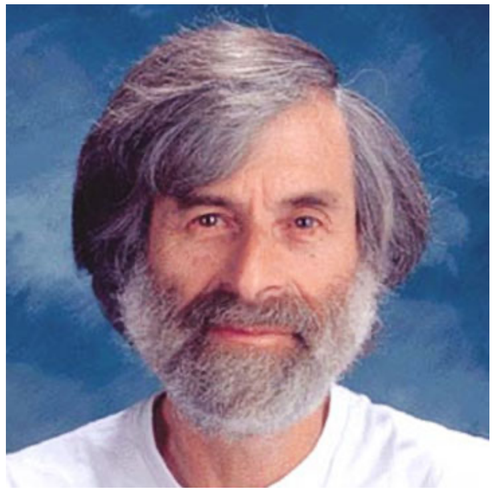
Protocol design with Alloy

Alcino Cunha

Protocol design

- Distributed algorithms (protocols) are hard to design
- Many critical systems nowadays are distributed
- Testing is ineffective
  - Too many interleavings
  - Bugs are subtle, due to specific race conditions
- Formal verification is mandatory

“If you’re not writing a program, don’t use a programming language.”



–Leslie Lamport

Protocol design

1. Model the (static) network configuration

2. Model the behaviour of the protocol with a transition system

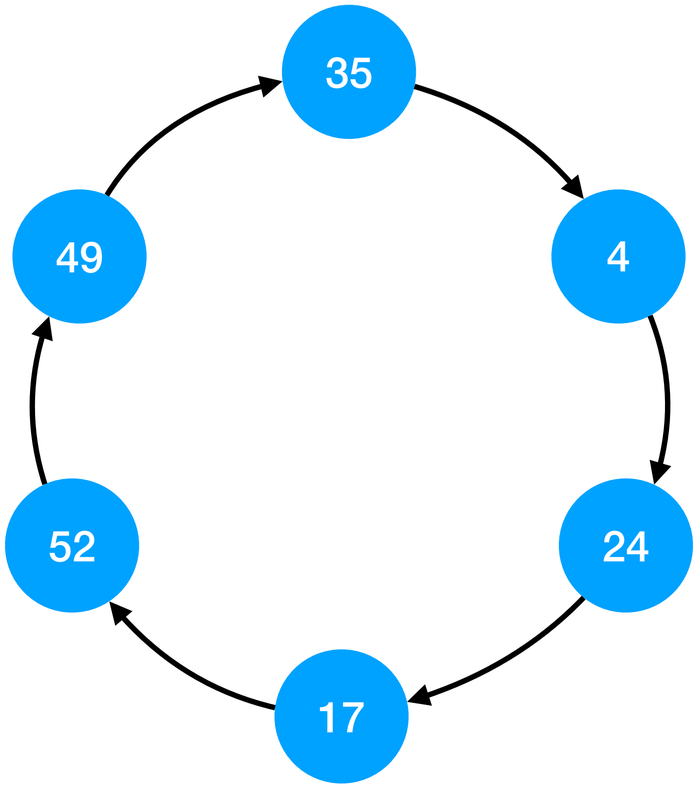
   - Declare the mutable data structures of the state

   - Specify the initial conditions and events

3. Validate the model

4. Specify and verify expected properties

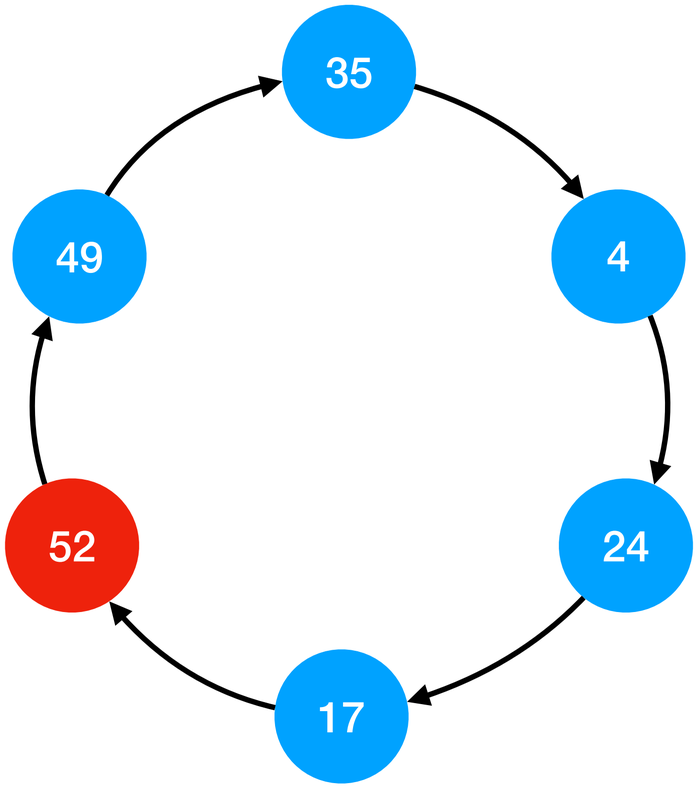
Leader election in a ring



Leader election in a ring

One and at most one leader will be elected!

Leader election in a ring



Network configuration

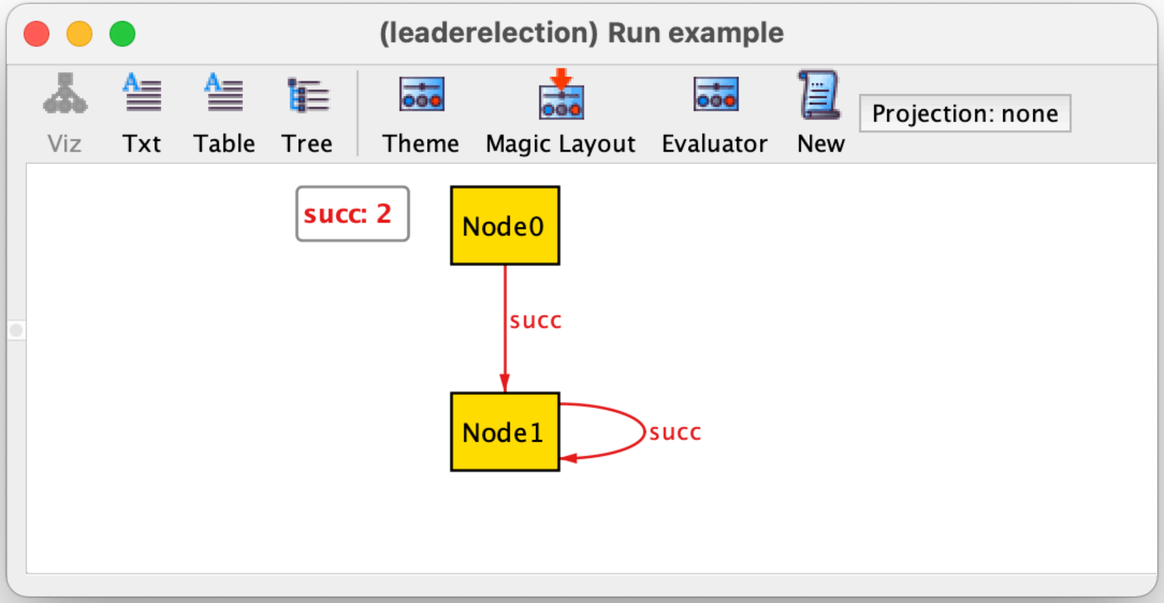
Network configuration

- Nodes are organised in a ring
- Nodes have unique comparable ids

Ring network

```
sig Node {
  succ : one Node
}
```

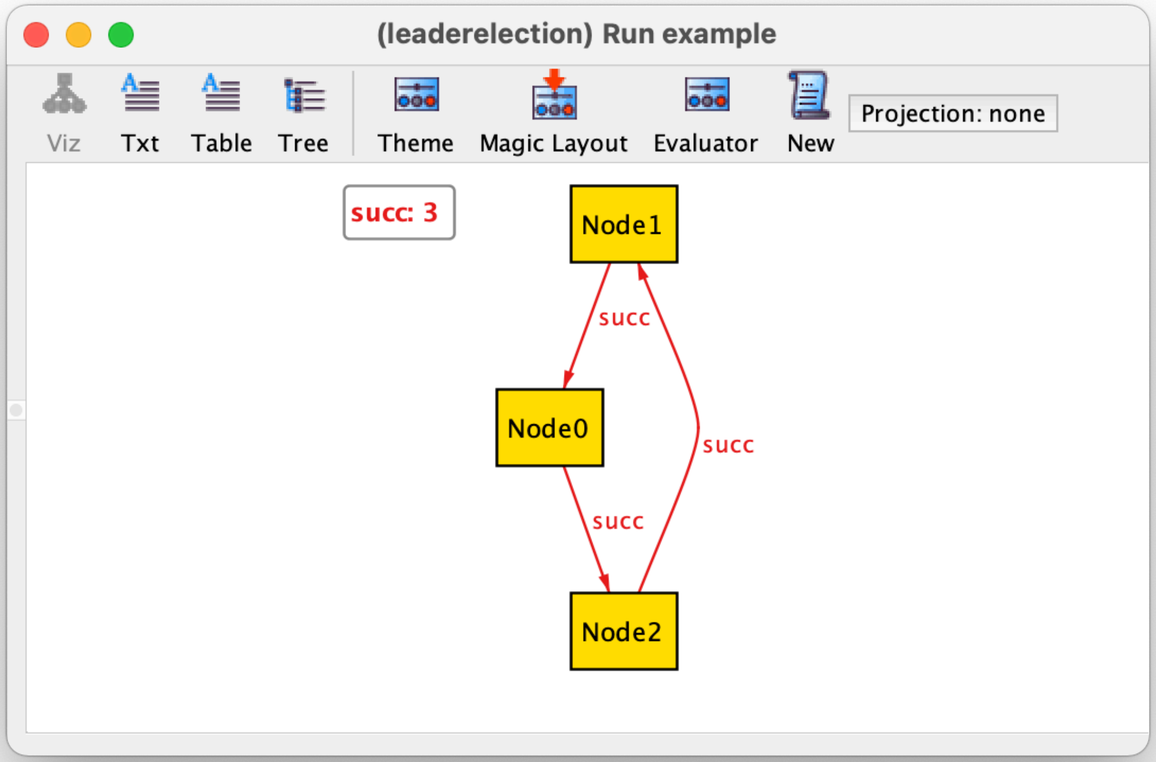
Ring network



Ring network

```
sig Node {
  succ : one Node
}

fact {
  // all nodes reachable from each node
  all n : Node | Node in n.^succ
  // at least one node
  some Node
}
```

