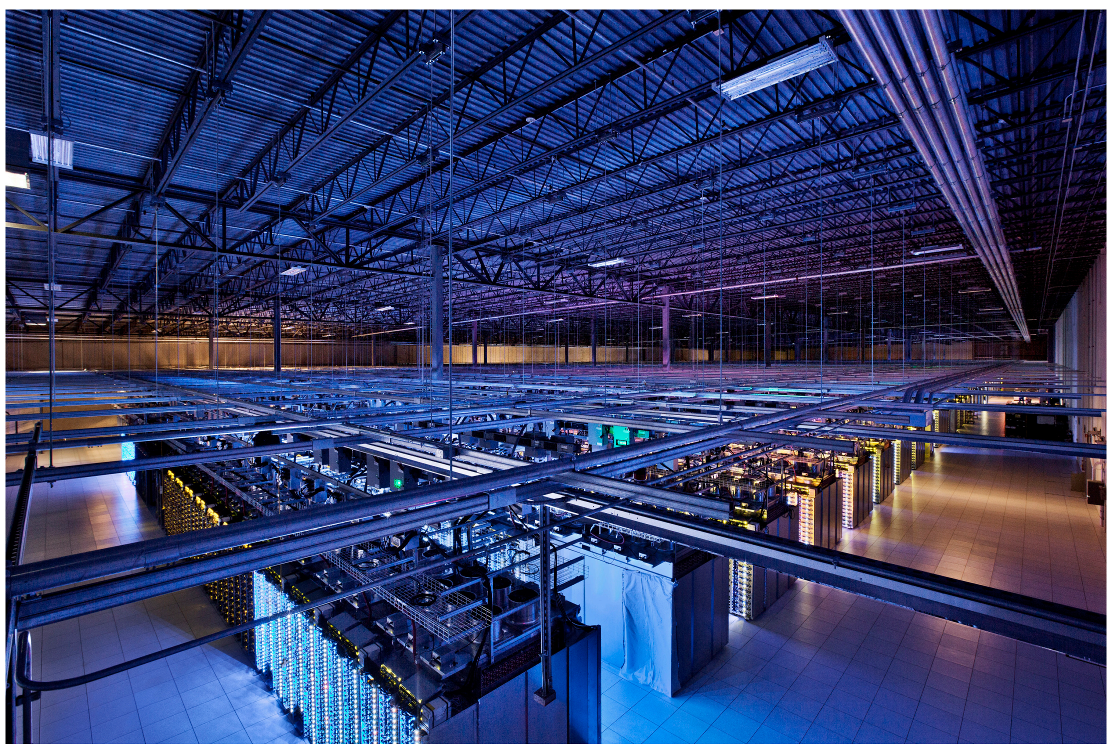
Distributed systems are widely-used

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

[1] Mark **Cavage**. 2013. *There's Just No Getting around It: You're Building a Distributed System*. Queue 11, 4, Pages 30 (April 2013)

Cloud systems/apps ecosystem

- Distributed systems are widely deployed [1]

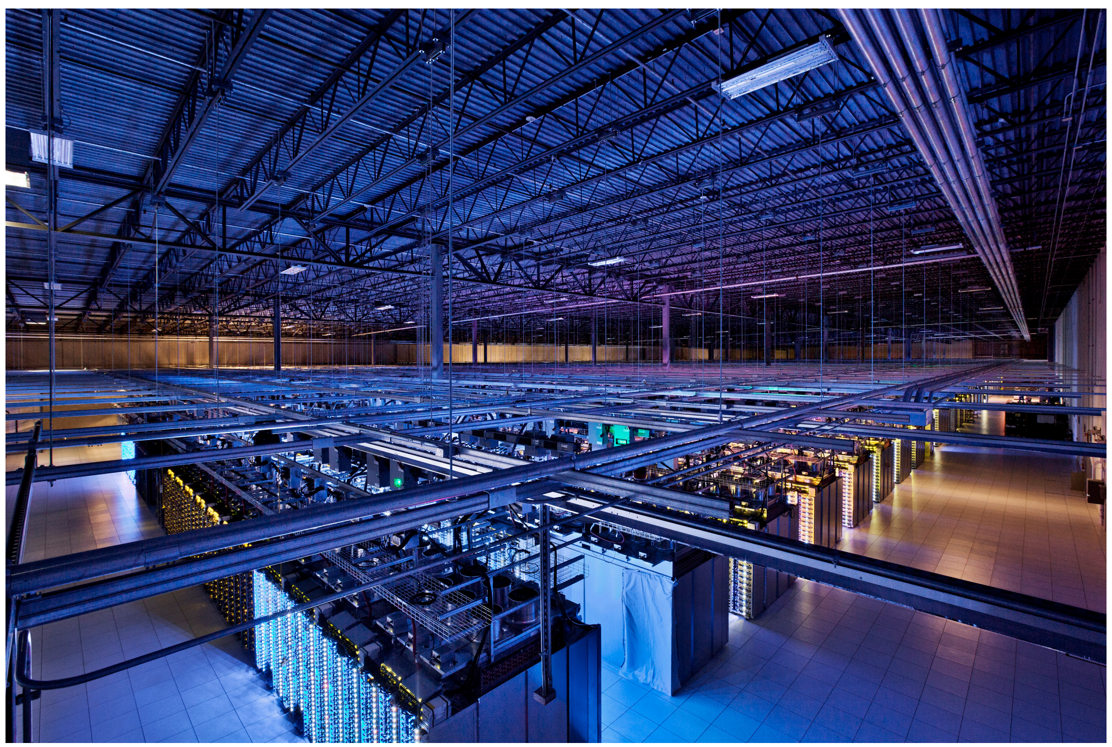


Google's data center, Council Bluffs, IA
https://www.google.com/about/datacenters/gallery

[1] Mark **Cavage**. 2013. *There's Just No Getting around It: You're Building a Distributed System*. Queue 11, 4, Pages 30 (April 2013)

Cloud systems/apps ecosystem

- Distributed systems are widely deployed [1]



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

Large-scale apps

Cloud abstraction

[1] Mark **Cavage**. 2013. *There's Just No Getting around It: You're Building a Distributed System*. Queue 11, 4, Pages 30 (April 2013)

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]

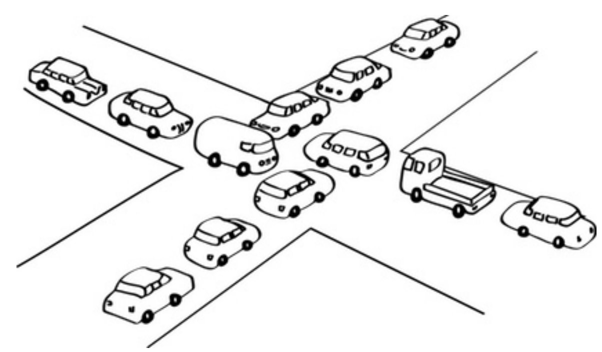
  - S3's outage in 2017 caused loss of millions of dollars [3]

[1] Mark **Cavage**. 2013. *There's Just No Getting around It: You're Building a Distributed System*. Queue 11, 4, Pages 30 (April 2013)
[2] Fletcher **Babb**. *Amazon's AWS DynamoDB Experiences Outage, Affecting Netflix, Reddit, Medium, and More*. en-US. Sept. 2015
[3] Shannon **Vavra.** *Amazon outage cost S&P 500 companies $150M*. axios.com, Mar 3, 2017

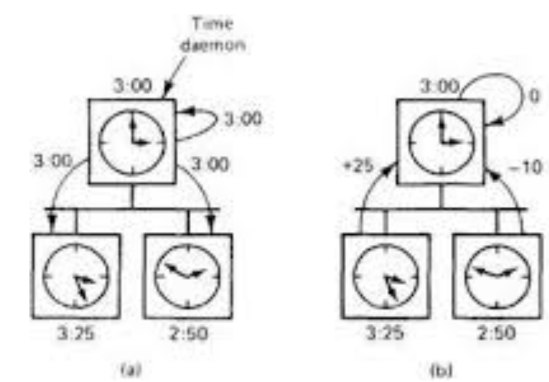
Issue 2: Distribution challenges

"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

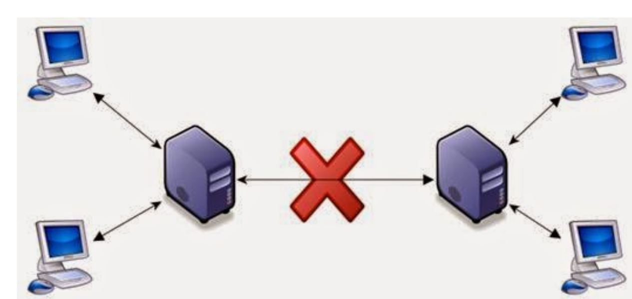
- Distributed systems are hard to **design** and **build**
- **Non-deterministic** sequence of events
- Processes make decisions based on **local state**
- A variety of **failures**



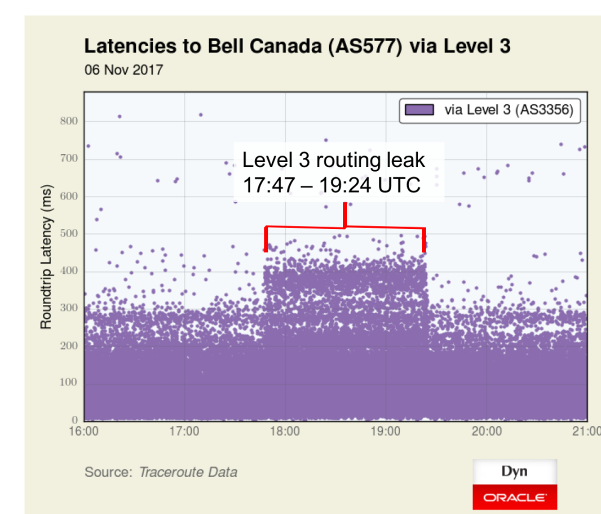
Concurrency



No central clock



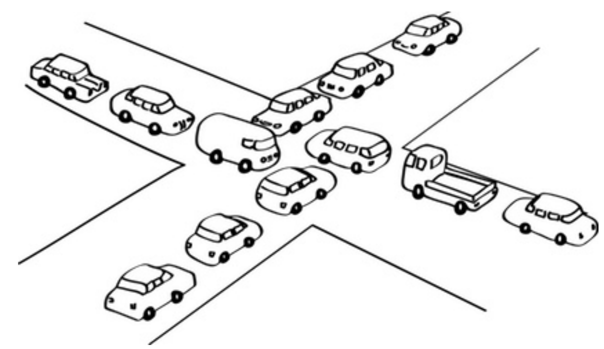
Partial failures



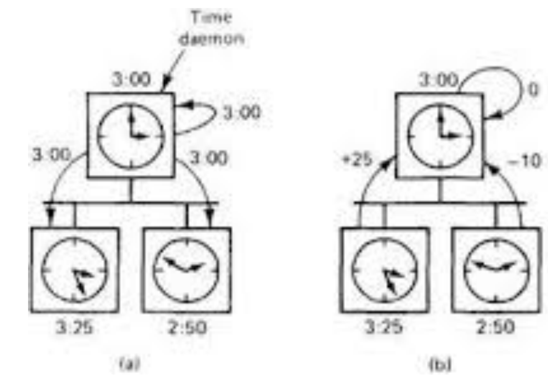
Perf variation

Overall: High essential complexity

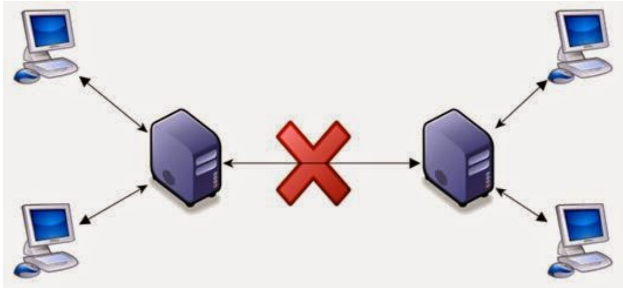
Failures can be very **costly**



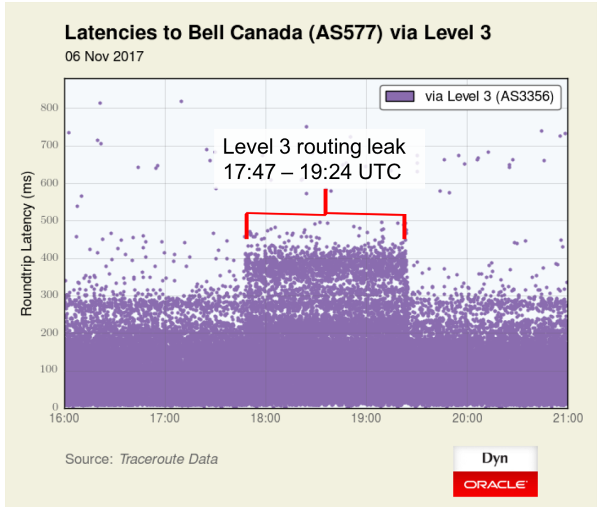
Concurrency



No central clock



Partial failures

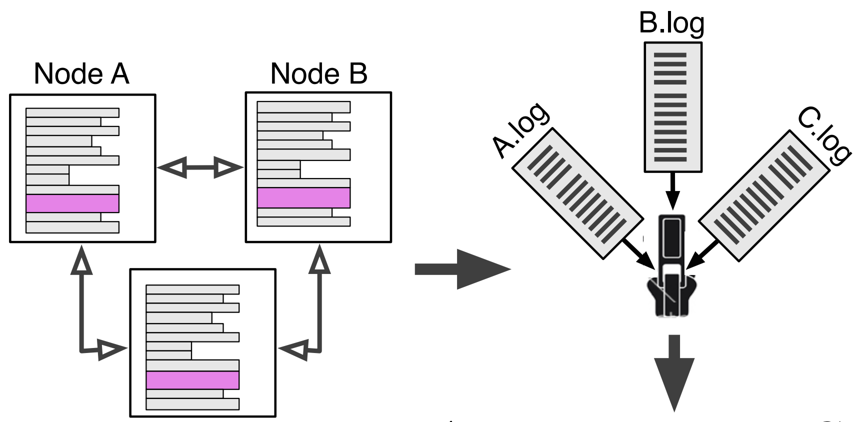


Perf variation

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

Program analysis for distributed systems

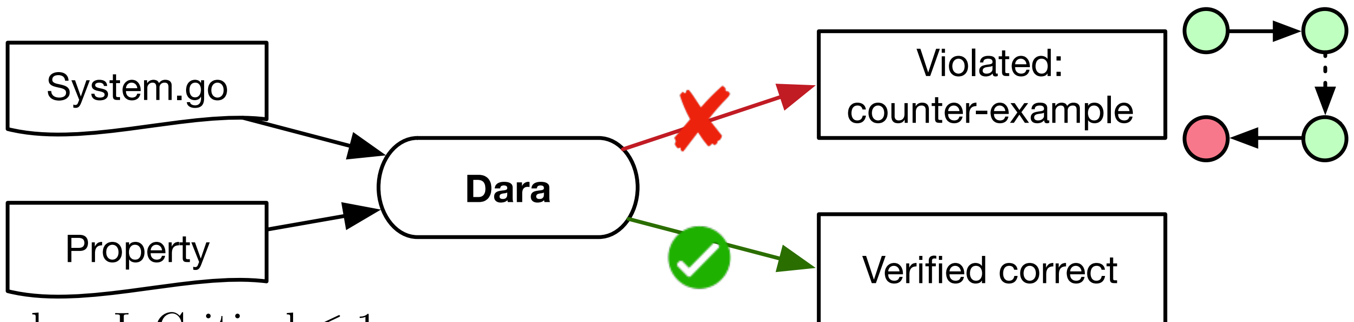
1. Dinv
Spec miner



$A.seq \leq B.seq \leq C.seq$

[ICSE 2018]

2. Dara
Model checker



$\forall$ nodes, $\text{InCritical} \leq 1$

3. PGo
Compiler

PlusCal model

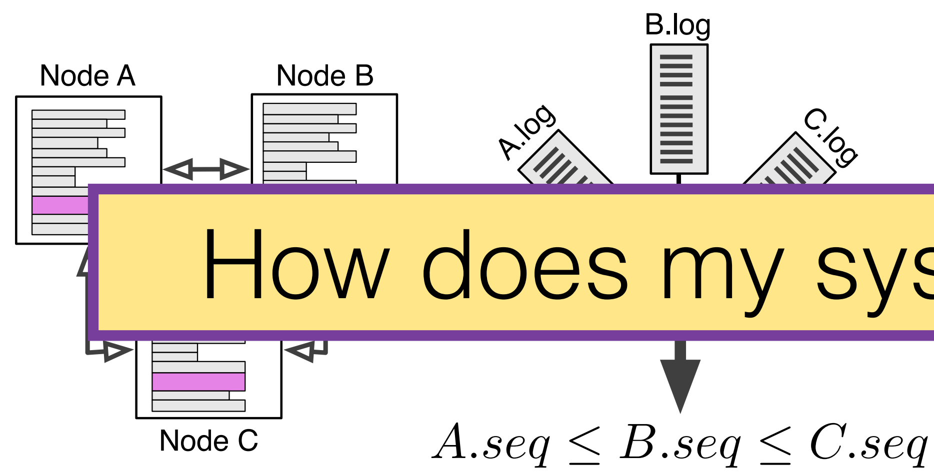
```
--algorithm Euclid { \** @PGo{ arg int N }@PGo
  (** @PGo{ var int u }@PGo
      @PGo{ var int v }@PGo
      @PGo{ var int v_init }@PGo
  **)
  variables u = 24;
            v \in 1 .. N;
            v_init = v;
  {
    while (u # 0) {
      if (u < v) {
          u := v || v := u;
      };
    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

Go lang

```
var v int
var u int
var v_init int
var N int


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 := v
        v_new := u
        u = u_new
        v = v_new
      }
      u = u − v
    }
    fmt.Printf("24 %v have gcd %v\n", v_init, v)
  }
}
```

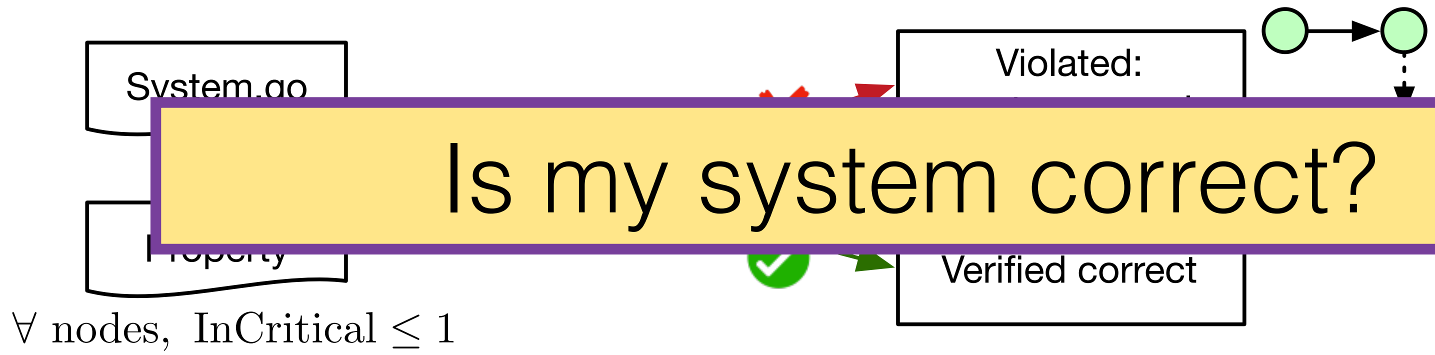
How these tools empower developers

1. Dinv
Spec miner



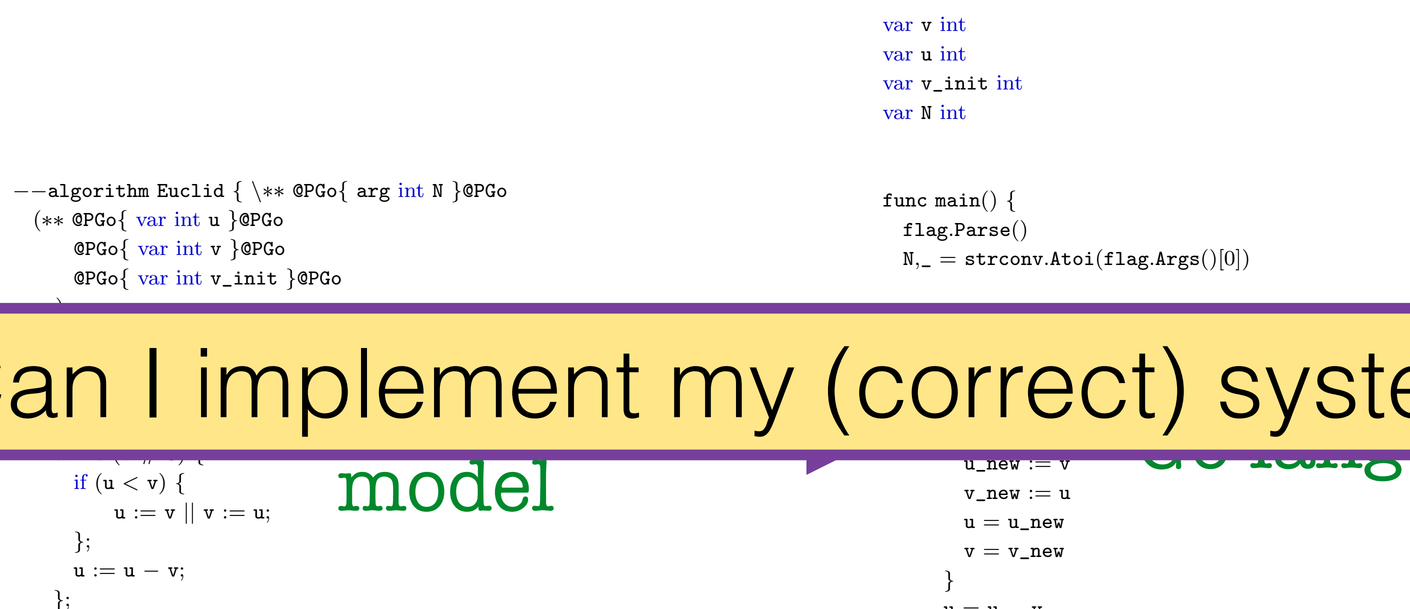
How does my system behave?

2. Dara
Model checker



Is my system correct?

3. PGo
Compiler



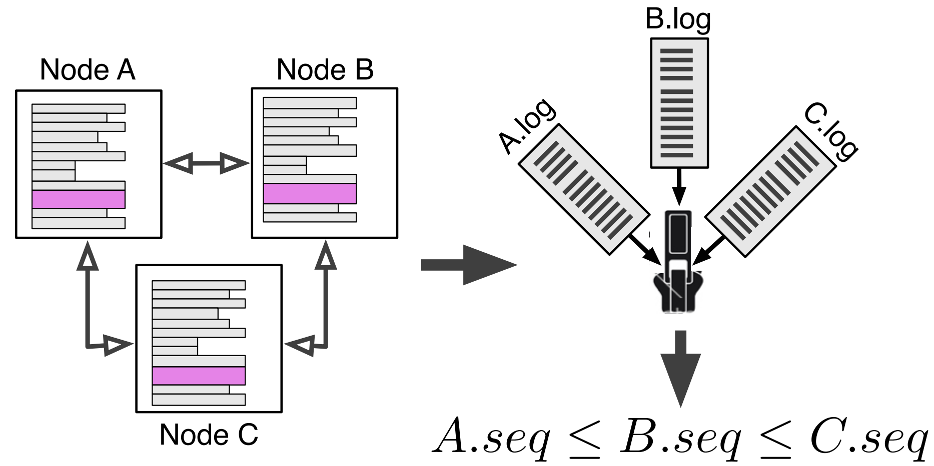
Can I implement my (correct) system faster?

Bridging gap between design and implementation

First up: distributed spec mining

1. Dinv

Spec miner

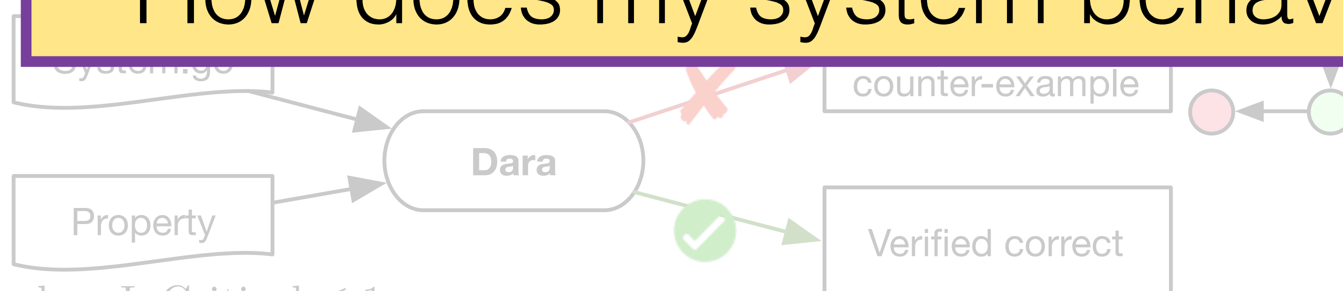


[ICSE 2018]

How does my system behave?

2. Dara

Model checker



3. PGo

Compiler

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

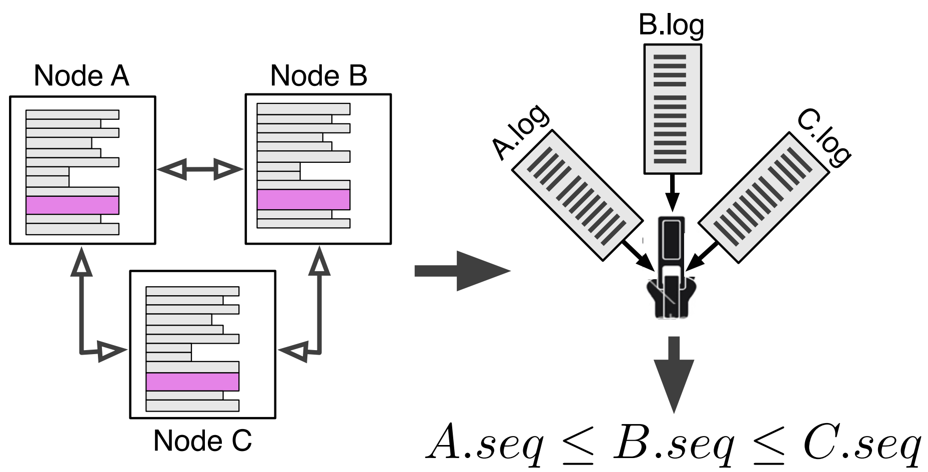
PlusCal model → Go lang

