A *formal* framework for the design of *software components* with the B method

David Déharbe - UFRN, Natal, Brazil
j.w.w. Stephan Merz - LORIA & U. Lorraine, Nancy, France

http://DavidDeharbe.github.io

Aspiration

- Software engineering
  - Design for quality
  - Traceability
  - Improve productivity

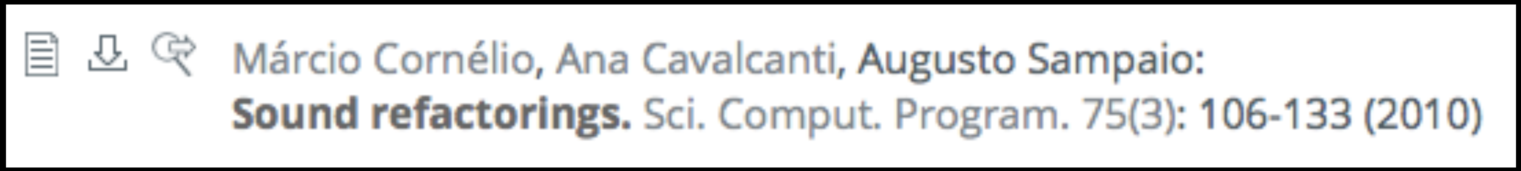
components for safety-critical systems

MDD

Automate some design tasks

Approach

- Components for safety-critical systems
  - Formal methods
- MDD
  - Formal specification
  - Successive refinements
- Improve productivity
  - Refactoring rules
  - Refinement rules



Márcio Cornélio, Ana Cavalcanti, Augusto Sampaio:
Sound refactorings. Sci. Comput. Program. 75(3): 106-133 (2010)

Paulo Borba, Augusto Sampaio, Márcio Cornélio:
A Refinement Algebra for Object-Oriented Programming. ECOOP 2003: 457-482

Goal

- Formal methods + MDD :
  - ⭑ the B method
- Improve productivity:
  - ⭑ refinement rules, refactoring rules
- Machine-controlled formalisation of the semantics
- Formal model of component behaviour

Other applications

- Verify
  - code generators
  - abstraction techniques

Overview

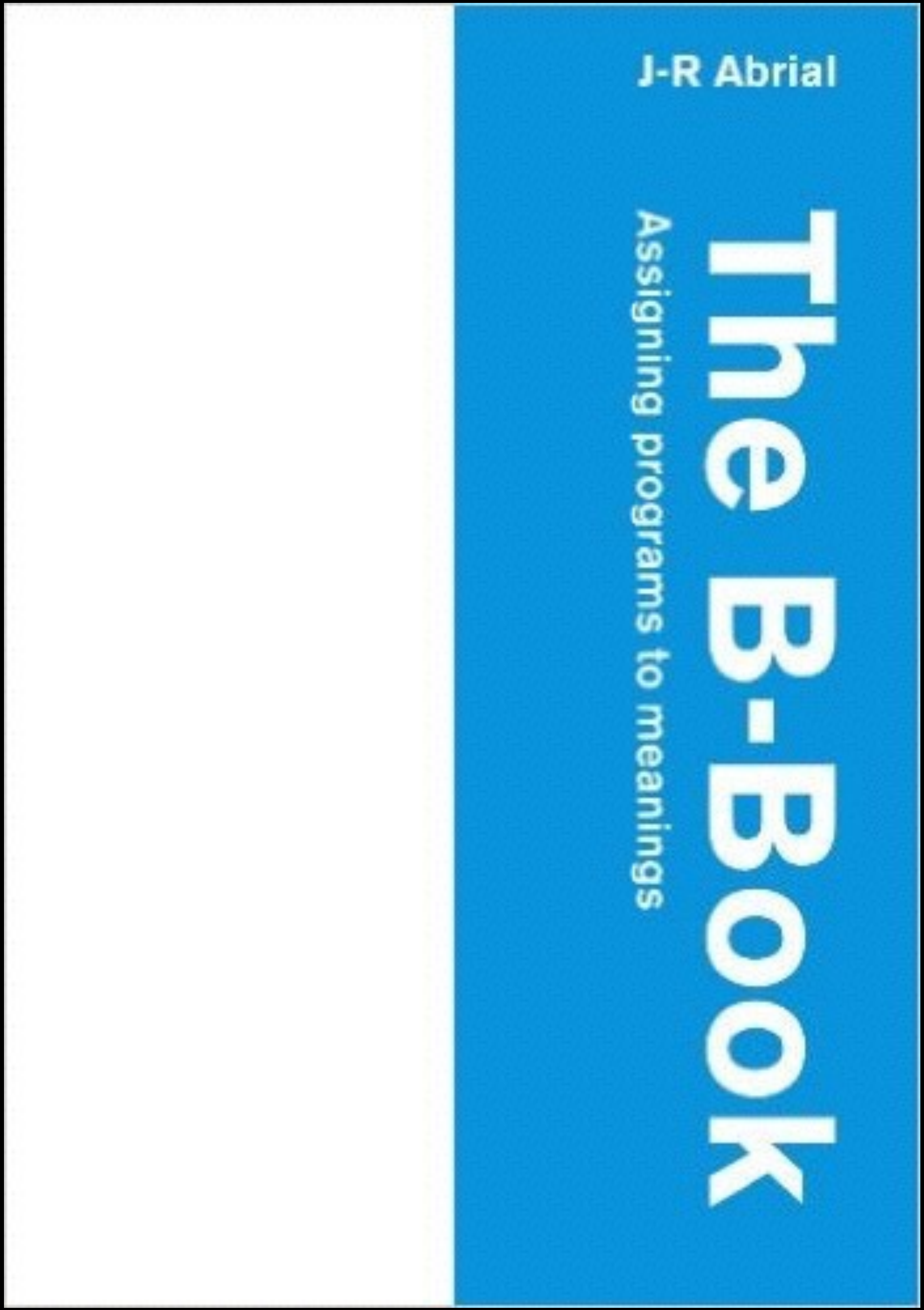
- Ingredients
  - B method
  - Isabelle/HOL
- Formal framework
  - Labeled transition systems as models for components
  - Simulation as model for refinement
  - B method

Overview

- Ingredients
    - ⋆ B method
    - ⋆ Isabelle/HOL
- Formal framework
    - ⋆ Labeled transition systems as models for components
    - ⋆ Simulation as model for refinement
    - ⋆ B method

B method

a method for specifying, designing and coding software components



Assigning programs to meanings

- Specify
  - ⋆ Mathematical language: FOL, set theory, integer arithmetics, substitutions
- Verify
  - ⋆ Weakest-precondition calculus
- Assign programs
  - ⋆ Refinement calculus
  - ⋆ Imperative programming

Tool support

- IDE
    - Atelier-B (free, partly open-source)
    - B-Toolkit (free, open-source)
- animation: BRAMA, ProB
- model checking (ProB)
- code generation (b2llvm)
- etc.

