Discourje: Runtime Verification of Communication Protocols in Clojure [TACAS'20]

Ruben Hamers and Sung-Shik Jongmans

Open Univ. Netherlands || CWI, Amsterdam

Long-term **research agenda:**

Development of theoretical **foundations** and practical **tools** to help programmers use concurrency

Long-term **research agenda:**

Development of theoretical **foundations** and practical **tools** to help programmers use shared-memory concurrency, with channels

Suppose that we have a **specification** S of...

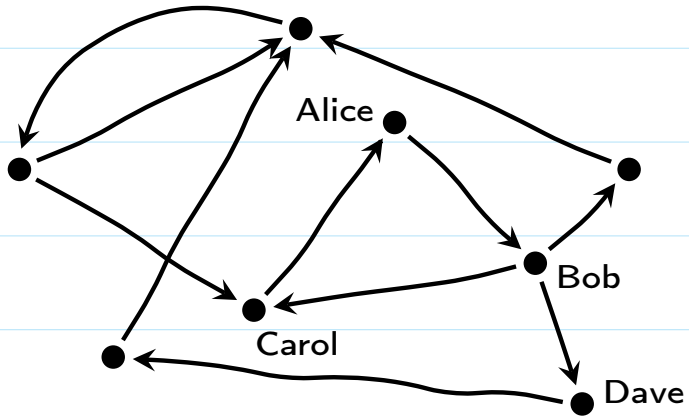
1. ??? **2.** ??? **3.** ???

...that **implementation** I should consist of

Suppose that we have a **specification** S of...

1. threads
2. ???
3. ???

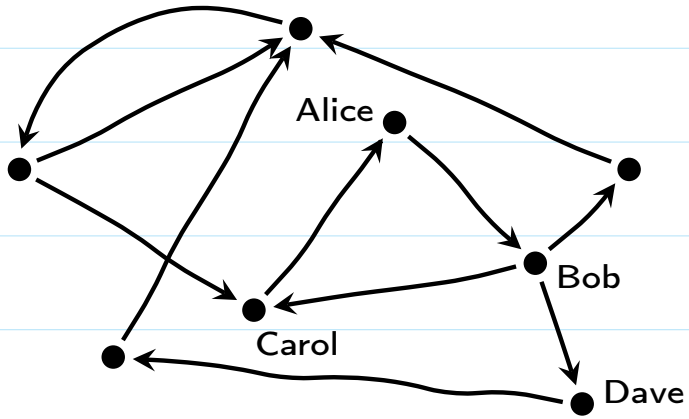
...that **implementation** I should consist of



Suppose that we have a **specification** S of...

1. threads **2.** channels **3.** ???

...that **implementation** I should consist of



First, a number from Alice to Bob.

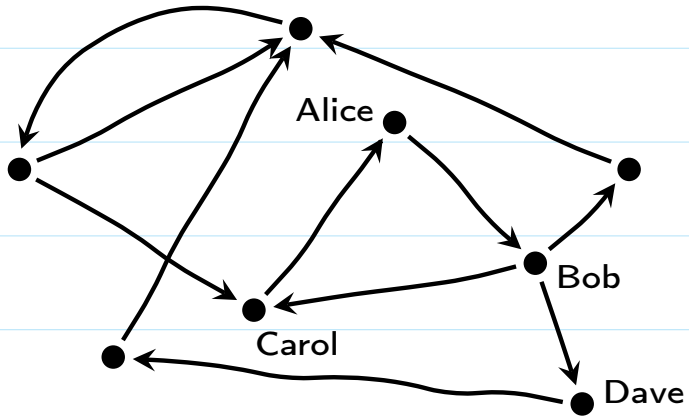
Then, a number from Bob to either Carol or Dave.

Then, ...

Suppose that we have a **specification** S of...

1. threads 2. channels 3. protocols (communications)

...that **implementation** I should consist of



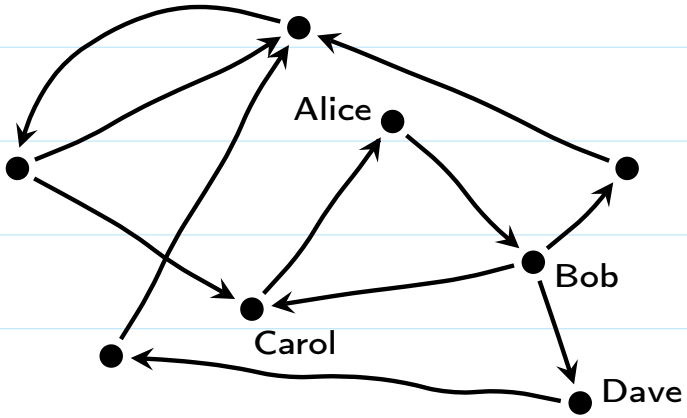
First, a number from Alice to Bob.

Then, a number from Bob to either Carol or Dave.

Then, ...

How to ensure that I is safe and live relative to S?

("bad"/"good" channel actions never/eventually happen)



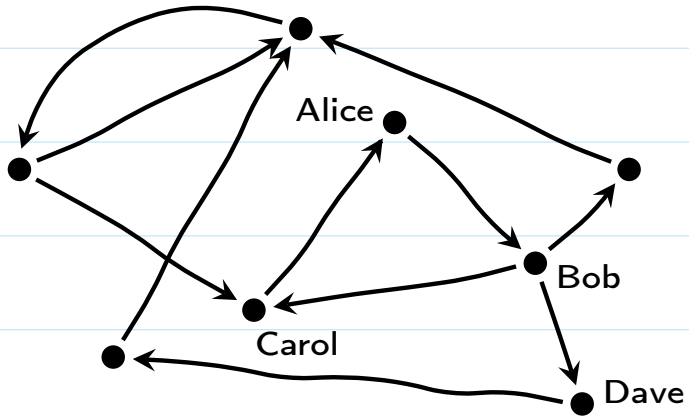
First, a number from Alice to Bob.

Then, a number from Bob to either Carol or Dave.