```
var v int
var u int
var v_init int
var N int

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 := v
        v_new := u
        u = u_new
        v = v_new
      }
      u = u - v
    }
    fmt.Printf("24 %v have gcd %v\n", v_init, v)
  }
}
```

Why distributed spec mining?

Dinv

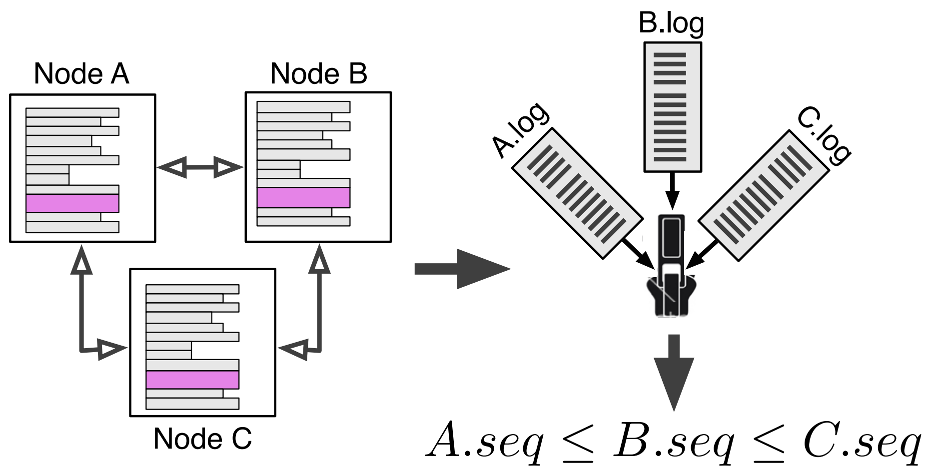
Spec miner



Why distributed spec mining?

Dinv

Spec miner



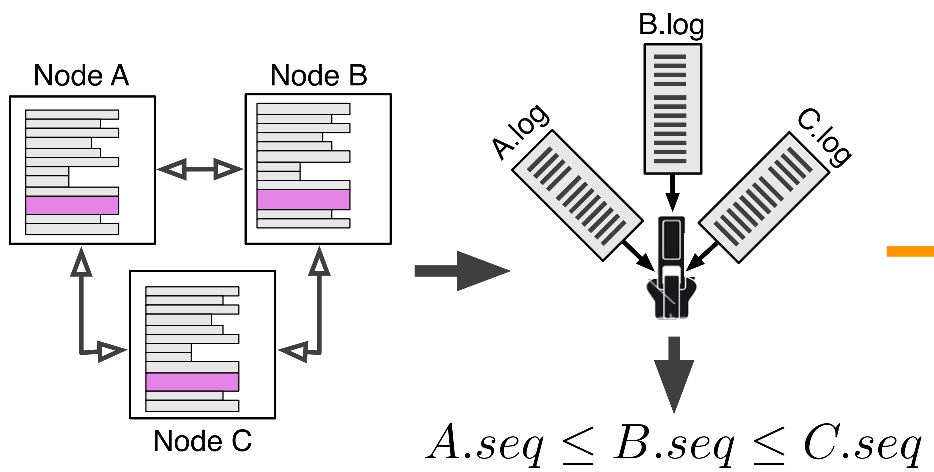
Sampler of state of the art in building robust distributed systems:

- **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:

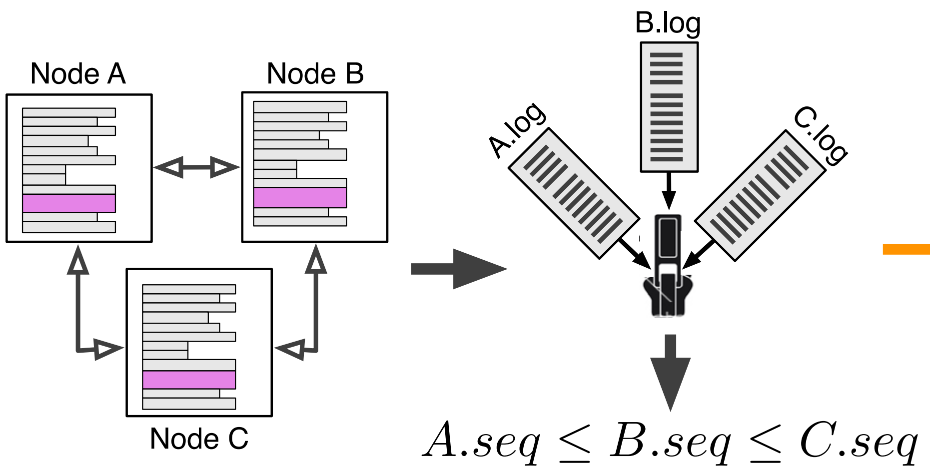
- **Verification** [*Verification:* Bagpipe OOPSLA'16, IronFleet SOSP'15, Verdi PLDI'15, Chapar POPL'16; *Modeling*: Lamport et.al SIGOPS'02, Holtzman IEEE TSE'97]
- **Bug detection** [SAMC OSDI'14, MODIST NSDI'09, CrystalBall NSDI'09, MaceMC NSDI'07]
- **Runtime checkers** [D3S NSDI'18]
- **Tracing** [PivotTracing SOSP'15, XTrace NSDI'07, Dapper TR'10]
- **Log analysis** [Pensieve SOSP'17, Demi NSDI'16, ShiViz CACM '16]

Require specifications

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]

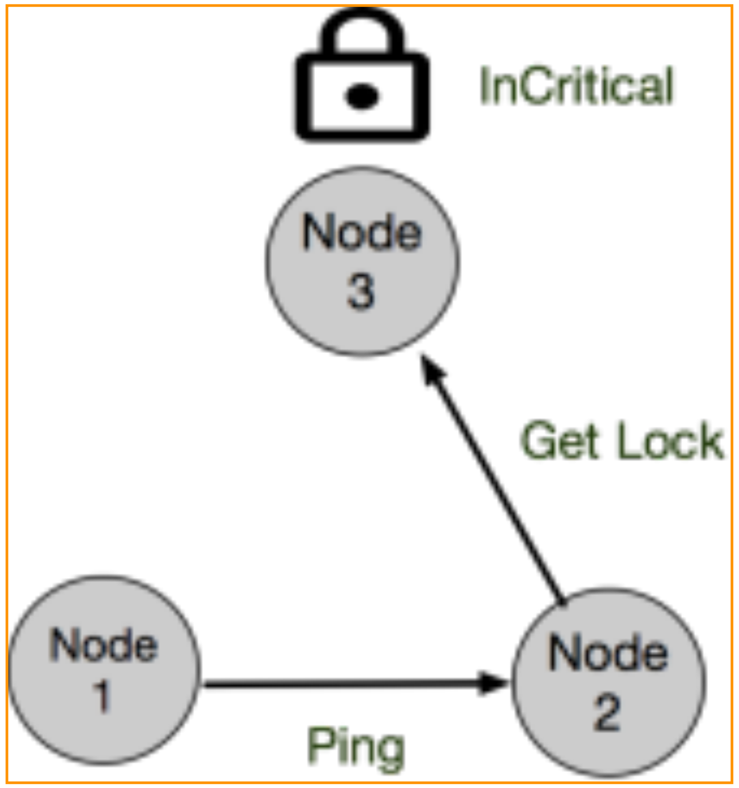
- Avenger SRDS'11
  - High manual effort
- CSight ICSE'14
  - Temporal model
- Udon ICSE'15
  - Multithreaded sh-state

Require specifications

Goal: infer correctness properties

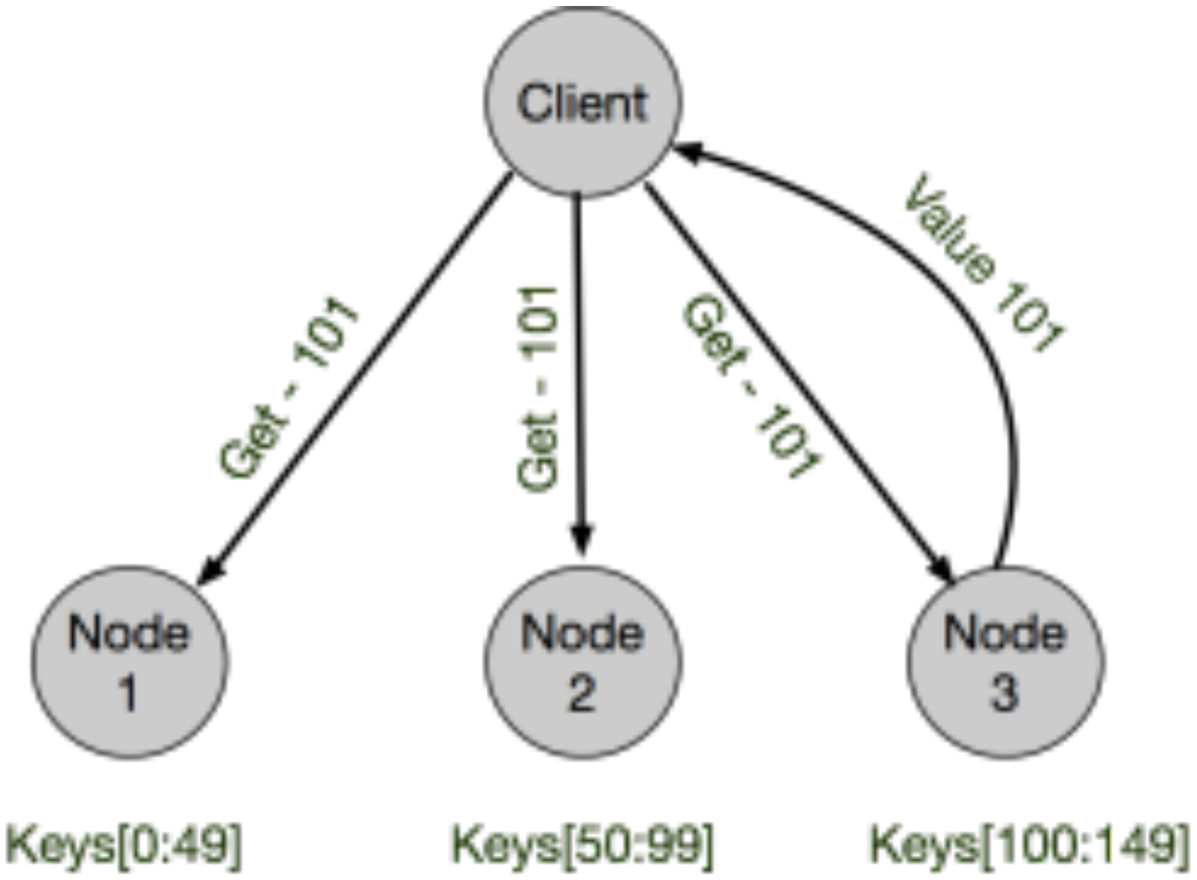
Mutual exclusion:

$\forall$ nodes, i, j $InCritical_i \rightarrow \neg\ InCritical_j$



Key Partitioning:

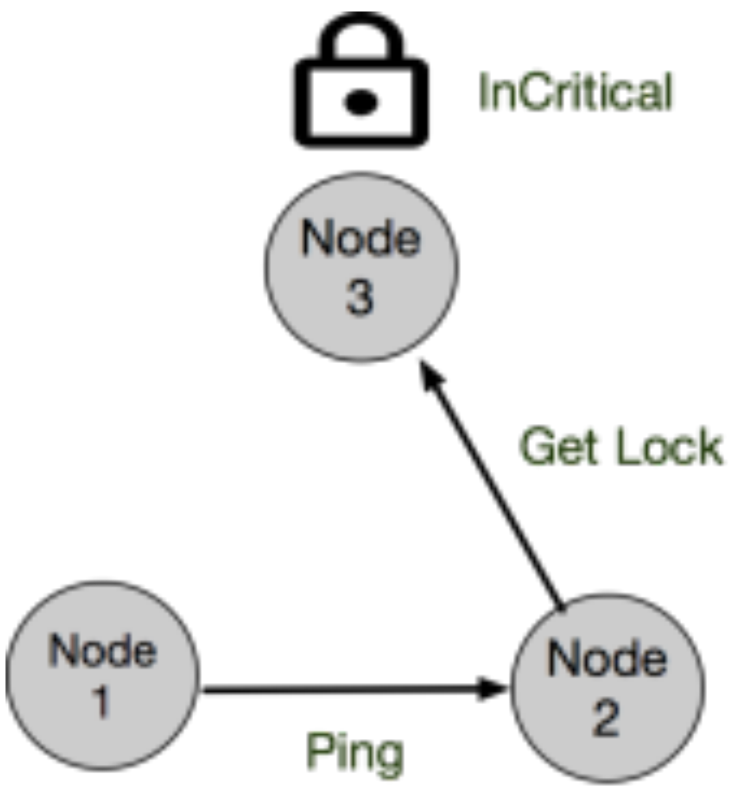
$\forall$ nodes, i, j $keys_i \neq keys_j$



Goal: infer correctness properties

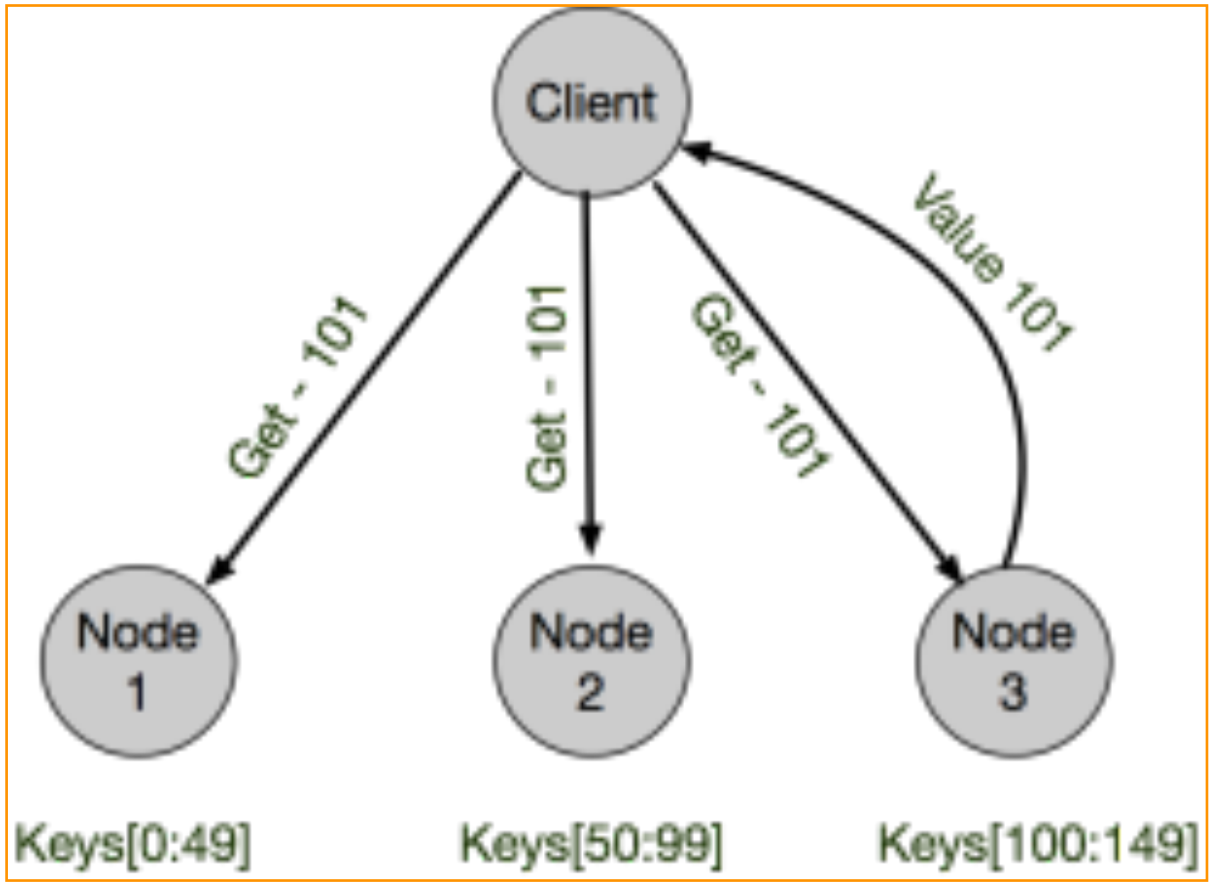
Mutual exclusion:

$\forall$ nodes, i, j $InCritical_i \rightarrow \neg \, InCritical_j$



Key Partitioning:

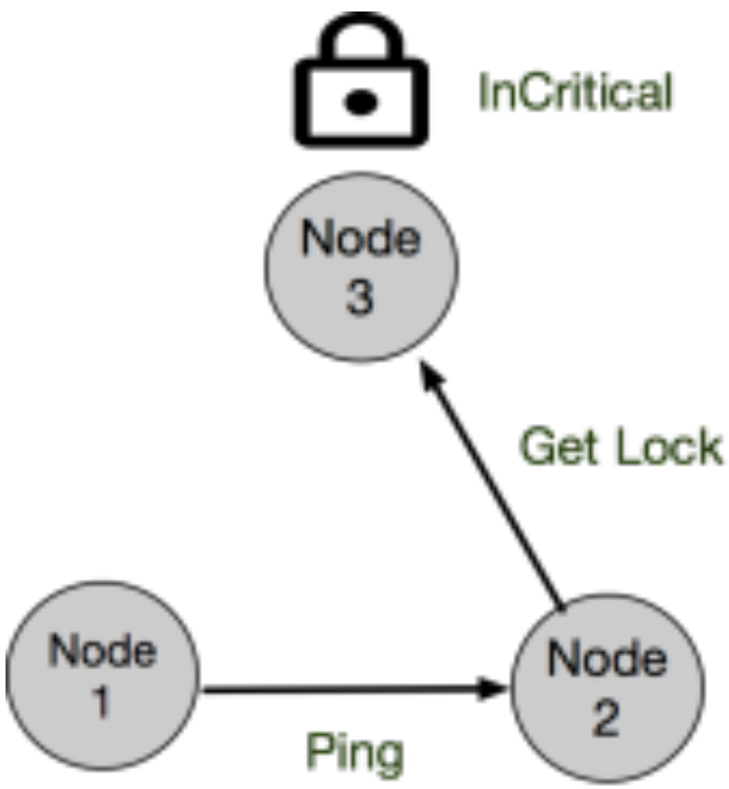
$\forall$ nodes$, i, j \; keys_i \neq keys_j$



Goal: infer correctness properties

Mutual exclusion:

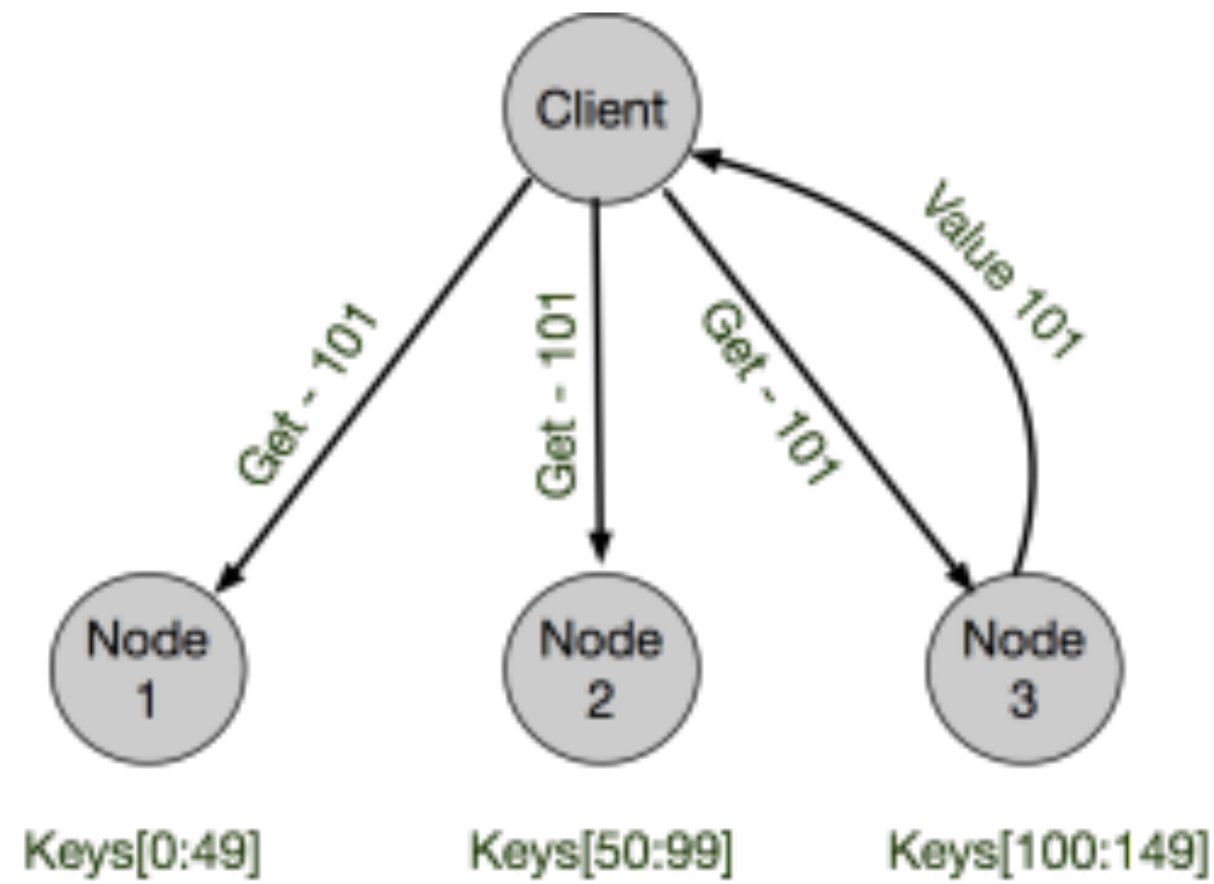
$$\forall \text{ nodes, i, j } InCritical_i \rightarrow \neg\, InCritical_j$$



Running example

Key Partitioning:

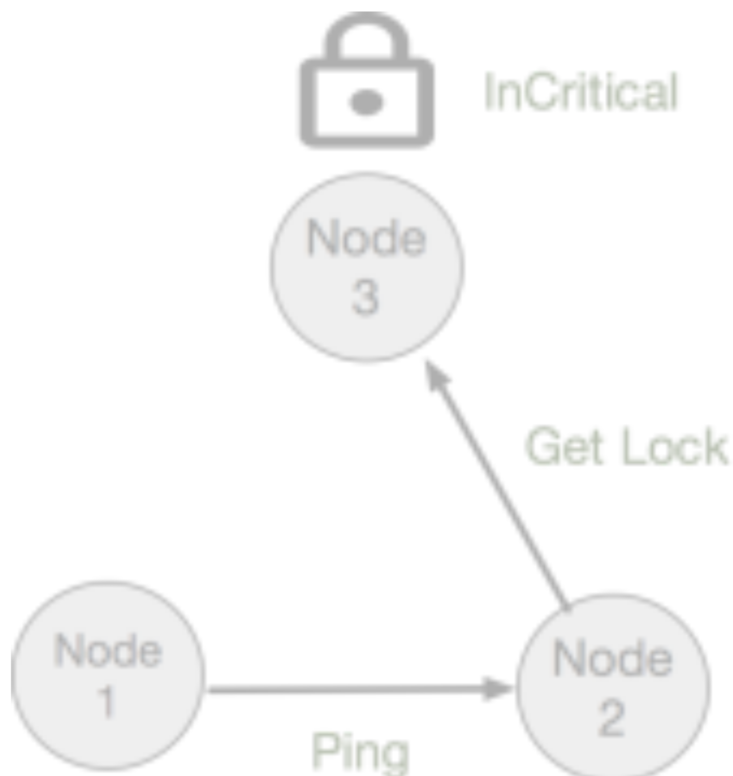
$$\forall \text{ nodes}, i, j \; keys_i \neq keys_j$$



Dist. correctness + Dist. state

Mutual exclusion:

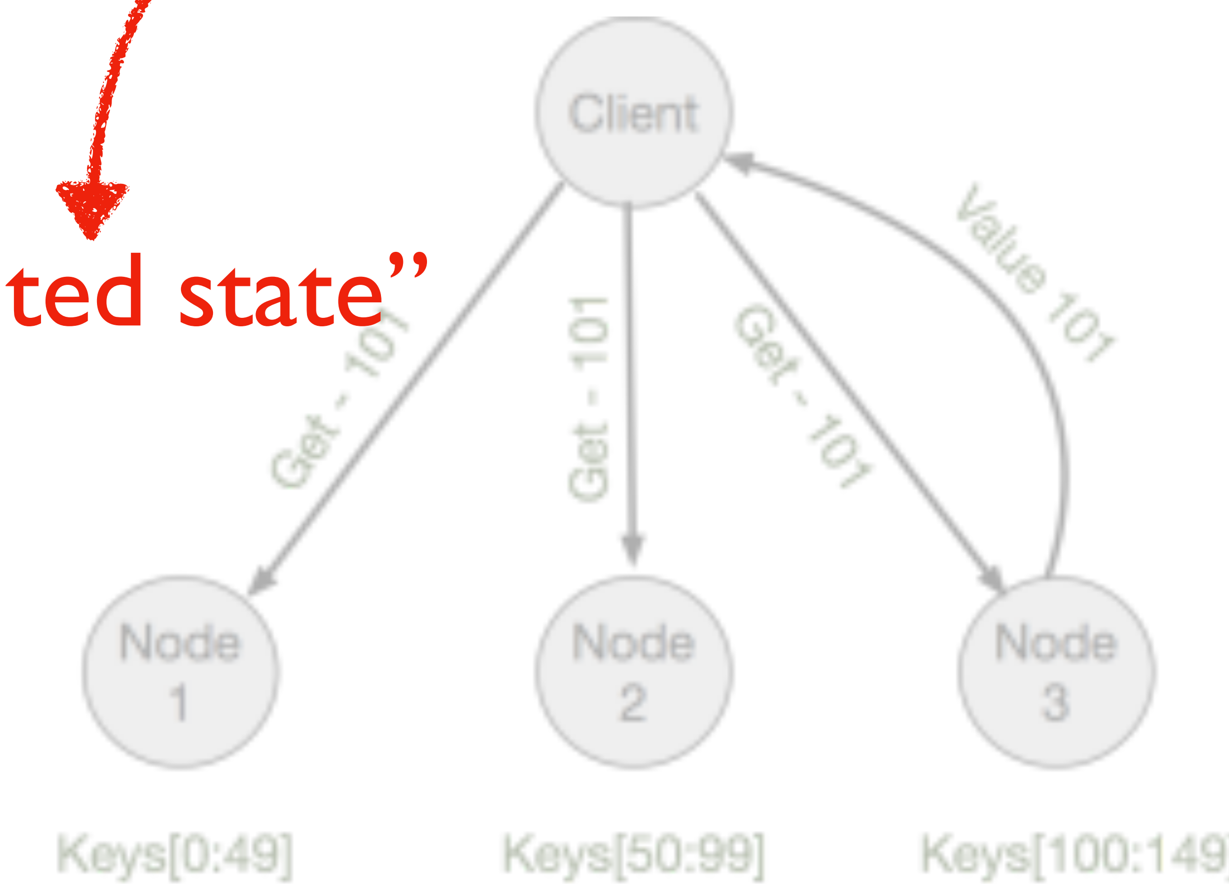
$\forall$ nodes, i, j $\boxed{InCritical_i} \rightarrow \neg \boxed{InCritical_j}$



Key Partitioning:

$\forall$ nodes, i, j $\boxed{keys_i \neq keys_j}$

“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

[1] John Ousterhout. *The Role of Distributed State*. CMU-TR. 1991

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

[1] John Ousterhout. *The Role of Distributed State*. CMU-TR. 1991

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.

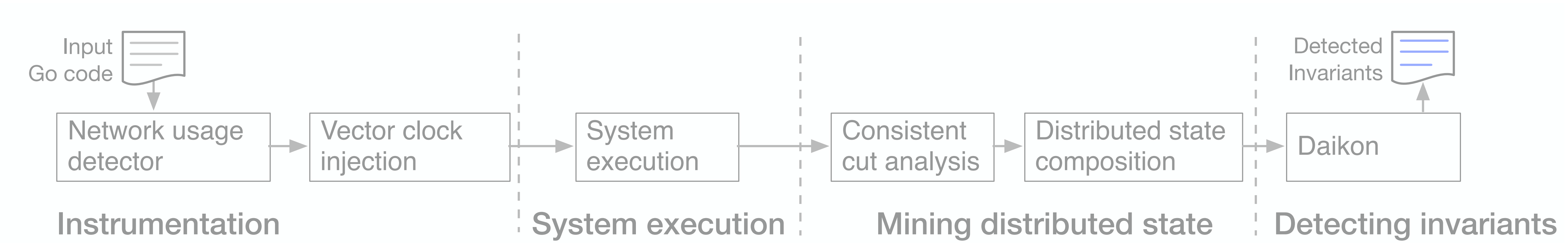
- John Ousterhout

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

Dinv: captures distributed state and reveals distributed state runtime properties

[1] John Ousterhout. *The Role of Distributed State*. CMU-TR. 1991

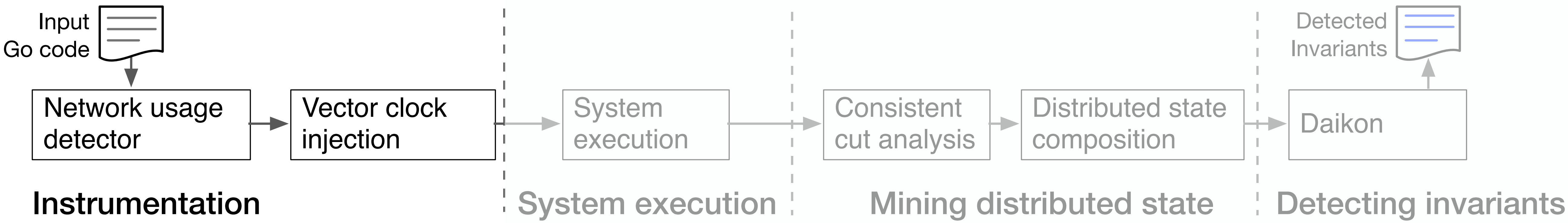
Dinv approach: static+dynamic analysis



Static analysis

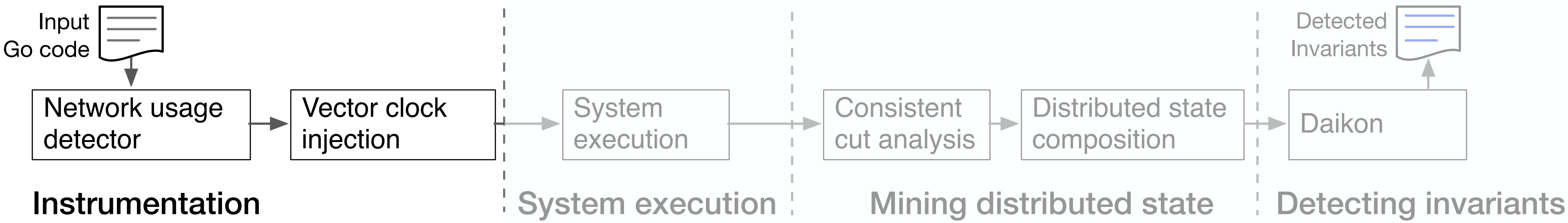
Dynamic analysis

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

1