Ring network



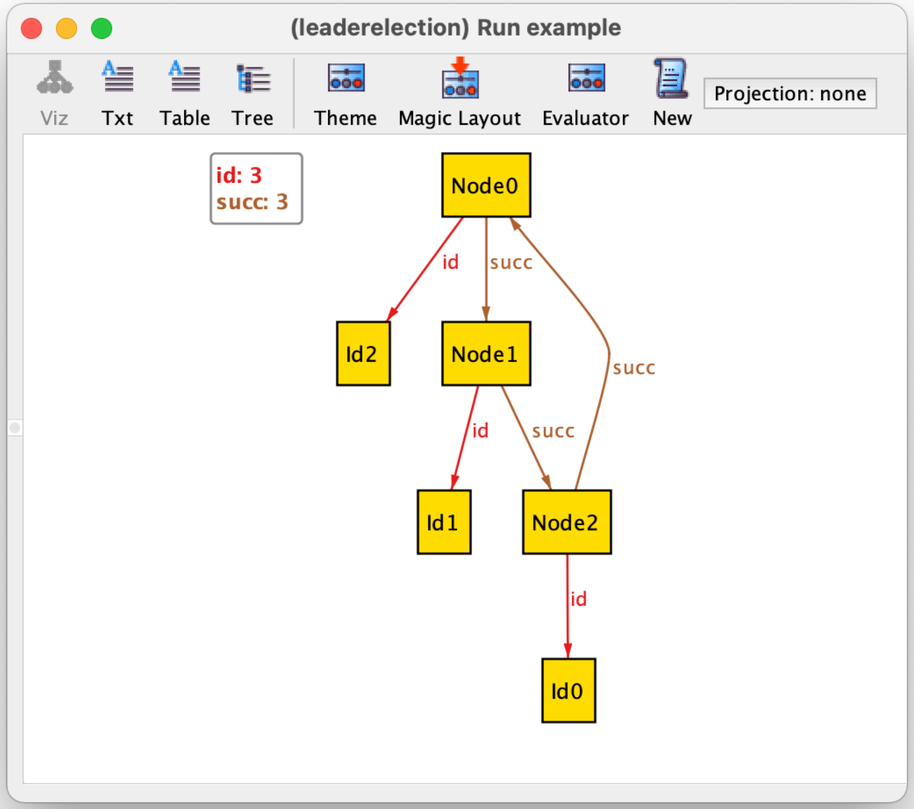
Node identifiers

- Node ids must be comparable
- No need to use numbers
- Any totally ordered set suffices
- `util/ordering` can be used to impose a total order on a signature

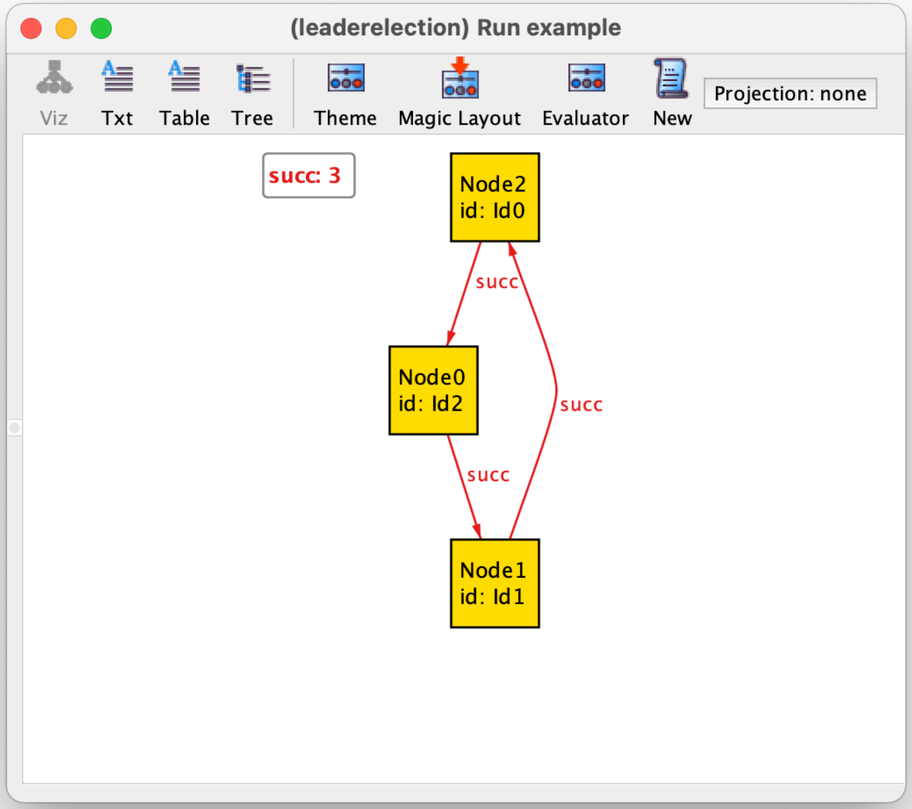
Unique identifiers

```
open util/ordering[Id]
sig Id {}

sig Node {
  succ : one Node,
  id : one Id
}

fact {
  // ids are unique
  all i : Id | lone id.i
}
```

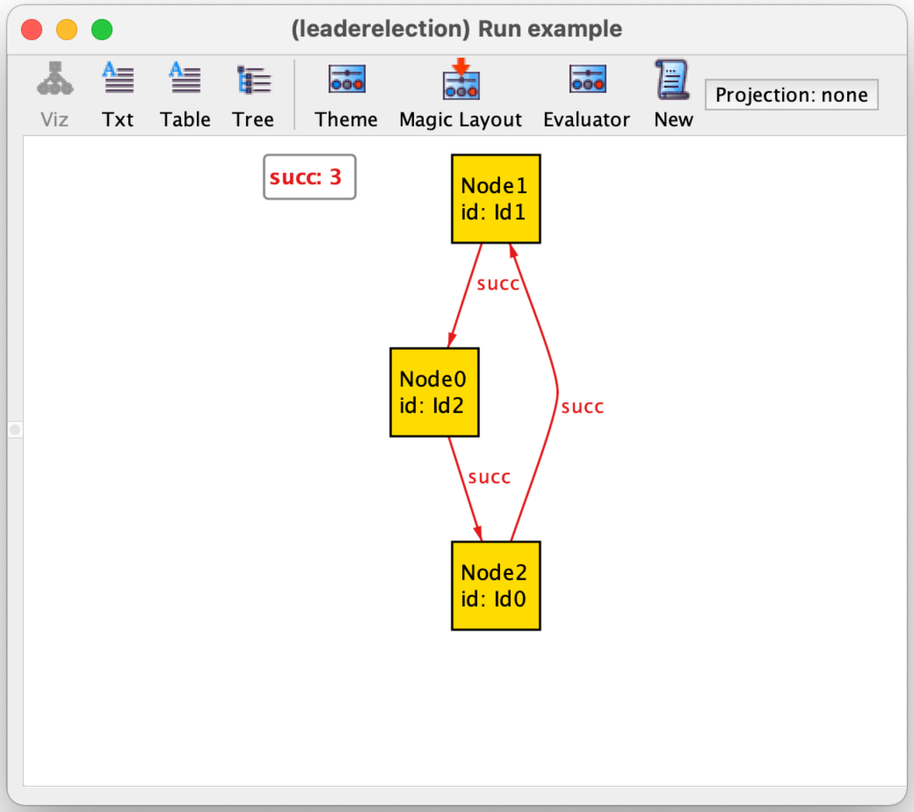
Network configuration



Network configuration



Network configuration



Mutable structures

Mutability

- In Alloy 6 mutable signatures and fields can be declared with keyword **`var`**
  - Previously only possible with the Electrum extension
  - It was possible to model behaviour in Alloy 5 by explicitly modelling the concept of state (confusing and error prone)
- Static field inside mutable signature yields a warning
- Same for static signature extending or inside mutable one

Mutable structures

```
open util/ordering[Id]
sig Id {}

sig Node {
   succ : one Node,
   id : one Id,
   var inbox : set Id,
   var outbox : set Id
}
var sig Elected in Node {}
```

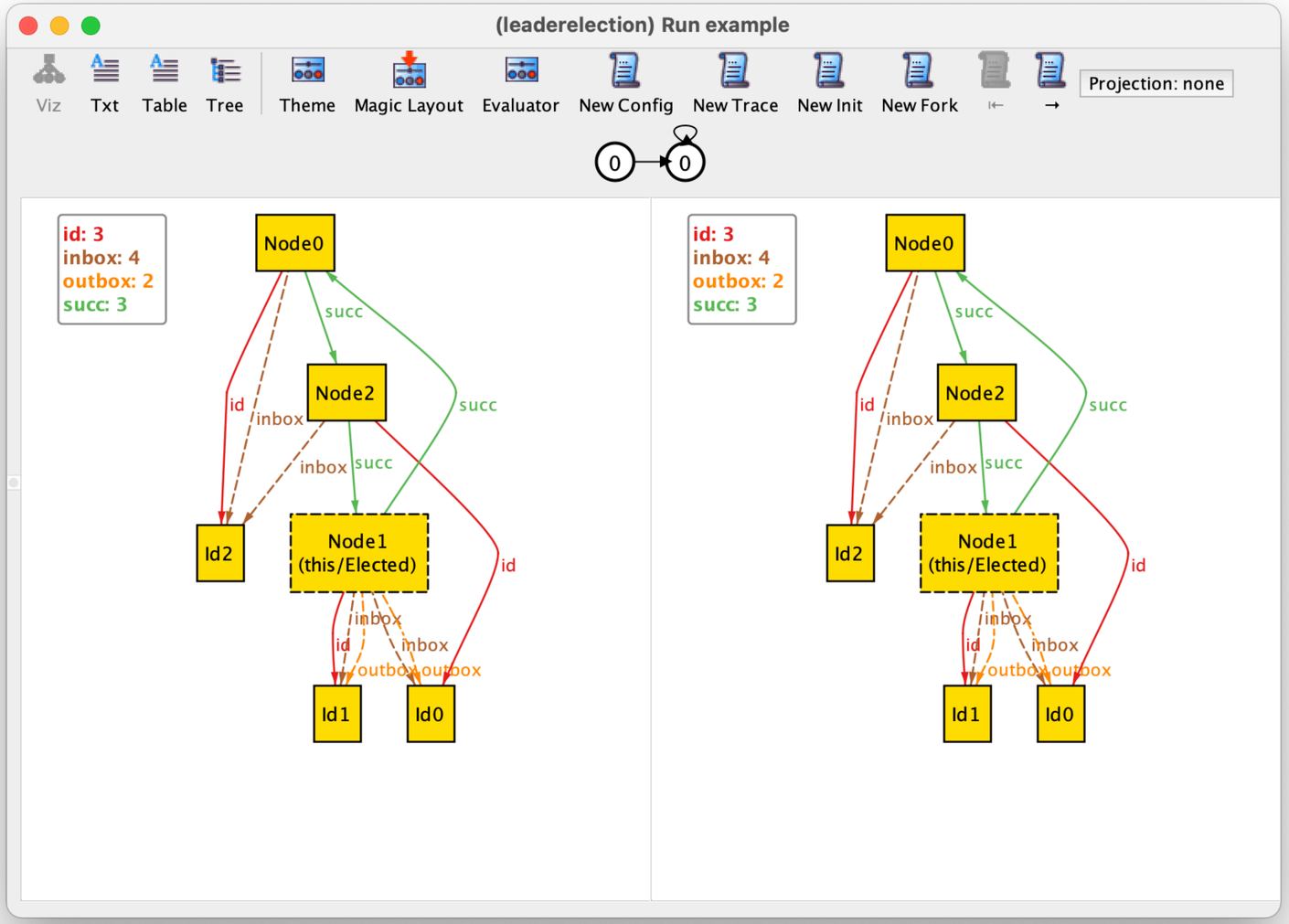
Instances

- Instances are now **infinite** sequences (*traces*) of *snapshots*
- A snapshot (*state*) is a valuation for all signatures and fields
- Analysis commands only return traces that can be represented finitely, traces that loop back at some point
- Static signatures and fields have the same value in all states
- The scope of a signature sets the maximum number of different atoms in the full trace, not a maximum per state
- **`univ`** (and **`iden`**) can be mutable, if there are mutable top-level signatures