~~**Then,**~~ ...

$$S = ???$$

$$I_1 = ???$$



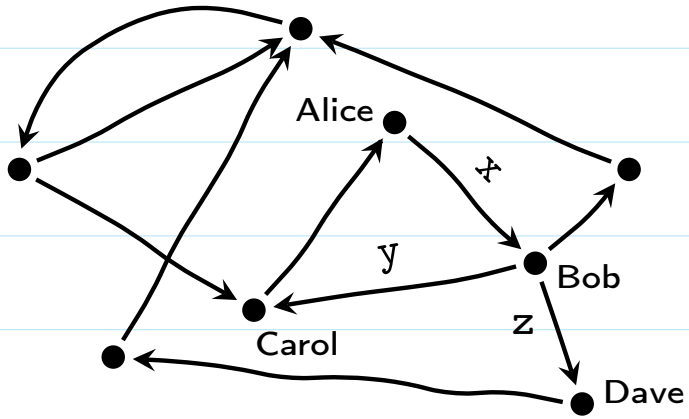
First, a number from Alice to Bob.

Then, a number from Bob to either Carol or Dave.

~~**Then,**~~ ...

$$S = \mathtt{A} \rightarrowtriangle \mathtt{B}:\mathtt{Nat} \cdot (\mathtt{B} \rightarrowtriangle \mathtt{C}:\mathtt{Nat} + \mathtt{B} \rightarrowtriangle \mathtt{D}:\mathtt{Nat})$$

$$I_1 = ???$$



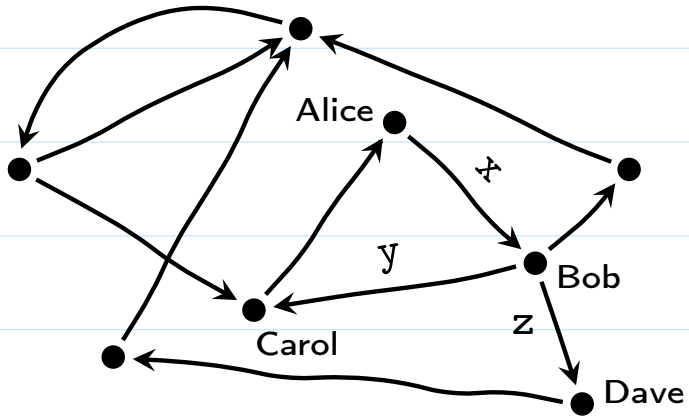
First, a number from Alice to Bob.

Then, a number from Bob to either Carol or Dave.

~~**Then,**~~

$$S = \mathtt{A} \rightarrowtriangle \mathtt{B}:\mathtt{Nat} \cdot (\mathtt{B} \rightarrowtriangle \mathtt{C}:\mathtt{Nat} + \mathtt{B} \rightarrowtriangle \mathtt{D}:\mathtt{Nat})$$

$$I_1 = (\textbf{let}\ \mathtt{x}\ (\textbf{chan}\ 1)\ (\textbf{let}\ \mathtt{y}\ (\textbf{chan}\ 1)\ (\textbf{let}\ \mathtt{z}\ (\textbf{chan}\ 1)\ ($$



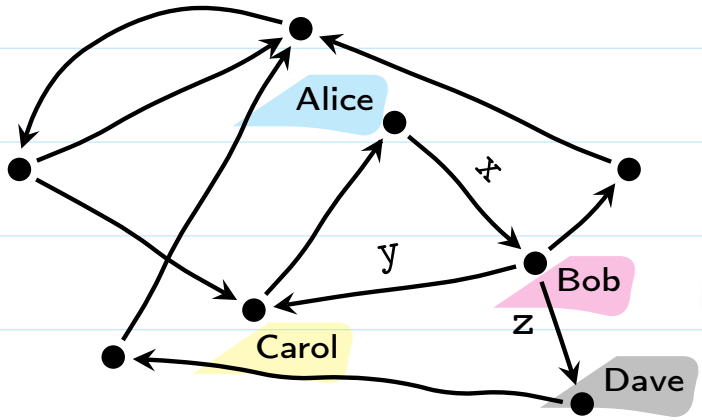
First, a number from Alice to Bob.

Then, a number from Bob to either Carol or Dave.

~~**Then,**~~ ...

$$S = \mathtt{A} \rightarrowtriangle \mathtt{B} : \mathtt{Nat} \cdot (\mathtt{B} \rightarrowtriangle \mathtt{C} : \mathtt{Nat} + \mathtt{B} \rightarrowtriangle \mathtt{D} : \mathtt{Nat})$$

$$I_1 = (\mathbf{let}\ \mathtt{x}\ (\mathbf{chan}\ 1)\ (\mathbf{let}\ \mathtt{y}\ (\mathbf{chan}\ 1)\ (\mathbf{let}\ \mathtt{z}\ (\mathbf{chan}\ 1)\ (\\ (\mathbf{send}\ \mathtt{x}\ 5) \parallel (\mathbf{recv}\ \mathtt{y}) \parallel \mathbf{nil} \parallel \\ (\mathbf{if}\ (\mathtt{odd?}\ (\mathbf{recv}\ \mathtt{x}))\ (\mathbf{send}\ \mathtt{y}\ 6)\ (\mathbf{send}\ \mathtt{z}\ \mathtt{true})))))$$



First, a number from Alice to Bob.

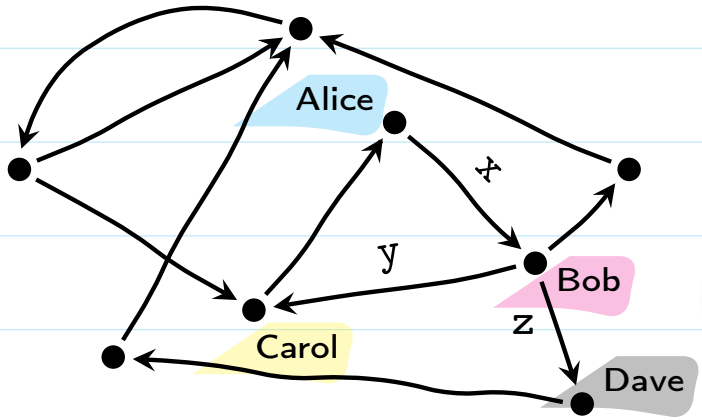
Then, a number from Bob to either Carol or Dave.