```
recv(n)
i:= 1
sum := 0
product := 1
for i <= n {
    sum := sum + 1
    product := product * i
    i := i + 1
}
send(sum)
// @ dump
send (product)
```

Developer adds dump annotations at key program points

2.a

```
recv(n)
i:= 1

product := 1
for i <= n {

    product := product * i
    i := i + 1
}

// @ dump
send (product)
```

Backward slice: code affecting the sent product variable

2.b

```
recv(n)
i:= 1

product := 1
for i <= n {

    product := product * i
    i := i + 1
}

// @ dump
send (product)
```

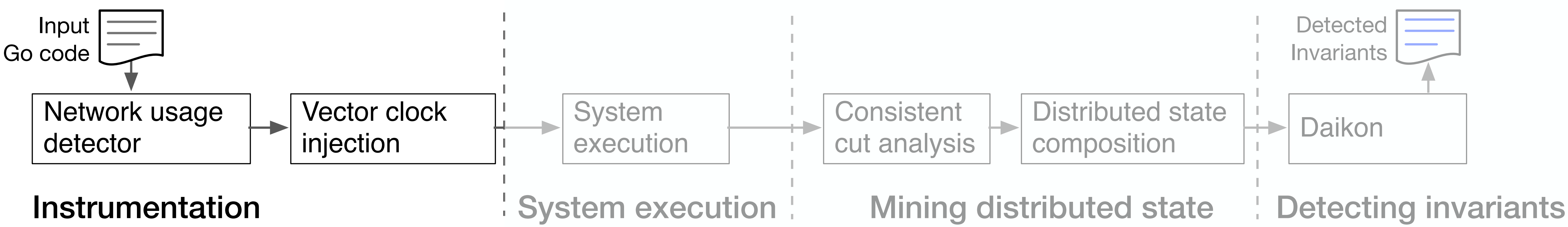
Variables appearing in the slice: i, n, product

2

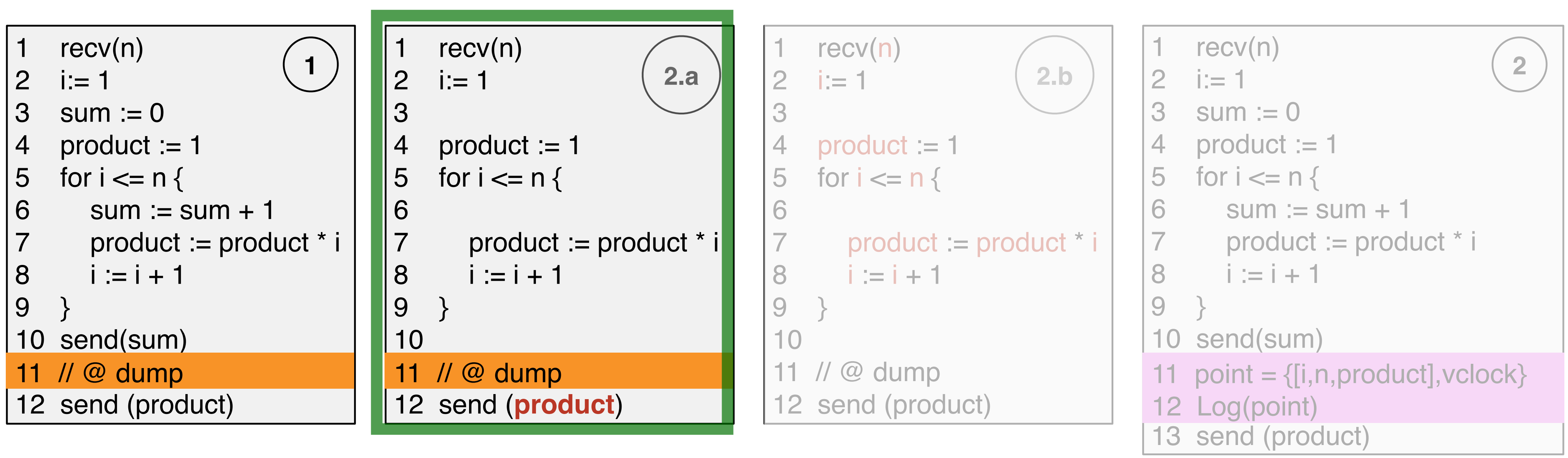
```
recv(n)
i:= 1
sum := 0
product := 1
for i <= n {
    sum := sum + 1
    product := product * i
    i := i + 1
}
send(sum)
point = {[i,n,product],vclock}
Log(point)
send (product)
```

Injected code to log product-affecting vars

Dinv static analysis



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



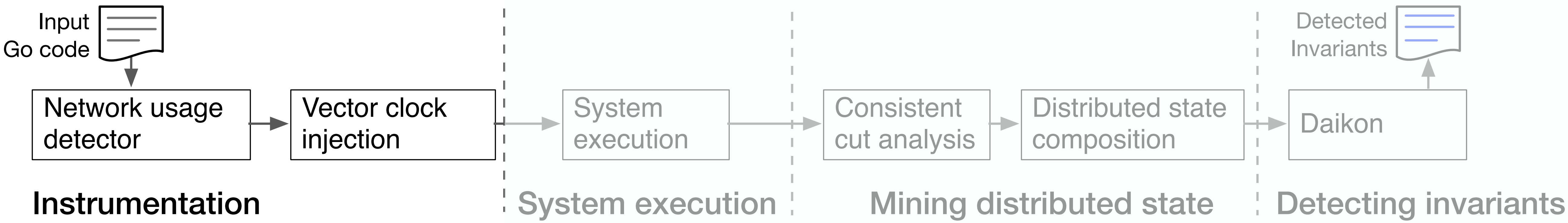
Developer adds dump annotations at key program points

Backward slice: code affecting the sent product variable

Variables appearing in the slice: i, n, product

Injected code to log product-affecting vars

Dinv static analysis



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

(1)

```
recv(n)
i:= 1
sum := 0
product := 1
for i <= n {
    sum := sum + 1
    product := product * i
    i := i + 1
}
send(sum)
// @ dump
send (product)
```

Developer adds dump annotations at key program points

(2.a)

```
recv(n)
i:= 1

product := 1
for i <= n {

    product := product * i
    i := i + 1
}

// @ dump
send (product)
```

Backward slice: code affecting the sent product variable

(2.b)

```
recv(n)
i:= 1

product := 1
for i <= n {

    product := product * i
    i := i + 1
}

// @ dump
send (product)
```

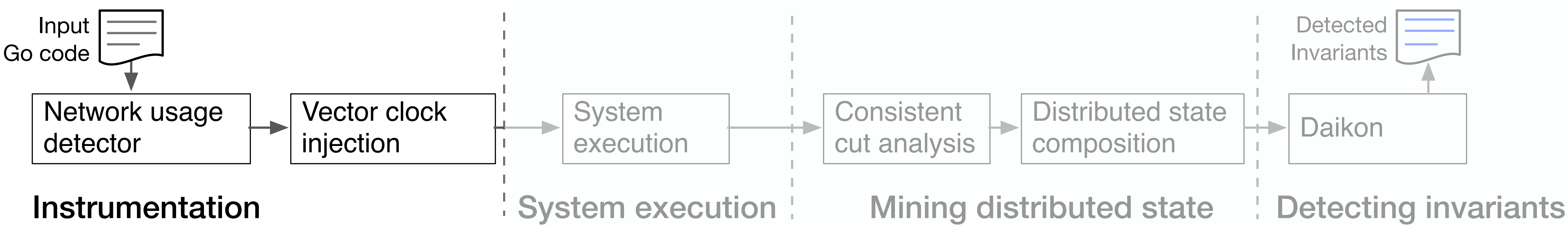
Variables appearing in the slice: i, n, product

(2)

```
recv(n)
i:= 1
sum := 0
product := 1
for i <= n {
    sum := sum + 1
    product := product * i
    i := i + 1
}
send(sum)
point = {[i,n,product],vclock}
Log(point)
send (product)
```

Injected code to log product-affecting vars

Dinv static analysis



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

1

```
recv(n)
i:= 1
sum := 0
product := 1
for i <= n {
    sum := sum + 1
    product := product * i
    i := i + 1
}
send(sum)
// @ dump
send (product)
```

Developer adds dump annotations at key program points

2.a

```
recv(n)
i:= 1

product := 1
for i <= n {

    product := product * i
    i := i + 1
}

// @ dump
send (product)
```

Backward slice: code affecting the sent product variable

2.b