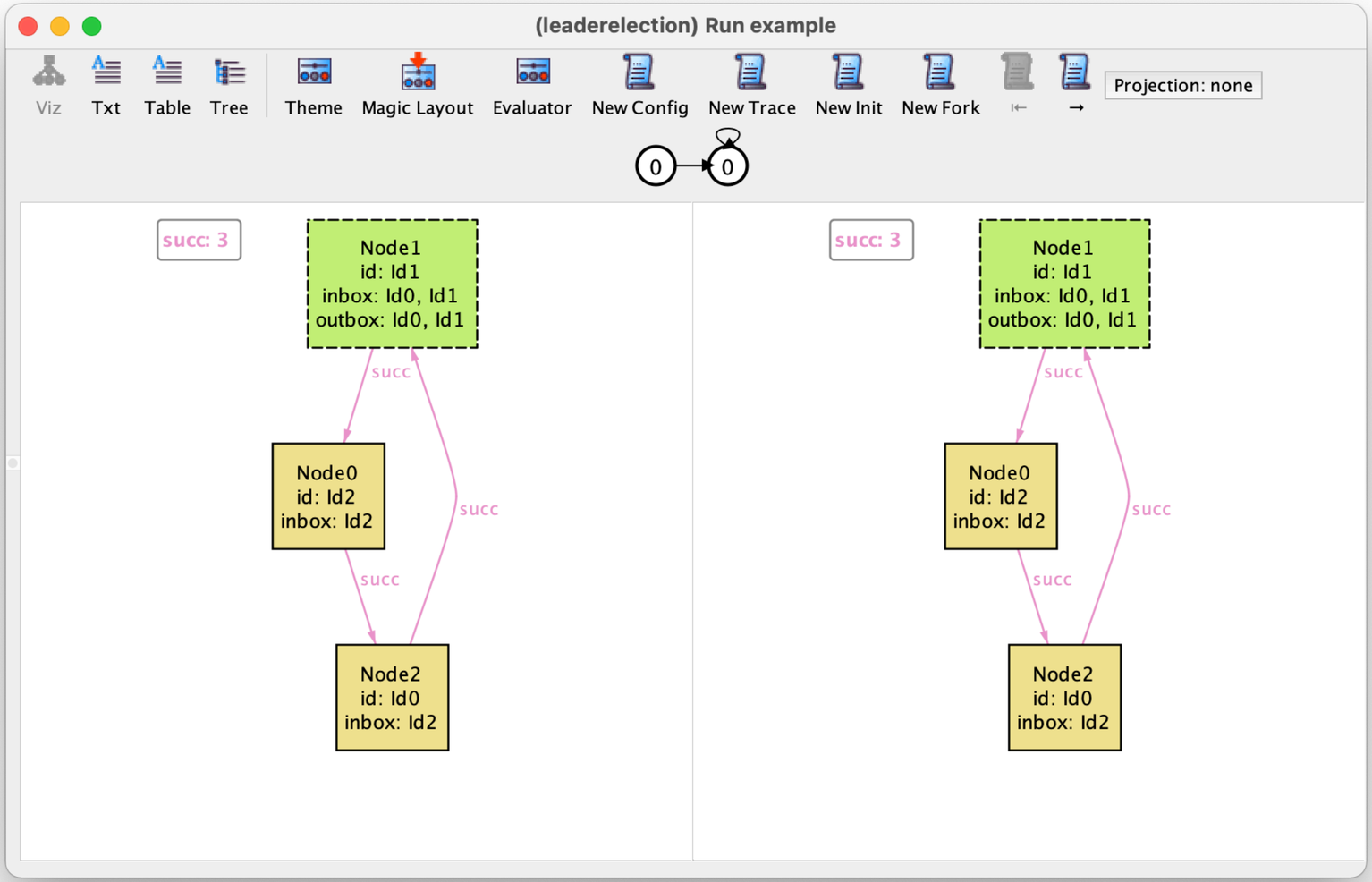
Trace visualisation

- When mutable structures are declared the visualisation changes
- It now depicts two consecutive states of the trace side-by-side
  - By default mutable structures are depict with dashed lines
- A representation of the infinite trace is shown above
  - Different states have different numbers and the loop back is explicitly depicted
  - Clicking on a state focus on that (and succeeding) state
  - It also possible to move forwards and backwards in the trace with the buttons → and ←
- We now have four different New instance buttons (more on that later…)

Trace visualisation



Trace visualisation



Property specification

Temporal logic

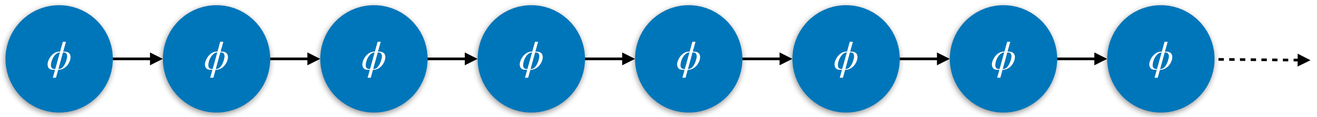
- To specify properties about traces we need a *temporal logic*
- Temporal logic adds *temporal operators,* that allow us to “quantify” the validity of a formula over time
- A formula without temporal operators is only required to hold in the initial state
- Alloy 6 has both future and past temporal operators

Temporal operators

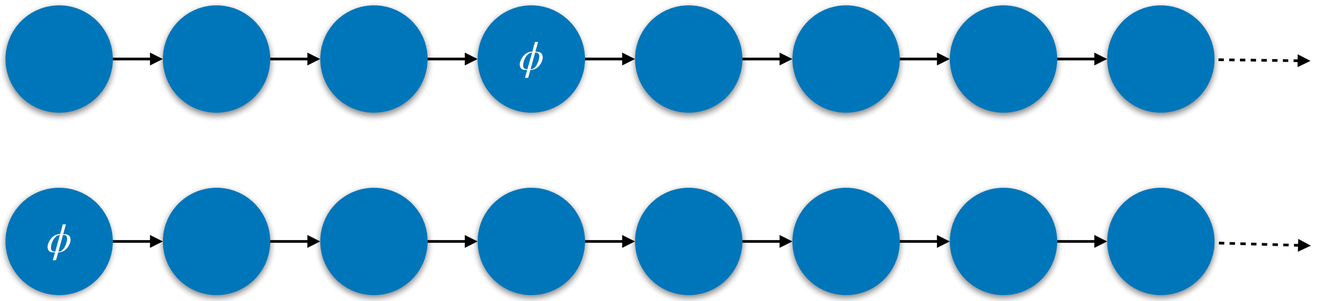
```
always ϕ            // ϕ will always be true
eventually ϕ        // ϕ will eventually be true
after ϕ             // ϕ will be true in the next state

historically ϕ      // ϕ was always true
once ϕ              // ϕ was once true
before ϕ            // ϕ was true in previous state
```

Future operators

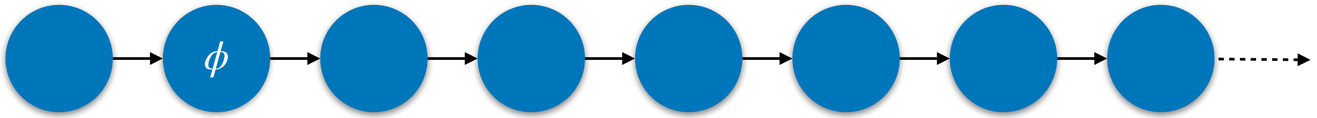
always ϕ



eventually ϕ

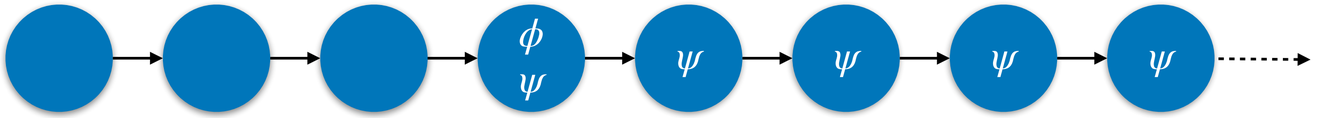


after ϕ

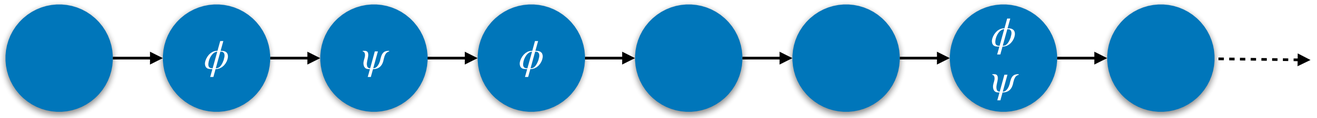


Mixing operators

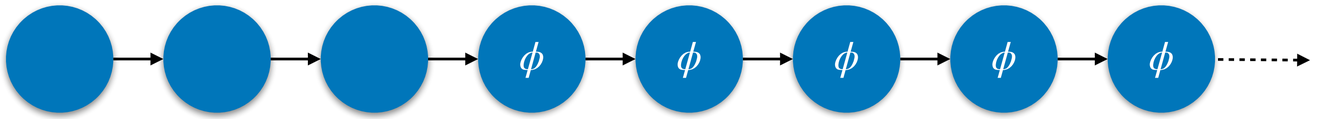
always (ϕ implies always ψ)



always (ϕ implies eventually ψ)

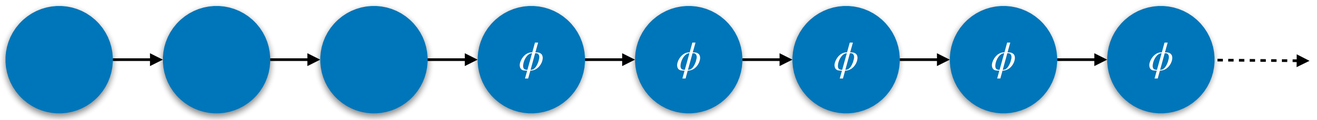


eventually (always ϕ)

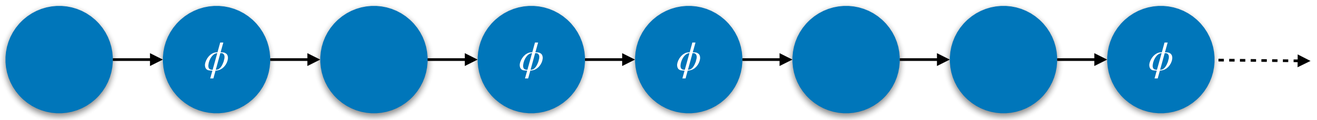


Mixing operators

eventually (always ϕ)

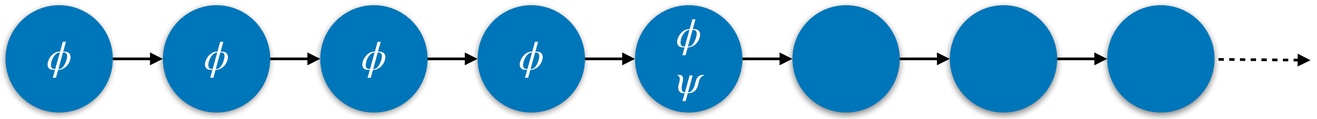


always (eventually ϕ)