~~**Then,** ...~~

$$S = \mathtt{A} \rightarrowtriangle \mathtt{B} : \mathtt{Nat} \cdot (\mathtt{B} \rightarrowtriangle \mathtt{C} : \mathtt{Nat} + \mathtt{B} \rightarrowtriangle \mathtt{D} : \mathtt{Nat})$$

$$I_1 = (\mathbf{let}\ \mathtt{x}\ (\mathbf{chan}\ 1)\ (\mathbf{let}\ \mathtt{y}\ (\mathbf{chan}\ 1)\ (\mathbf{let}\ \mathtt{z}\ (\mathbf{chan}\ 1)\ (\\ (\mathbf{send}\ \mathtt{x}\ 5) \| (\mathbf{recv}\ \mathtt{y}) \| \mathbf{nil} \| \\ (\mathbf{if}\ (\mathtt{odd?}\ (\mathbf{recv}\ \mathtt{x}))\ (\mathbf{send}\ \mathtt{y}\ 6)\ (\mathbf{send}\ \mathtt{z}\ \mathtt{true}))))))$$

	safe	live
I_1	✓	✓



First, a number from Alice to Bob.

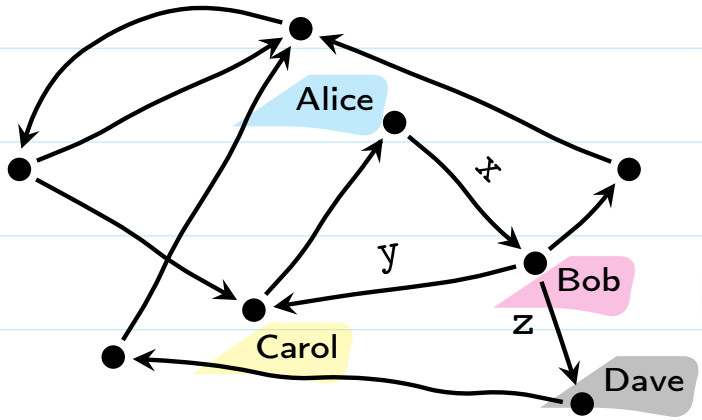
Then, a number from Bob to either Carol or Dave.

~~**Then,** ...~~

$$S = \mathtt{A} \rightarrowtriangle \mathtt{B} : \mathtt{Nat} \cdot (\mathtt{B} \rightarrowtriangle \mathtt{C} : \mathtt{Nat} + \mathtt{B} \rightarrowtriangle \mathtt{D} : \mathtt{Nat})$$

$$I_1 = (\mathbf{let}\ \mathtt{x}\ (\mathbf{chan}\ 1)\ (\mathbf{let}\ \mathtt{y}\ (\mathbf{chan}\ 1)\ (\mathbf{let}\ \mathtt{z}\ (\mathbf{chan}\ 1)\ (\\ (\mathbf{send}\ \mathtt{x}\ 5) \| (\mathbf{recv}\ \mathtt{y}) \| \mathbf{nil} \| \\ (\mathbf{if}\ (\mathtt{odd?}\ (\mathbf{recv}\ \mathtt{x}))\ (\mathbf{send}\ \mathtt{y}\ 6)\ (\mathbf{send}\ \mathtt{z}\ \mathtt{true}))))))$$

	safe	live
I_1	✓	✓
I_2	✓	–



First, a number from Alice to Bob.

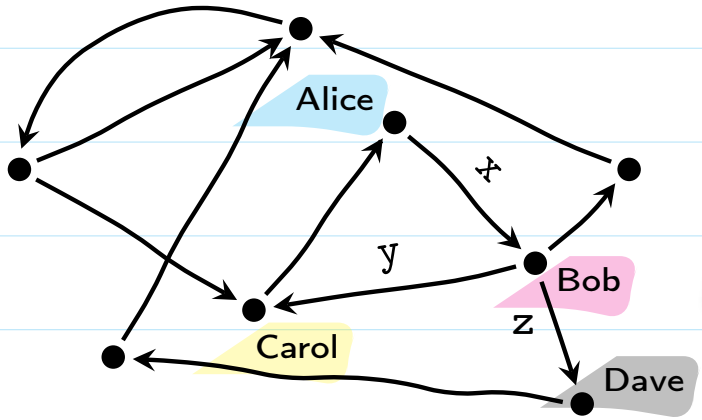
Then, a number from Bob to either Carol or Dave.

~~**Then,** ...~~

$$S = \mathtt{A} \rightarrowtriangle \mathtt{B} : \mathtt{Nat} \cdot (\mathtt{B} \rightarrowtriangle \mathtt{C} : \mathtt{Nat} + \mathtt{B} \rightarrowtriangle \mathtt{D} : \mathtt{Nat})$$

$$I_2 = (\textbf{let}\ \mathtt{x}\ (\textbf{chan}\ 1)\ (\textbf{let}\ \mathtt{y}\ (\textbf{chan}\ 1)\ (\textbf{let}\ \mathtt{z}\ (\textbf{chan}\ 1)\ (\\ (\textbf{send}\ \mathtt{x}\ 5) \parallel (\textbf{recv}\ \mathtt{y}) \parallel (\textbf{recv}\ \mathtt{z}) \parallel \\ (\textbf{if}\ (\mathtt{odd?}\ (\textbf{recv}\ \mathtt{x}))\ (\textbf{send}\ \mathtt{y}\ 6)\ (\textbf{send}\ \mathtt{z}\ \mathtt{true}))))))$$

	safe	live
I_1	✓	✓
I_2	✓	–
I_3	–	✓



First, a number from Alice to Bob.

Then, a number from Bob to either Carol or Dave.

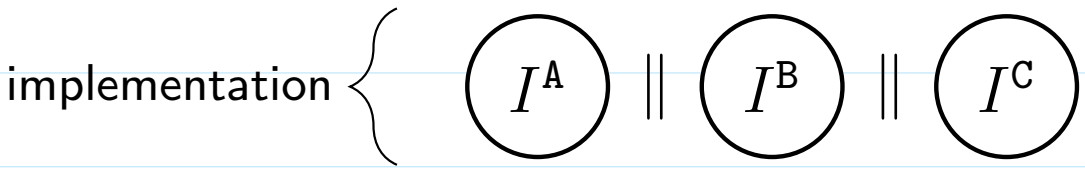
~~Then, ...~~

$$S = \mathtt{A} \rightarrowtriangle \mathtt{B}:\mathtt{Nat} \cdot (\mathtt{B} \rightarrowtriangle \mathtt{C}:\mathtt{Nat} + \mathtt{B} \rightarrowtriangle \mathtt{D}:\mathtt{Nat})$$

$$I_3 = (\mathbf{let}\ \mathtt{x}\ (\mathbf{chan}\ 1)\ (\mathbf{let}\ \mathtt{y}\ (\mathbf{chan}\ 1)\ (\mathbf{let}\ \mathtt{z}\ (\mathbf{chan}\ 1)\ (\\ (\mathbf{send}\ \mathtt{x}\ 4) \| \mathbf{nil} \| (\mathbf{recv}\ \mathtt{z}) \| \\ (\mathbf{if}\ (\mathtt{odd?}\ (\mathbf{recv}\ \mathtt{x}))\ (\mathbf{send}\ \mathtt{y}\ 6)\ (\mathbf{send}\ \mathtt{z}\ \mathtt{true}))))))$$