ATELIER B

METROS AND TRAINS EQUIPPED WITH **B SIL4 SOFTWARE**

NEW YORK
SAN JUAN
MEXICO
CARACAS
SANTIAGO
SAO PAULO
PANAMA
TORONTO

LAUSANNE
MILANO
BARCELONA
MADRID
LISBON
ALGIERS
CAIRO
BUDAPEST
PARIS
MALAGA
DUBAÏ

BENGALORE
DELHI
SEOUL
BEJING
SHANGAÏ
HONK-KONG
SINGAPOUR
NINGBO
TAICHUNG
KUNMING
SHENZHEN
GUANGHOU

B FORMAL METHOD

- DEVELOPMENT OF SAFETY CRITICAL SIL4 SOFTWARE
- FREE DOWNLOAD OF ATELIER B 4.0
- BUG FREE PROVEN

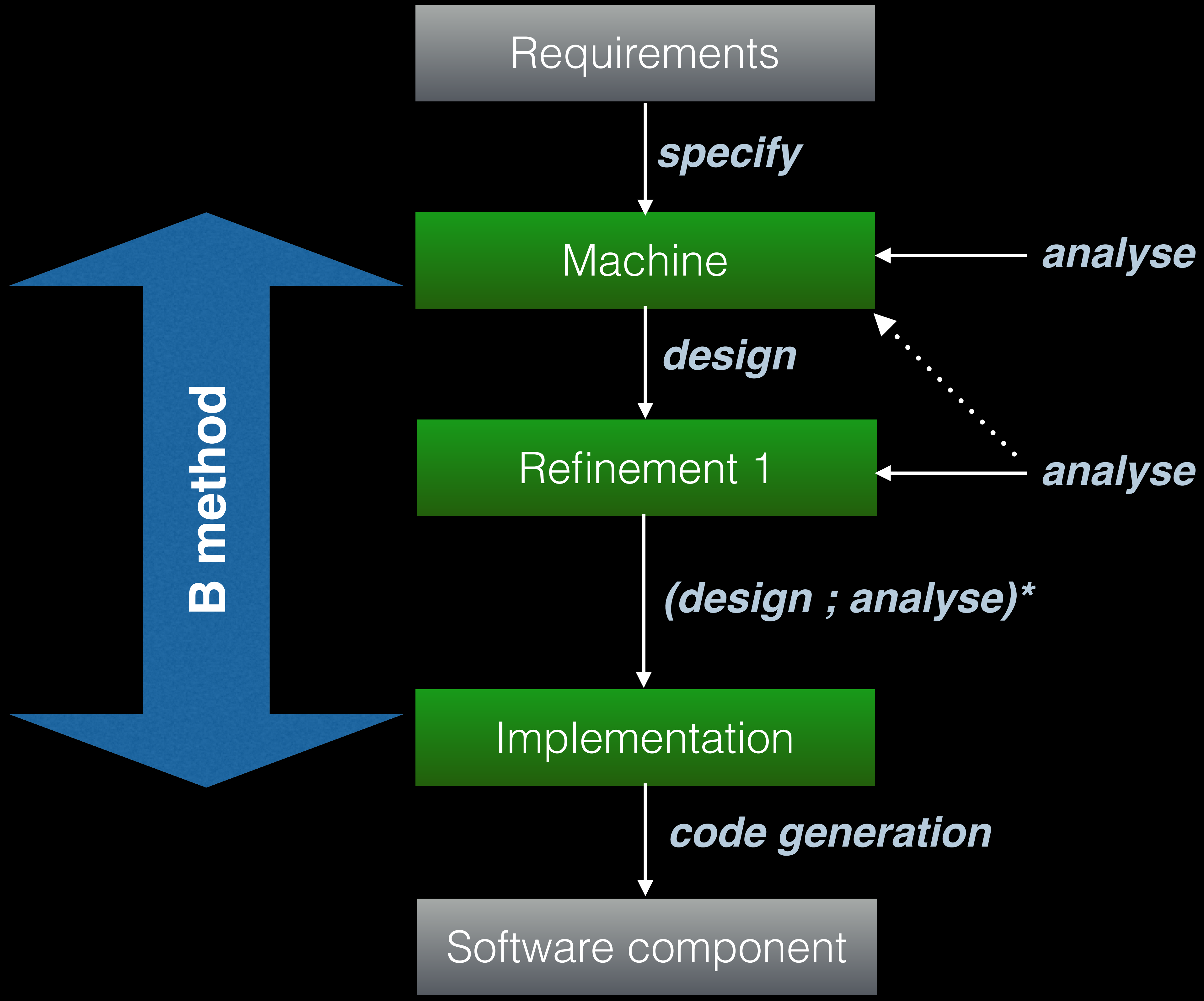
CLEARSY