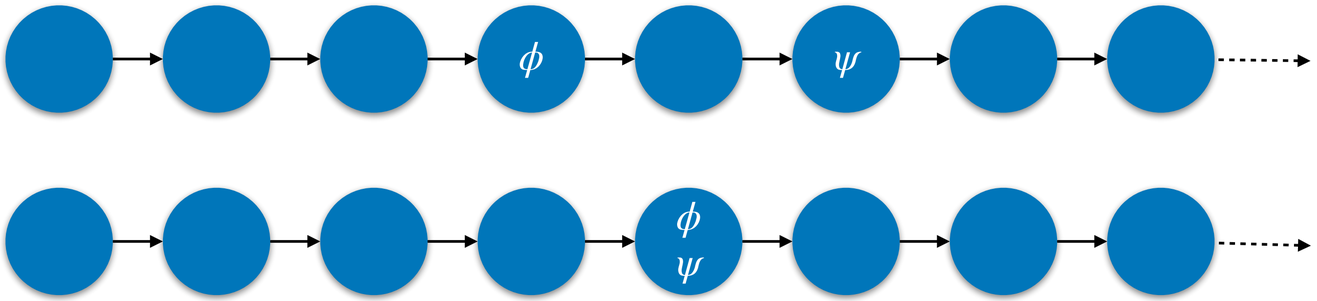


Past operators

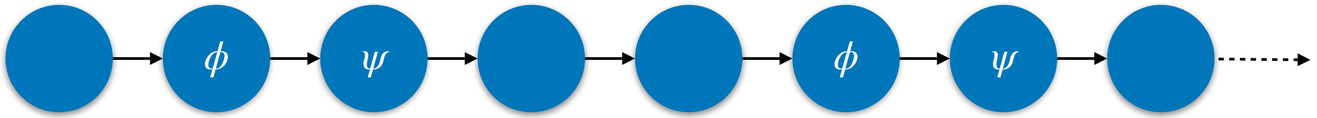
always (ψ implies historically ϕ)



always (ψ implies once ϕ)



always (ψ implies before ϕ)



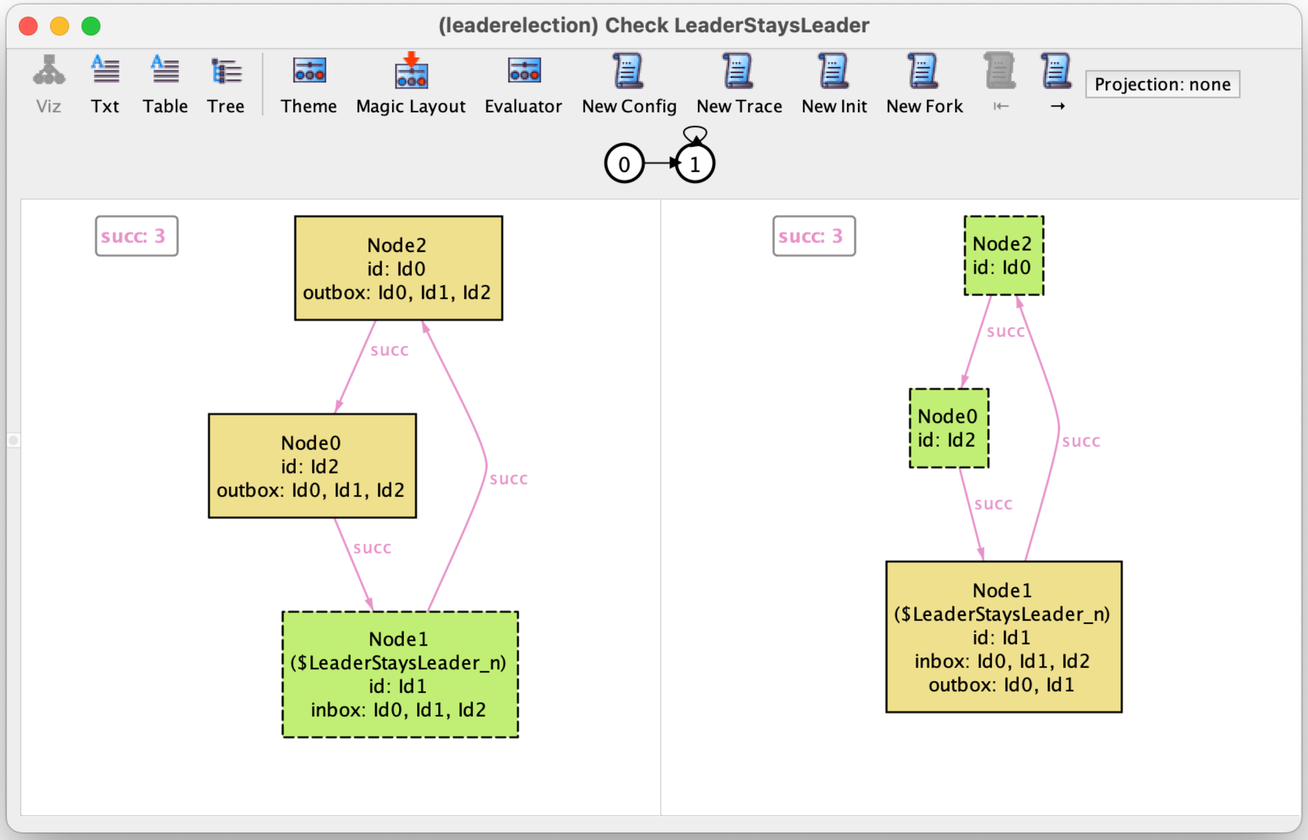
Expected properties

- One and at most one leader will be elected
  - There will never be more than one leader
  - Eventually there will be at least one leader
  - Once a leader is elected it stays elected

Expected properties

```
assert AtMostOneLeader {
  always (lone Elected)
}

assert AtLeastOneLeader {
  eventually (some Elected)
}

assert LeaderStaysLeader {
  always (all n : Elected | always n in Elected)
}
```

Counter-example



Transition systems

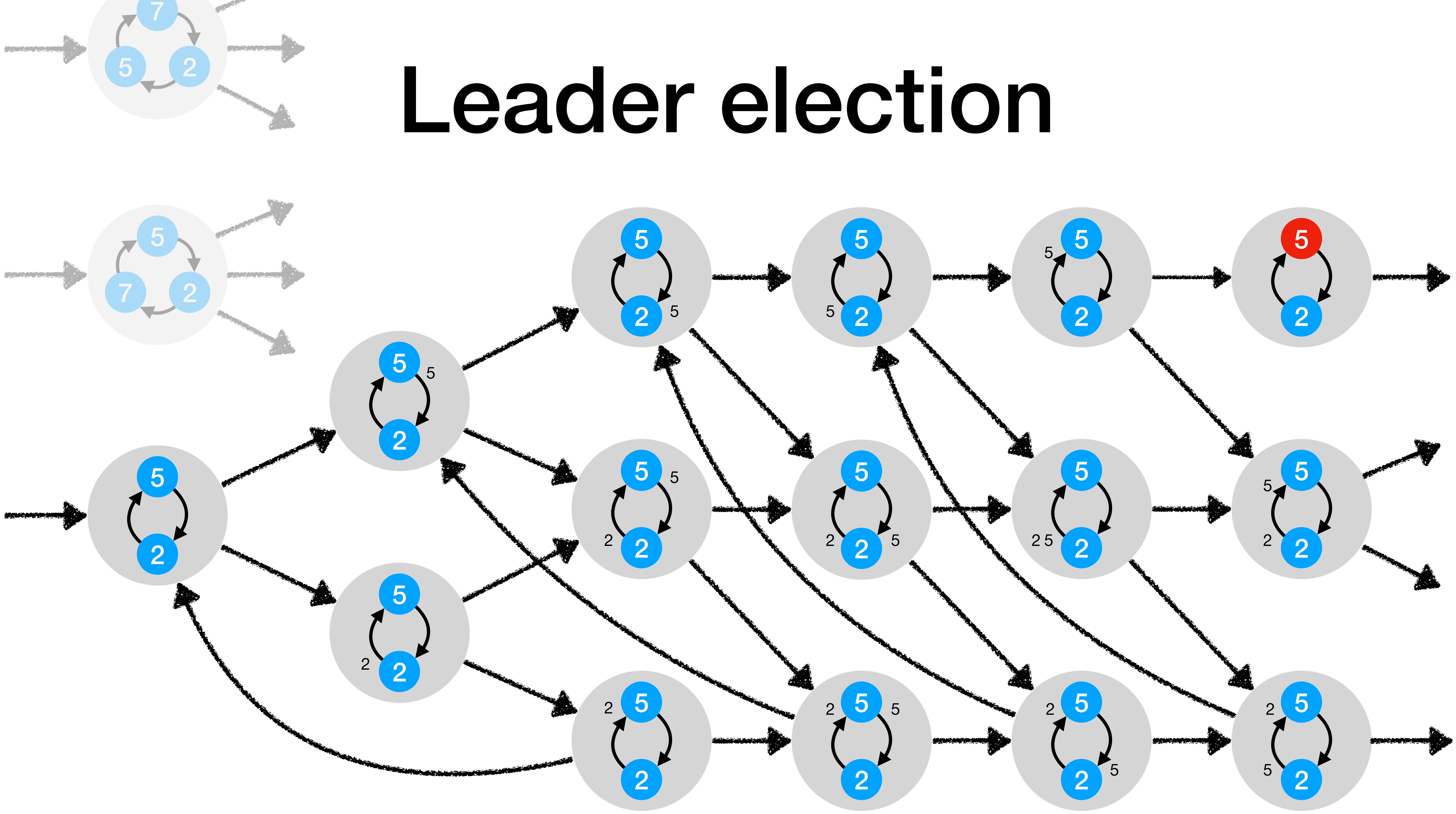
Transition systems

- The admissible behaviour can be modelled with a *transition system*
  - *Initial states* capture the starting conditions
  - *Transitions* originate from *events* performed by entities of the system or the environment
- Since traces are infinite every state must have at least one outgoing transition
  - If the system has nothing else a *stutter* transition must occur