```
recv(n)
i:= 1

product := 1
for i <= n {

    product := product * i
    i := i + 1
}

// @ dump
send (product)
```

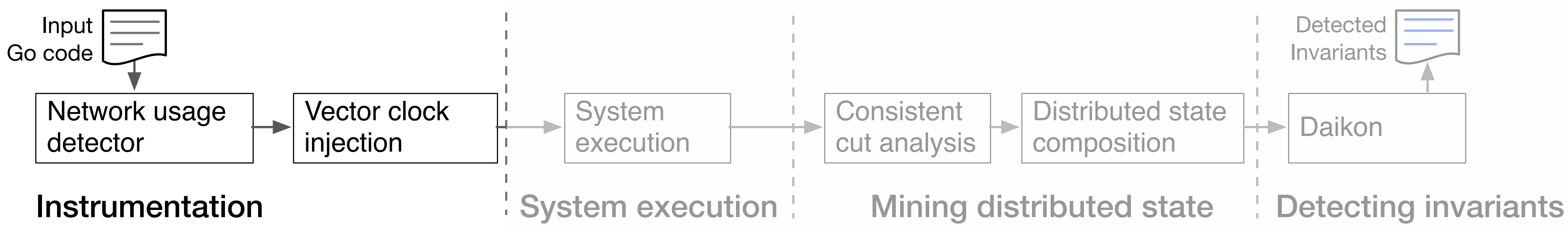
Variables appearing in the slice: i, n, product

2

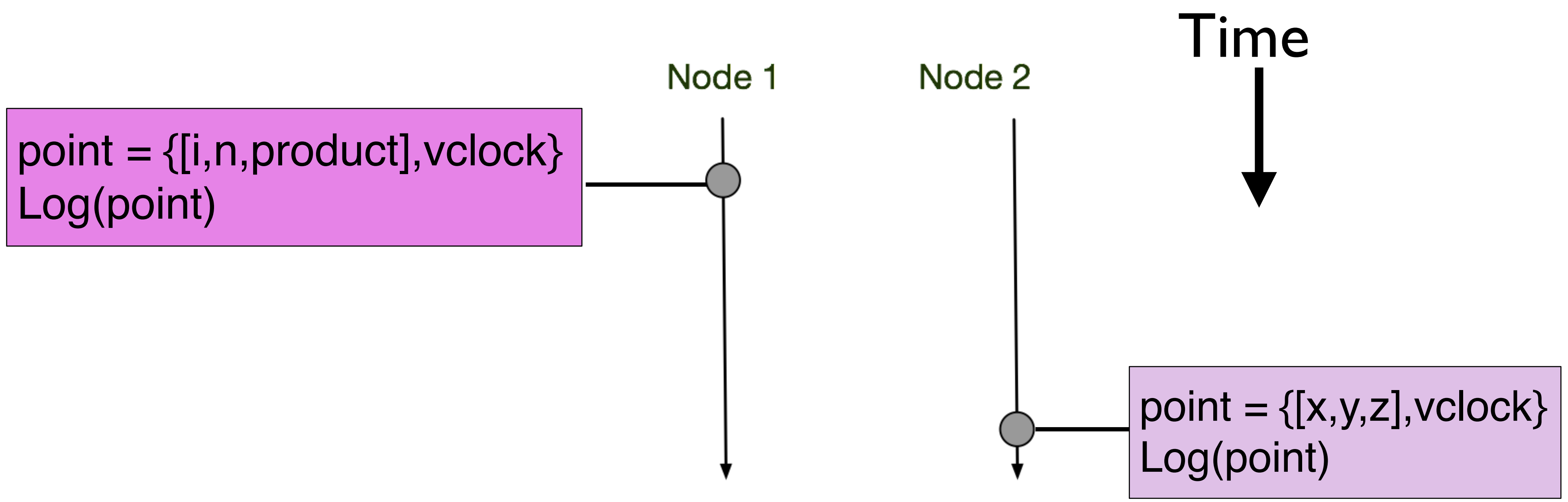
```
recv(n)
i:= 1
sum := 0
product := 1
for i <= n {
    sum := sum + 1
    product := product * i
    i := i + 1
}
send(sum)
point = {[i,n,product],vclock}
Log(point)
send (product)
```

Injected code to log product-affecting vars

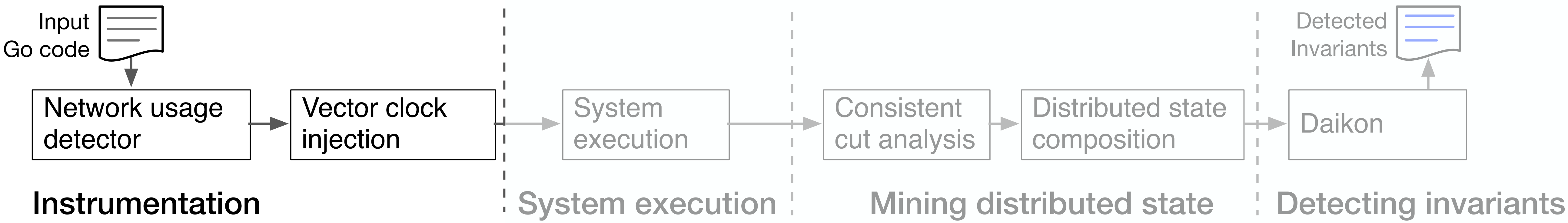
Dinv static analysis



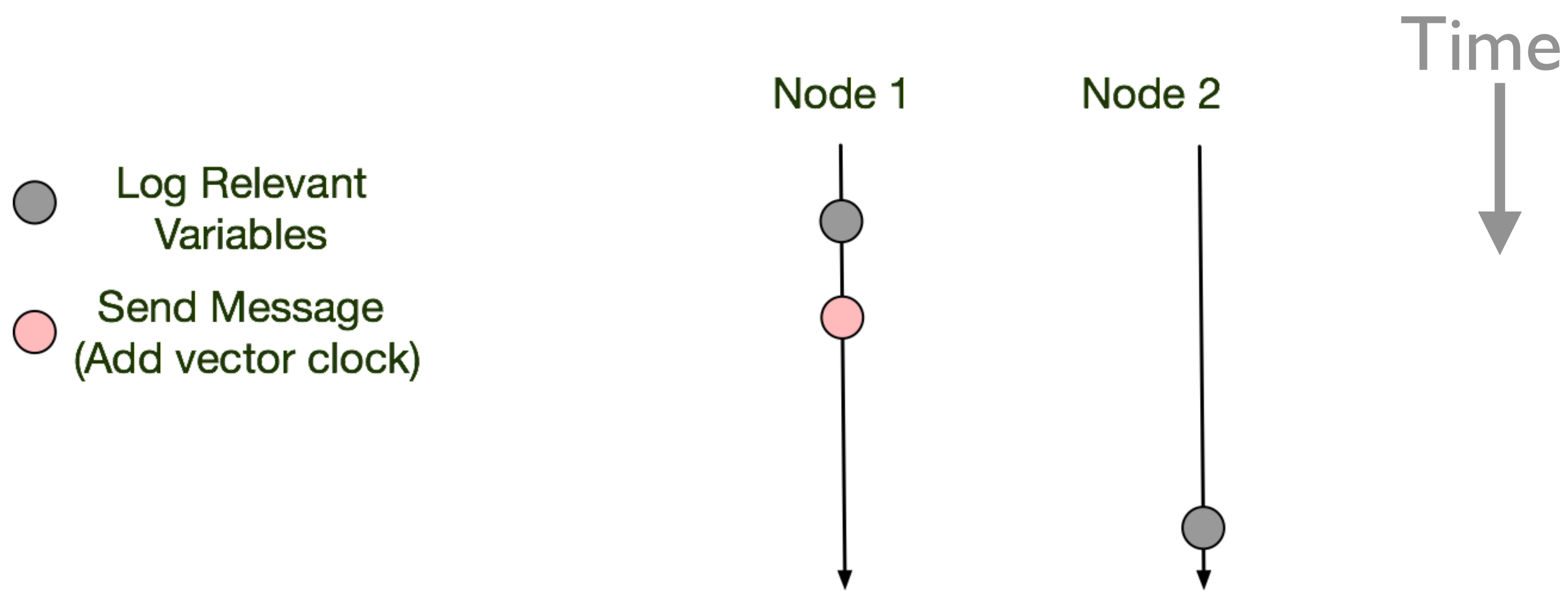
1. Interprocedural Program Slicing
2. Logging Code Injection
3. Vector Clock Injection



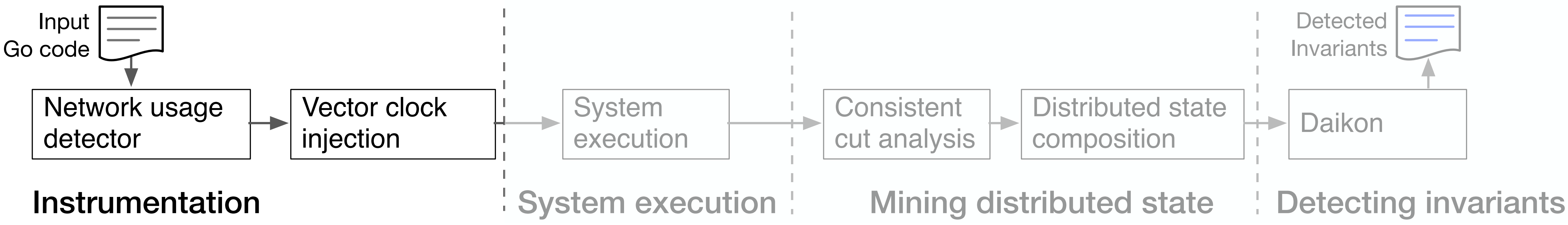
Dinv static analysis



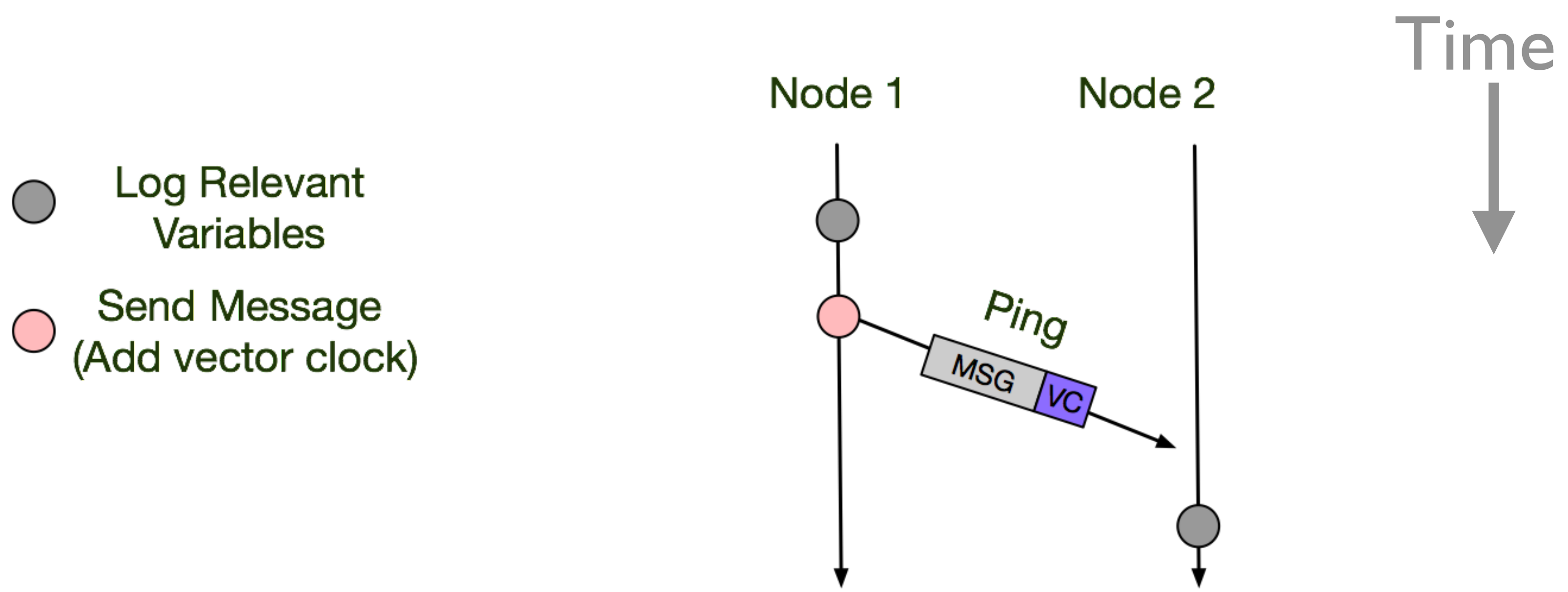
1. Interprocedural Program Slicing
2. Logging Code Injection
3. Vector Clock Injection



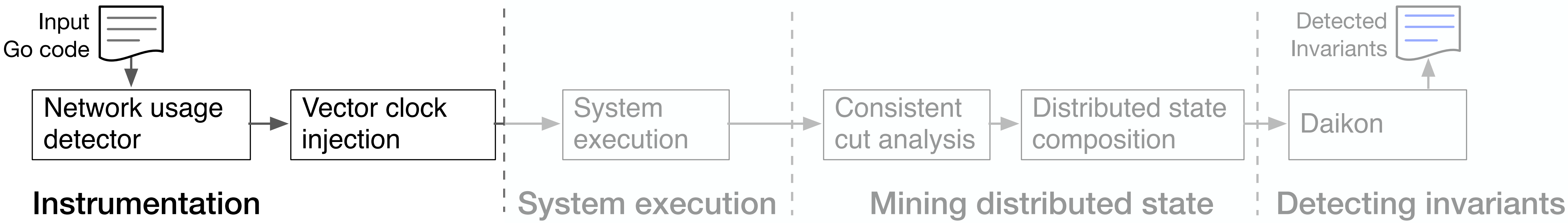
Dinv static analysis



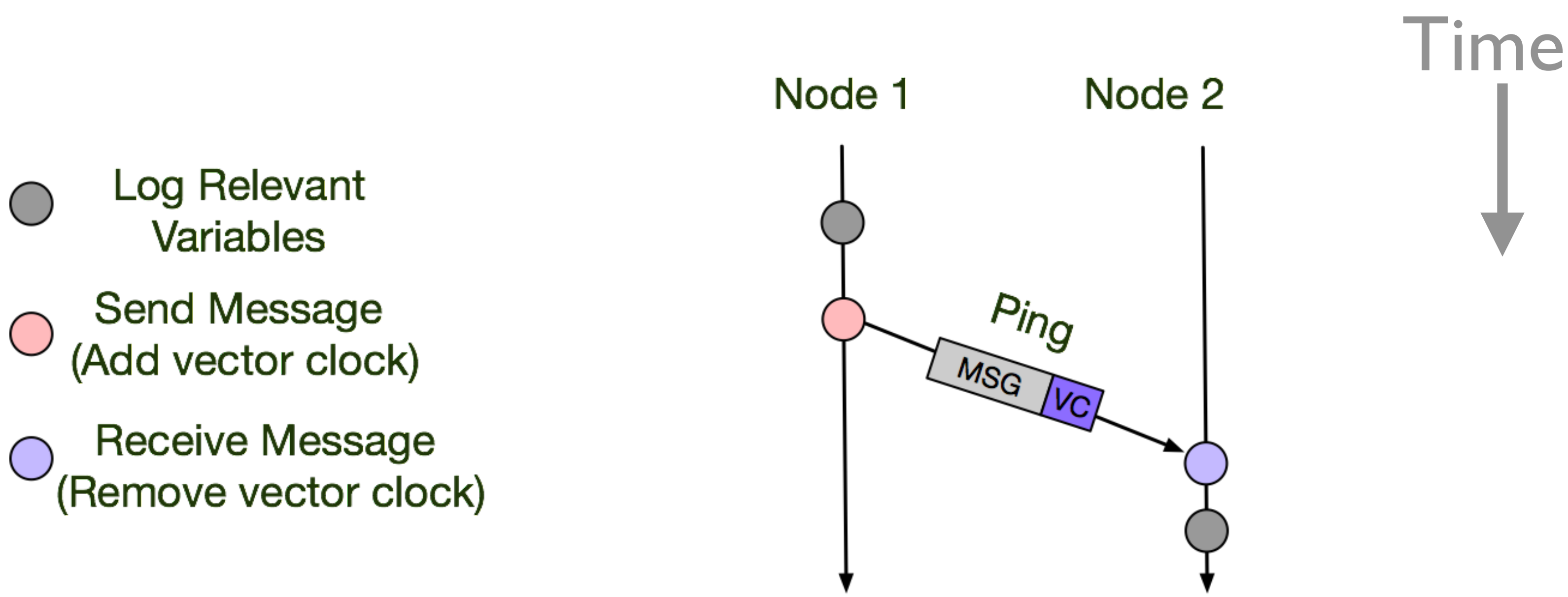
1. Interprocedural Program Slicing
2. Logging Code Injection
3. Vector Clock Injection



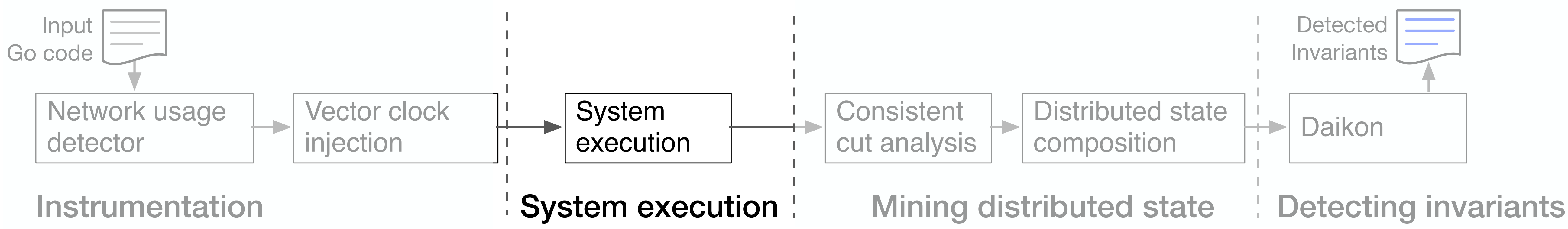
Dinv static analysis

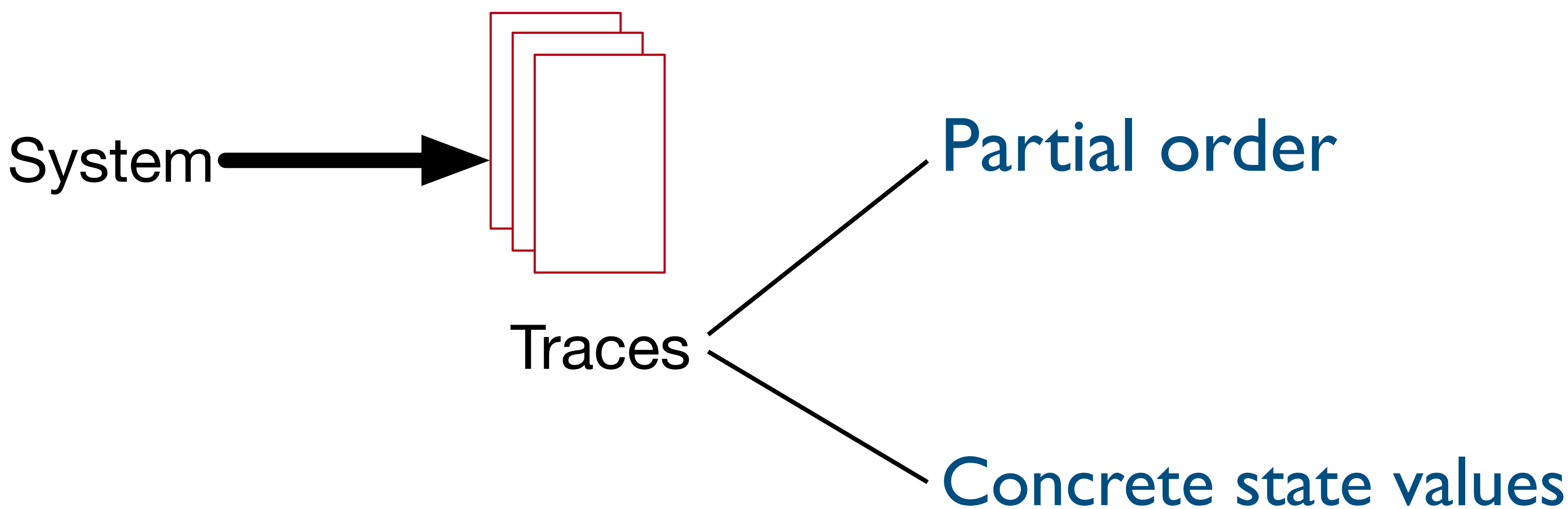


1. Interprocedural Program Slicing
2. Logging Code Injection
3. Vector Clock Injection

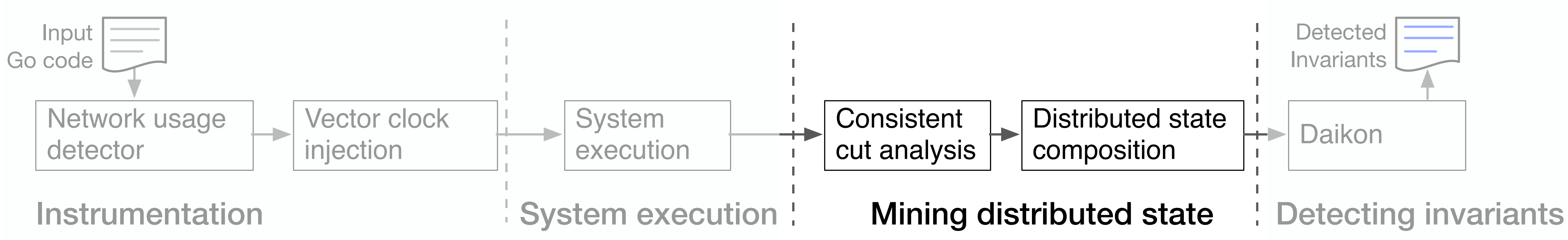


Run the system + collect traces

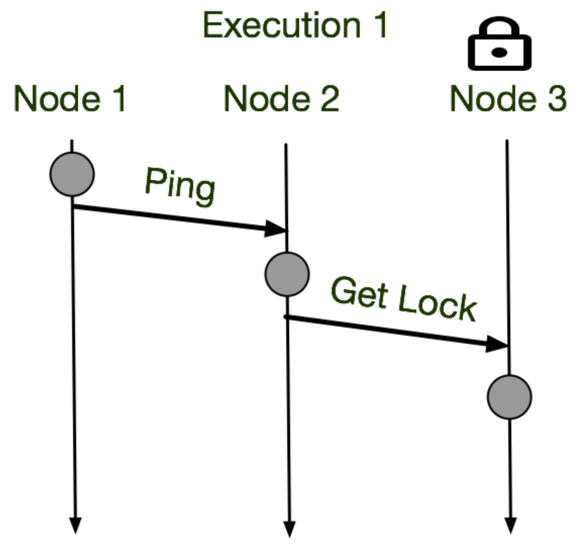


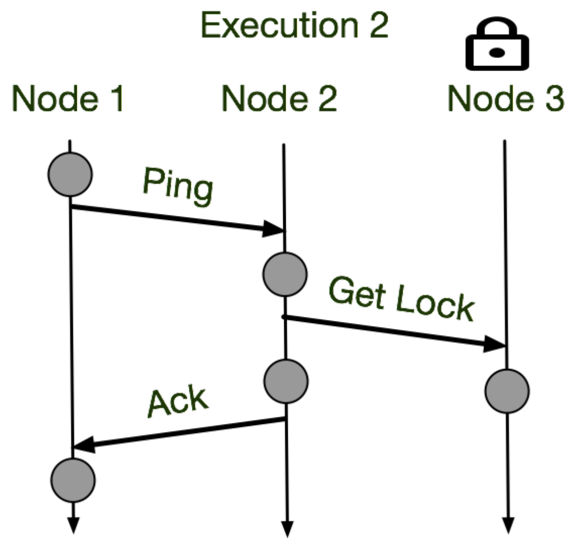


Reasoning about global state

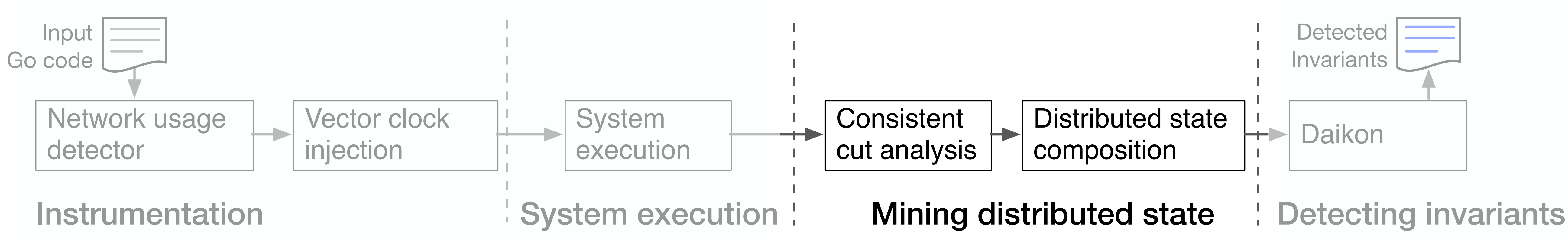


1. Consistent Cuts
2. Ground States
3. State Bucketing

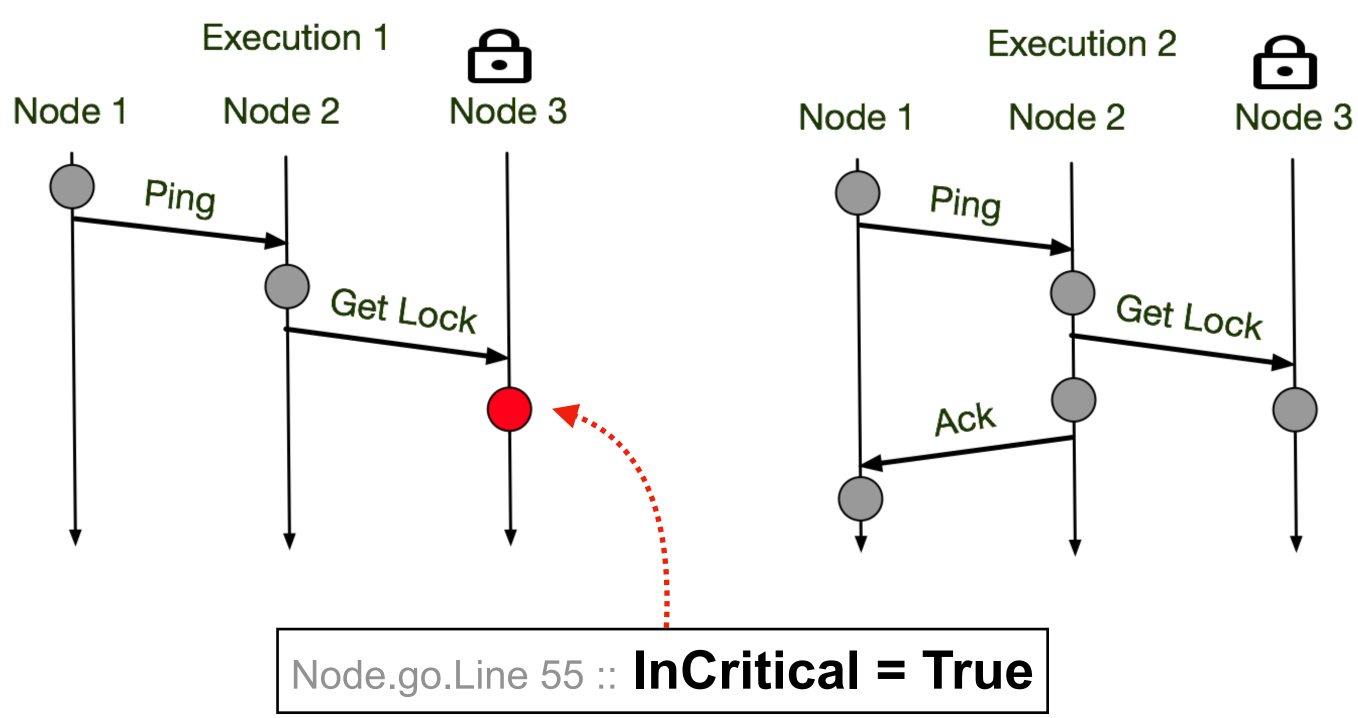




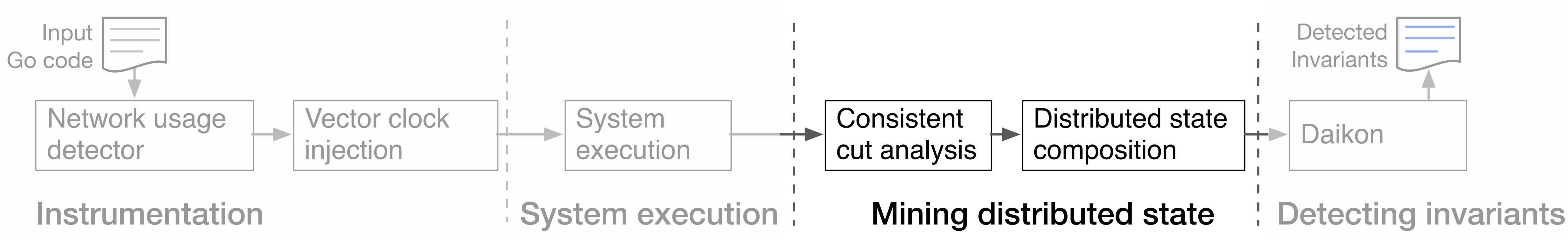
Reasoning about global state



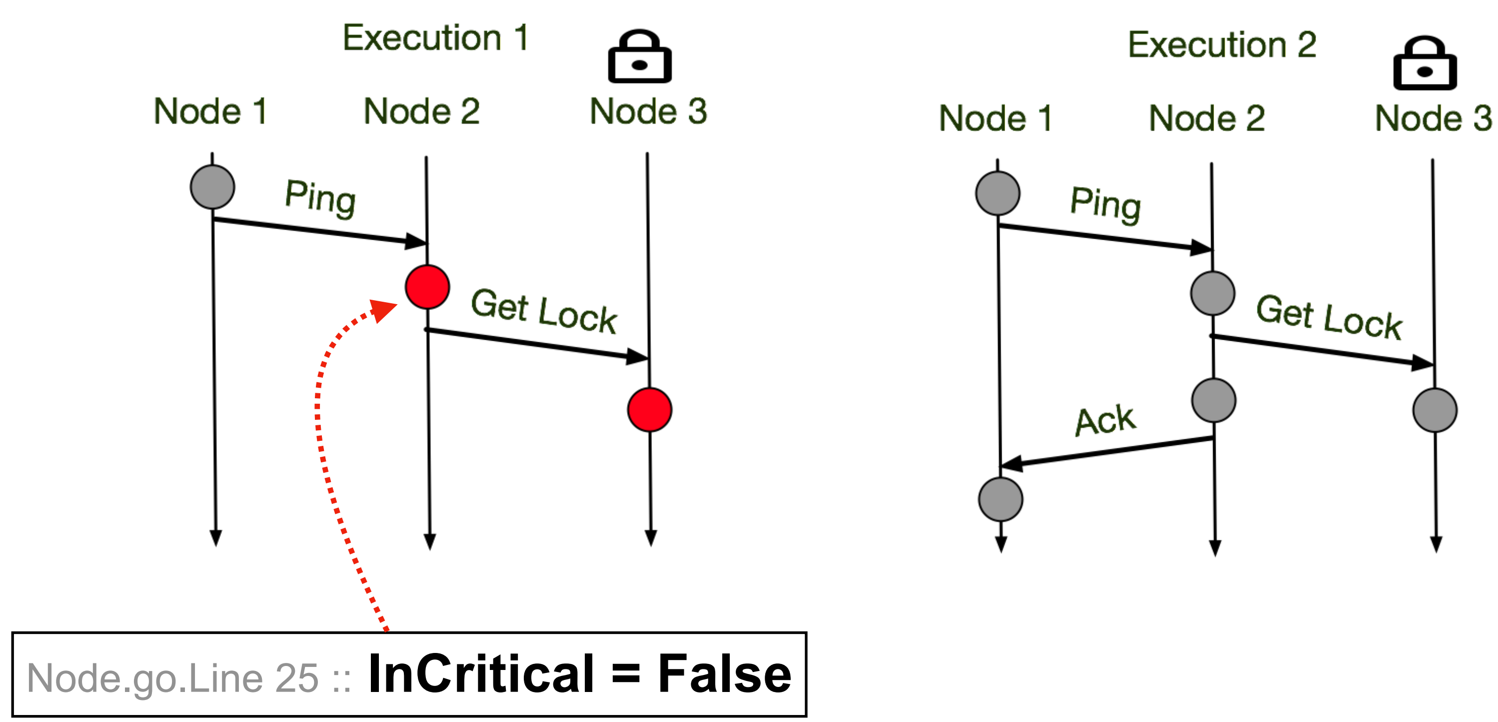
1. Consistent Cuts
2. Ground States
3. State Bucketing



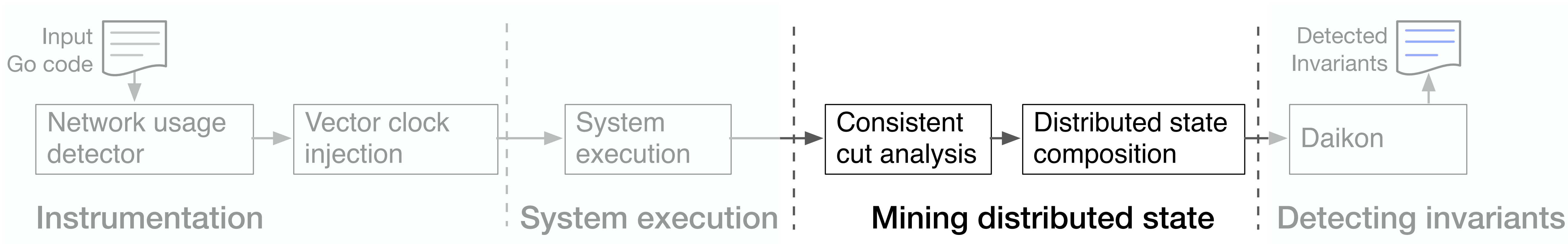
Reasoning about global state