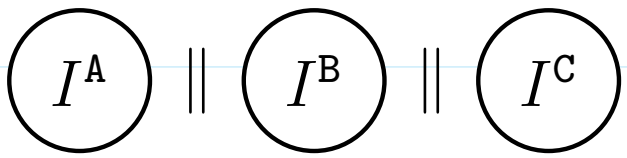
Influential approach to ensure safety/liveness: [Honda et al., POPL'08]

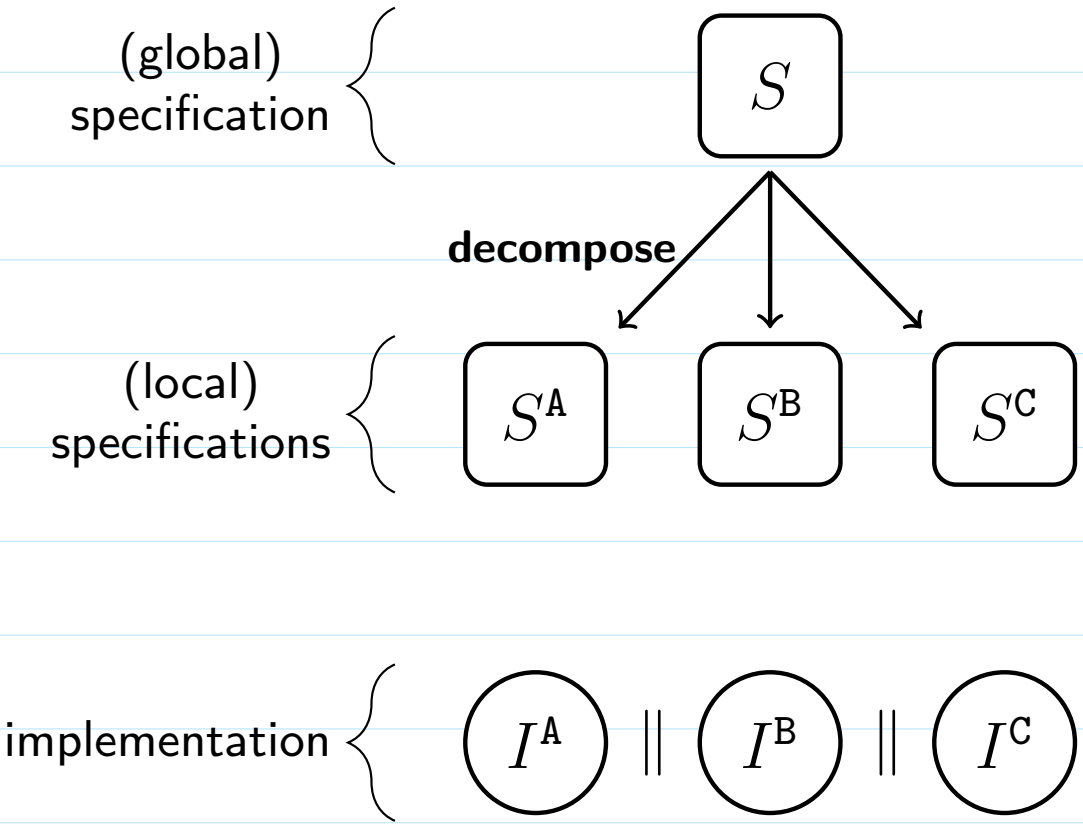
Multiparty session types


implementation
$I^{\mathtt{A}}$
$I^{\mathtt{B}}$
$I^{\mathtt{C}}$

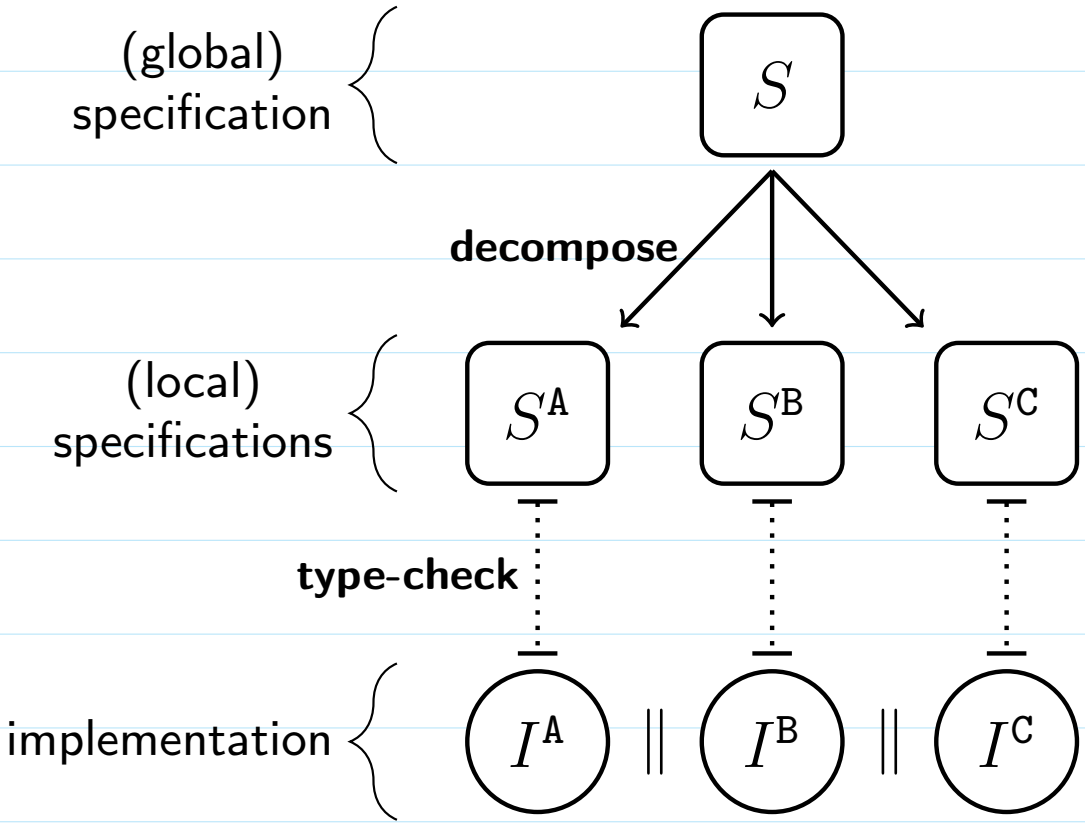
(global) specification

implementation





$$S^{\mathtt{A}} = \mathtt{B}!\mathtt{Nat}$$
$$S^{\mathtt{B}} = \mathtt{A}?\mathtt{Nat} \,.\, (\mathtt{C}!\mathtt{Nat} + \mathtt{D}!\mathtt{Nat})$$
$$S^{\mathtt{C}} = \mathtt{B}?\mathtt{Nat} + 1$$