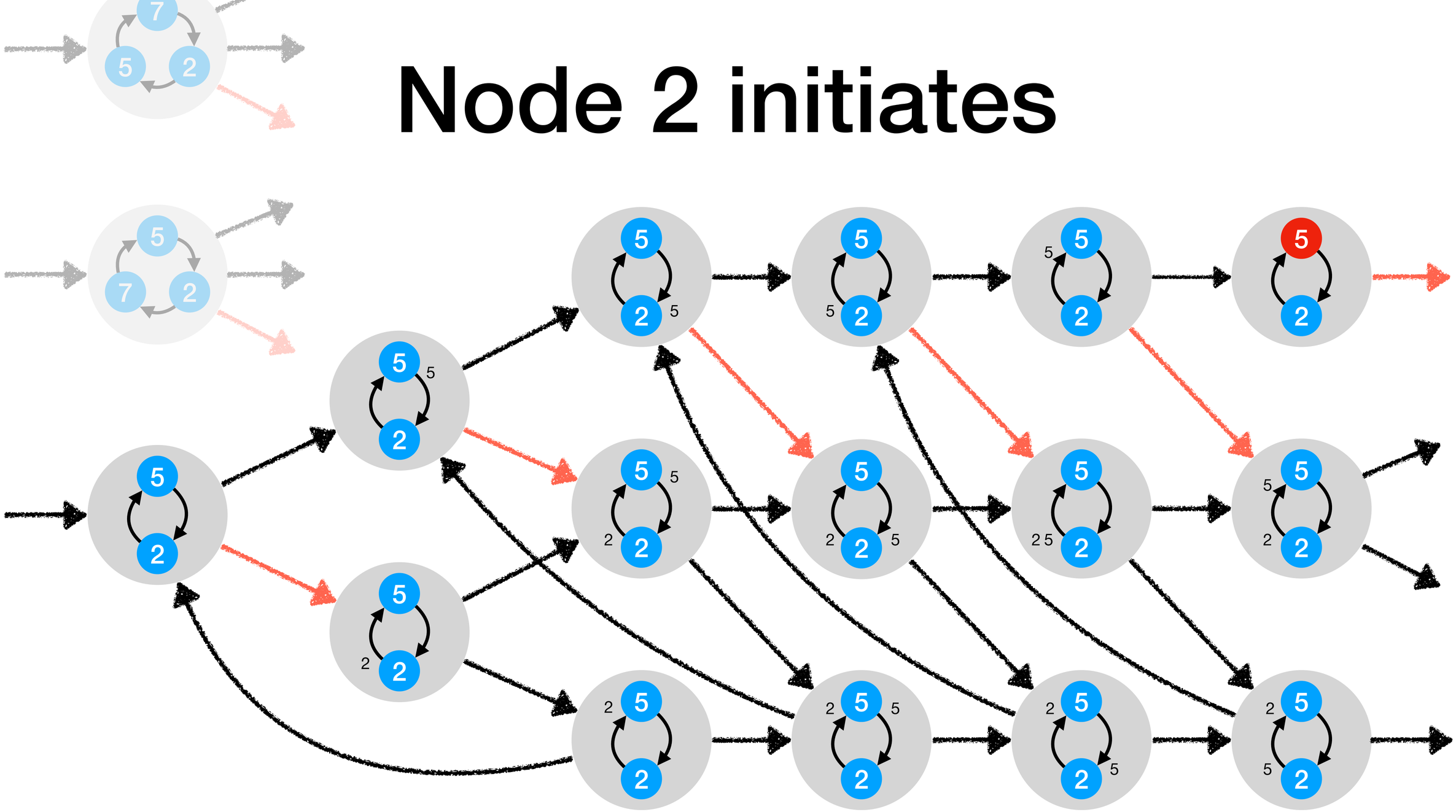
Leader election

- In the initial states
  - There are no messages in inboxes and outboxes
  - There are no elected nodes
- Besides stuttering, transitions originate from one of the following events
  - A node initiates the protocol, by putting its own identifier in the outbox
  - The network sends a message from a outbox to the inbox of the successor
  - A node reads and processes a message in its inbox