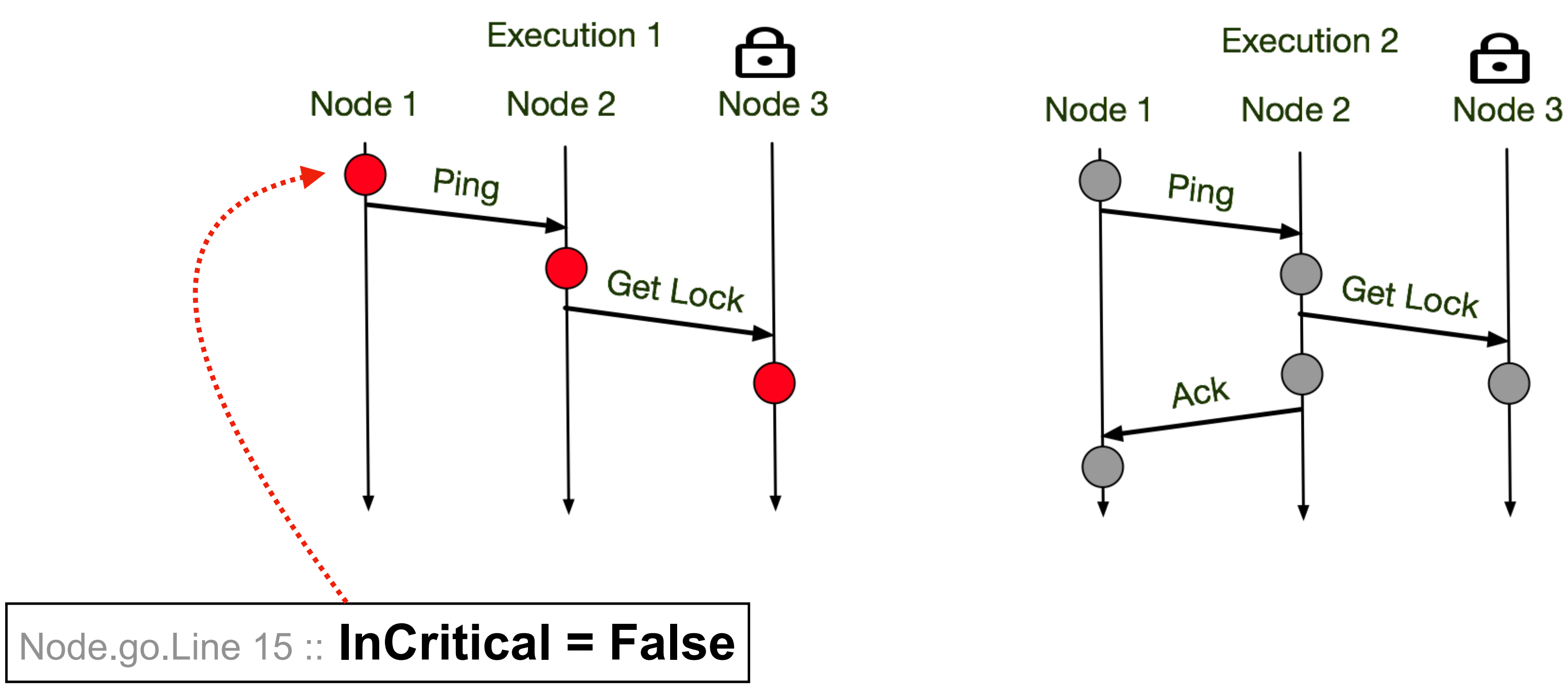
1. Consistent Cuts
2. Ground States
3. State Bucketing



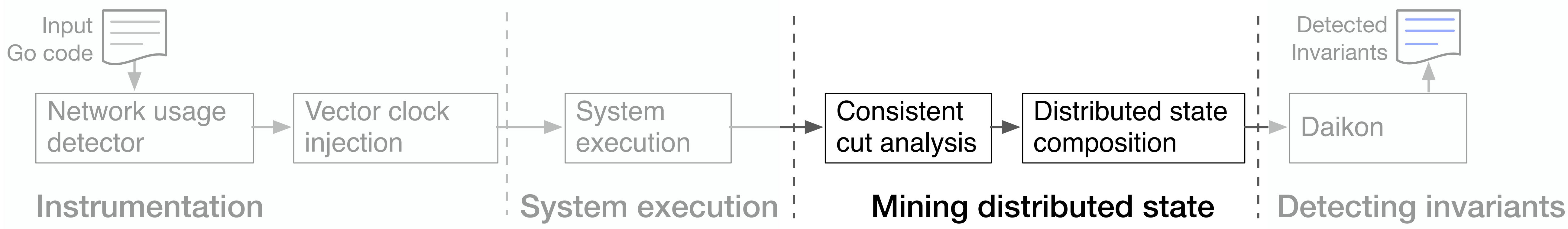
Reasoning about global state



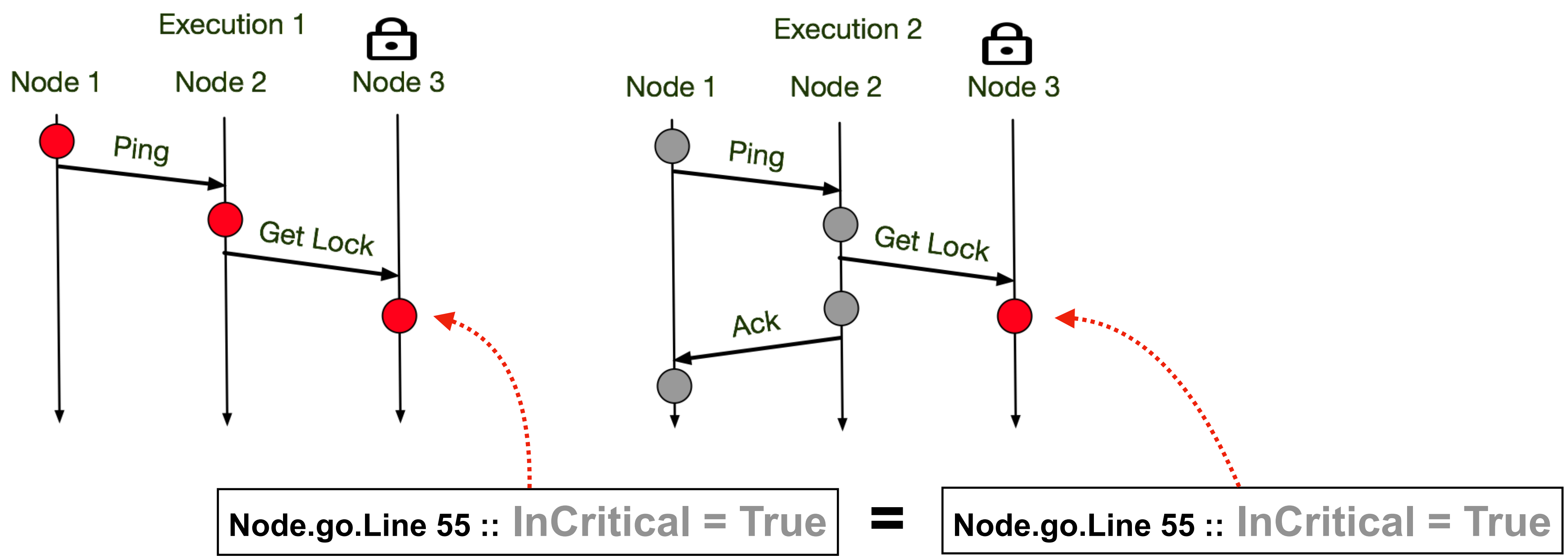
1. Consistent Cuts
2. Ground States
3. State Bucketing



Reasoning about global state

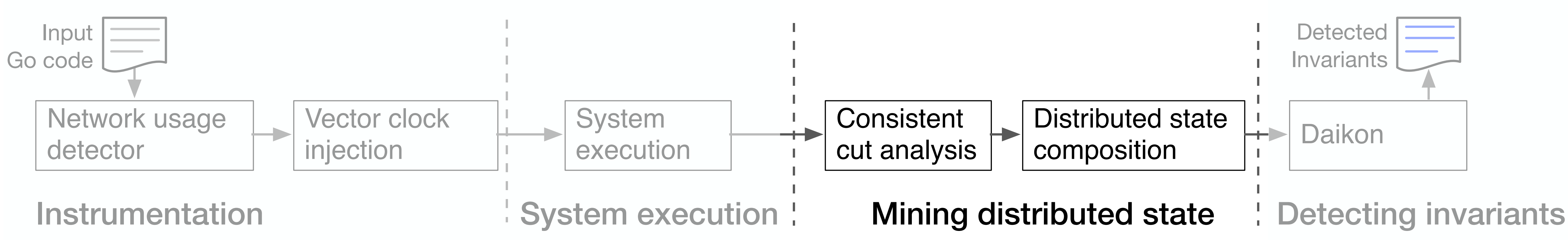


1. Consistent Cuts
2. Ground States
3. State Bucketing

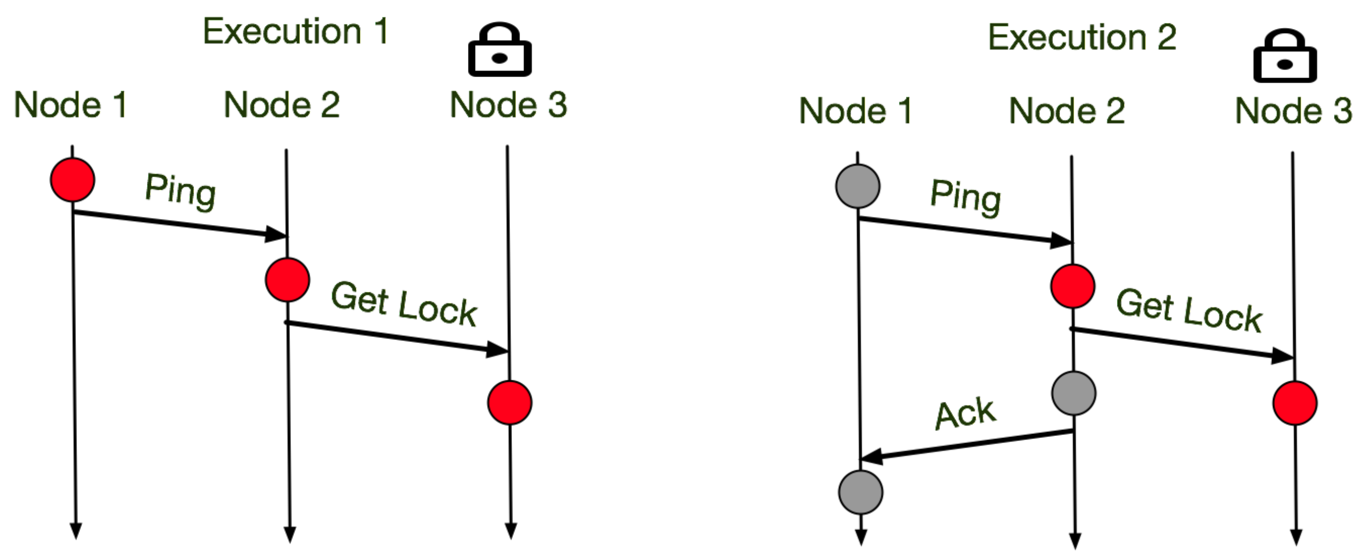


Matching logging locations

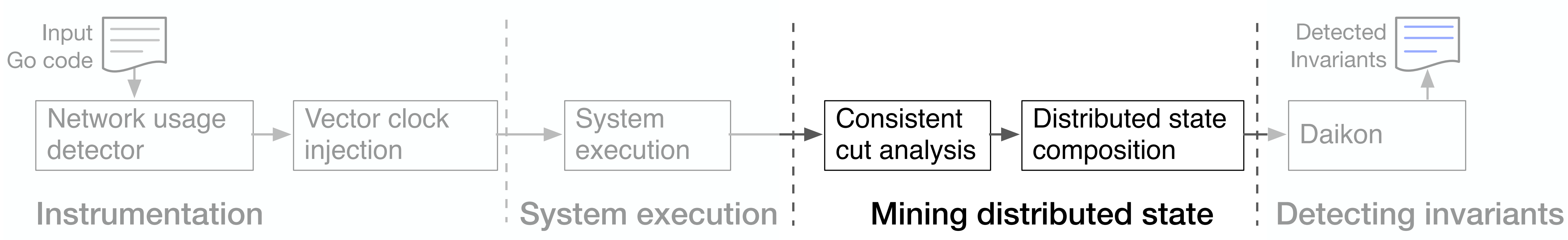
Reasoning about global state



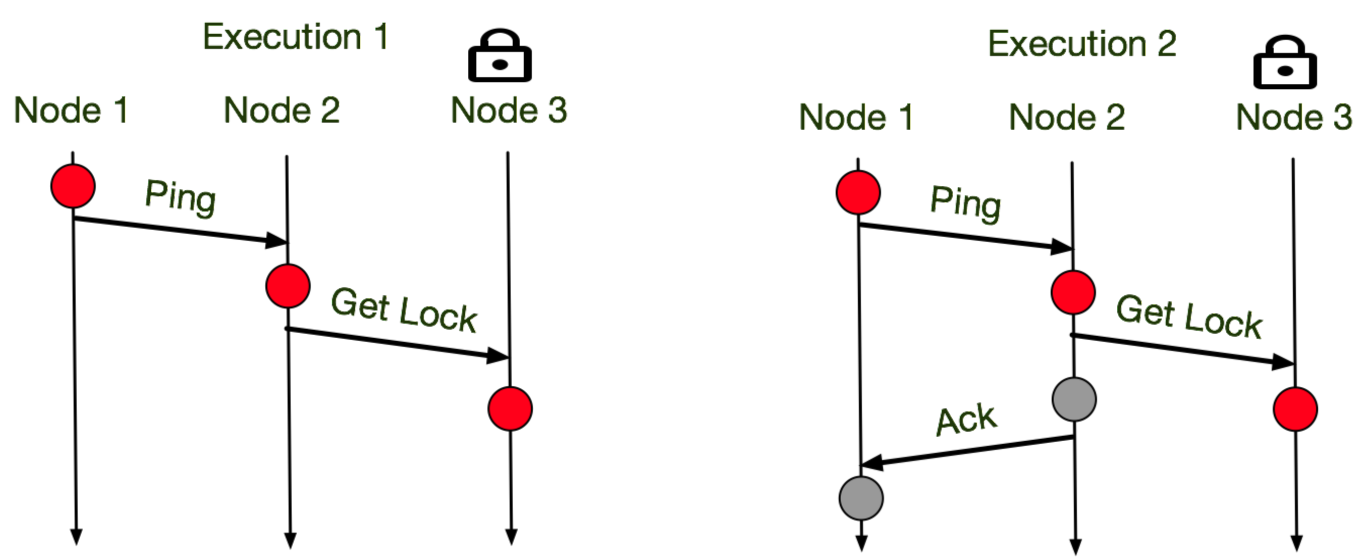
1. Consistent Cuts
2. Ground States
3. State Bucketing



Reasoning about global state

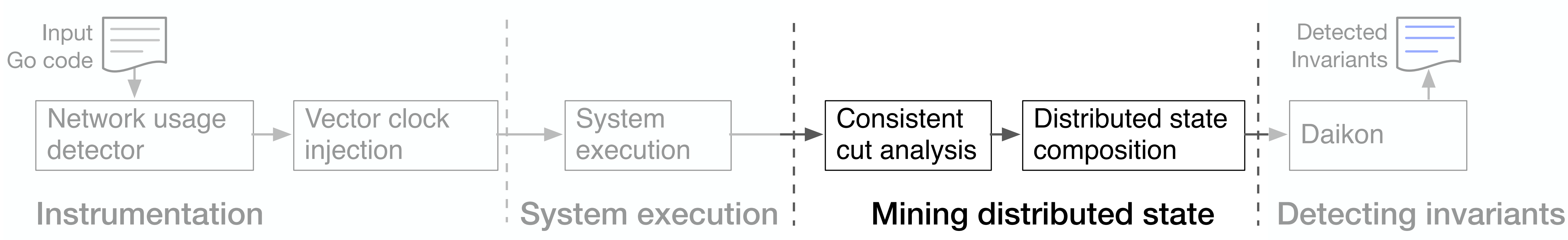


1. Consistent Cuts
2. Ground States
3. State Bucketing

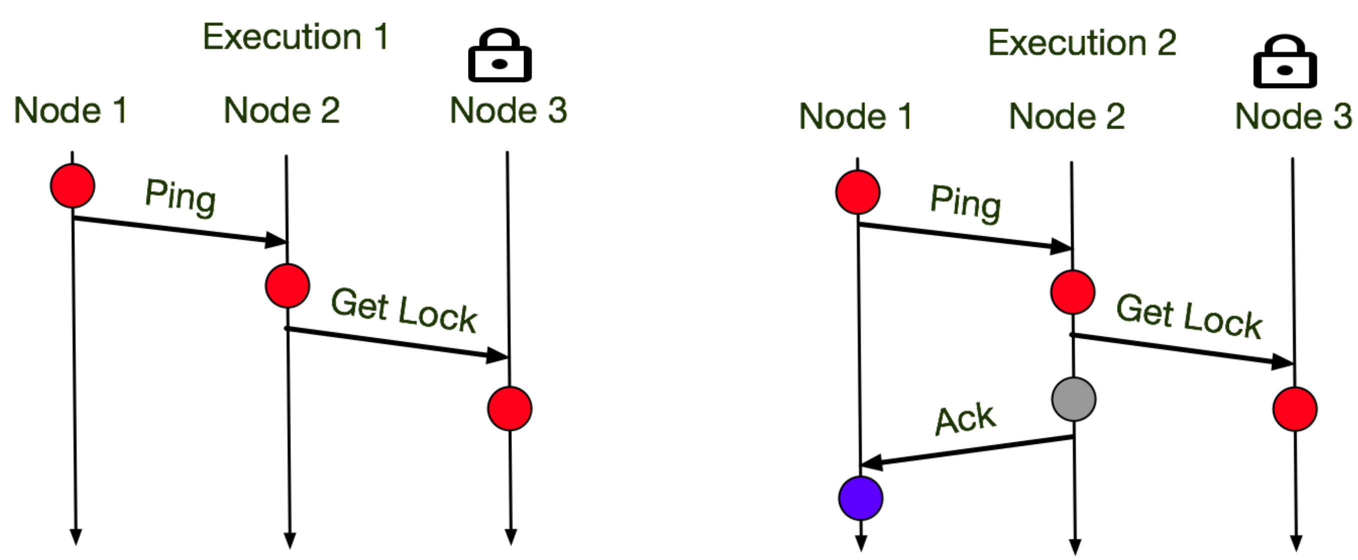


Matching consistent state cuts

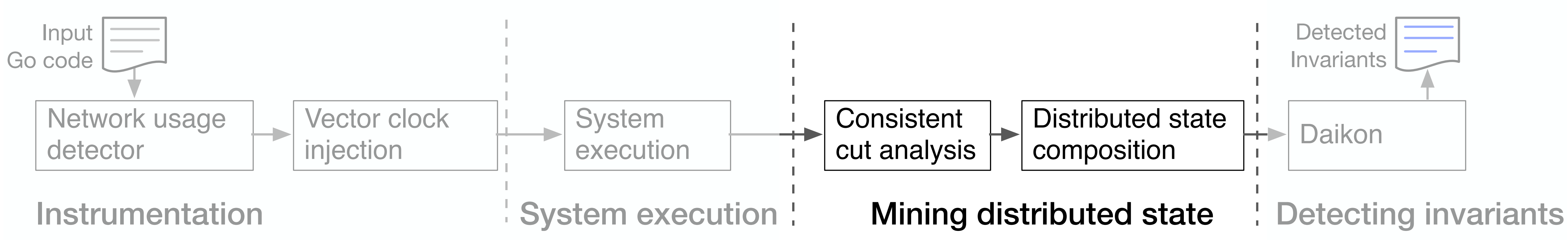
Reasoning about global state



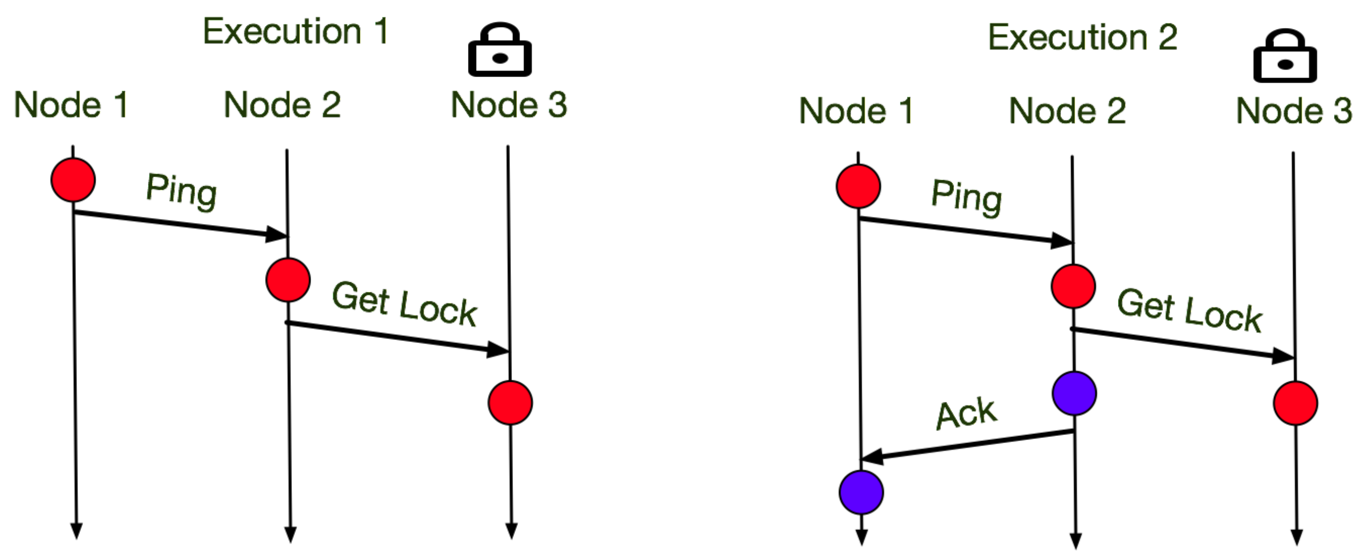
1. Consistent Cuts
2. Ground States
3. State Bucketing



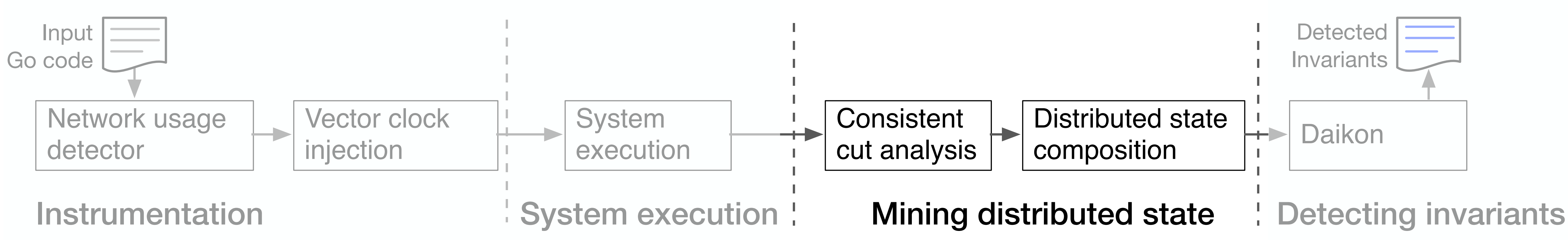
Reasoning about global state



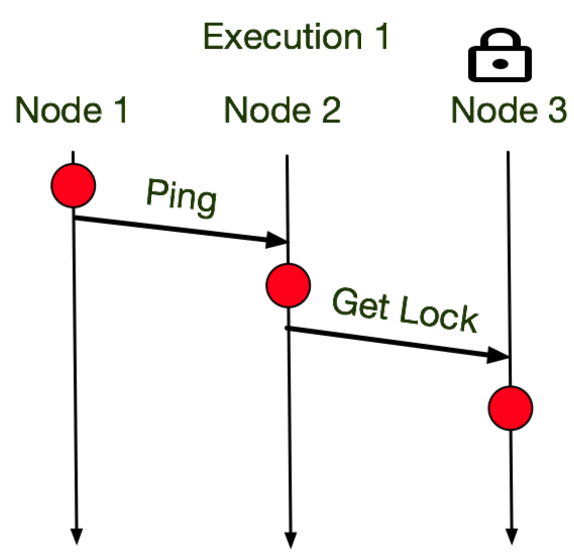
1. Consistent Cuts
2. Ground States
3. State Bucketing

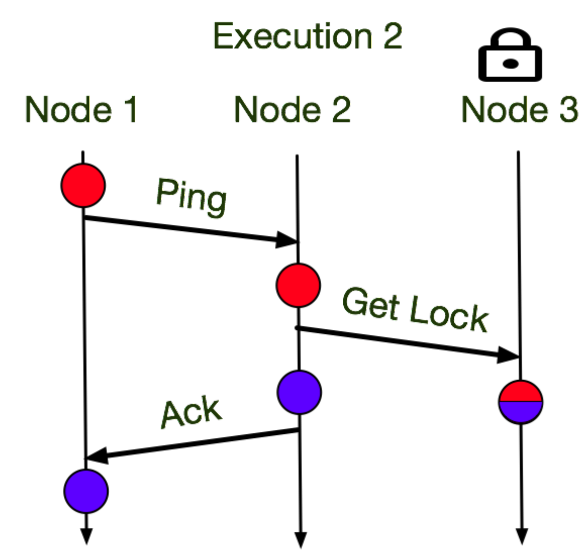


Reasoning about global state

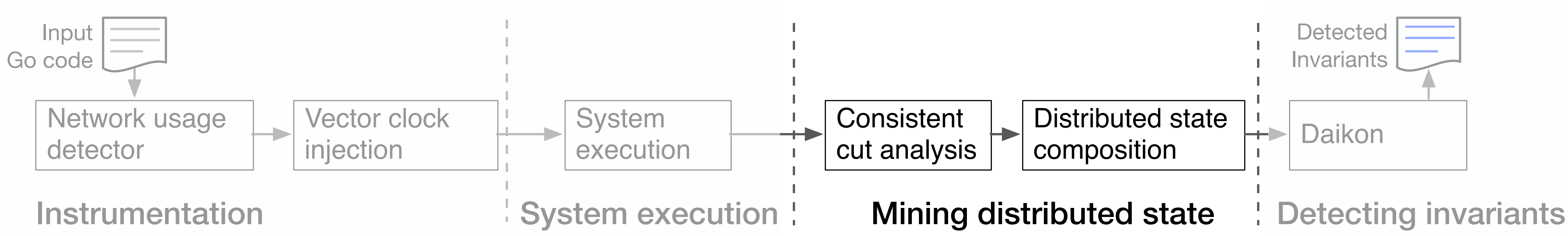


1. Consistent Cuts
2. Ground States
3. State Bucketing



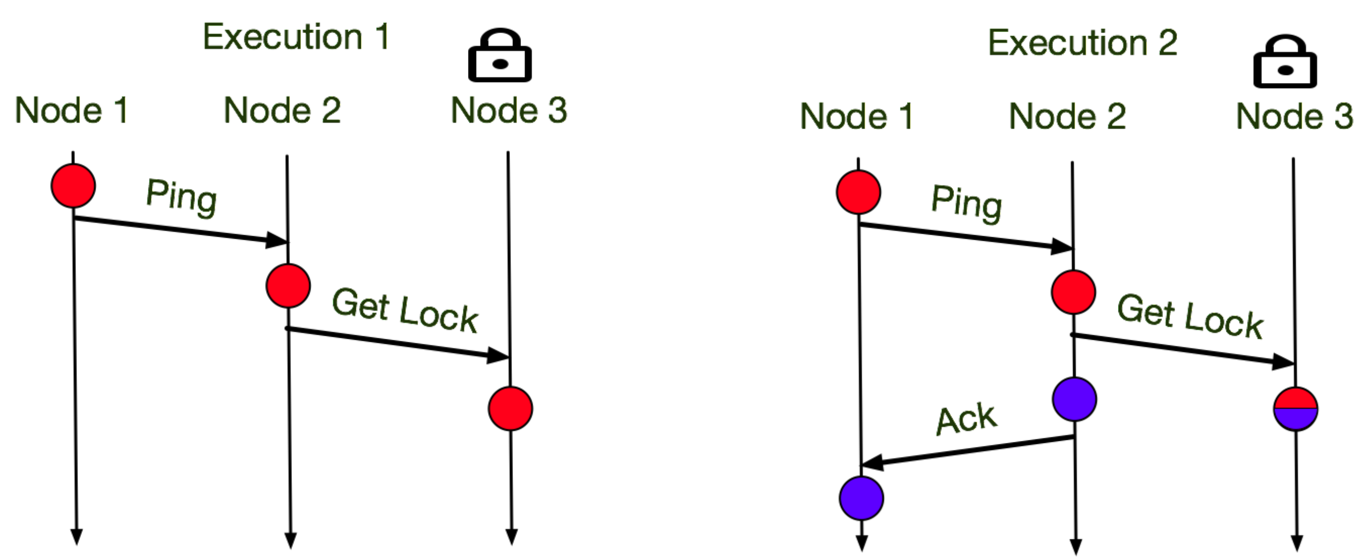


Reasoning about global state

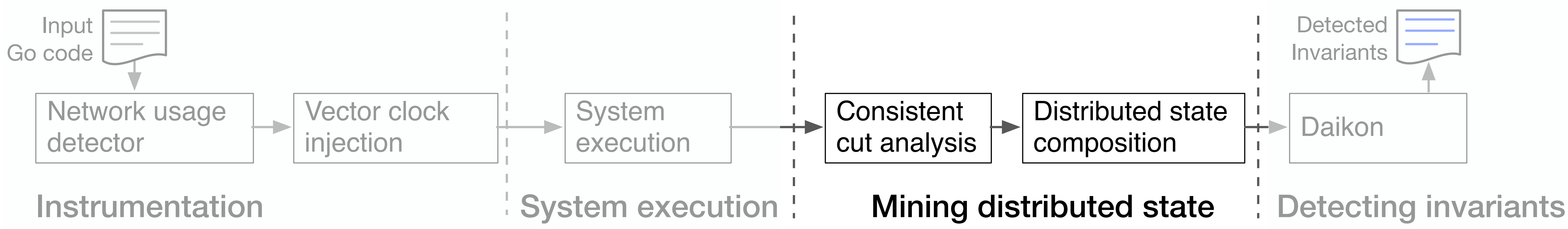


1. Consistent Cuts
2. Ground States
3. State Bucketing

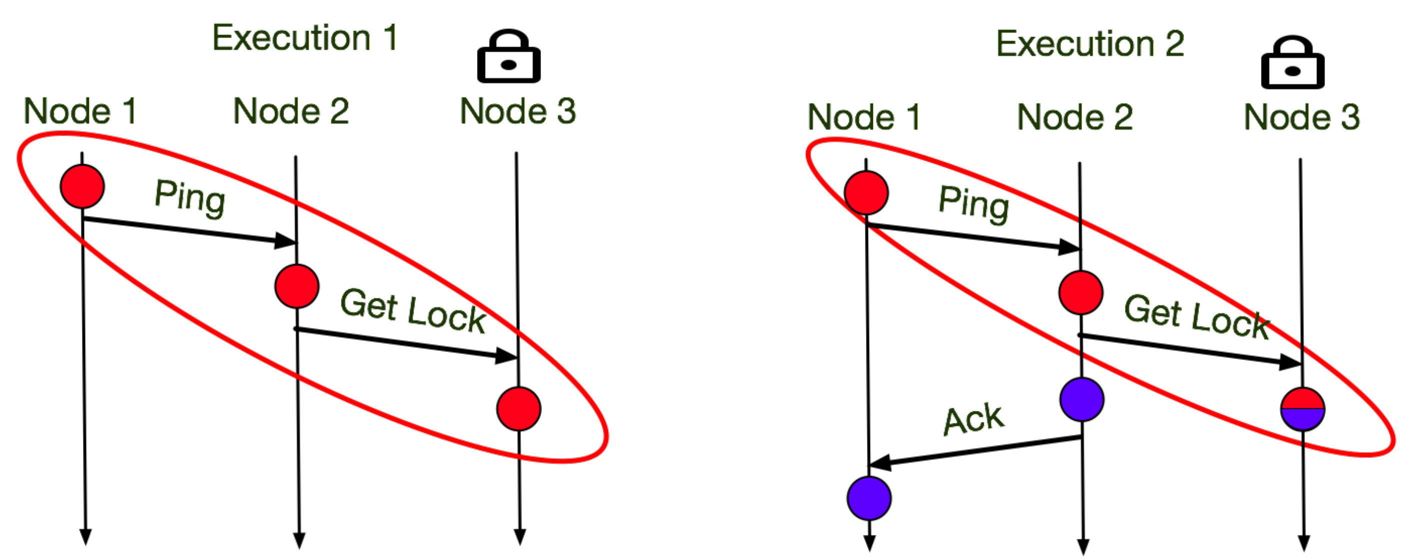
Scalability: only process "ground states" (no msgs in flight)



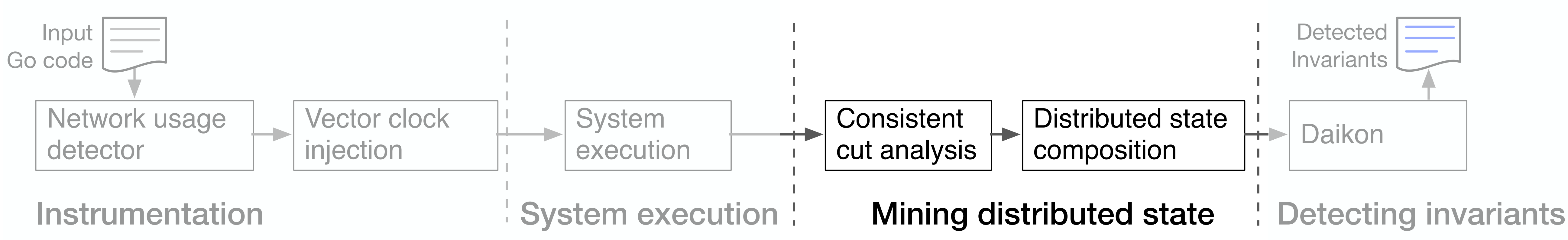
Reasoning about global state



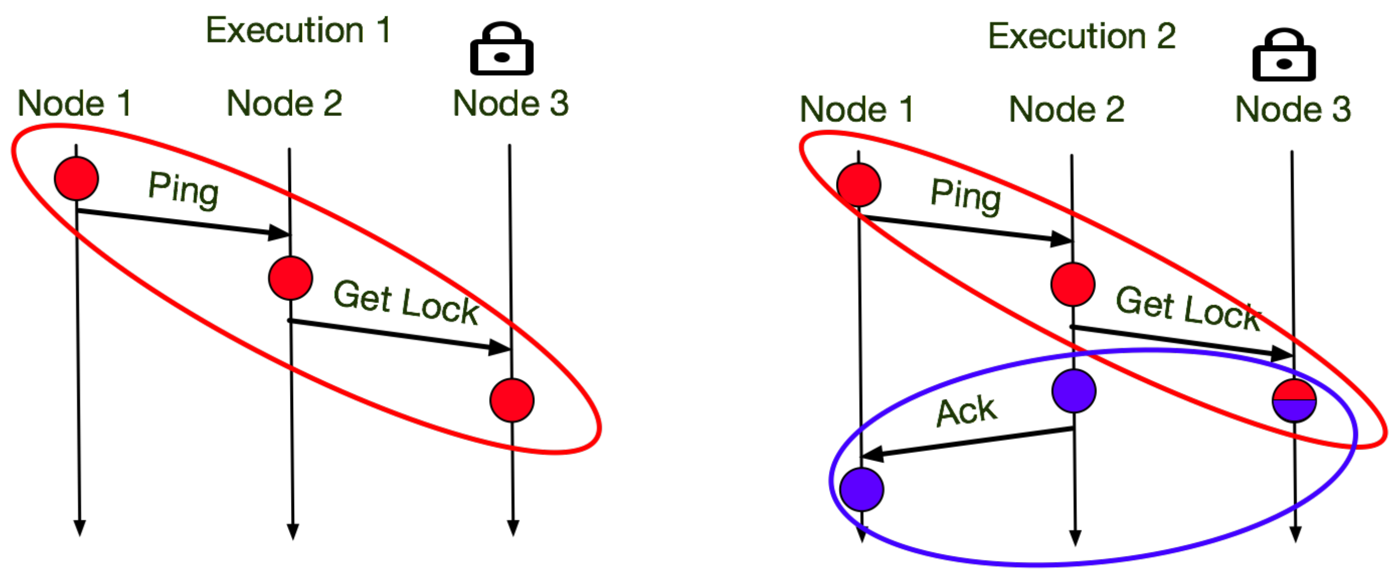
1. Consistent Cuts
2. Ground States
3. State Bucketing



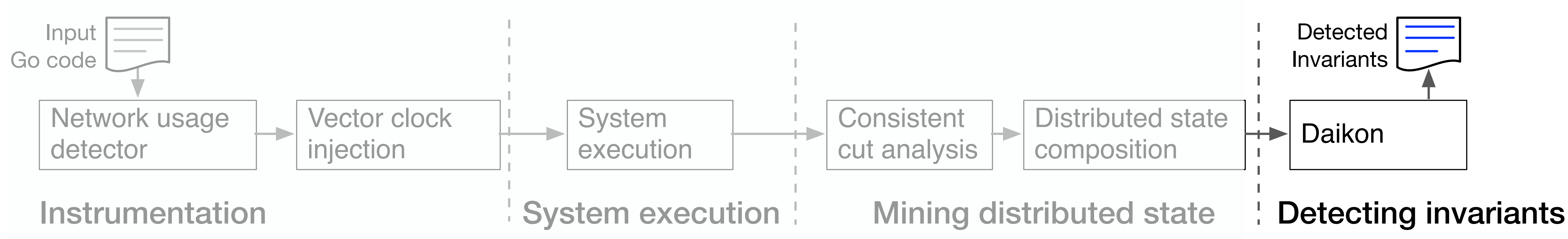
Reasoning about global state