$S^{\mathtt{A}} = \mathtt{B}!\mathtt{Nat}$

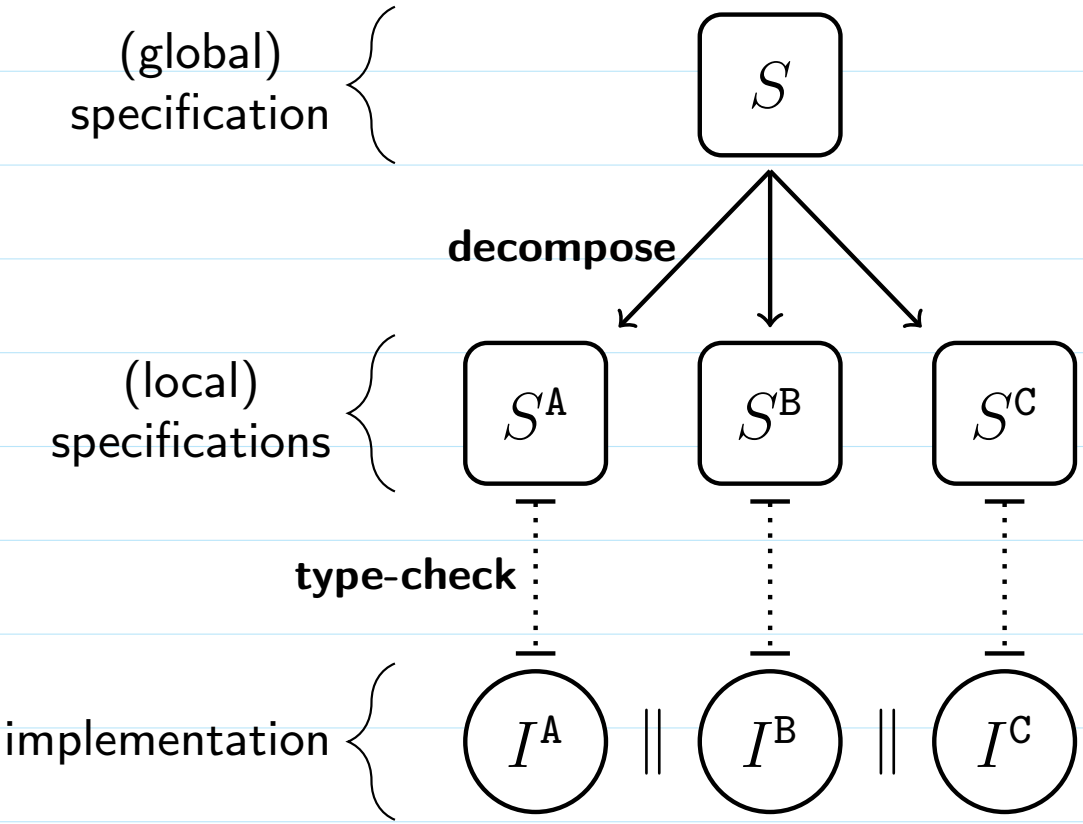
$S^{\mathtt{B}} = \mathtt{A}?\mathtt{Nat} \,.\, (\mathtt{C}!\mathtt{Nat} + \mathtt{D}!\mathtt{Nat})$

$S^{\mathtt{C}} = \mathtt{B}?\mathtt{Nat} + 1$

$\Gamma \vdash (\textbf{send}\ \mathtt{x}\ 5) : S^{\mathtt{A}}$

$\Gamma \vdash (\textbf{if}\ \ldots) : S^{\mathtt{B}}$

$\Gamma \vdash (\textbf{recv}\ \mathtt{y}) : S^{\mathtt{C}}$



$S^{\mathtt{A}} = \mathtt{B}!\mathtt{Nat}$

$S^{\mathtt{B}} = \mathtt{A}?\mathtt{Nat} \,.\, (\mathtt{C}!\mathtt{Nat} + \mathtt{D}!\mathtt{Nat})$

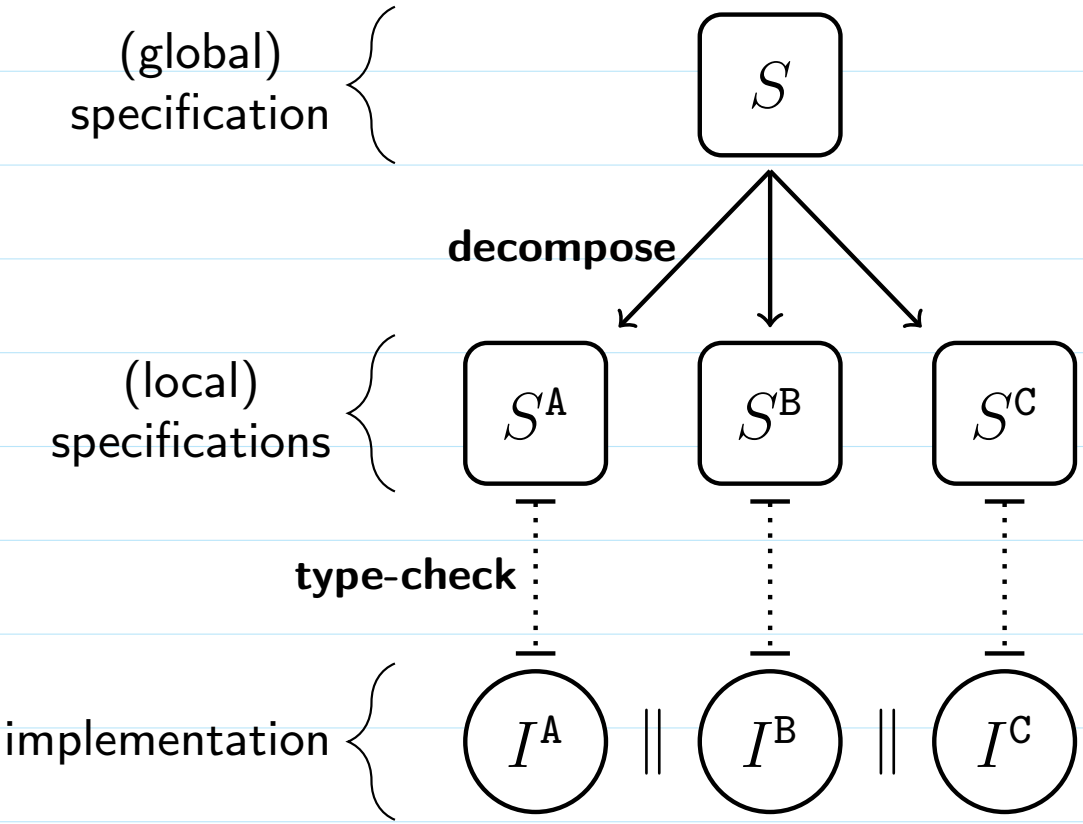
$S^{\mathtt{C}} = \mathtt{B}?\mathtt{Nat} + 1$