SYSTEM ENGINEERING

ATELIER B

WWW.CLEARSY.COM

WWW.ATELIERB.EU

Overview



Requirements

specify

Machine

analyse

design

Refinement 1

analyse

*(design ; analyse)**

Implementation

code generation

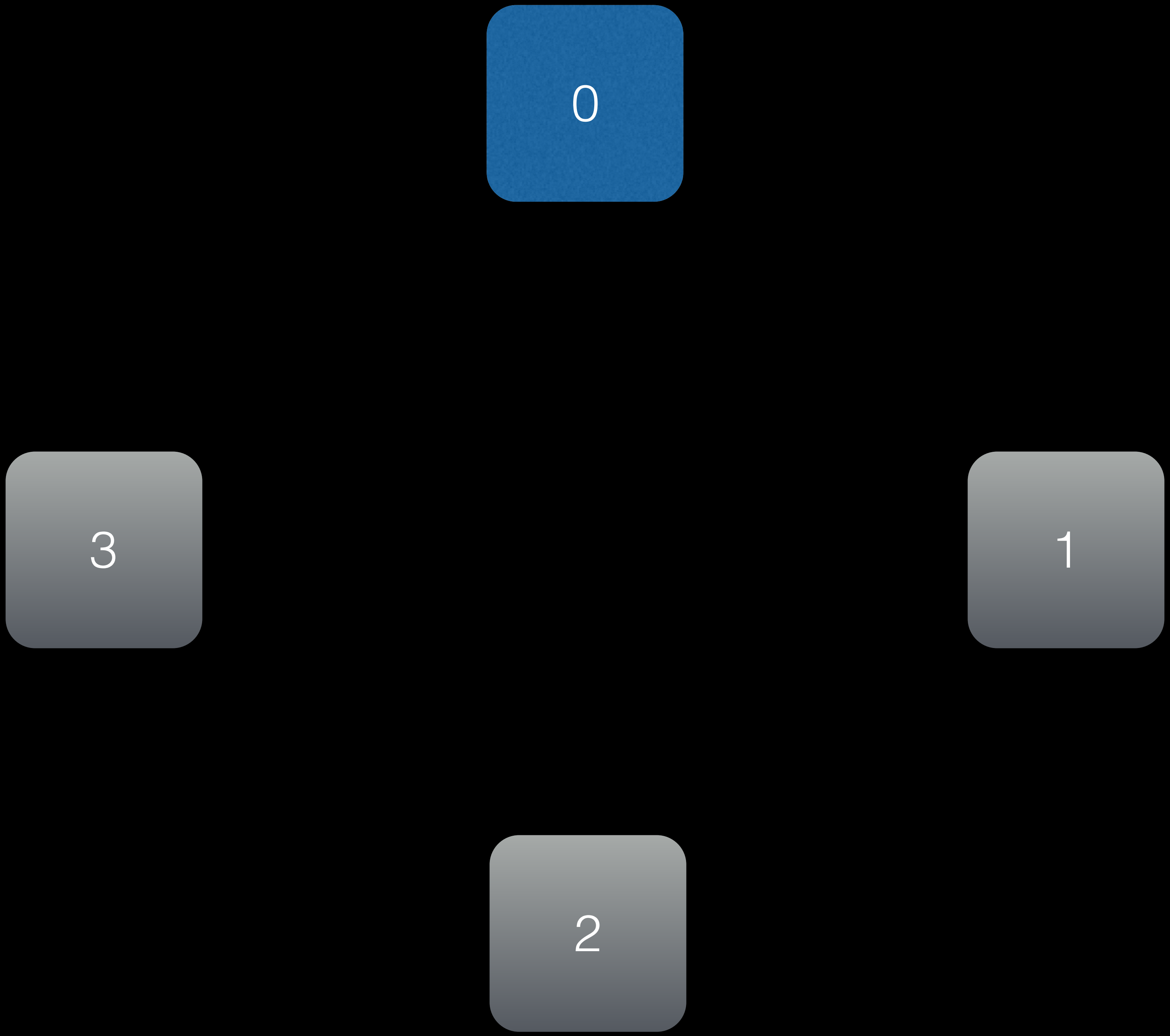
Software component

B method

B by example

- Distributed termination detection

Processors: 0, 1, 2 e 3



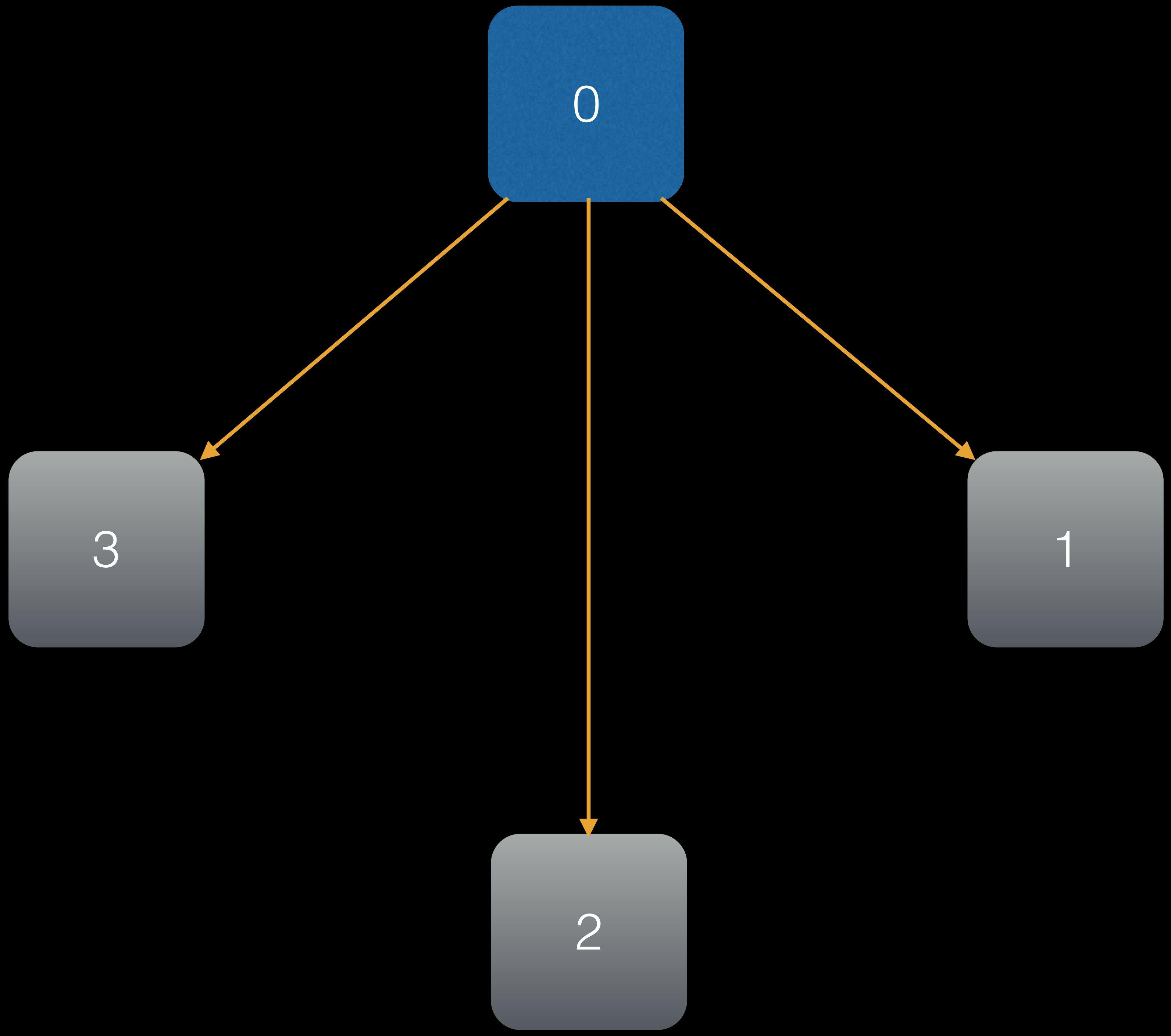
0

3

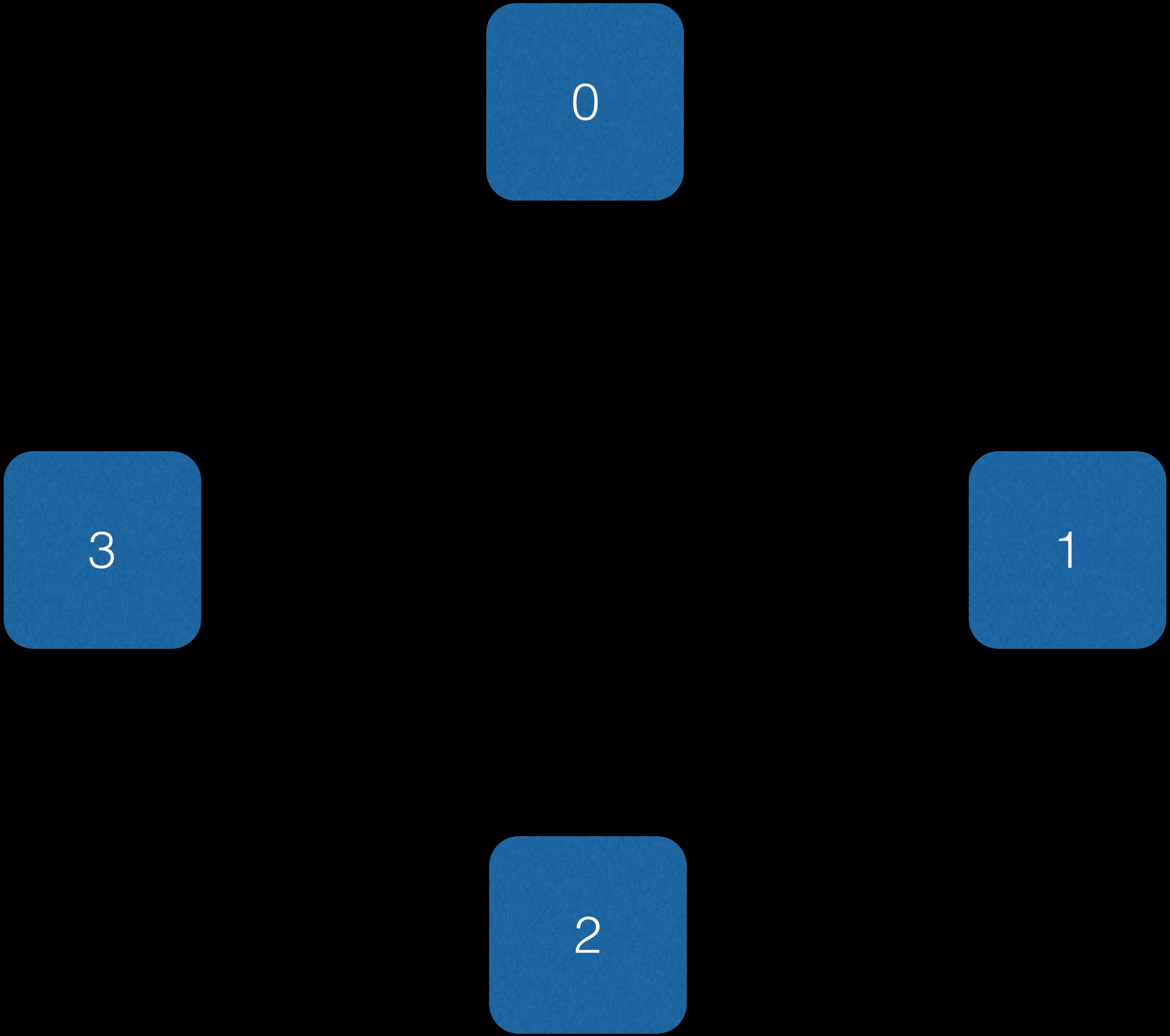
1

2

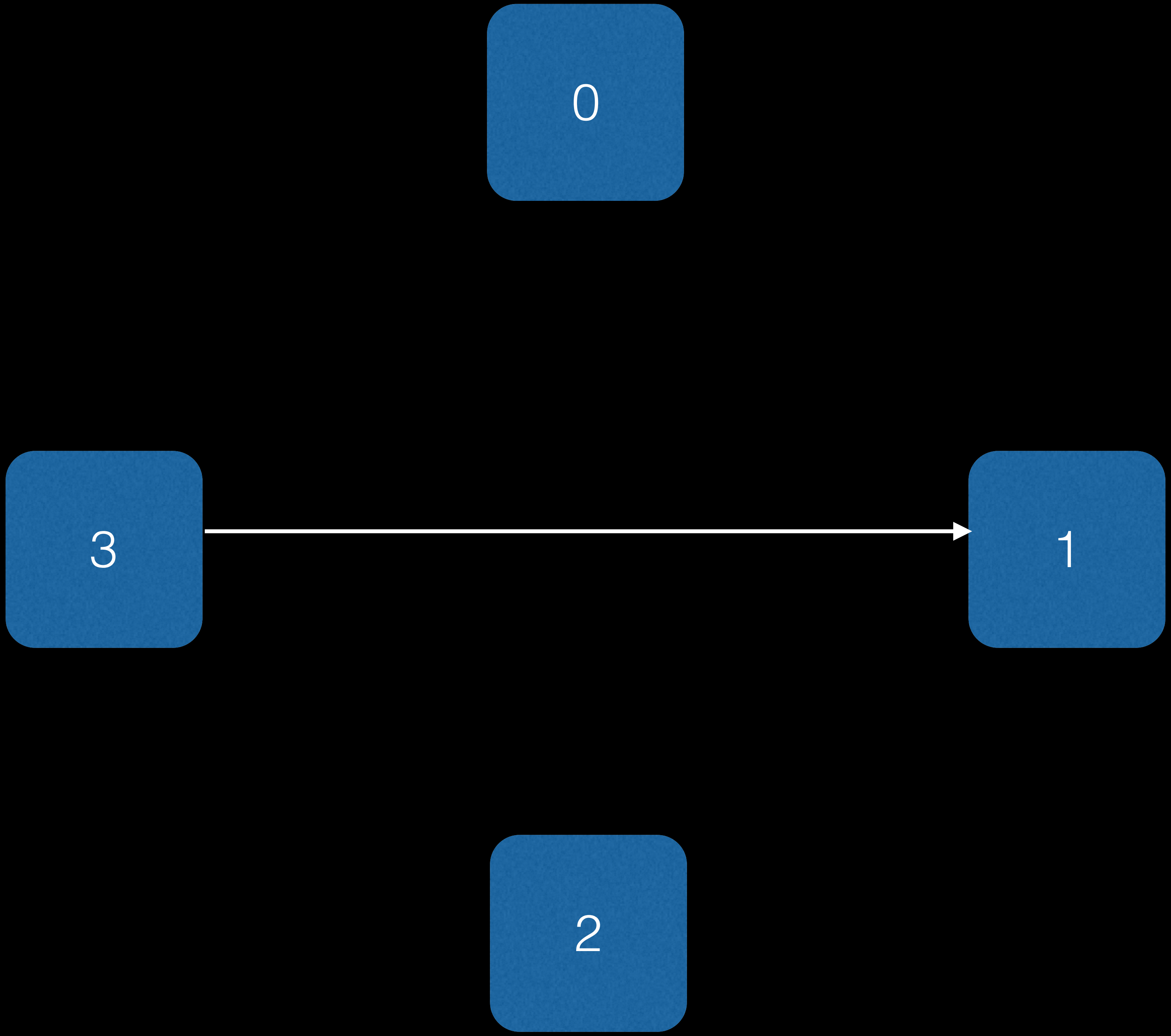