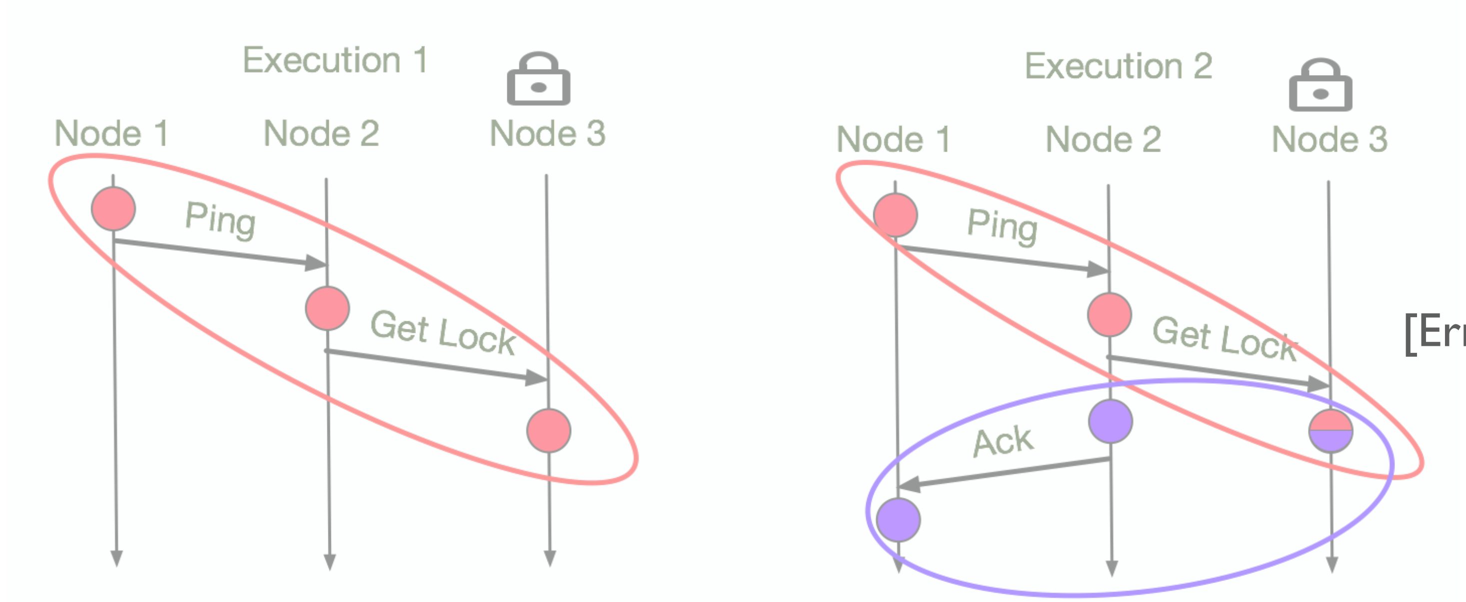


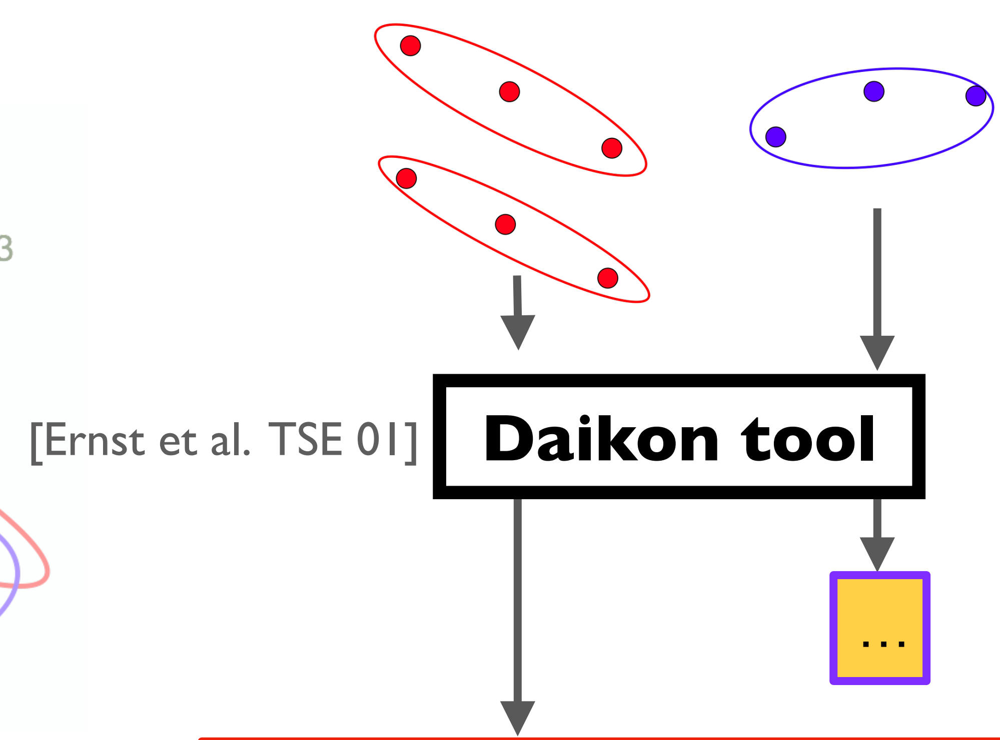
1. Consistent Cuts
2. Ground States
3. State Bucketing



From concrete values to abstract relations



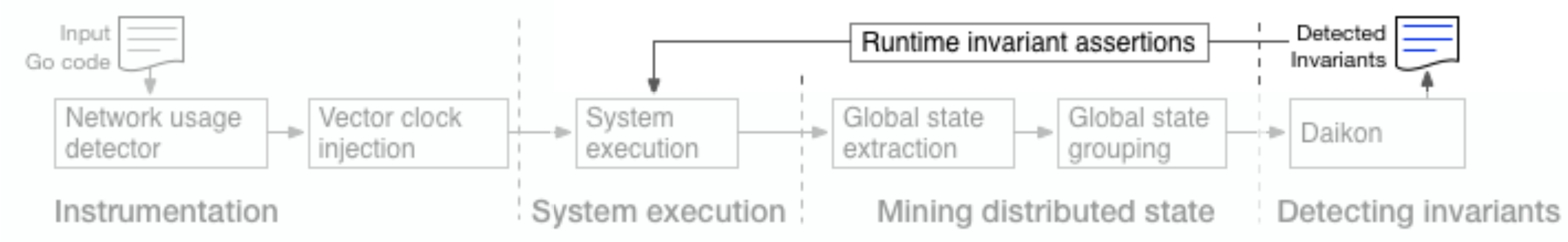




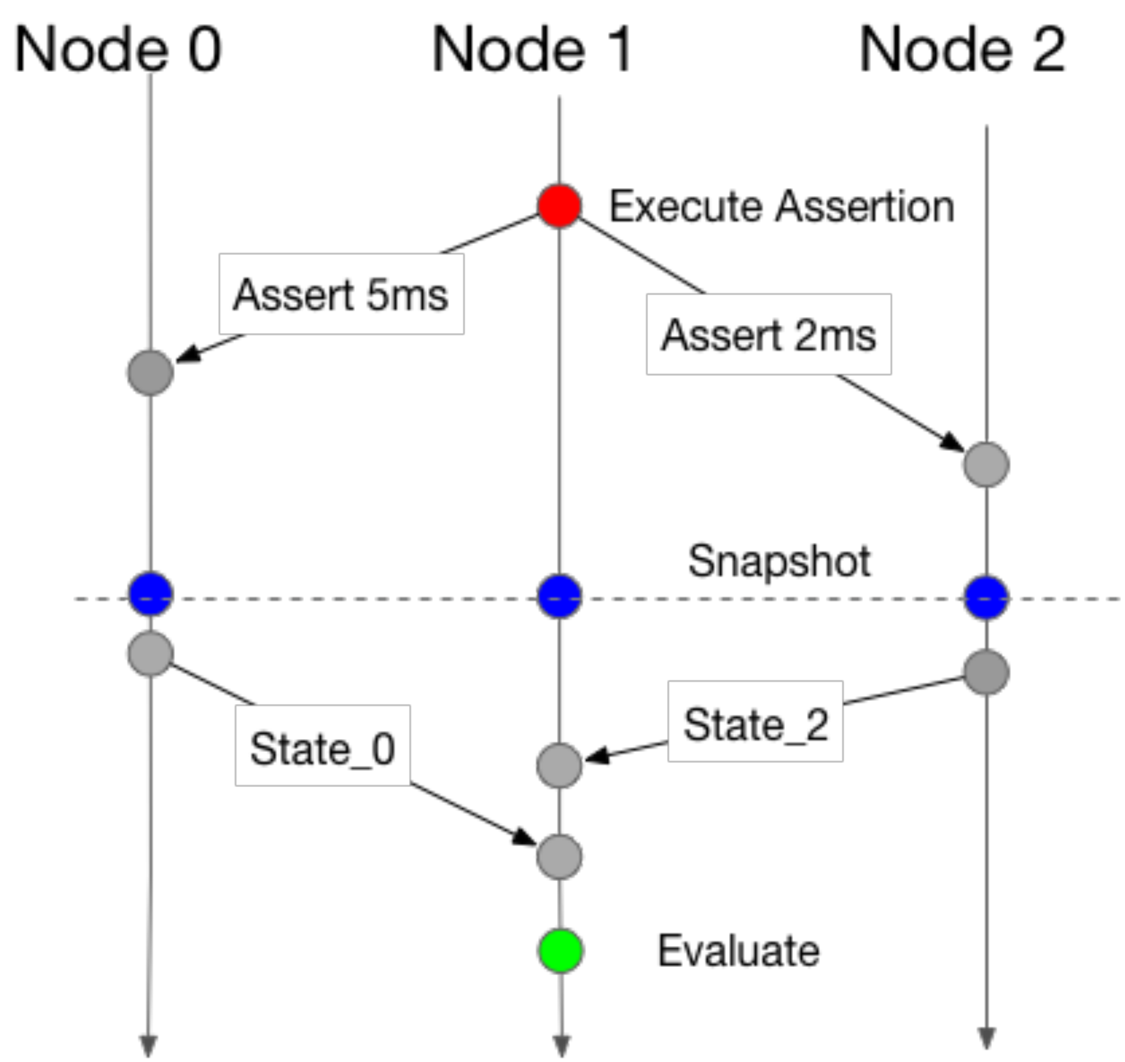
"likely" invariants

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 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

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

UBC

e.g., Etcd ~ 120K Lines of Code

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 key RAFT correctness properties
  - Just 2 annotations sufficient to detect all invs
  - Traces from YCSB-A workload generate enough diversity

Probabilistic assertions

Raft invariant
Strong leadership
Leadership agreement
Log matching

Constructed and injected silent bugs for each invariant into a running etcd system

Probabilistic assertions

Raft invariant	LOC
Strong leadership	11
Leadership agreement	13
Log matching	72

LOC in assertion
(developer must write)

Probabilistic assertions

Raft invariant	LOC	P=1.0	P=0.1	P=0.01
Strong leadership	11	0.07	0.05	2.96
Leadership agreement	13	0.36	0.34	6.75
Log matching	72	2.22	4.35	6.07

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

Probabilistic assertions

Raft invariant	LOC	P=1.0	P=0.1	P=0.01
Strong leadership	11	0.07	0.05	2.96
Leadership agreement	13	0.36	0.34	6.75
Log matching	72	2.22	4.35	6.07

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

Future work

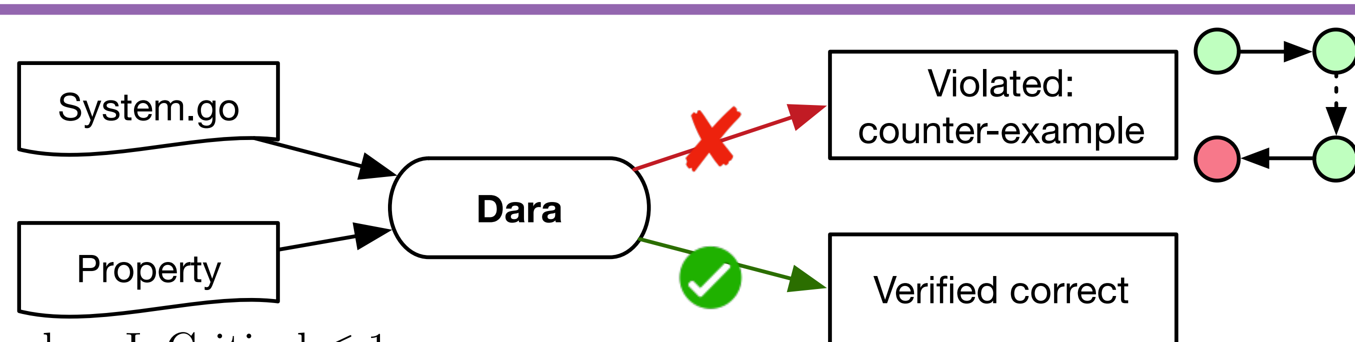
- Root cause analysis\impact analysis\etc
- Distributed test case generation
- Extend analysis to temporal invariants

Ongoing: distributed model checking

1. Dinv
Spec miner

[ICSE 2018]

2. Dara
Model checker



Is my system correct?

3. PGo
Compiler

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

PlusCal model → Go lang