$\Gamma \vdash (\mathbf{send}\ \mathtt{x}\ 5) : S^{\mathtt{A}}$

$\Gamma \vdash (\mathbf{if}\ \ldots) : S^{\mathtt{B}}$

$\Gamma \vdash (\mathbf{recv}\ \mathtt{y}) : S^{\mathtt{C}}$

well-typedness $\Rightarrow$ safety $\wedge$ liveness



$S^{\mathtt{A}} = \mathtt{B}!\mathtt{Nat}$

$S^{\mathtt{B}} = \mathtt{A}?\mathtt{Nat} \,.\, (\mathtt{C}!\mathtt{Nat} + \mathtt{D}!\mathtt{Nat})$

$S^{\mathtt{C}} = \mathtt{B}?\mathtt{Nat} + 1$

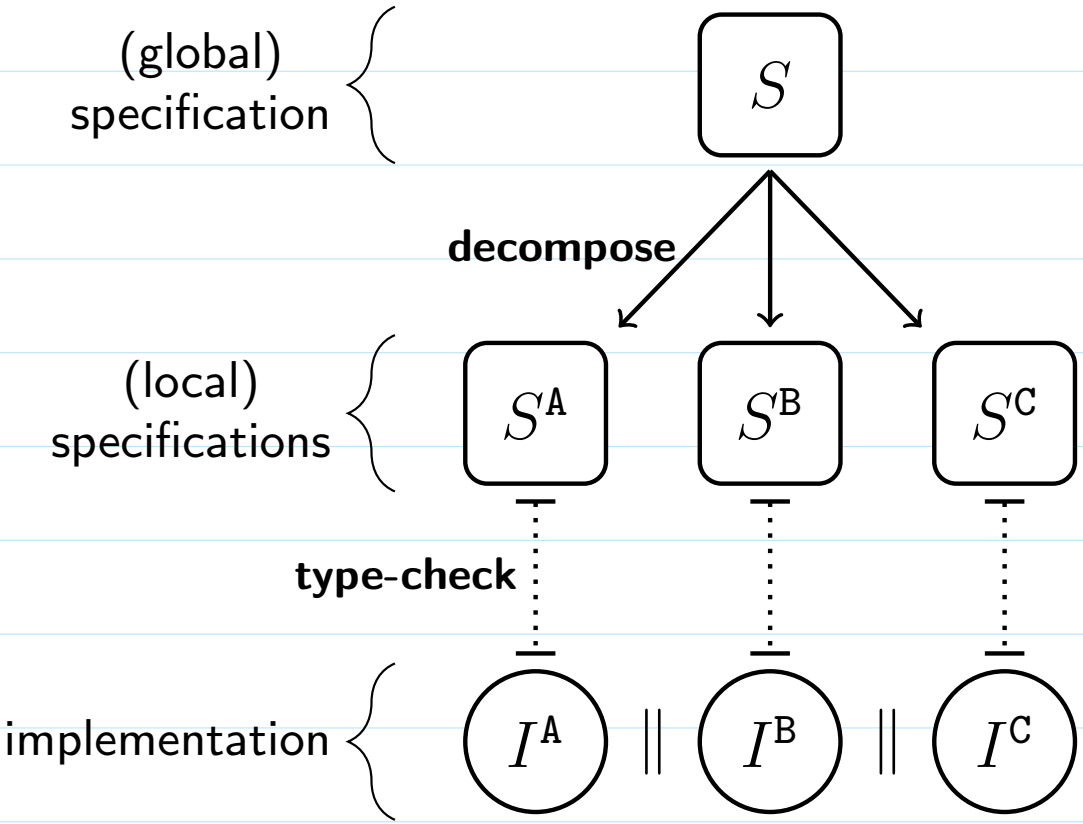
$\Gamma \vdash (\textbf{send}\ \mathtt{x}\ 5) : S^{\mathtt{A}}$

$\Gamma \vdash (\textbf{if}\ \ldots) : S^{\mathtt{B}}$

$\Gamma \vdash (\textbf{recv}\ \mathtt{y}) : S^{\mathtt{C}}$

well-typedness $\Rightarrow$ safety $\wedge$ liveness

(But, expressiveness is an issue)



$S^{\mathtt{A}} = \mathtt{B}!\mathtt{Nat}$

$S^{\mathtt{B}} = \mathtt{A}?\mathtt{Nat} \,.\, (\mathtt{C}!\mathtt{Nat} + \mathtt{D}!\mathtt{Nat})$