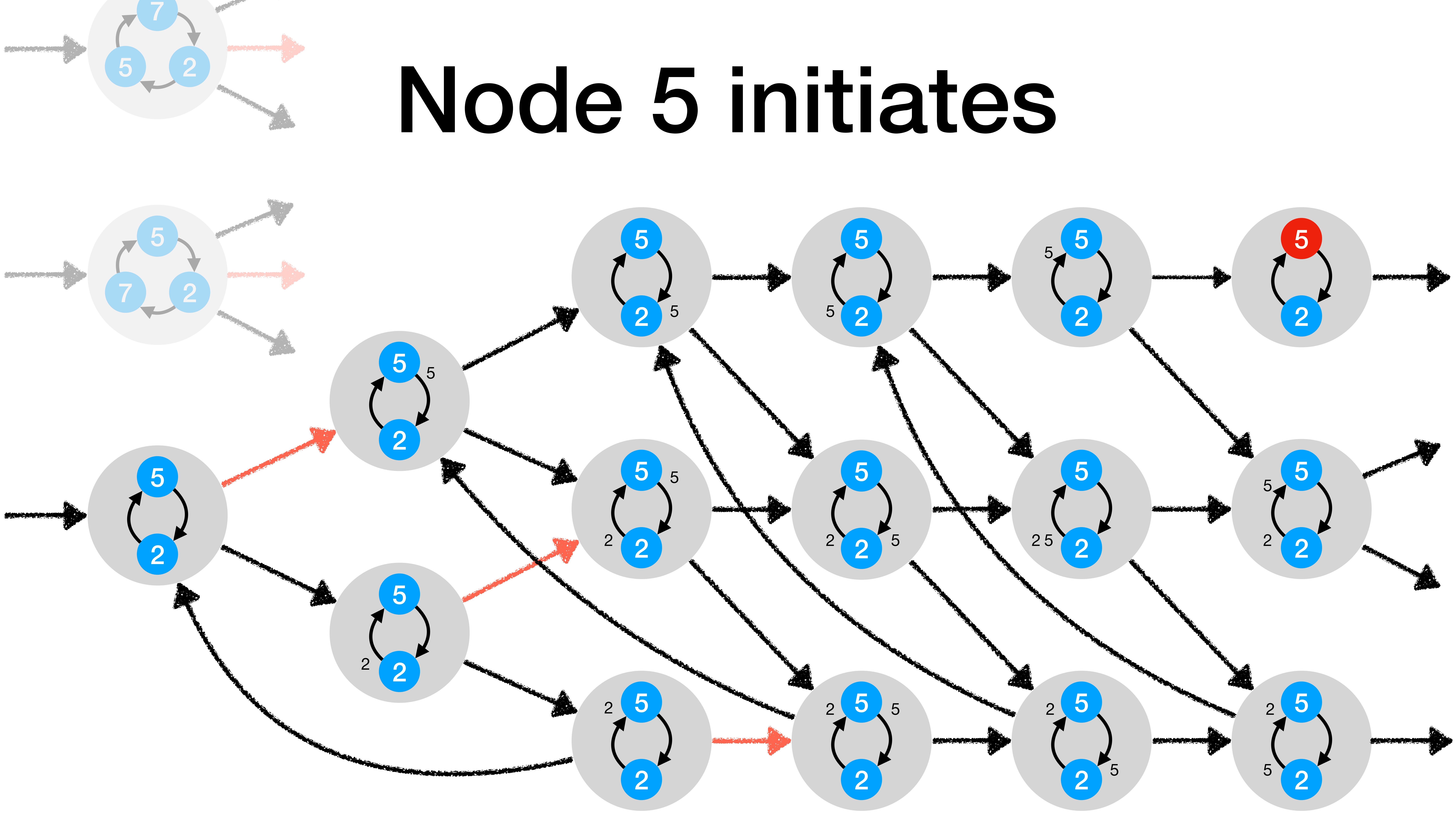
Leader election



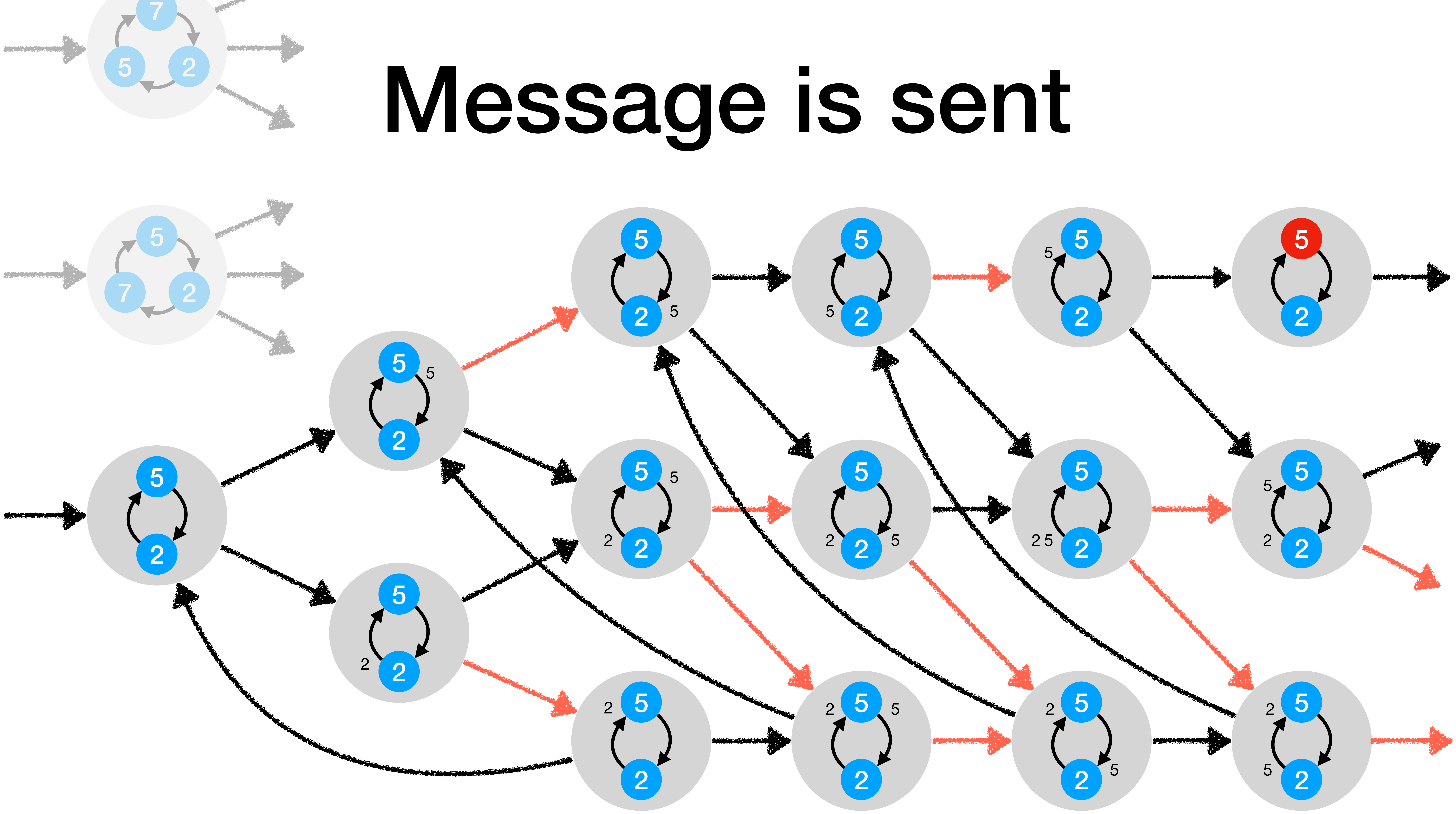
Node 2 initiates



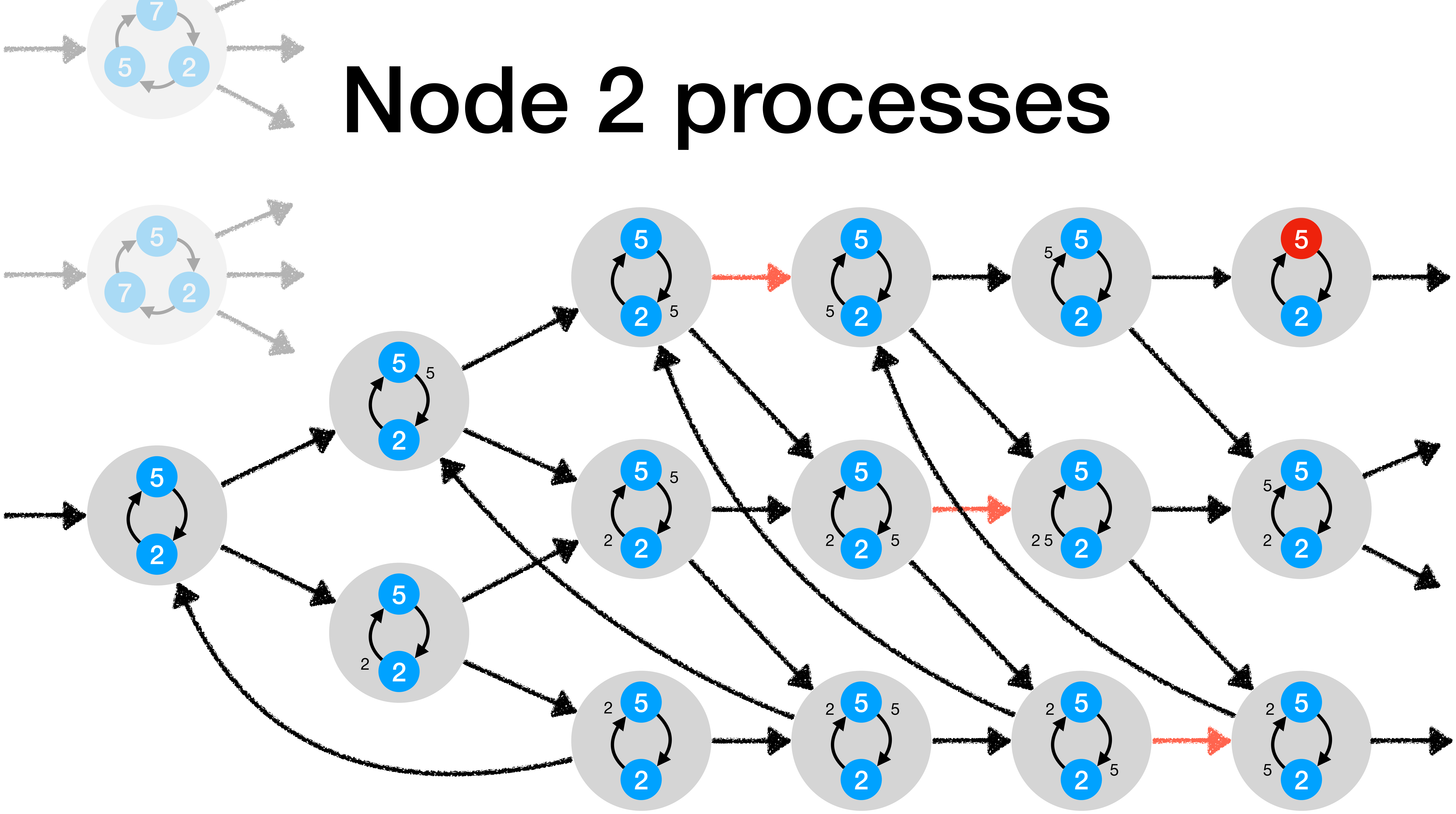
Node 5 initiates



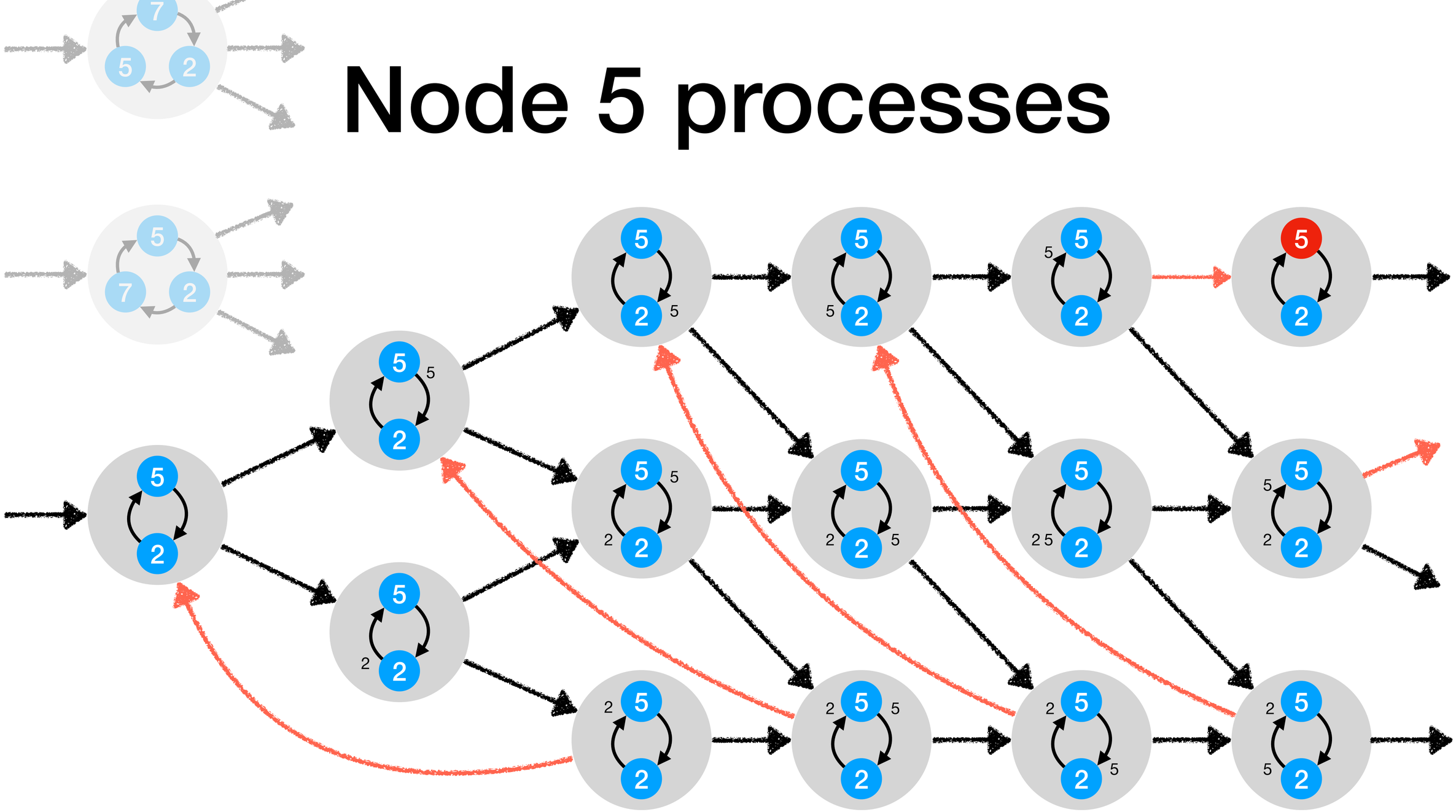
Message is sent



Node 2 processes



Node 5 processes



Declarative modelling

- A transition system can be modelled by a temporal logic formula that specifies what are the valid traces
  - Initial states are specified by formulas without temporal operators
  - Events are specified by formulas that relate consecutive states
  - Besides **`after`**, operator `'` can be used to evaluate expressions in the next state

```
pred init { … }
pred event1 { … and after … }
pred event2 { … and after … }
…
fact { init and always (event1 or event2 or …) }
```

Leader election

```
fact init {
  no inbox
  no outbox
  no Elected
}

fact events {
 always (
    some n : Node | initiate[n] or
    some n : Node, i : Id | send[n,i] or
    some n : Node, i : Id | process[n,i]
  )
}
```

Anatomy of an event

- The specification of an event is a conjunction of three kinds of formulas
  - *Guards*, that specify when can an event occur
  - *Effects*, that specify what changes when an event occurs
  - *Frame conditions*, that specify what does not change when an event occurs
- Guards usually have no temporal operators
  - But can use past temporal operators to recall something about the past
- Effects and frame conditions use only **`after`** and `‘`

Initiate

```
pred initiate [n : Node] {
  // guard
  historically n.id not in n.outbox

  // effect
  n.outbox' = n.outbox + n.id

  // frame conditions
  all m : Node - n | m.outbox' = m.outbox
  inbox'   = inbox
  Elected' = Elected
}
```

Initiate

```
pred initiate [n : Node] {
  // guard
  historically n.id not in n.outbox

  // effect
  outbox' = outbox + n->n.id

  // frame conditions
  inbox'   = inbox
  Elected' = Elected
}
```

Send

```
pred send [n : Node, i : Id] {
  // guard
  i in n.outbox

  // effects
  outbox' = outbox - n->i
  inbox'  = inbox  + n.succ->i

  // frame conditions
  Elected' = Elected
}
```

Process

```
pred process [n : Node, i : Id] {
  // guard
  i in n.inbox

  // effects
  inbox' = inbox - n->i
  gt[i,n.id] implies outbox'  = outbox  + n->i
  i = n.id   implies Elected' = Elected + n

  // frame conditions
  lte[i,n.id] implies outbox'  = outbox
  i != n.id   implies Elected' = Elected
}
```

Process

```
pred process [n : Node, i : Id] {
  // guard
  i in n.inbox

  // effects
  inbox'    = inbox    - n->i
  outbox'   = outbox   + n->(i & n.id.nexts)
  Elected' = Elected + (n & id.i)
}
```

Validation

Validation

- **`run`** commands should be used to validate the model
  - Optionally a formula can be given to look for specific scenarios
- It is also possible to perform “simulation” with the New instance buttons
  - *New config*, returns a trace with a different configuration (a different value to the immutable structures)
  - *New trace*, returns any different trace with the same configuration
  - *New init*, returns a trace with the same config, but a different initial state
  - *New fork*, returns a trace with the same prefix, but a different next state

Consistency check

```
run example {}
```

```
Executing "Run example"
   Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   1..10 steps. 88603 vars. 1895 primary vars. 220976 clauses. 2776ms.
   No instance found. Predicate may be inconsistent. 1386ms.
```

Inconsistency

- The model does not allow any (infinite) trace
- Once the protocol completes no event is possible
- At least a stuttering event should be possible at that point

A possible fix

```
pred stutter {
  // guards
  no inbox and no outbox
  all n : Node | once initiate[n]

  // frame conditions
  outbox'  = outbox
  inbox'   = inbox
  Elected' = Elected
}
```

A possible fix

```
fact events {
 always (
    stutter or
    some n : Node | initiate[n] or
    some n : Node, i : Id | send[n,i] or
    some n : Node, i : Id | process[n,i]
  )
}
```

Stuttering

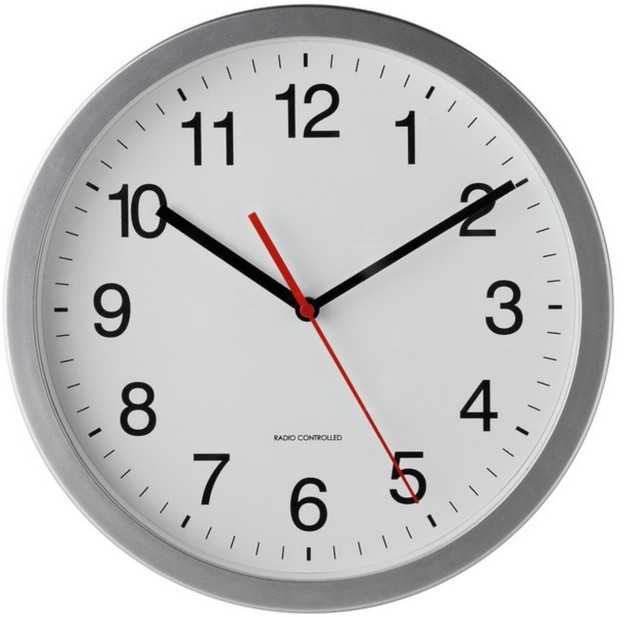
A clock specification

```
pred clock_spec {
   h = 0 and m = 0
   always {
      m'=(m+1)%60 and
      m=59  implies h'=(h+1)%12 and
      m!=59 implies h'=h
  }
}
```