0 send messages to 1, 2 e 3



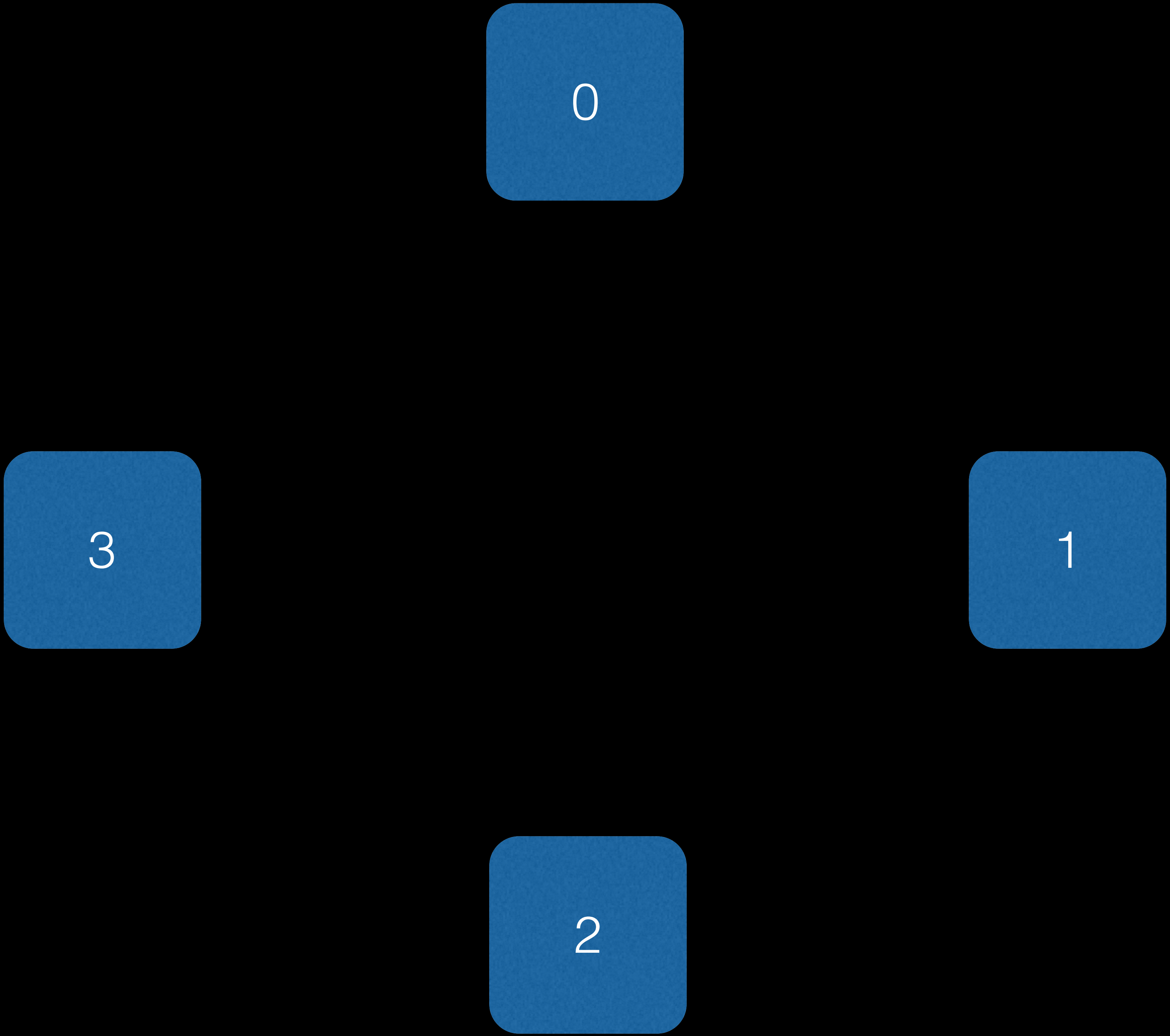
1, 2 e 3 compute



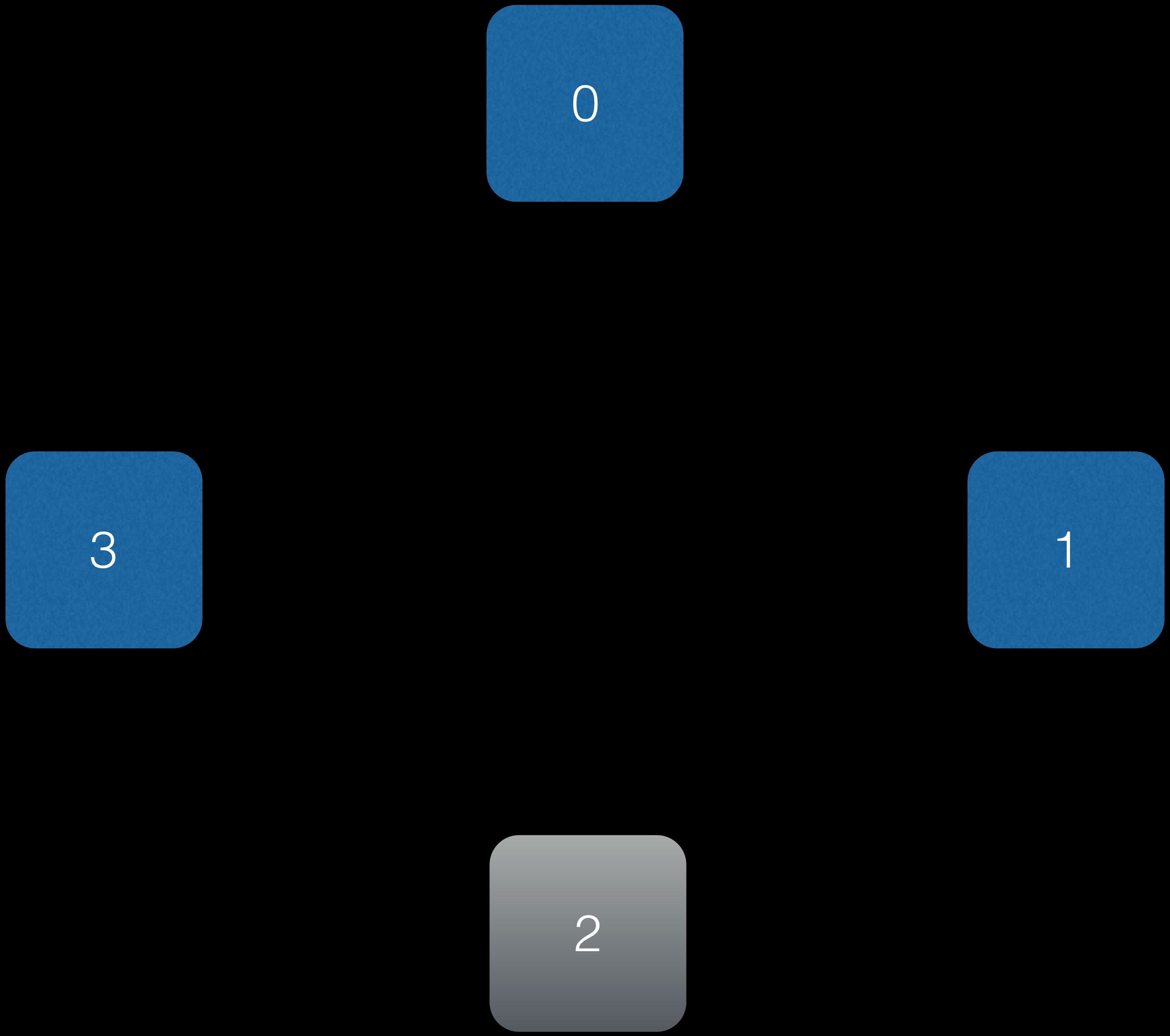
3 sends data to 1



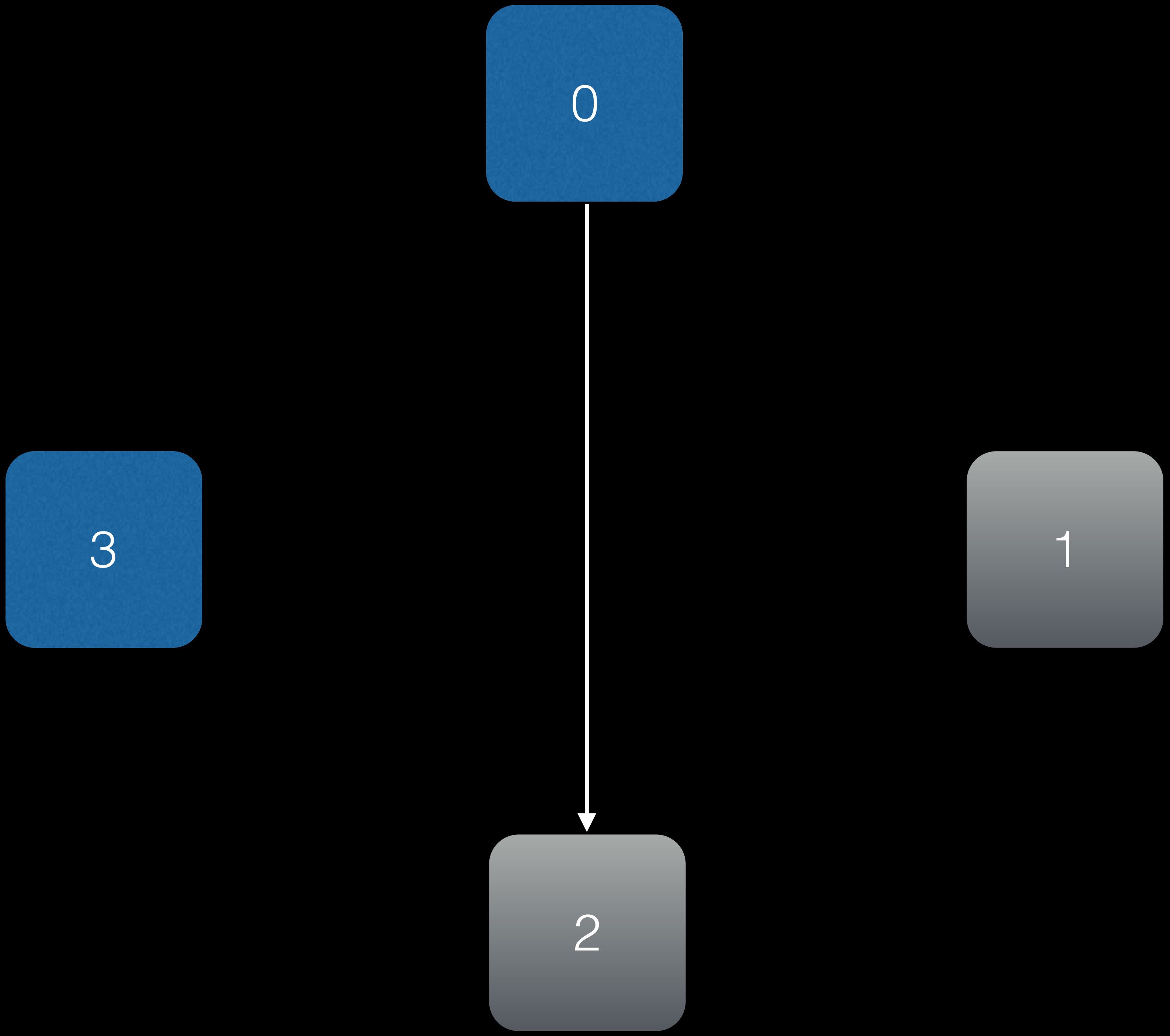
computation proceeds



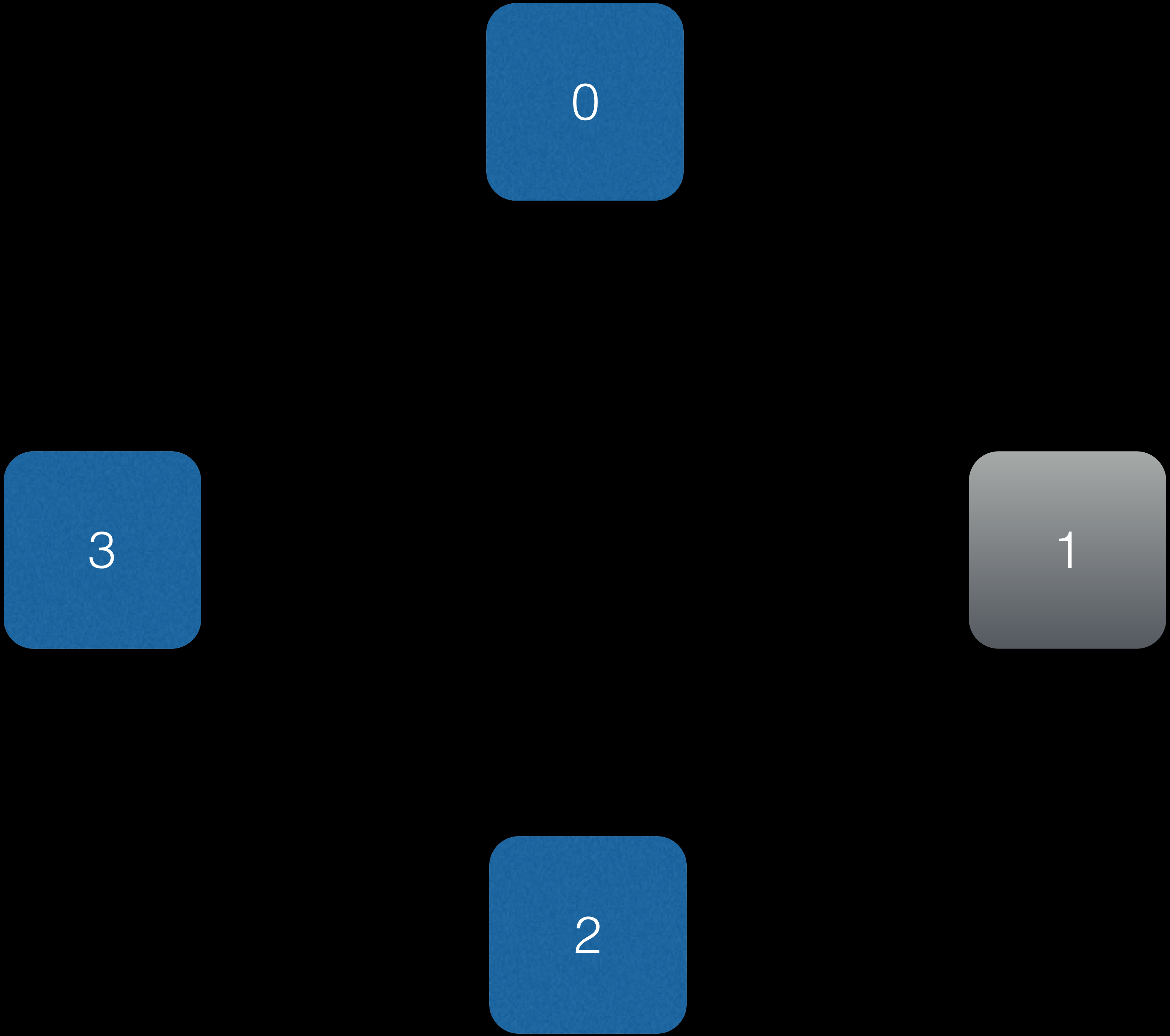
2 terminates, becomes idle



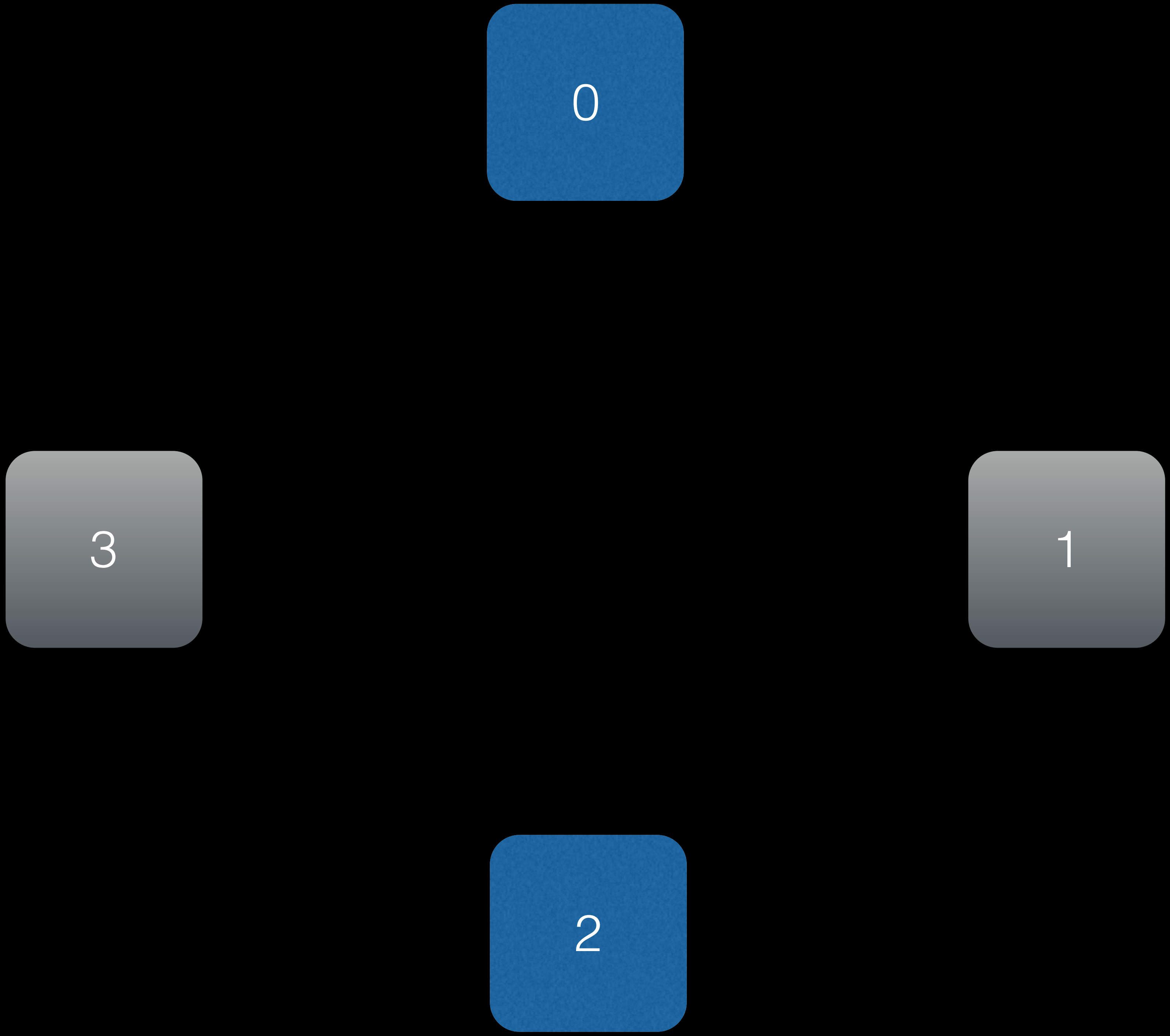
1 becomes idle, 0 sends data to 2



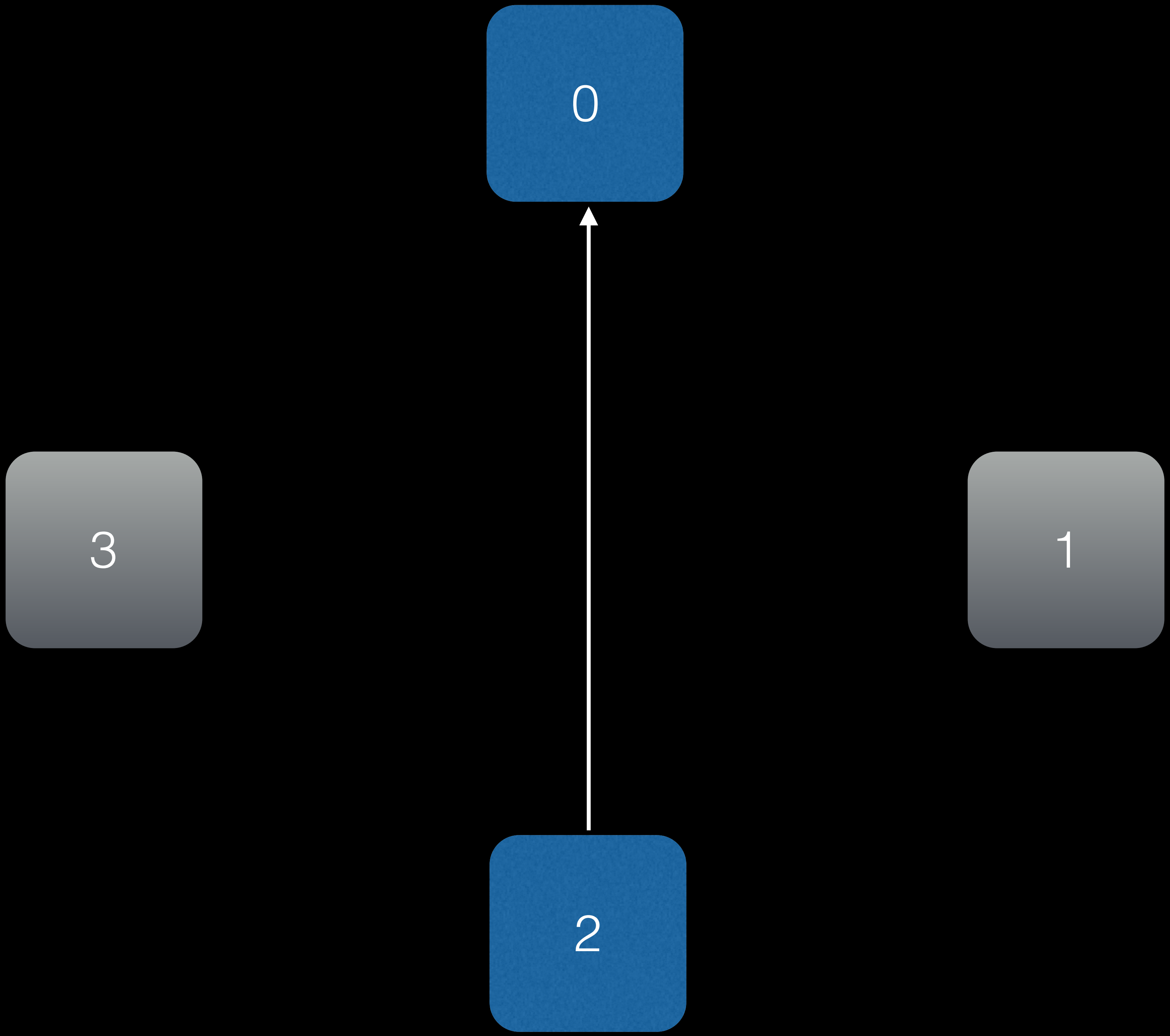
upon receiving a message, 2 becomes active again