$S^{\mathtt{C}} = \mathtt{B}?\mathtt{Nat} + \mathbf{1}$ ~~$\mathbf{1}$~~ $\bot$

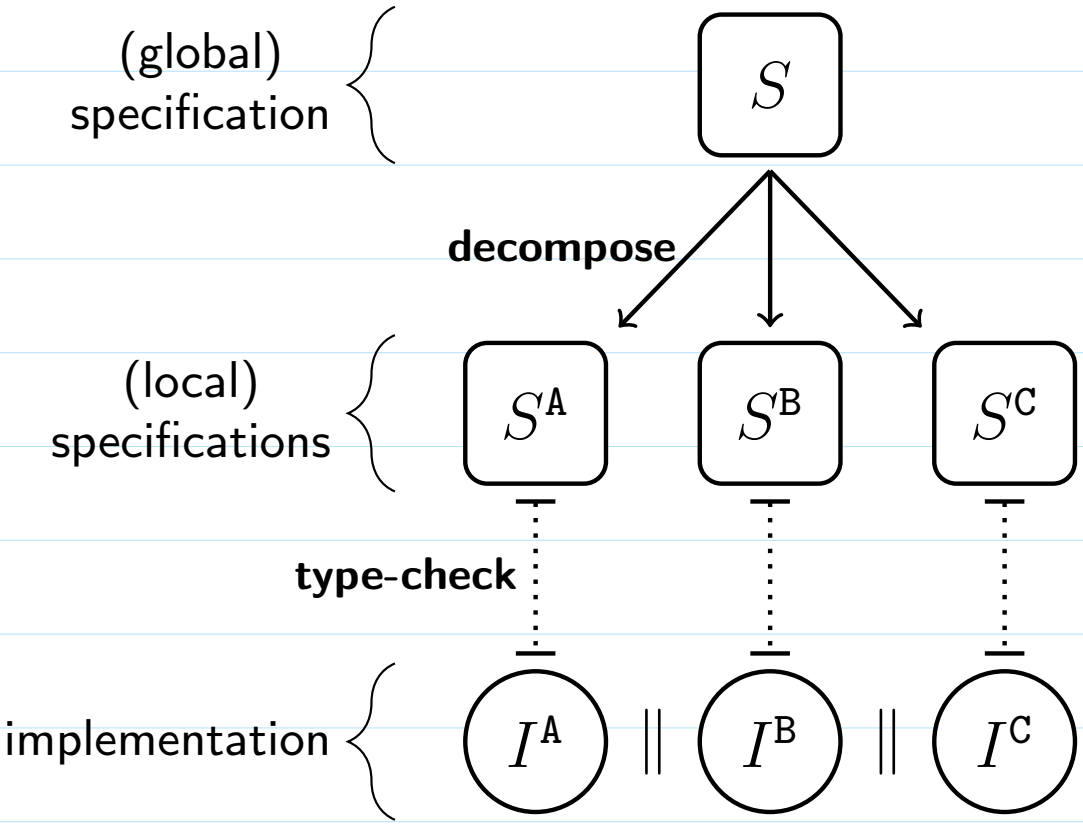
$\Gamma \vdash (\mathbf{send}\ \mathtt{x}\ 5) : S^{\mathtt{A}}$

$\Gamma \vdash (\mathbf{if}\ \ldots) : S^{\mathtt{B}}$

$\Gamma \vdash (\mathbf{recv}\ \mathtt{y}) : S^{\mathtt{C}}$

well-typedness $\Rightarrow$ safety $\wedge$ liveness

(But, expressiveness is an issue)



Many **specifications** cannot be decomposed (modulo $\equiv$)

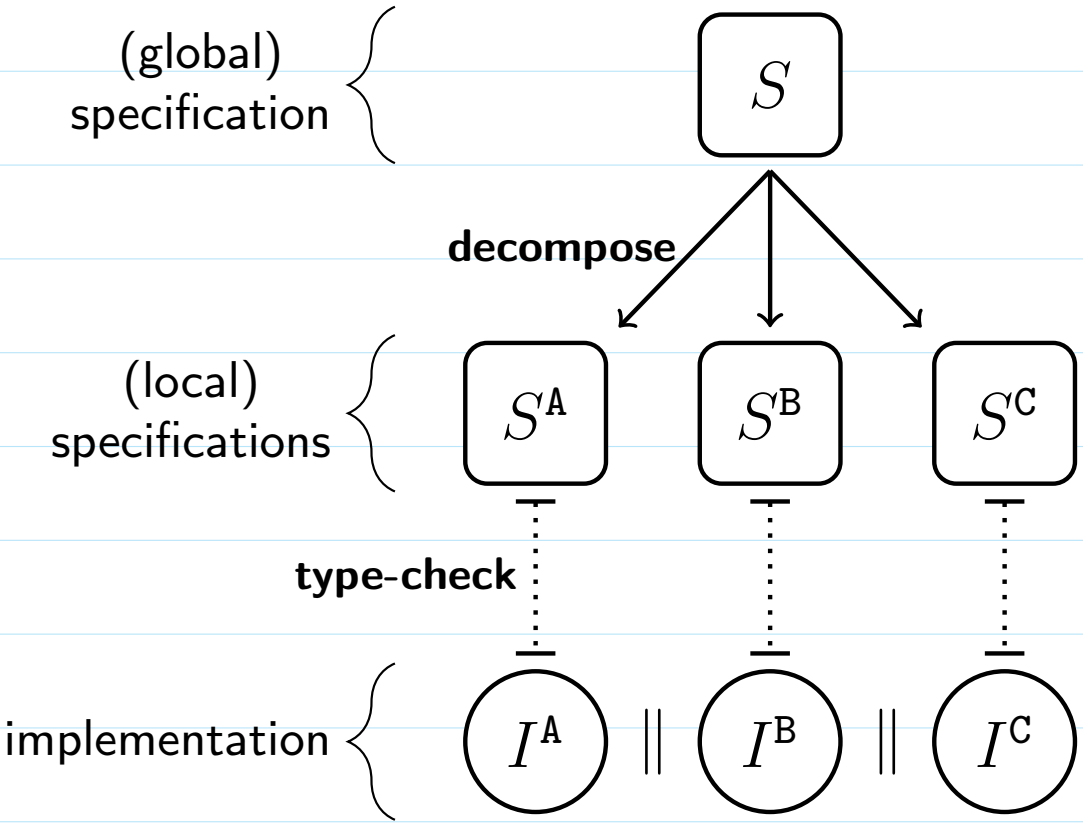
$\Gamma \vdash (\mathbf{send}\ \mathtt{x}\ 5) : S^{\mathtt{A}}$

$\Gamma \vdash (\mathbf{if}\ \ldots) : S^{\mathtt{B}}$

$\Gamma \vdash (\mathbf{recv}\ \mathtt{y}) : S^{\mathtt{C}}$

well-typedness $\Rightarrow$ safety $\wedge$ liveness

(But, expressiveness is an issue)



Many **specifications** cannot be decomposed (modulo $\equiv$)