Ceci n'est pas une montre?!

```
check clock_spec
```

```
Executing "Check clock_spec"
   Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   1..2 steps. 55 vars. 12 primary vars. 59 clauses. 3ms.
   Counterexample found. Assertion is invalid. 3ms.
```



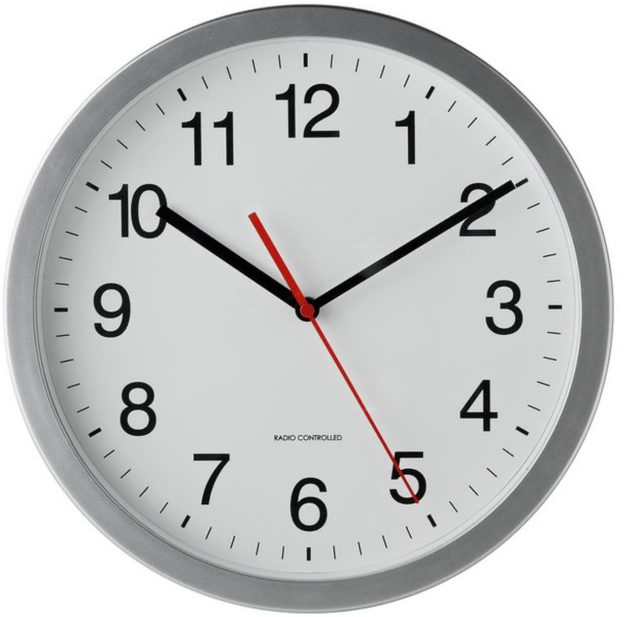
A clock specification

```
pred clock_spec {
   h = 0 and m = 0
   always {
      m'=(m+1)%60 and
      m=59  implies h'=(h+1)%12 and
      m!=59 implies h'=h
      or
      m'=m and h'=h
  }
}
```

A clock

```
check clock_spec
```

```
Executing "Check clock_spec"
   Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   1..10 steps. 151901 vars. 1875 primary vars. 413006 clauses. 1042ms.
   No counterexample found. Assertion may be valid. 298ms.
```



Stuttering

- It is good practice to allow the system to stutter in every state
- Stuttering can represent events by the environment not (yet) modelled
- Stuttering allow us to check refinements

Stuttering

```
pred stutter {
  // frame conditions
  outbox'   = outbox
  inbox'    = inbox
  Elected' = Elected
}
```

Back to validation

Simulation

Model checking

Model checking

- *Model checking* is the process of automatically verifying if a temporal logic specification holds in a finite transition system model of a system
- If the specification is false a counter-example is returned
  - A finite transition system may have infinite non-looping traces
  - But every false specification can be falsified with a looping trace
- *Bounded model checking* explores only a finite number of steps before looping back
  - The default verification method in Alloy 6 is bounded model checking via SAT
  - The default number of steps is 10 but can be changed with keyword **`steps`** in scopes
  - Alloy 6 also supports normal model checking if model checkers `NuSMV` or `nuXmv` are installed

Expected properties

```
assert AtMostOneLeader {
  always (lone Elected)
}

assert AtLeastOneLeader {
  eventually (some Elected)
}

assert LeaderStaysLeader {
  always (all n : Elected | always n in Elected)
}
```

Safety vs Liveness

- `AtMostOneLeader` and `LeaderStaysLeader` are *safety* properties
  - They prevent some undesired behaviours from happening
  - Easier to model-check, since it suffices to search for a finite sequence of steps that lead to a bad state
  - It is irrelevant what happens afterwards, and any continuation leads to a counter-example
- `AtLeastOneLeader` is a *liveness* property
  - If forces some desired behaviours to happen
  - Harder to model-check, since it is necessary to search for a complete infinite trace where the desired behaviour never happened

At most one leader

```
assert AtMostOneLeader {
   always (lone Elected)
}
check AtMostOneLeader
```

```
Executing "Check AtMostOneLeader"
   Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   1..10 steps. 79442 vars. 1675 primary vars. 206995 clauses. 337ms.
   No counterexample found. Assertion may be valid. 99ms.
```

At most one leader

```
assert AtMostOneLeader {
   always (lone Elected)
}
check AtMostOneLeader for 3 but 20 steps
```

```
Executing "Check AtMostOneLeader for 3 but 20 steps"
   Solver=sat4j Steps=1..20 Bitwidth=4 MaxSeq=3 SkolemDepth=1 Symmetry=20 Mode=batch
   1..20 steps. 494769 vars. 6050 primary vars. 1275400 clauses. 4146ms.
   No counterexample found. Assertion may be valid. 450ms.
```

At most one leader

```
assert AtMostOneLeader {
   always (lone Elected)
}
check AtMostOneLeader for 3 but 1.. steps
```

```
Option Solver changed to Electrod/nuXmv
Executing "Check AtMostOneLeader for 3 but 1.. steps"
   Solver=nuXmv Steps=1..2147483647 Bitwidth=4 MaxSeq=3 SkolemDepth=1 Symmetry=20 Mode=batch
   No translation information available. 44ms.
   No counterexample found. Assertion may be valid. 2260ms.
```

Leader stays leader

```
assert LeaderStaysLeader {
  always (all n : Elected | always n in Elected)
}
check LeaderStaysLeader for 3 but 1.. steps
```

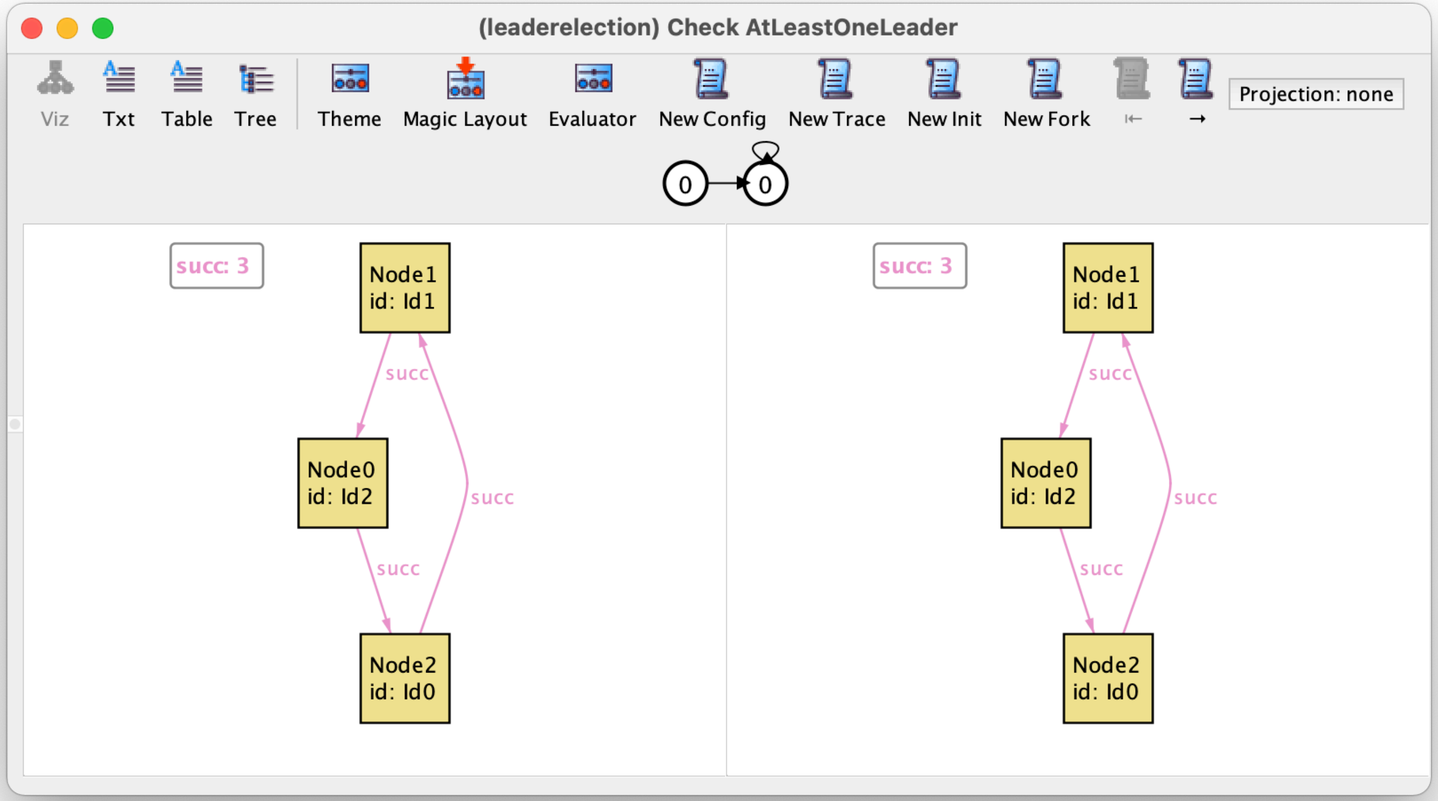
```
Executing "Check LeaderStaysLeader for 3 but 1.. steps"
   Solver=nuXmv Steps=1..2147483647 Bitwidth=4 MaxSeq=3 SkolemDepth=1 Symmetry=20 Mode=batch
   No translation information available. 12ms.
   No counterexample found. Assertion may be valid. 1749ms.
```

At least one leader

```
assert AtLeastOneLeader {
  eventually (some Elected)
}
check AtLeastOneLeader
```

```
Executing "Check AtLeastOneLeader"
   Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   1..1 steps. 1178 vars. 44 primary vars. 2526 clauses. 9ms.
   Counterexample found. Assertion is invalid. 7ms.
```

At least one leader



Fairness

- *Fairness* assumptions are necessary for verifying most liveness properties
- The goal is to exclude counter-examples where an event becomes “continuously” enabled but never occurs
  - In *weak fairness* “continuously” means permanently
  - In *strong fairness* “continuously” means infinitely often

Fairness

```
// Weak fairness
always ((always enabled) implies (eventually happens))
(eventually always enabled) implies (always eventually happens)

// Strong fairness
(always eventually enabled) implies (always eventually happens)
```

Fair leader election

```
pred fairness {
  all n : Node, i : Id {
    eventually always (historically n.id not in n.outbox)
    implies
    always eventually initiate[n]

    eventually always (i in n.inbox)
    implies
    always eventually process[n,i]

    eventually always (i in n.outbox)
    implies
    always eventually send[n,i]
  }
}
```

At least one leader

```
assert AtLeastOneLeader {
  fairness implies eventually (some Elected)
}
check AtLeastOneLeader for 3 but 1.. steps
```

```
Executing "Check AtLeastOneLeader for 3 but 1.. steps"
   Solver=nuXmv Steps=1..2147483647 Bitwidth=4 MaxSeq=3 SkolemDepth=1 Symmetry=20 Mode=batch
   No translation information available. 43ms.
   No counterexample found. Assertion may be valid. 11682ms.
```

Abstraction

At least one leader

```
assert AtLeastOneLeader {
   fairness implies eventually (some Elected)
}
check AtLeastOneLeader for 4 but 1.. steps
```

```
Executing "Check AtLeastOneLeader for 4 but 1.. steps"
   Solver=nuXmv Steps=1..2147483647 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   No translation information available. 15ms.
   No counterexample found. Assertion may be valid 95382ms.
```