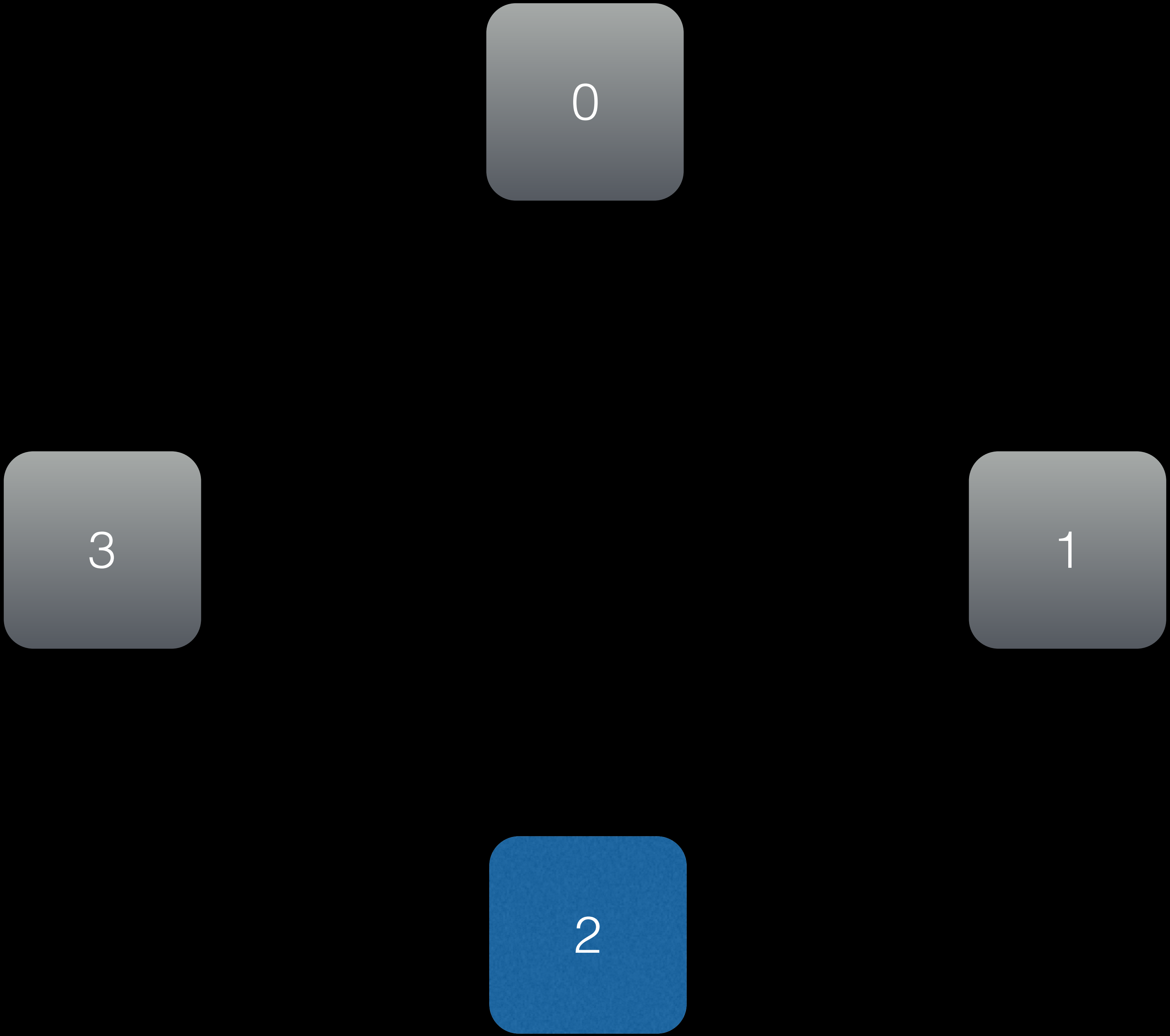
3 becomes idle



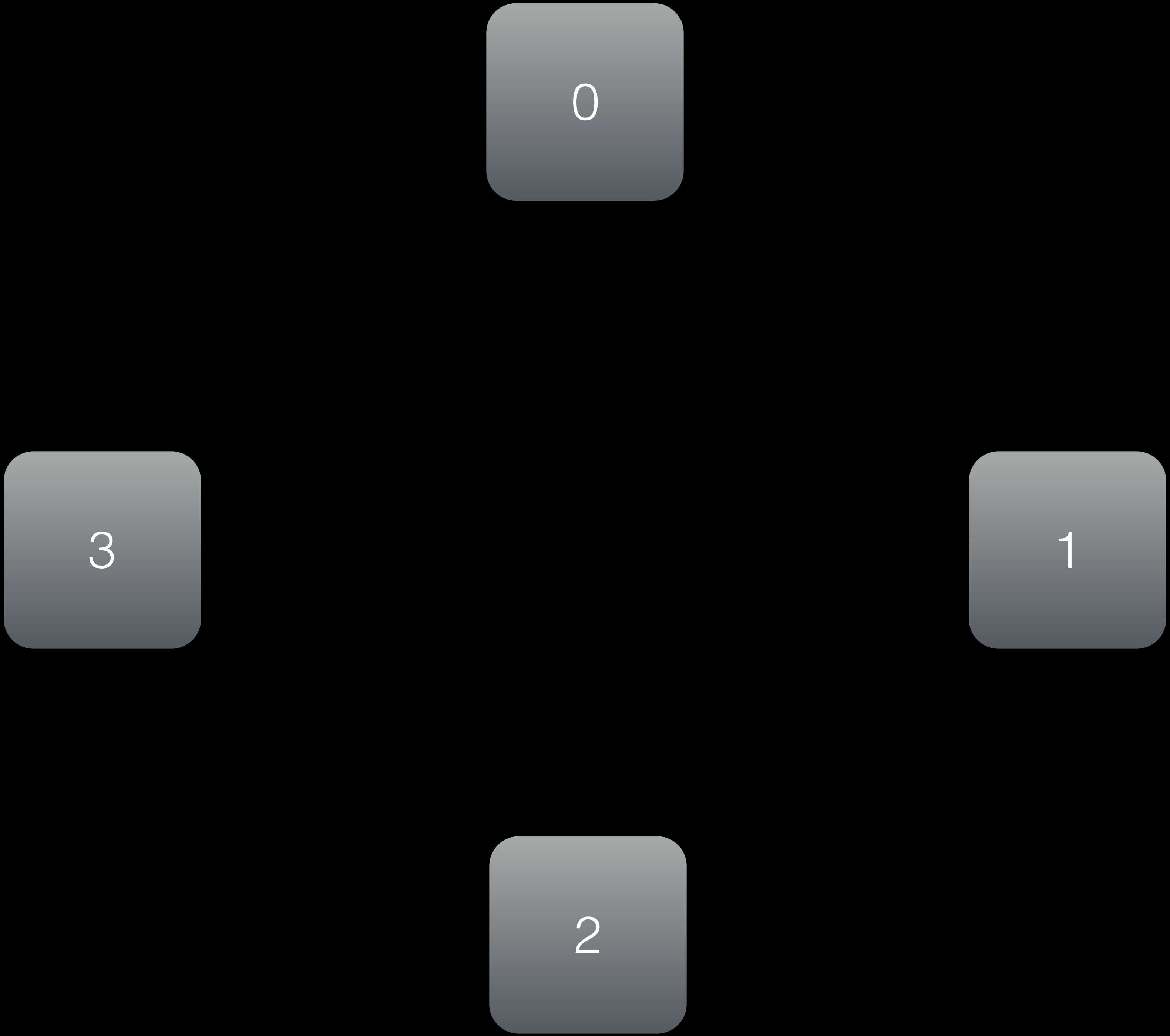
2 sends a message to 0



0 becomes idle



2 becomes idle



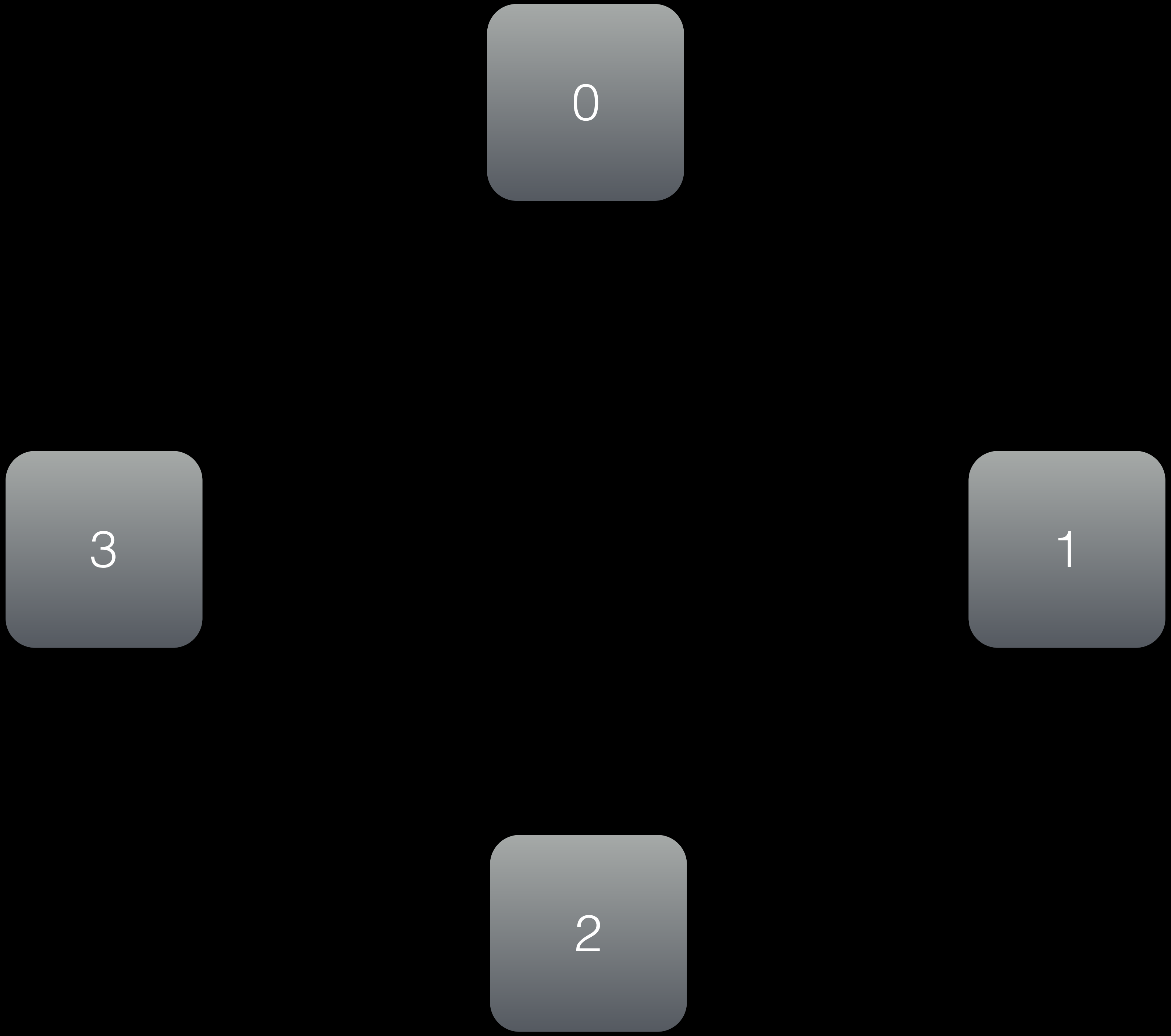
0

3

1

2

How can 0 can detect that 1 2 and 3 are idle to report end of computation?



Solution EWD 840

- Edsger W. Dijkstra, W.H.J. Feijen, A.J.M. van Gasteren

- Derivation of a termination detection algorithm for distributed computations

B model

```
MACHINE EWD840

CONSTANTS NUM

PROPERTIES NUM ∈ NAT1

DEFINITIONS PROC ==  0 .. (NUM-1)
```

- NUM processors [#0, #1, #2, … #(NUM-1)]

B model

- State of processors
  - active
  - idle

```
VARIABLES idle
INVARIANT
 idle ⊆ PROC
INITIALISATION
  idle :∈ 𝒫(PROC)
```

B model

- Detect termination

```
VARIABLES terminated
INVARIANT
  terminated ∈ BOOL
INITIALISATION
  terminated := FALSE
```

B model

INVARIANT
terminated = TRUE ⇒ idle = PROC

- Functional requirement
  - termination occurs when all processors are idle

Approach

- A token circulates among processors
- When processor terminates, forward token to successor

```
VARIABLES token
INVARIANT token ∈ PROC
INITIALISATION token := 0
```

- Hypothesis:
  - Instantaneous communication
  - #0 → #(N-1) … #2 →#1 →#0

Approach

INVARIANT token+1..num ⊆ idle

but… what if a message reaches P after token has left?

- Desired property
  - ★ All processors above token position are idle.

Solutions

- but… what if a message reac left?
- Token already
  - The sender
  - It must be i
  - “tag” the token to indicate that one more run is required

```
SETS
  COLOR = { BLACK, WHITE }
VARIABLE
  token_color
INVARIANT
  token_color ∈ COLOR ∧
  (token+1..num ⊆ idle ∨ token_color = BLACK)
INITIALISATION
  color_token := WHITE
```

but… what if the sender does not have the token?

Solutions

- but… what if the sender does not have the token?
- The sender is in 0..token
  - ⋆ When a processor sends a her processor with higher inde
  - ⋆ When a flag is set on the processor with the token, it tags the token.

```
VARIABLES
  tainted
INVARIANT
  tainted ⊆ PROC ∧
  (token+1..num ⊆ idle ∨
   tainted ∩ 0..token ≠ ∅ ∨
   token_color = BLACK)
INITIALISATION
  tainted := ∅
```

Partial synthesis

```
MACHINE EWD840
CONSTANTS NUM
PROPERTIES NUM ∈ NAT1
SETS
  COLOR = { BLACK, WHITE }
DEFINITIONS PROC ==  0 .. (NUM-1)
VARIABLES
  idle, terminated, token, tainted, token_color
INVARIANT
  idle ⊆ PROC ∧ terminated ∈ BOOL ∧ token ∈ PROC ∧
  tainted ⊆ PROC ∧ token_color ∈ COLOR ∧
 (token+1..num ⊆ idle ∨
   tainted ∩ 0..token ≠ ∅ ∨
   token_color = BLACK)
INITIALISATION
  idle :∈ 𝒫(PROC) || terminated := FALSE || token := 0 ||
color_token := WHITE  || tainted := ∅
```

Behaviour

- An active processor might become idle anytime

```
Finish(pr) =
PRE pr ∈ PROC ∧ pr ∉ idle THEN
  idle := idle ∪ {pr}