```
var v int
...
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 := v
      v_new := u
      u = u_new
      v = v_new
    }
    u = u - v
  }
  fmt.Printf("24 %v have gcd %v\n", v_init, v)
}
}
```

Model checking (MC)

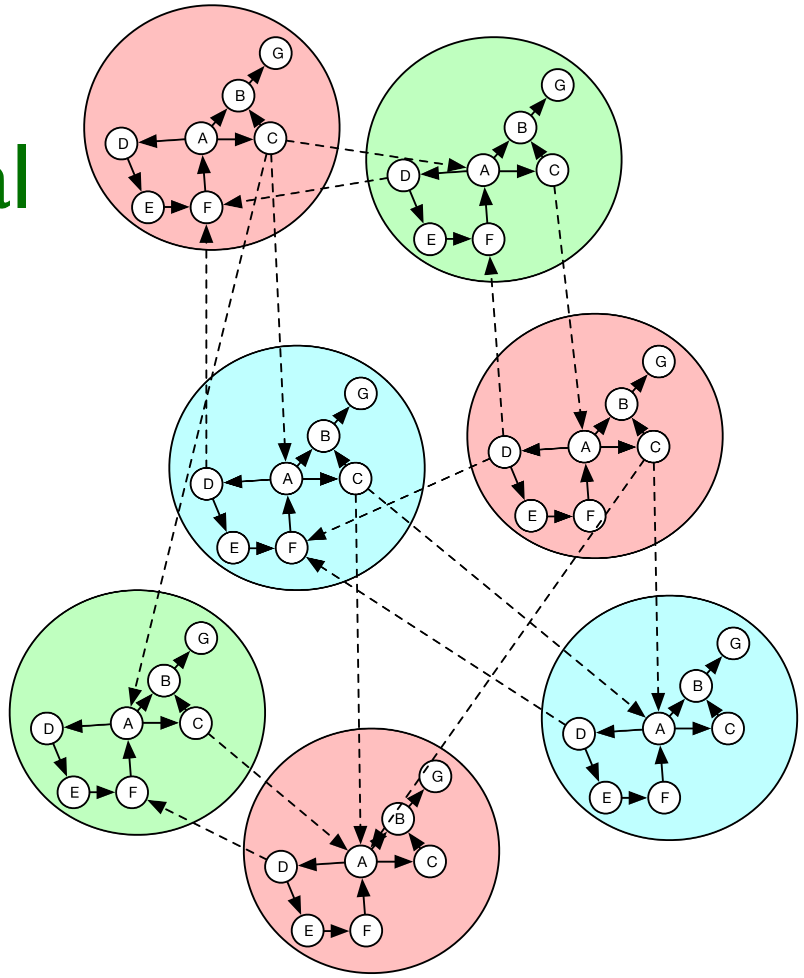
- *"Exhaustive testing"*
- Explore the state space of a system w.r.t some **model**
- **Check** predicate at each state (safety property) for violation
- Violation is a path = bug in the model: output to developer
- Main challenge: *state space explosion*

Trade-offs in model checking (MC)

Concrete (implementation-level) MC

[SAMC OSDI'14, MODIST NSDI'09, Demi NSDI'16]

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

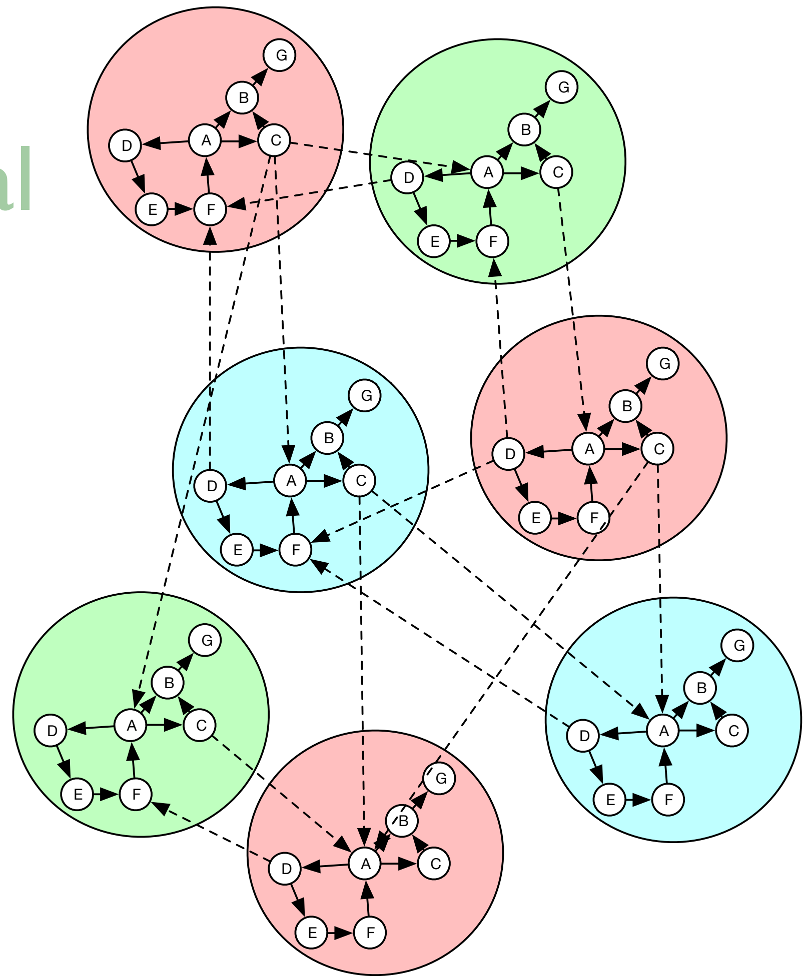


Trade-offs in model checking (MC)

Concrete (implementation-level) MC

[SAMC OSDI'14, MODIST NSDI'09, Demi NSDI'16]

- 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

[Chapar POPL'16, IronFleet SOSP'15, VerdiPLDI'15, Lamport et.al SIGOPS'02, Holtzman 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?

Dara

Concrete traces → Abstract model

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

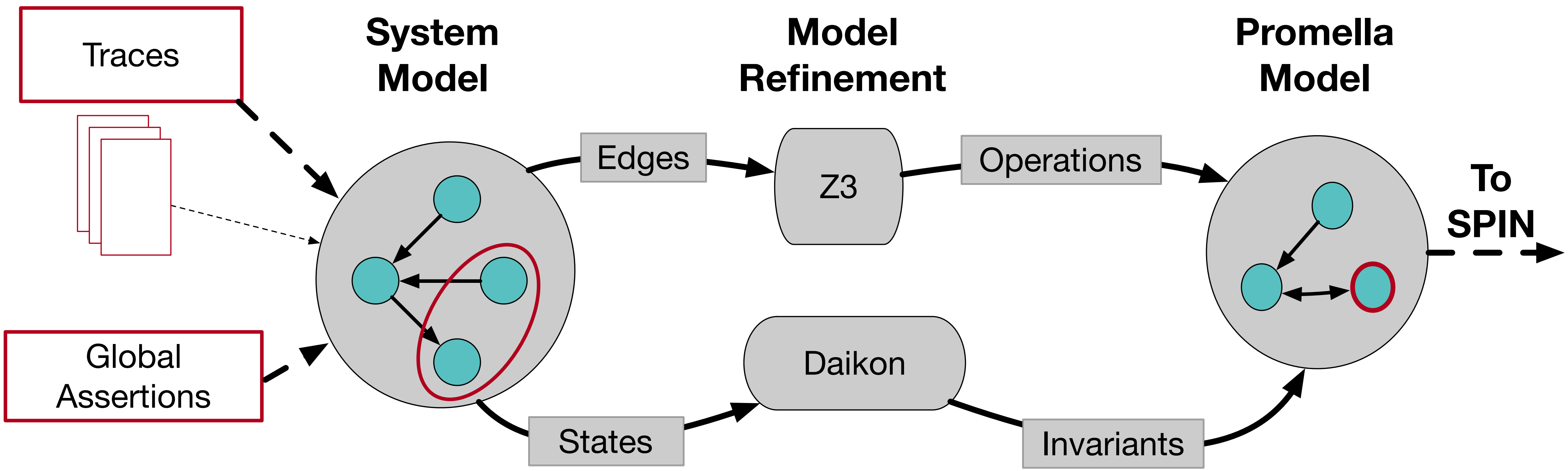
- Map each abstract violation into a concrete violation (replay)
    - Attempt to reproduce the abstract execution by replaying it on the actual system

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

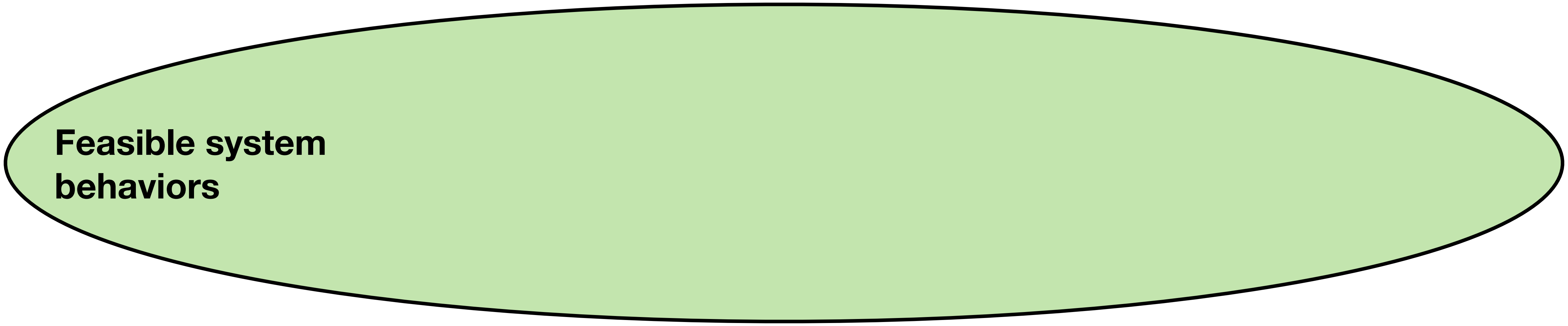
Idea 3: refine the abstract model with counter-examples

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

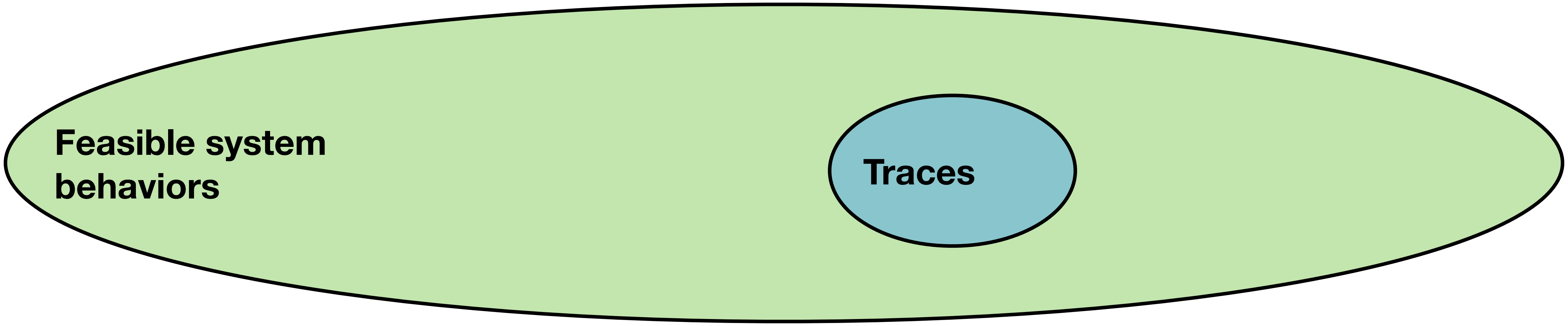
Concrete traces → Abstract model



High-level view of the approach

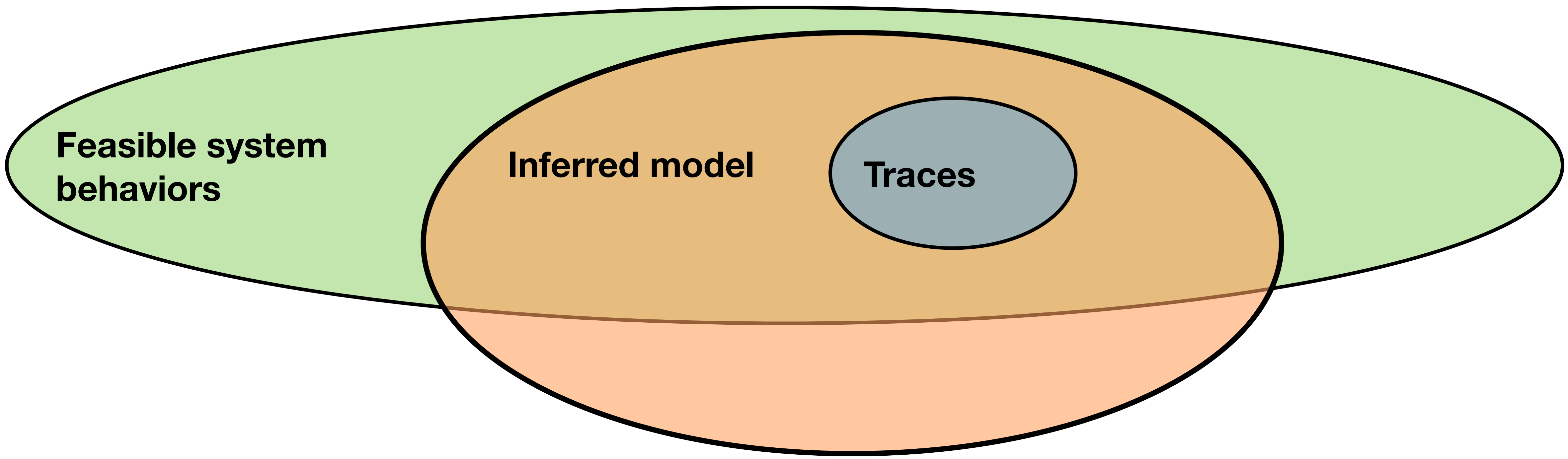


High-level view of the approach



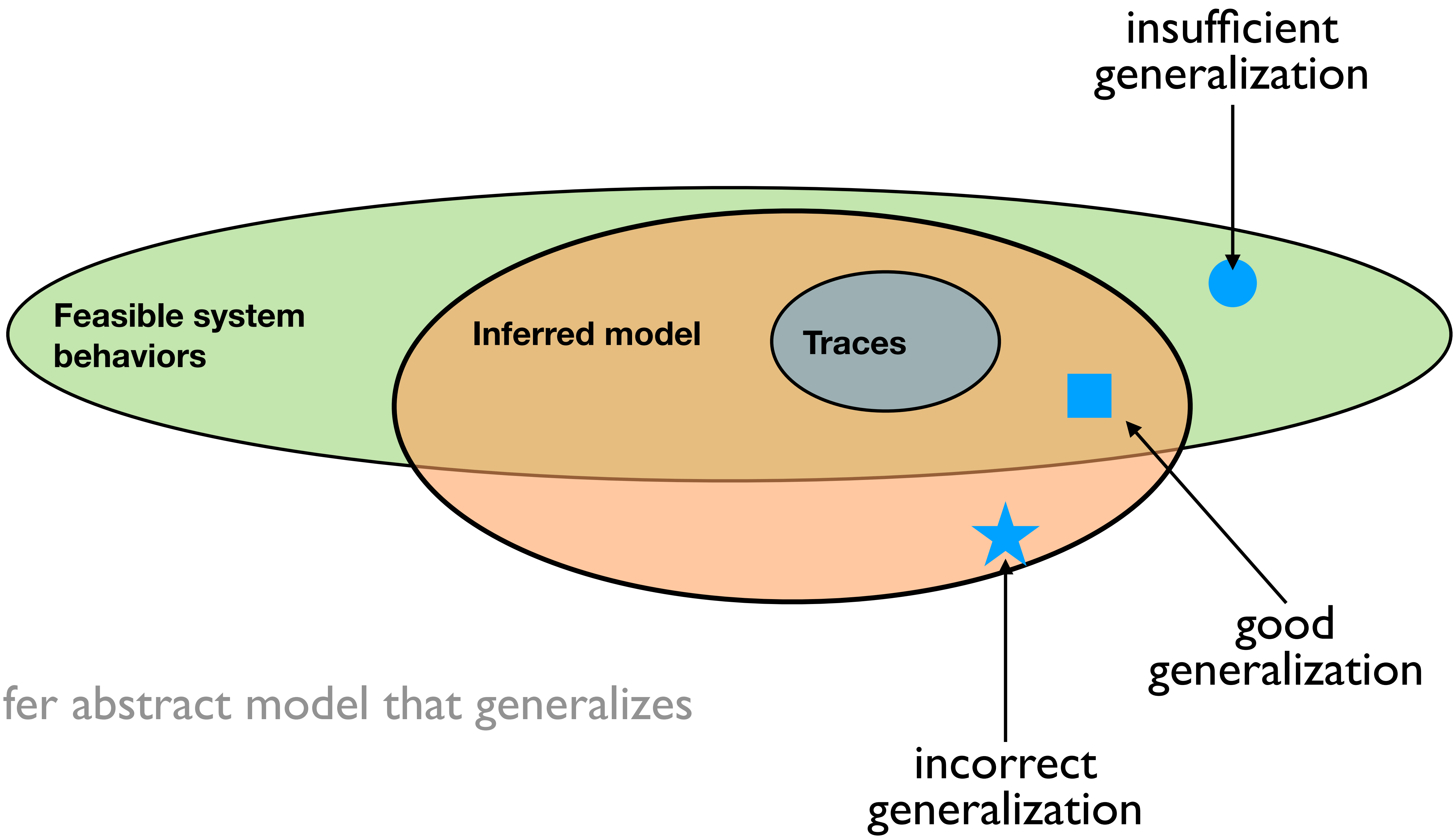
Generate traces using the concrete MC: exhaustive.. but bounded/incomplete

High-level view of the approach



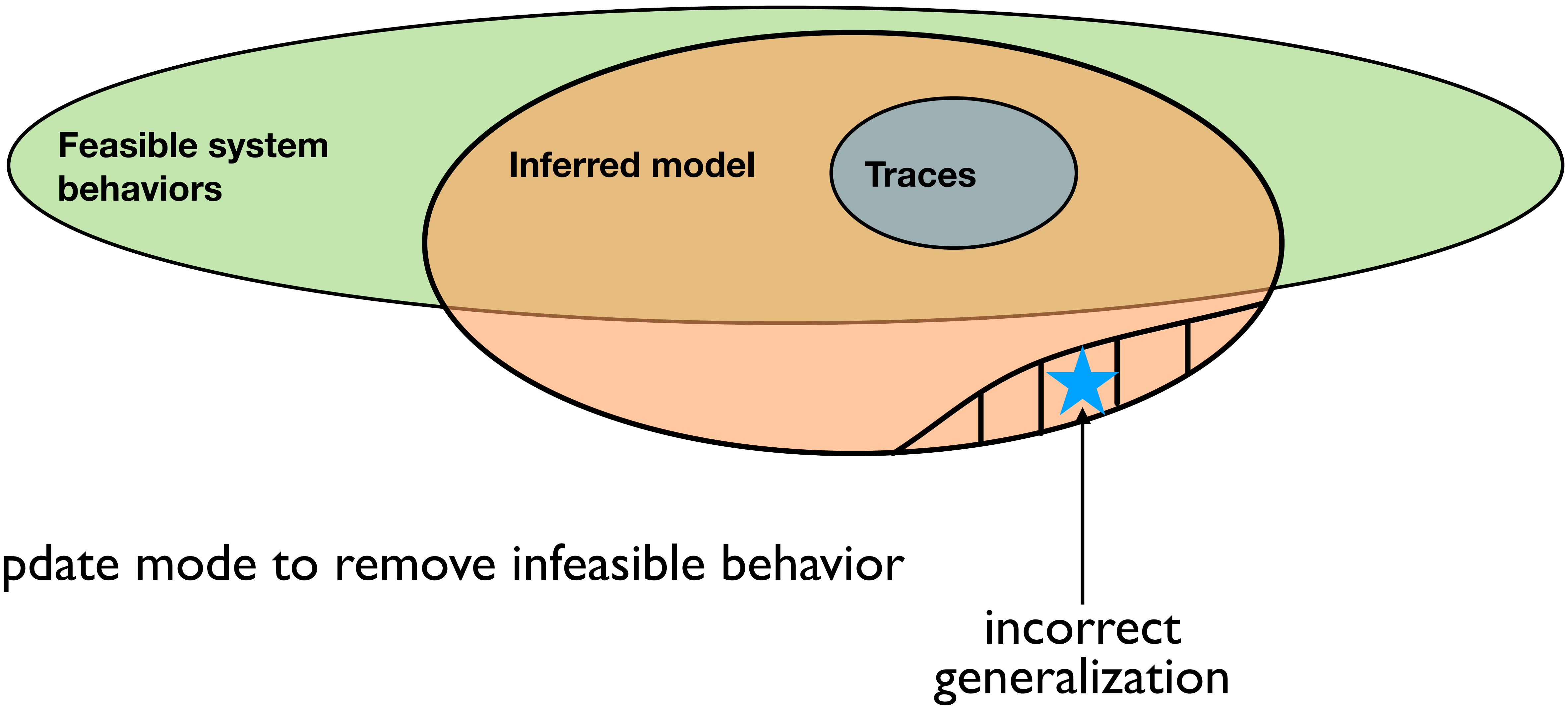
Infer abstract model that generalizes

High-level view of the approach



Infer abstract model that generalizes

High-level view of the approach

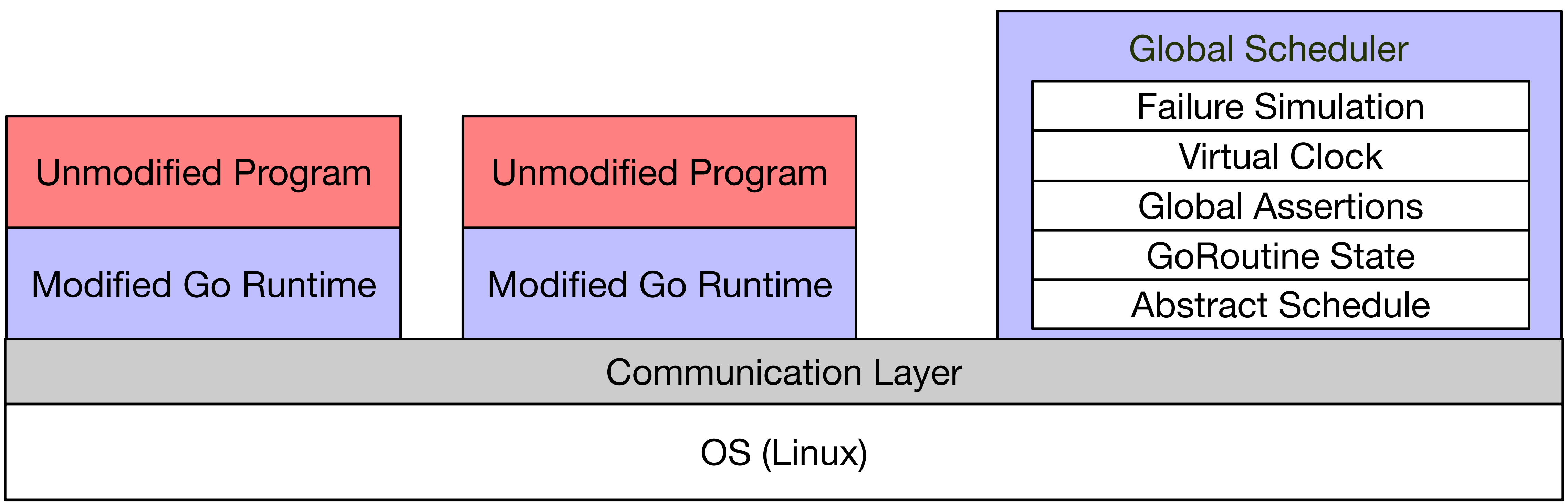


Update mode to remove infeasible behavior

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



Dara current status

- 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

Ongoing: compiling distributed systems

1. Dinv
Spec miner

[ICSE 2018]

2. Dara
Model checker

3. PGo
Compiler