Abstraction

- Why?
  - Improve efficiency
  - Improve generality
  - Improve understandability
- How?
  - Merge events (if interleaving is not likely a problem)
  - Remove structures
  - Make the specification more declarative
  - Make the specification more liberal

Merging send

```
open util/ordering[Id]
sig Id {}

sig Node {
  succ : one Node,
  id : one Id,
  var inbox : set Id,
  var outbox : set Id
}

var sig Elected in Node {}
```

Merging send

```
fact init {
  no inbox
  no outbox
  no Elected
}

fact events {
 always (
    some n : Node | initiate[n] or
    some n : Node, i : Id | send[n,i] or
    some n : Node, i : Id | process[n,i]
  )
}
```

Merging send

```
pred initiate [n : Node] {
  // guard
  historically n.id not in n.succ.inbox

  // effect
  inbox' = inbox + n.succ->n.id

  // frame conditions
  Elected' = Elected
}
```

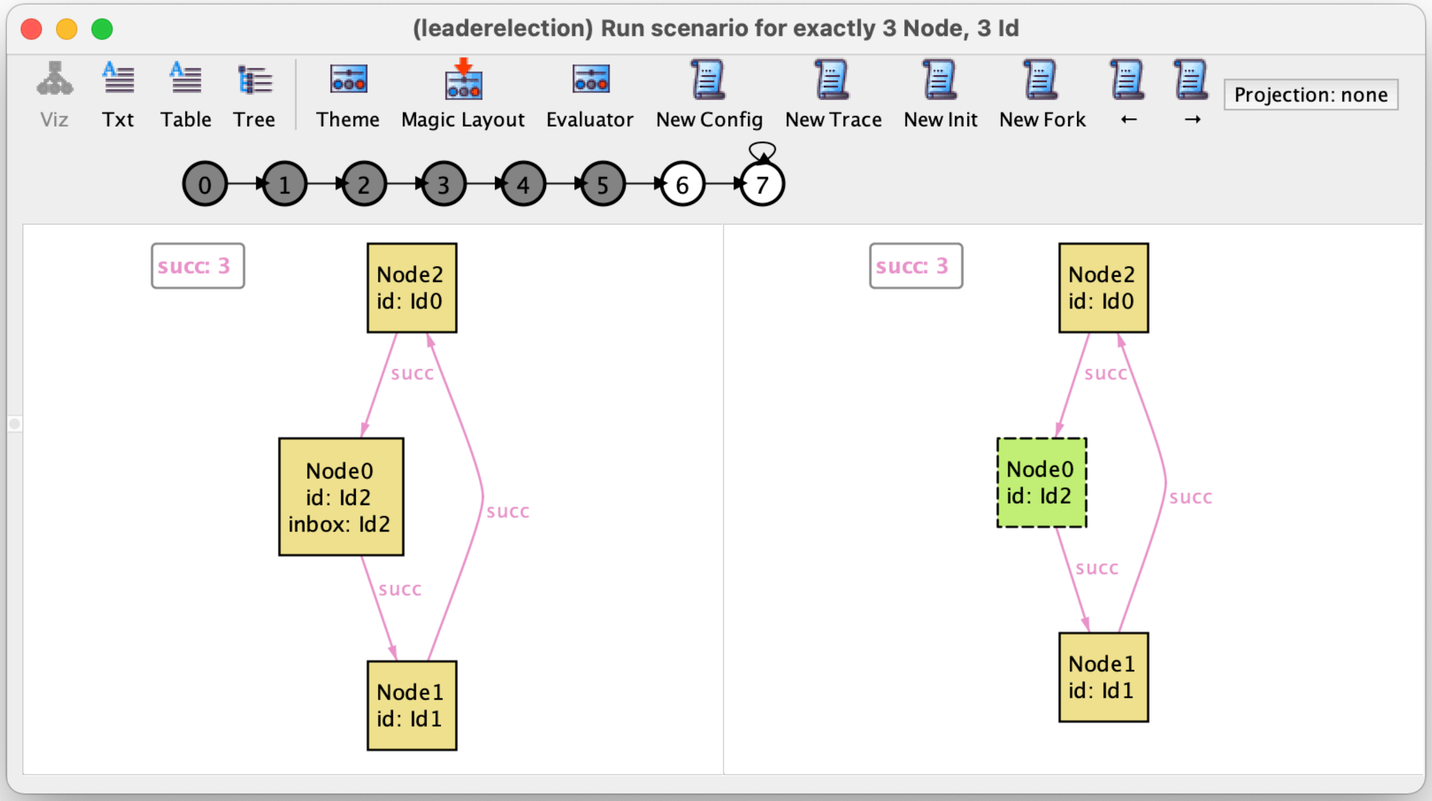
Merging send

```
pred process [n : Node, i : Id] {
  // guard
  i in n.inbox

  // effects
  inbox'   = inbox - n->i + n.succ->(i & n.id.nexts)
  Elected' = Elected + (n & id.i)
}
```

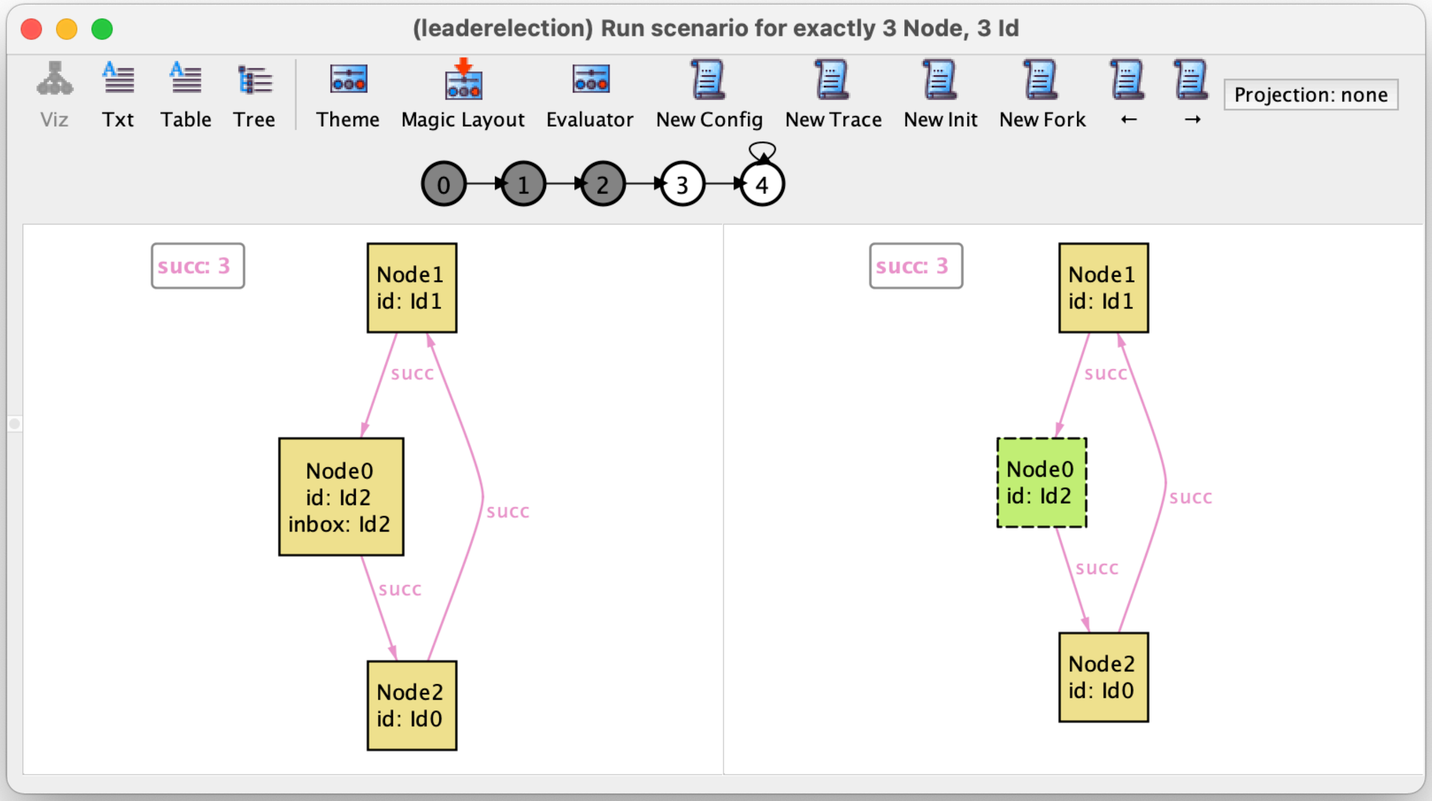
Merging send

```
pred stutter {
  // frame conditions
  inbox'   = inbox
  Elected' = Elected
}
```

Scenario exploration



Scenario exploration



Removing Id

```
open util/ordering[Node]
sig Id {}

sig Node {
   succ : one Node,
   id : one Id,
   var inbox : set Node
}

var sig Elected in Node {}
```

Removing Id

```
pred initiate [n : Node] {
  // guard
  historically n not in n.succ.inbox

  // effect
  inbox' = inbox + n.succ->n

  // frame conditions
  Elected' = Elected
}
```

Removing Id

```
pred process [n : Node, i : Node] {
  // guard
  i in n.inbox

  // effects
  inbox'   = inbox - n->i + n.succ->(i & n.nexts)
  Elected' = Elected + (n & i)
}
```

Removing Elected

```
open util/ordering[Node]

sig Node {
  succ : one Node,
  var inbox : set Node,
}
~~var sig Elected in Node {}~~

fun Elected : set Node {
  { n : Node | n not in n.inbox and once (n in n.inbox) }
}
```

Removing Elected

```
fact init {
   no inbox
   no Elected
}

fact events {
 always (
     some n : Node | initiate[n] or
     some n : Node, i : Node | process[n,i]
   )
}
```

Removing Elected

```
pred initiate [n : Node] {
  // guard
  historically n not in n.succ.inbox
  // effect
  inbox' = inbox + n.succ->n
}
pred process [n : Node, i : Node] {
  // guard
  i in n.inbox
  // effects
  inbox' = inbox - n->i + n.succ->(i & n.nexts)
}
pred stutter {
  // frame conditions
  inbox' = inbox
}
```

At least one leader

```
assert AtLeastOneLeader {
   fairness implies eventually (some Elected)
}
check AtLeastOneLeader for 4 but 1.. steps
```

```
Executing "Check AtLeastOneLeader for 4 but 1.. steps"
   Solver=nuXmv Steps=1..2147483647 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   No translation information available. 15ms.
   No counterexample found. Assertion may be valid 95382ms.
```

At least one leader

```
assert AtLeastOneLeader {
   fairness implies eventually (some Elected)
}
check AtLeastOneLeader for 4 but 1.. steps
```

```
Executing "Check AtLeastOneLeader for 4 but 1.. steps"
   Solver=nuXmv Steps=1..2147483647 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   No translation information available. 8ms.
   No counterexample found. Assertion may be valid. 10942ms.
```

Liberating initiate

```
pred initiate [n : Node] {
  // guard
  historically n not in n.succ.inbox

  // effect
  inbox' = inbox + n.succ->n
}

fun Elected : set Node {
  { n : Node | once (n not in n.inbox and once (n in n.inbox)) }
}
```

“The core of software development, therefore, is the design of abstractions. An abstraction is [...] an idea reduced to its essential form.”

–Daniel Jackson

“31. Simplicity does not precede complexity, but follows it.”

–Alan Perlis