END;
```

Behaviour

- A
  a

```
Send_Message(pri,prj) =
PRE pri ∈ PROC ∧ prj ∈ PROC ∧ pri ∉ idle THEN
  IF prj ∈ idle THEN
    idle := idle - {prj}
  END ||
  IF prj > pri THEN
    tainted := tainted ∪ {pri}
  END
END
```

Behaviour

- When a processor becomes idle and it has the token, the token goes to the next processor.
  - ★ Processor 0 has a special behaviour (resets the token, etc.)

```
Pass_Token = PRE $token \neq 0 \land token \in idle$ THEN
  token := token - 1 ||
  IF $token \in tainted$ THEN
    color_token := BLACK
  END ||
  tainted := tainted - { token }
END
```

Behaviour

- Processor 0:
  - resets the token
  - forwards the token to the processor with higher index

```
Initiate_Probe = PRE token = 0 ∧ color_token = BLACK THEN
  tainted := tainted - {0} ||
  color_token := WHITE ||
  token := NUM - 1
END
```

Behaviour

```
Terminated = PRE token = 0 ∧ color_token = WHITE ∧ 0 ∉ tainted ∧ 0 ∈ idle
THEN
terminated := TRUE
END
```

- token has returned to 0
- token is not tagged
- 0 is not flagged
- 0 is idle

Next steps

- Syntax and type checking
- D
  expressions -defin
- reachable s satisfy the invariant
- Ani (small instances)
- Model check: no deadlock (small instances)

Welcome to Atelier B

Please select from the following sections where you would like to start...

Projects

New Projects

- Create a Project Software Developement
- Create a Project System Modelisation
- Create a Project Data Generation

Recent Projects

- timer
- EWD840

Help and Tutorials

- Atelier B 4.1 in a nutshell
- Tutorials
- Examples
- FAQ

Web Sites

- Atelier B
- B-Method

Next ste

- Design implementation
-
  - ★
  - ★ Pr
- Each compon may be designed individually

Untitled — Edited

Welcome to Atelier B

Please select from the following sections where you would like to start...

Projects

New Projects

- Create a Project Software Developement
- Create a Project System Modelisation
- Create a Project Data Generation

Recent Projects

- timer

Help and Tutorials

- Atelier B 4.1 in a nutshell
- Tutorials
- Examples
- FAQ

Web Sites

- Atelier B
- B-Method

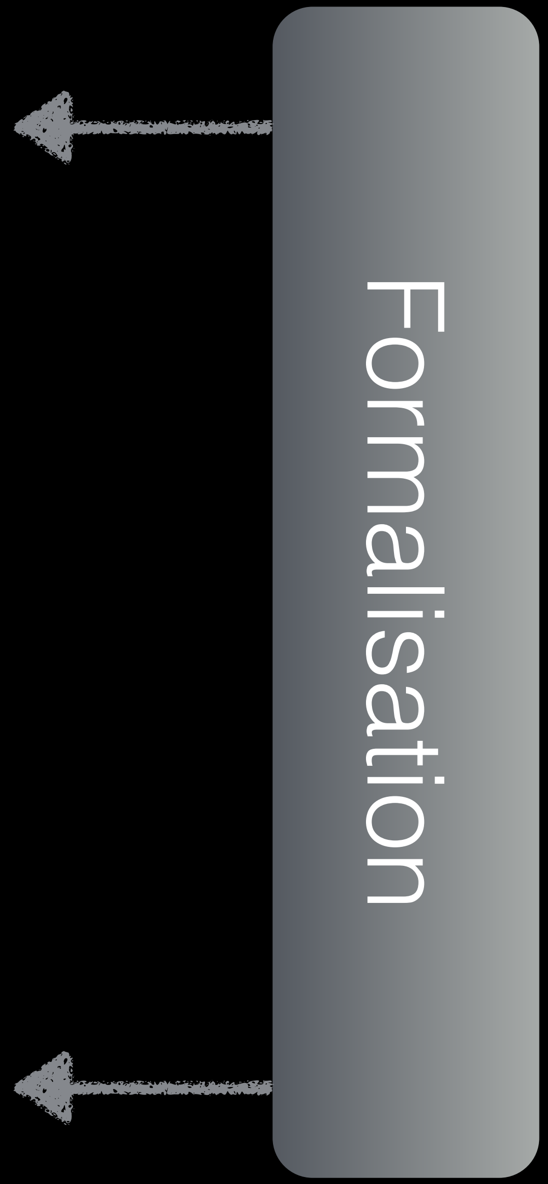
Workspaces

- local
  - Biblioteca
  - C
  - CBSOFT
  - CONTAINERS
  - CRM
  - CentralDoCidadao
  - EWD840
  - ILP
  - Jb
  - Jukebox
  - SbC
  - TestCodeGen
  - Tutoriel
  - b2llvm_ts
  - epoca
  - exemplo1
  - freertos
  - gdc
  - gdc2007
  - gdc2011

Errors: No errors

B-method: synthesis

- Specification:
  - non-deterministic state-machines
  - state = valuation of state variables
  - transition = operation execution
- Design:
  - refinement relation


Formalisation

Overview

- Ingredients
  - ⋆ B method
  - ⋆ Isabelle/HOL
- Formal framework
  - ⋆ Labeled transition systems as models for components
  - ⋆ Simulation as model for refinement
  - ⋆ B method