```
--algorithm Euclid { \** @PGo{ arg int N }@PGo
 (** @PGo{ var int u }@PGo
     @PGo{ var int v }@PGo
     @PGo{ var int v_init }@PGo
 **)
 variables u = 24;
           v \in 1 .. N;
           v_init = v;
 {
   while (u # 0) {
     if (u < v) {
        u := v || v := u;
     };
     u := u − v;
```

PlusCal model

```
var v int
var u int
var v_init int
var N int

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 := v
        v_new := u
        u = u_new
        v = v_new
```

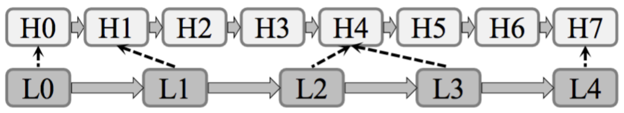
Go lang

Can I implement my (correct) system faster?

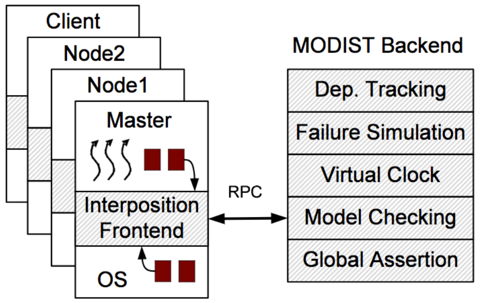
Existing verification approaches

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

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



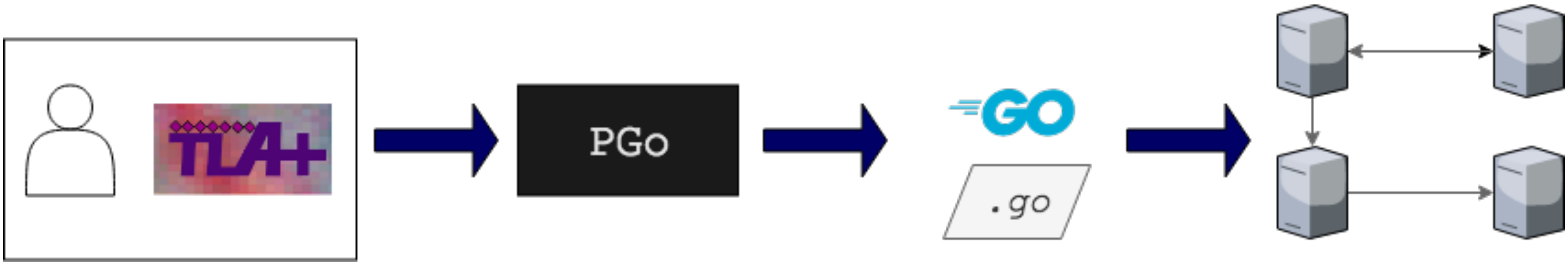
- **`MODIST`** checks the implementation rather than a specification [NSDI'09]



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!

TLC

PGo

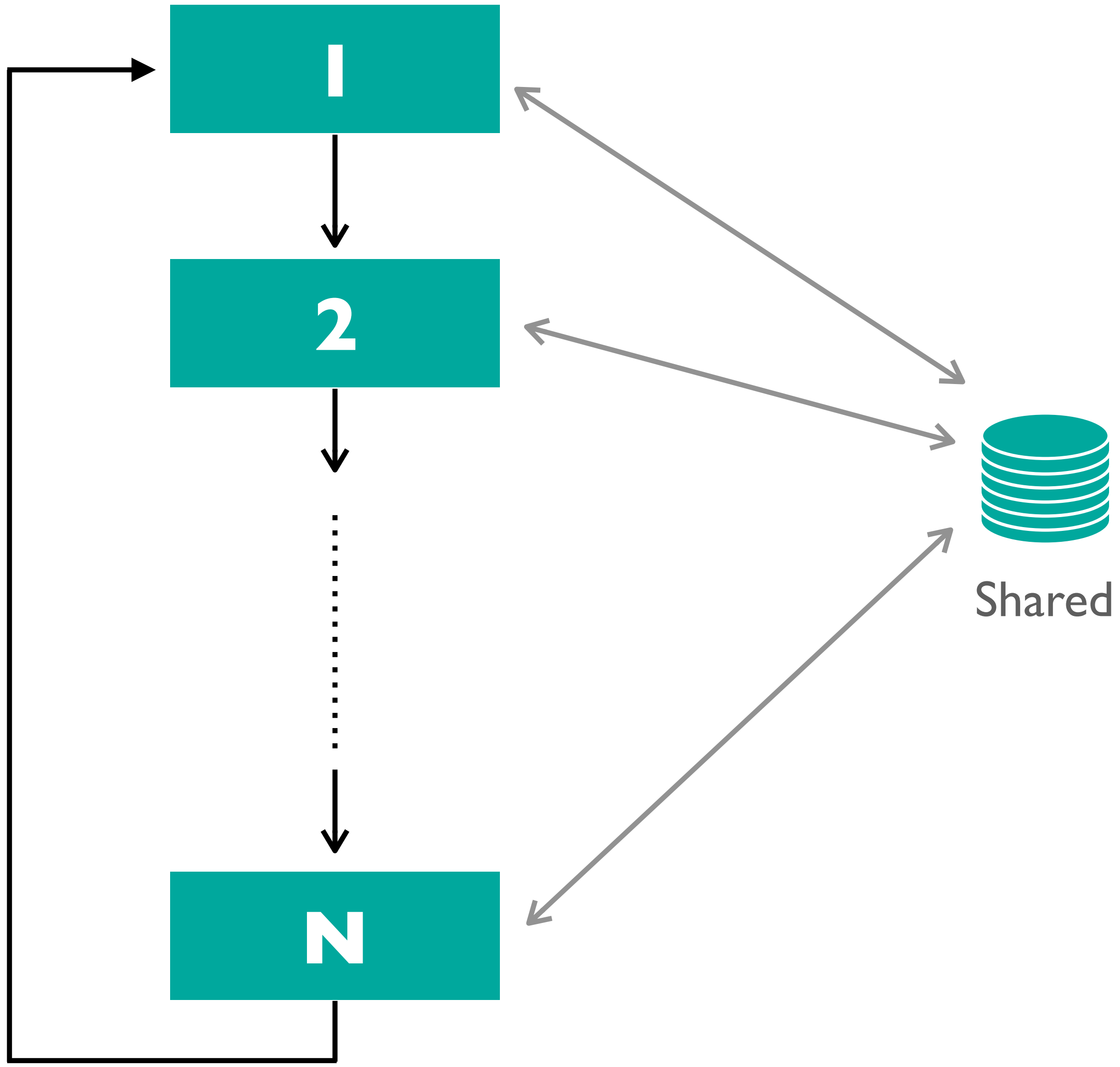
Transition from *design* (specification) to *implementation* is **automated**

[1] Killian et al. *Mace: Language Support for Building Distributed Systems*. *PLDI 2007*

PGo Workflow: (1) Example System

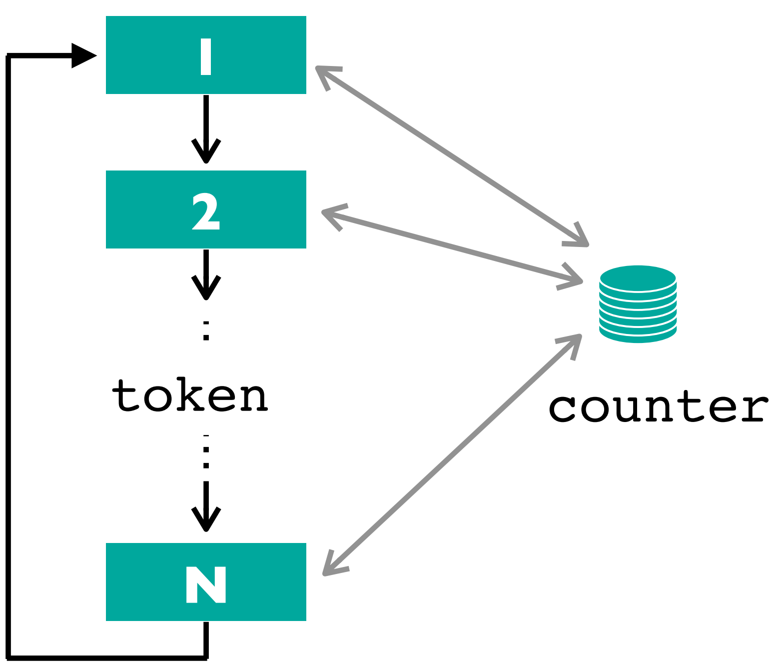
Round-Robin Resource Sharing

Developer writes
specification



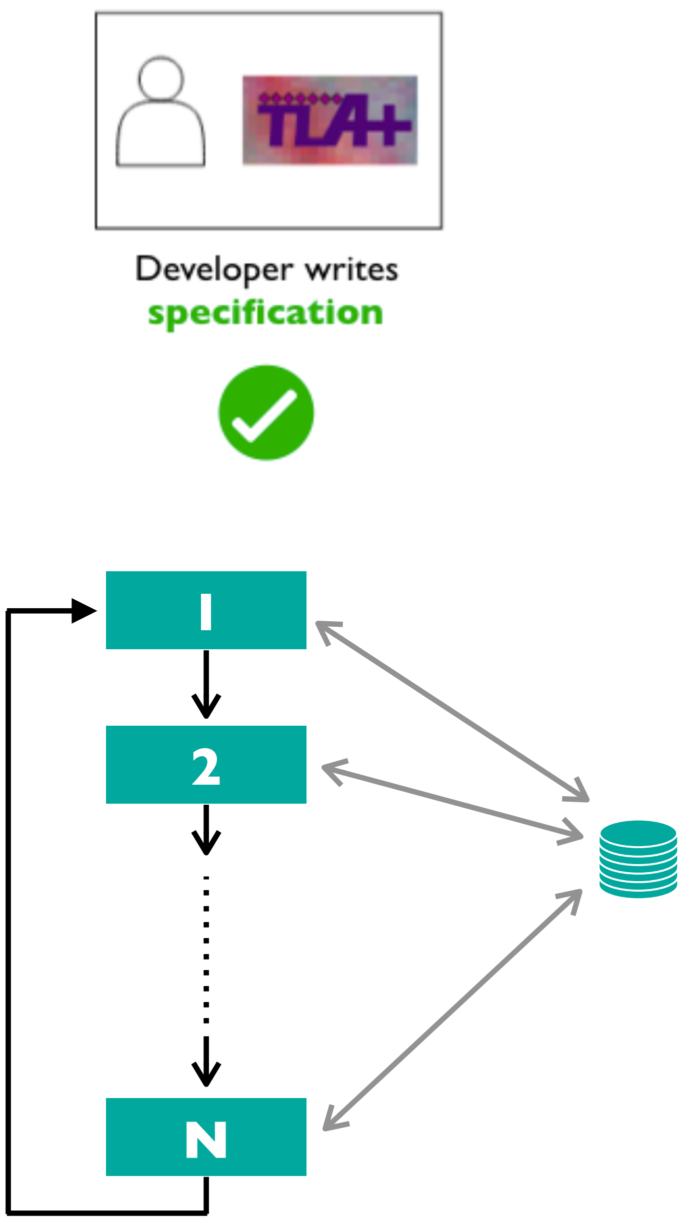
PGo Workflow: (1) PlusCal Spec

Developer writes **specification**




```
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

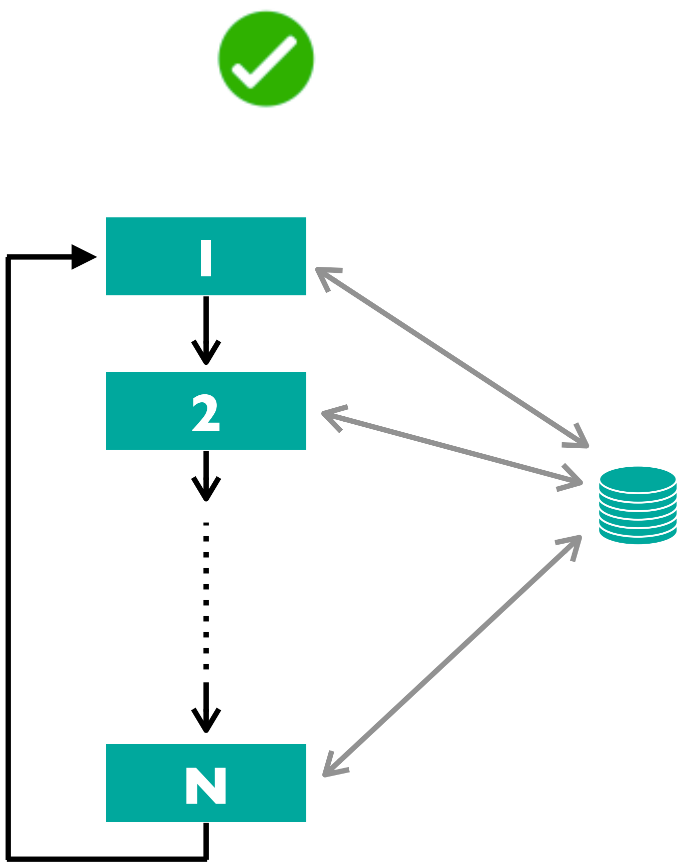
Token is within bounds	`token \in 0..procs-1`

Properties

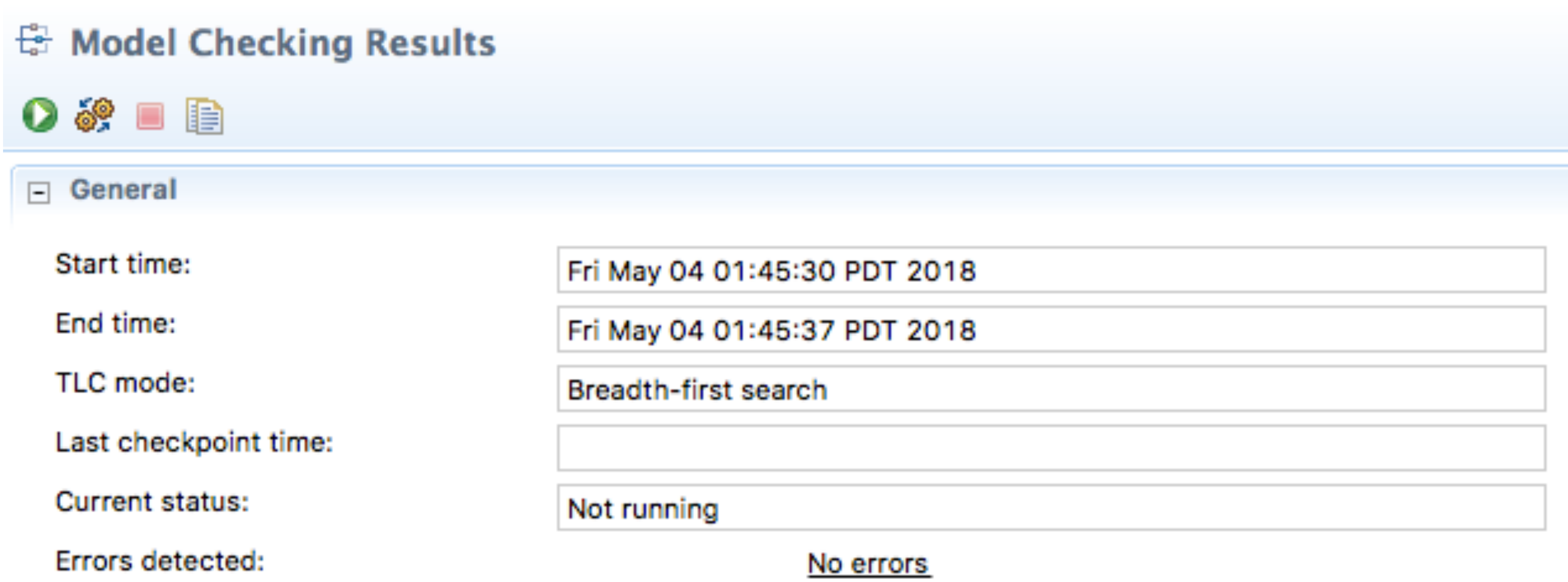
Counter Converges	`Termination => (counter = procs * iters)`
Processes Get the Token	`\A p \in \ProcSet : <>(token = p)`

PGo Workflow: (1) Verifying

Developer writes **specification**

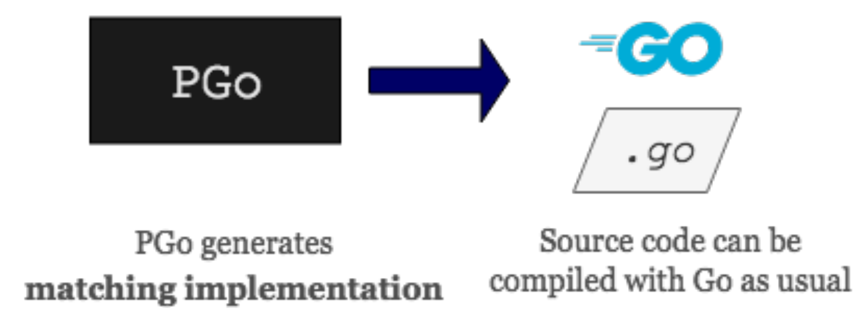


Model Checked with TLC!



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

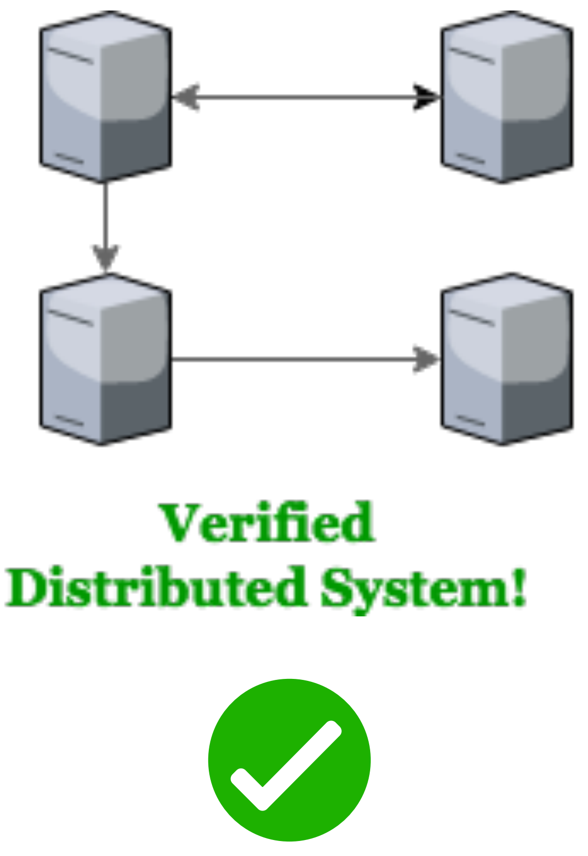



```
fair process (P \in 0..procs-1)
variable i = 0;
{
    w: while ( i < iters) {
        inc: await token = self;
             counter := counter + 1;
             token := (self + 1) % procs;
             i := i + 1;
    }
}}
```

PGo Workflow: (3) Using Compiled Code

- 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
```



Current Status

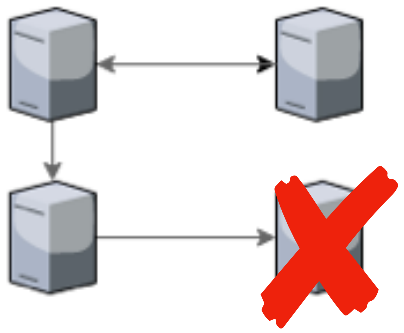
- PGo is 25K LOC (compiler) and 3K (runtime)
- Able to compile concurrent and distributed systems
- Support for different strategies to deal with global state in a distributed system
- Designing *ModularPlusCal:* extending PlusCal with more modularity features for large systems, and more separation of 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

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 * # 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)

PGo work in progress

- 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)



MODULE *SyncQueue*

CONSTANT *Message*
VARIABLES *in*, *out*
$Internal(q) \triangleq$ INSTANCE *SyncQueueInternal*

$Fifo \triangleq \exists\, q : Internal(q)!FifoI$

PGo Limitations

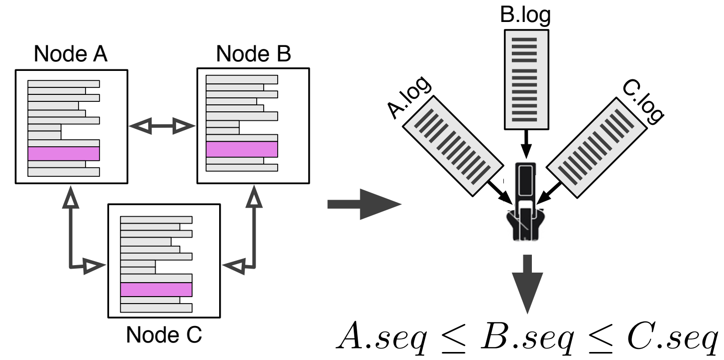
- Specifications are very high level: not everything can be compiled efficiently
- Requires developers to also specify environment during compilation (e.g., number of processes, transport protocol, etc).
- Both the PGo compiler and the associated runtime need to be trusted to claim correctness

Program analysis for distributed systems

1. Dinv [ICSE 2018]

Spec miner

https://bitbucket.org/bestchai/dinv



Thanks to our Funders:

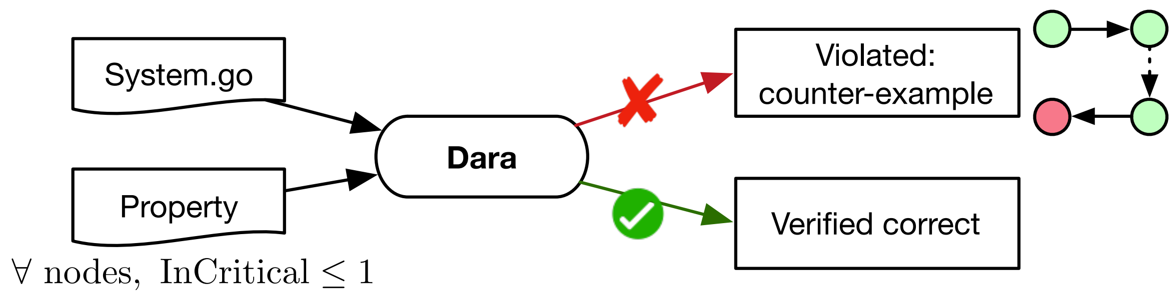
HUAWEI

NSERC CRSNG

2. Dara

Model checker

https://github.com/DARA-Project



3. PGo

Compiler

https://github.com/UBC-NSS/pgo

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

PlusCal model

```
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 := v
        v_new := u
        u = u_new
        v = v_new
      }
      u = u - v
    }
    fmt.Printf("24 %v have gcd %v\n", v_init, v)
  }
}
```

Go lang

Bridging gap between design and implementation

Backup slides

Dinv runtime overhead

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

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

Dinv runtime overhead

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

All Raft invariants can be detected with just two annotations

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

Dinv analysis time

System runtime (s)	Raft log (MB)	Raft analysis (s)
30	5.1	12.7
60	10.5	28.1
90	13.7	35.9
120	17.4	48.7
150	22.5	68.8
180	27.7	99.1

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