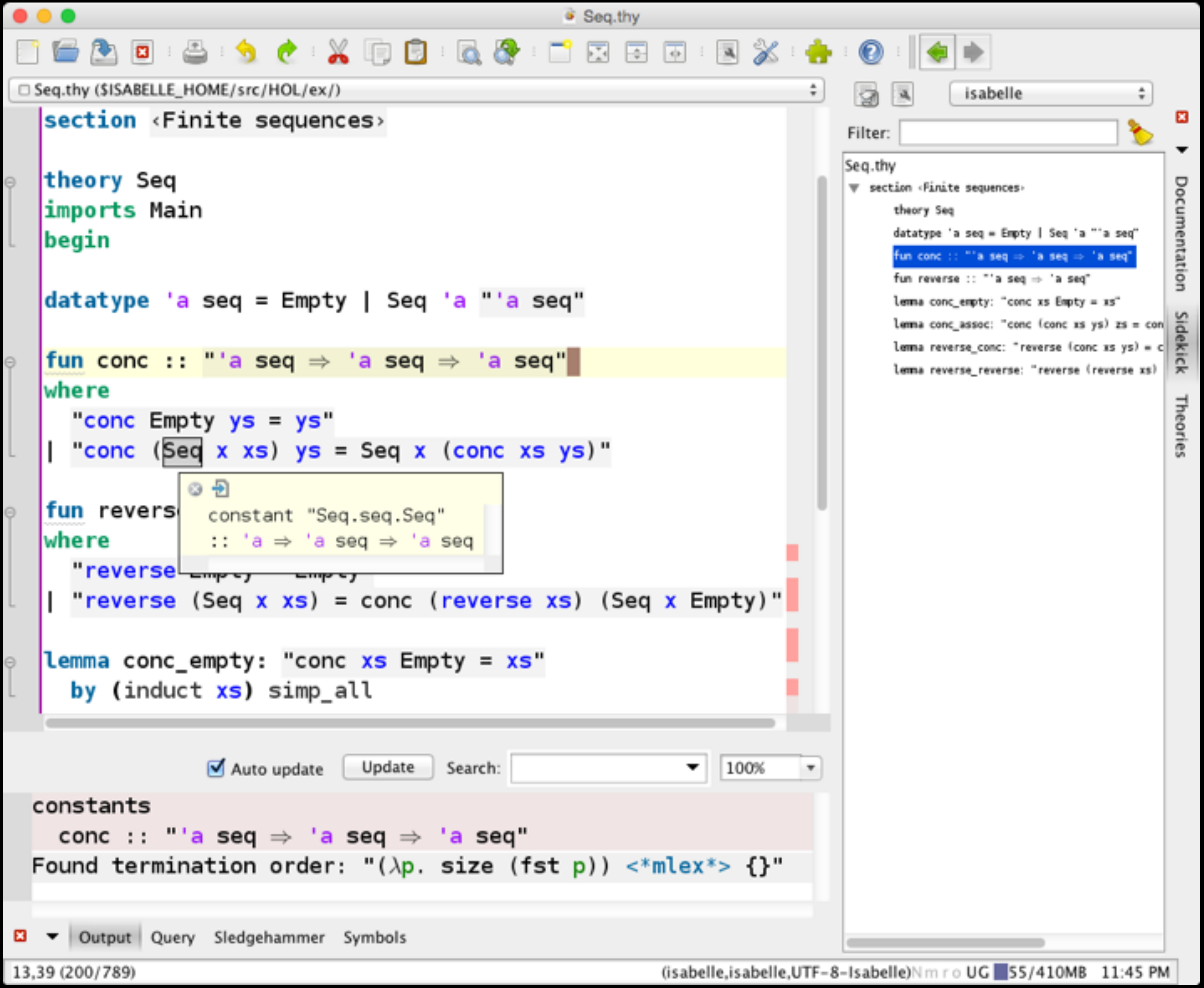
Formalisation

- Interactive theorem prover
  - ⭑ ACL2, Coq, Isabelle, LCF, Maude, PVS, etc.
  - ⭑ programming language: define inductive data types, recursive functions
  - ⭑ logic: specify properties of interest
  - ⭑ proof engine: verify properties
  - ⭑ code generation: execute defined functions

Isabelle/HOL

- functional programming language
- typed, higher order logic
- proof:
  - interactive
  - automatic: tableaux, rewriting systems, decision procedures
- code generation: SML, OCaml, Scala

Tool support



```
section ‹Finite sequences›

theory Seq
imports Main
begin

datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
where
  "conc Empty ys = ys"
| "conc (Seq x xs) ys = Seq x (conc xs ys)"

fun revers
where
  "reverse
| "reverse (Seq x xs) = conc (reverse xs) (Seq x Empty)"

lemma conc_empty: "conc xs Empty = xs"
  by (induct xs) simp_all
```

```
constant "Seq.seq.Seq"
:: 'a ⇒ 'a seq ⇒ 'a seq
```

```
Seq.thy
section ‹Finite sequences›
  theory Seq
  datatype 'a seq = Empty | Seq 'a "'a seq"
  fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
  fun reverse :: "'a seq ⇒ 'a seq"
  lemma conc_empty: "conc xs Empty = xs"
  lemma conc_assoc: "conc (conc xs ys) zs = con
  lemma reverse_conc: "reverse (conc xs ys) = c
  lemma reverse_reverse: "reverse (reverse xs)
```

```
constants
  conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
Found termination order: "(λp. size (fst p)) <*mlex*> {}"
```

```
13,39 (200/789)                (isabelle,isabelle,UTF-8-Isabelle)Nmro UG 55/410MB 11:45 PM
```

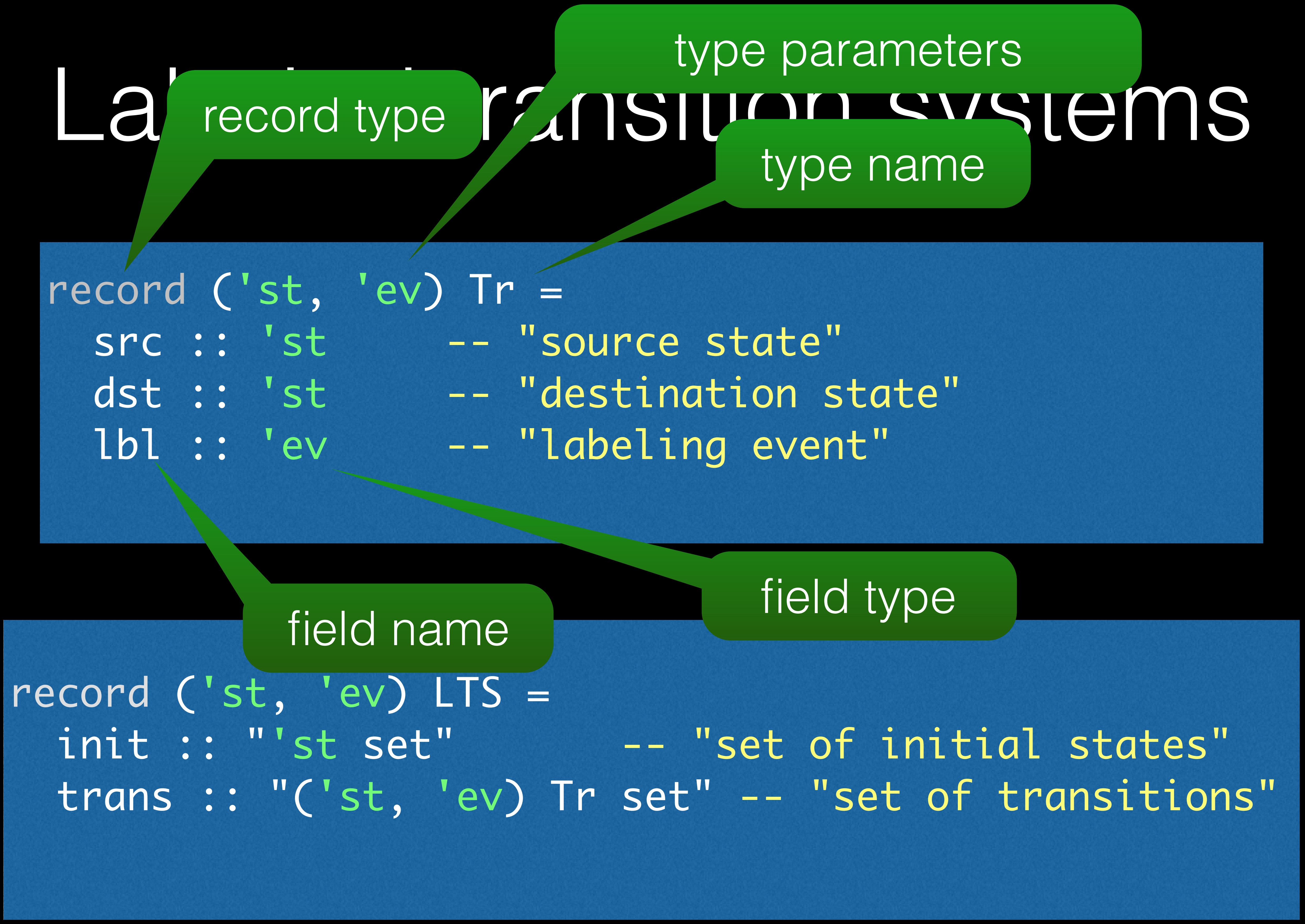
Formalisation

- LTS
- Simulation
- B method
  - without pre-condition
  - with pre-condition

Formalisation principles

- A B component is a labeled transition system
  - valuation of the variables ⟺ state
  - initial states, reachable states
- Transitions occur when an operation is applied
  - operation ⟺ event
  - state + event ⇸ ⟺ partial
  - state + event → states ⟺ non deterministic

La ransition systems



record type

type parameters

type name

```
record ('st, 'ev) Tr =
  src :: 'st       -- "source state"
  dst :: 'st       -- "destination state"
  lbl :: 'ev       -- "labeling event"
```

field name

field type

```
record ('st, 'ev) LTS =
  init :: "'st set"            -- "set of initial states"
  trans :: "('st, 'ev) Tr set" -- "set of transitions"
```

Labeled transition systems

defines a named value

the name

the type

```
definition successors ::
  "('st, 'ev) LTS ⇒ 'st set ⇒ 'st set"
where
  "successors l S ≡
     { dst t | t . t ∈ trans l ∧ src t ∈ S }"
```

the value

inductive (set) definition

parameter

ates

introduction rule

```
inductive_set states :: "('st, 'ev) LTS ⇒ 'st set"
  for l :: "('st, 'ev) LTS"
where
  base[elim!]: "s ∈ init l ⟹ s ∈ states l"
| step[elim!]:
   "⟦ t ∈ trans l; src t ∈ states l ⟧
    ⟹ dst t ∈ states l"
```

introduction rule

ces of case-split rules for inductive proofs:

```
inductive_cases base : "s ∈ states l"
inductive_cases step : "dst t ∈ states l"
```

we want to establish a property

name

expression

```
lemma reachable_init: "init l ⊆ states l"
by auto
```

proof construction

```
        ble_stable:
         ors l (states l) ⊆ states l"
unfolding successors_def
```

to prove a proposition **P** is true in all states of LTS l:

1. prove **P** is true in **init l**
2. prove **P** is preserved by **trans l**

```
                                 t:
                                 l"
```

```
lemma reachable_induct_predicate = states.induct
using assms
by (induct s) (auto simp: successors_def)
```

Internal behaviour

```
inductive_set runs :: "('st, 'ev) LTS ⇒ ('st, 'ev) Run set"
  for l :: "('st, 'ev) LTS"
where
| start: "⟦ t ∈ trans l; src t ∈ init l ⟧ ⟹ [t] ∈ runs l"
| step: "⟦ t ∈ trans l; ts ∈ runs l; ts ≠ []; src t = dst (last ts) ⟧
        ⟹ ts @ [t] ∈ runs l"
```

```
  type_synonym ('st, 'ev) Run = "('st, 'ev) Tr list"
```

```
inductive_cases empty_run : "[] ∈ runs l"
inductive_cases one_step_run : "[t] ∈ runs l"
inductive_cases multi_step_run : "ts @ [t] ∈ runs l"
```

Properties of internal behaviour

```
lemma "ts ∈ runs l ⟹ ts ≠ [] ⟹ src (hd ts) ∈ init l"
  by (induct rule: runs.induct, auto)
```

External behaviour

```
type_synonym 'ev Trace = "'ev list"
```

```
definition traces :: "('st, 'ev) LTS ⇒ 'ev Trace set"
where
  "traces l ≡ (map lbl) ` (runs l)
```

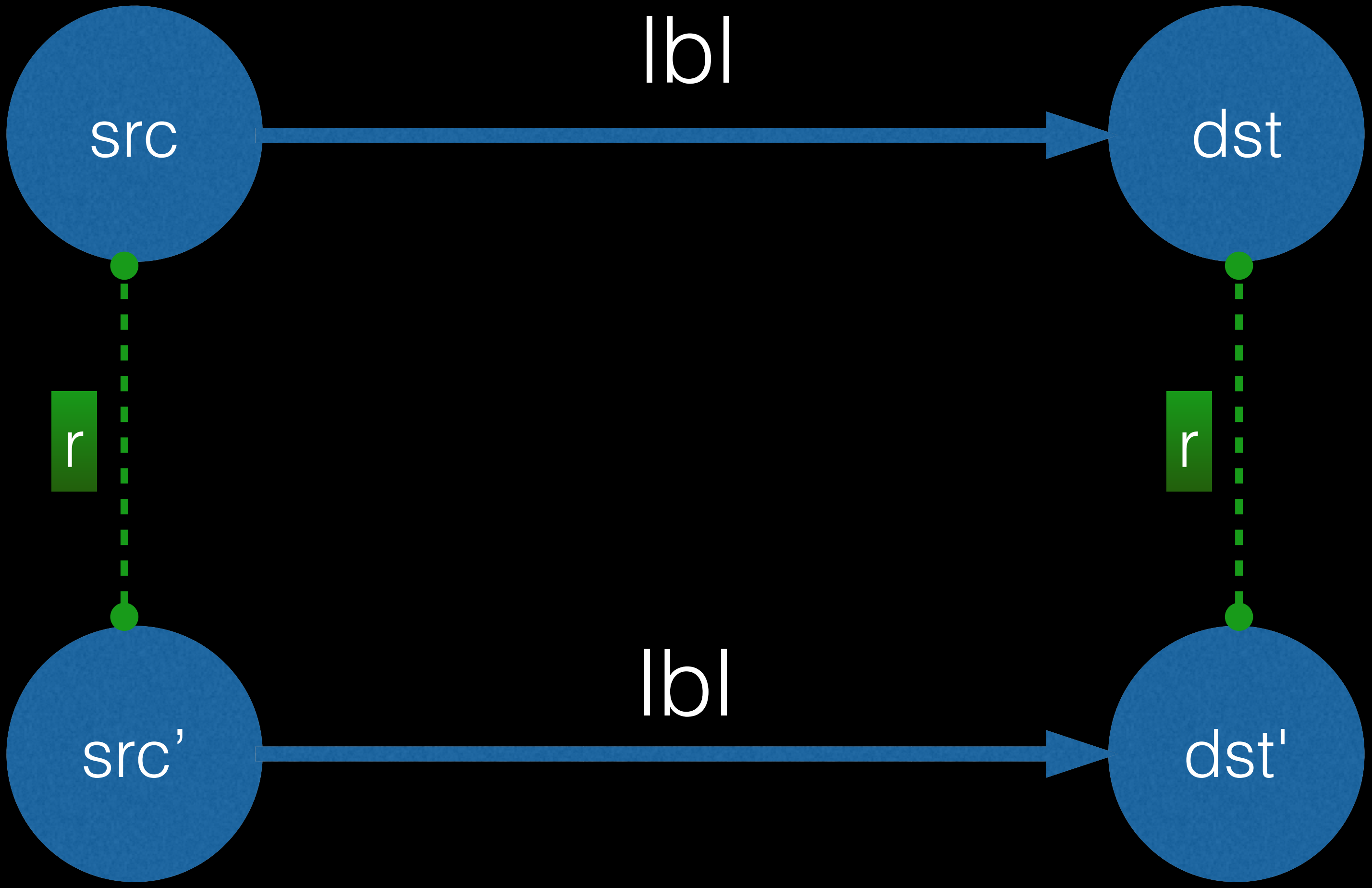
maps function **lbl** to each element of list and returns list of results
map lbl : converts run to trace

operator ` is relational image

Formalisation

- LTS
- Simulation
- B method
  - without pre-condition
  - with pre-condition

Simulation between transitions



```
definition sim_transition :: "'st rel ⇒ ('st, 'ev) Tr rel"
where
  "sim_transition r ≡
     { (t,t') | t t’. (src t, src t') ∈ r
       ∧ lbl t = lbl t' ∧ (dst t, dst t') ∈ r }"
```

Simulation between LTSes

```
definition simulation :: "'st rel ⇒ ('st, 'ev) LTS rel"
where
  "simulation r ≡ { (l,l') | l l'.
     (∀s ∈ in
   ∧ (∀s s'.
        (∀t ∈
          (∃t' ∈ trans l'. sr  ' = s' ∧
                           (t,  ') ∈ sim_transition r))) }"
```

Left-associative, infix operator ≼, with precedence 50, is syntactic sugar for **simulated**

```
definition simulated (infixl "≼" 50)
where "(l ≼ l') ≡ ∃r. (l,l') ∈ simulation r"
```

Properties of simulation

- The identity relation on LTS is a simulation.
- The relational composition of two simulations is a simulation.
- The simulated relation is reflexive and transitive.

Simulation and behavior

```
definition sim_run ::
  "'st rel ⇒ ('st, 'ev) Run rel”
where
  "sim_run r ≡
    {(ts, ts') | ts ts'.
     list_all2 (λt t'. (t,t') ∈ sim_transition r) ts ts'}"
```

Simulation and runs

- To two similar runs correspond the same trace (sequence of events).
- Similar runs have equal length.
- other properties have been shown

Simulation on LTSes and behaviour

```
theorem sim_run:
  assumes "(l,l') ∈ simulation r" and "ts ∈ runs l"
  obtains ts' where
    "ts' ∈ runs l'" "(ts,ts') ∈ sim_run r"
```

```
lemma sim_traces:
  assumes "(l,l') ∈ simulation r" and "t ∈ traces l"
  shows "t ∈ traces l’"
```

```
theorem sim_trace_inclusion:
  "(l,l') ∈ simulation r ⟹ traces l ⊆ traces l'"
```

```
corollary simulates_traces:
"l ≼ l' ⟹ traces l ⊆ traces l'"
```

Formalisation

- LTS
- Simulation
- Bmethod
  - without pre-condition
  - with pre-condition

B machine

```
record ('st, 'ev) B_machine =
  lts :: "('st, 'ev) LTS"
  invariant :: "'st ⇒ bool"
```

```
definition sound_B_machine ::
  "('st, 'ev) B_machine ⇒ bool"
where
  "sound_B_machine m ≡ ∀s ∈ states (lts m). invariant m s"
```

Correctness of B machines

```
theorem machine_po:
  assumes "⋀s. s ∈ init (lts m) ⟹ invariant m s"
      and "⋀t. ⟦t ∈ trans (lts m); invariant m (src t)⟧
               ⟹ invariant m (dst t)"
    shows "sound_B_machine m"
unfolding sound_B_machine_def
using assms
by (auto elim: states.induct)
```

B refinement

```
record ('st, 'ev) B_refinement =
  abstract :: "('st, 'ev) LTS"      -- "the abstract component"
  concrete :: "('st, 'ev) LTS"      -- "the concrete component "
  invariant :: "'st × 'st ⇒ bool" -- "gluing invariant"
```

```
definition sound_B_refinement ::
  "('st, 'ev) B_refinement ⇒ bool”
where
  "sound_B_refinement r ≡
  (concrete r, abstract r) ∈ simulation (Collect (invariant r))"
```

Properties of B refinement

```
lemma refinement_sim:
  "⟦ sound_B_refinement r ⟧ ⟹ concrete r ≼ abstract r"
```

```
lemma refinement_compose_soundness:
  "⟦ sound_B_refinement r ;
     sound_B_refinement r';
     concrete r = abstract r'⟧
   ⟹ sound_B_refinement (refinement_compose r r')"
```

```
lemma refinement_compose_associative:
  "refinement_compose (refinement_compose r r') r'' =
   refinement_compose r (refinement_compose r' r'')"
```

```
etc.
```

B development

```
type_synonym ('st, 'ev) B_design =
  "('st, 'ev) B_refinement list"
```

```
record ('st, 'ev) B_development =
  spec :: "('st, 'ev) B_machine"
  design :: "('st, 'ev) B_design"
```

B development

```
definition sound_B_design where
  "sound_B_design refs ≡ ∀i < size refs.
     sound_B_refinement (refs!i)
   ∧ (Suc i < size refs ⟶
      concrete (refs!i) = abstract (refs!(Suc i)))"
```

```
definition sound_B_development where
  "sound_B_development dev ≡
    sound_B_machine (spec dev) ∧
    sound_B_design (design dev) ∧
    (design dev ≠ [] ⟶
     (lts (spec dev)) = (abstract (hd (design dev))))"
```

B development

```
lemma design_sim:
  ⟦ "sound_B_design refs” ; "refs ≠ []" ⟧
    ⟹ "concrete (last refs) ≼ abstract (hd refs)"
```

```
theorem development_sim:
  ⟦ "sound_B_development d" ; "design d ≠ []" ⟧
  ⟹ "concrete (last (design d)) ≼ lts (spec d)"
```

Formalisation

- LTS
- Simulation
- Bmethod
  - without pre-condition
  - with pre-condition

Preconditions

- In B, operations may have a precondition.
- precondition ⟺ operation terminates in a safe state.
- ¬ precondition ⟺ no guarantee, even of termination.
    - ⋆ transition ⟺ valid application of operation
    - ⋆ notion of accepted events, outgoing transitions
- Operations in refinement may have weaker preconditions than in abstract counterpart.
    - ⋆ new notions of simulation and refinement

Accepted events

```
definition
  outgoing_tr                                 'ev) Tr set"
where
  "outgoing_t                                c t = s}"
```

Given

- a LTS l and
- a state s,

outgoing_trans l s : set of transitions leaving s

```
definition
  accepted_ev
where
  "accepted_events l s ≡ lbl ` (outgoing_trans l s)"
```

Given

- a LTS l and
- a state s,

accepted_events l s : set of events for transitions leaving s

Simulation and preconditions

```
definition simulation_B :: "'st rel ⇒ ('st, 'ev) LTS rel"
where
  "s
    ∧ (∀s s . (s, s ) ∈ r ⟶
          accepted_events l s ⊇ accepted_events l' s' ∧
          (∀t ∈ outgoing_trans l s.
              lbl t ∈ accepted_events l' s' ⟶
              (∃t' ∈ outgoing_trans l' s'.
                   src t' = s' ∧ lbl t' = lbl t ∧
                   (dst t, dst t') ∈ r))) }"
```

- lifting the notion of simulation between states to simulation between LTS…
- restricted to events accepted by simulating LTS.

```
definition simulated_B (infixl "≼B" 50)
  where "l ≼B l' ≡ ∃r. (l,l') ∈ simulation_B r"
```

Properties of simulation

```
lemma simulation_B_composition:
  assumes "(l, l') ∈ simulation_B r”
      and "(l', l'') ∈ simulation_B r'"
    shows "(l, l'') ∈ simulation_B (r O r')"
```

```
lemma simulates_B_transitive:
  assumes "l ≼B l'" and "l' ≼B l''"
  shows   "l ≼B l''"
```

Accepted events after a run

```
definition run_accepted_events ::
  "('st, 'ev) LTS ⇒ ('st, 'ev) Run ⇒ 'ev set"
where
"run_accepted_events l r ≡
   if r = [] then UNION (init l) (accepted_events l)
   else accepted_events l (dst (last r))"
```

B Trac

trace of observed events

accepted events

```
type_synonym 'ev TrB = "'ev list * 'ev set"
```

```
definition run_trace ::
  "('st, 'ev) LTS ⇒ ('st, 'ev) Run ⇒ 'ev TrB"
where
  "run_trace l r ≡ (map lbl r, run_accepted_events l r)"
```

```
definition traces_B ::
  "('st, 'ev) LTS ⇒ 'ev TrB set"
where
  "traces_B l ≡ (run_trace l) ` (runs l)"
```

Simulation and traces for B

```
lemma sim_traces_B:
  assumes "l ≼B l'"
      and "(tr, acc) ∈ traces_B l"
    shows "∃ (tr', acc') ∈ traces_B l' .
           acc ⊇ acc' ∧
           (tr = tr' ∨
            prefix tr' tr ∧ (∃ d ∈ acc'. d ∉ acc ∧
                                 prefixeq (tr' @ [d]) tr))"
```

B development

- Only change: substituted $\preccurlyeq$ by $\preccurlyeq\mathbf{B}$
- All results apply

Conclusion

- Semantic model for the behavioural aspects of component in the B method.
- Formalised in Isabelle/HOL.
- Two versions

Outlook

- Investigate other modelling approaches
    - include attribute “alphabet” of events in LTS
- Formalise derivation of semantic structure from syntactic structure
- Formalise refactoring and refinement rules

Thanks for your attention!

Questions?