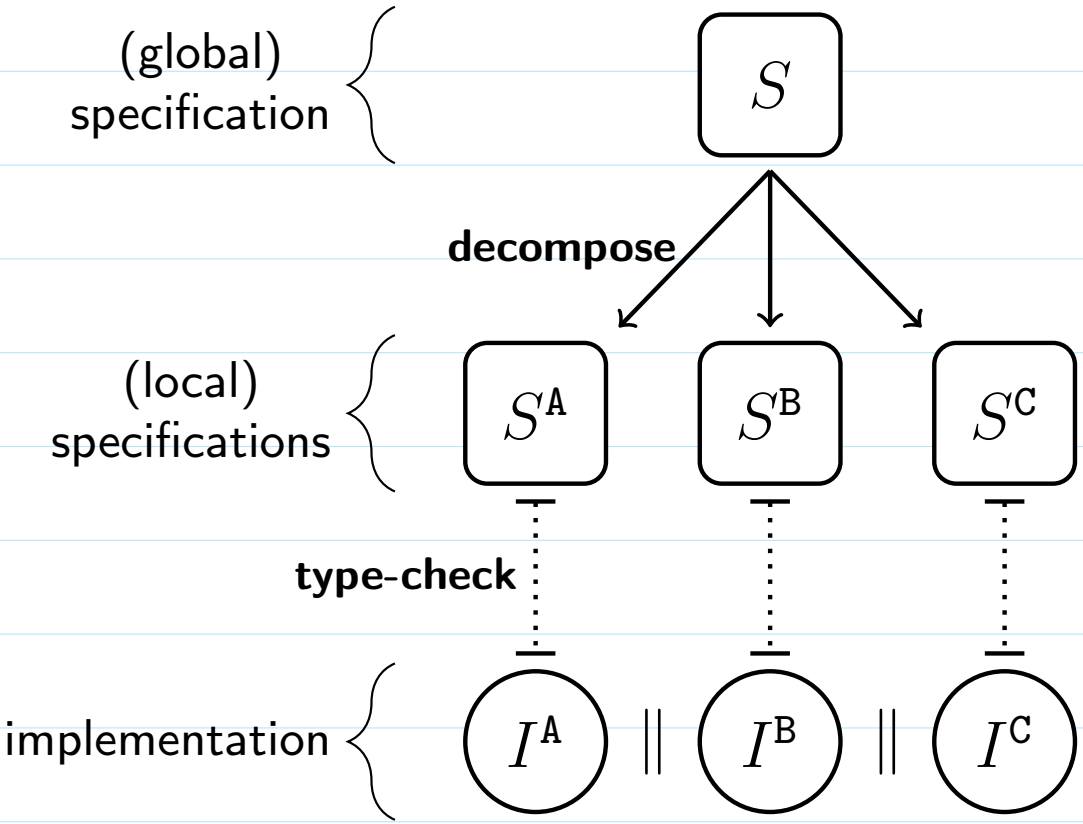
$\Gamma \vdash (\mathbf{send}\ \mathtt{x}\ 5) : S^{\mathtt{A}}$

$\Gamma \not\vdash (\mathbf{if}\ \ldots) : S^{\mathtt{B}}$

$\Gamma \vdash (\mathbf{recv}\ \mathtt{y}) : S^{\mathtt{C}}$

well-typedness $\Rightarrow$ safety $\wedge$ liveness

(But, expressiveness is an issue)

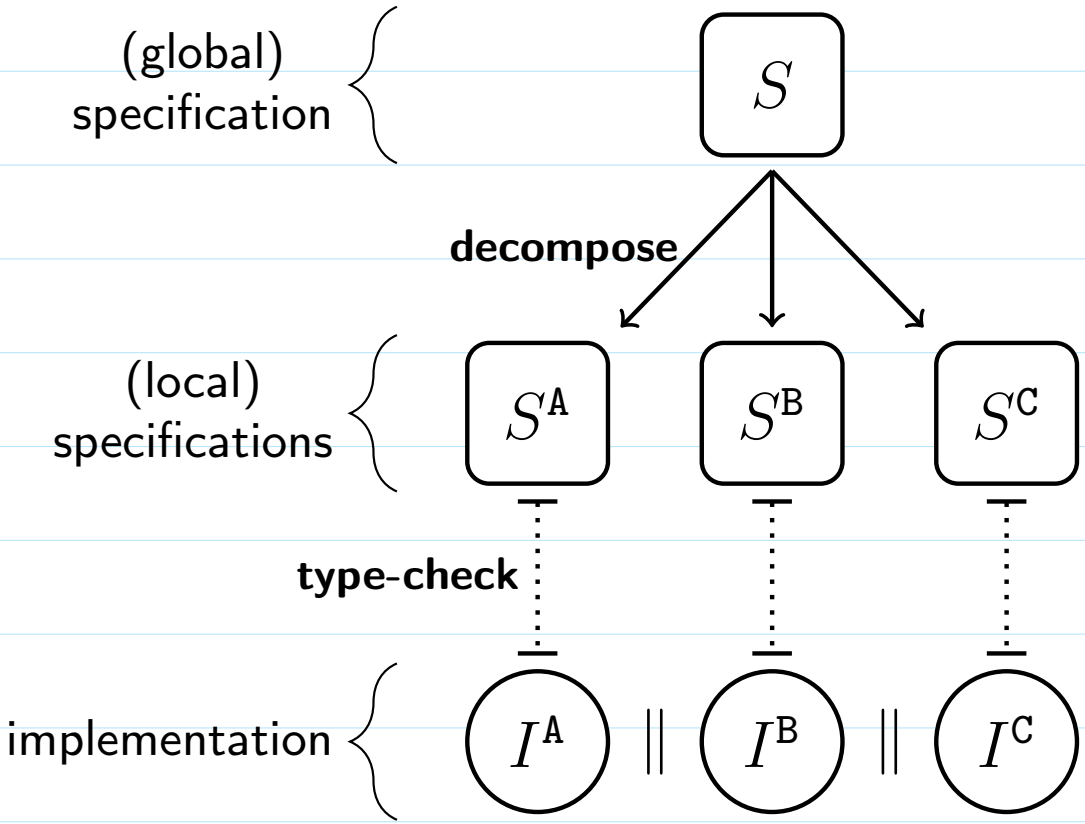


Many **specifications** cannot be decomposed (modulo $\equiv$)

Many **implementations** cannot be type-checked (conservative)

well-typedness $\Rightarrow$ safety $\wedge$ liveness

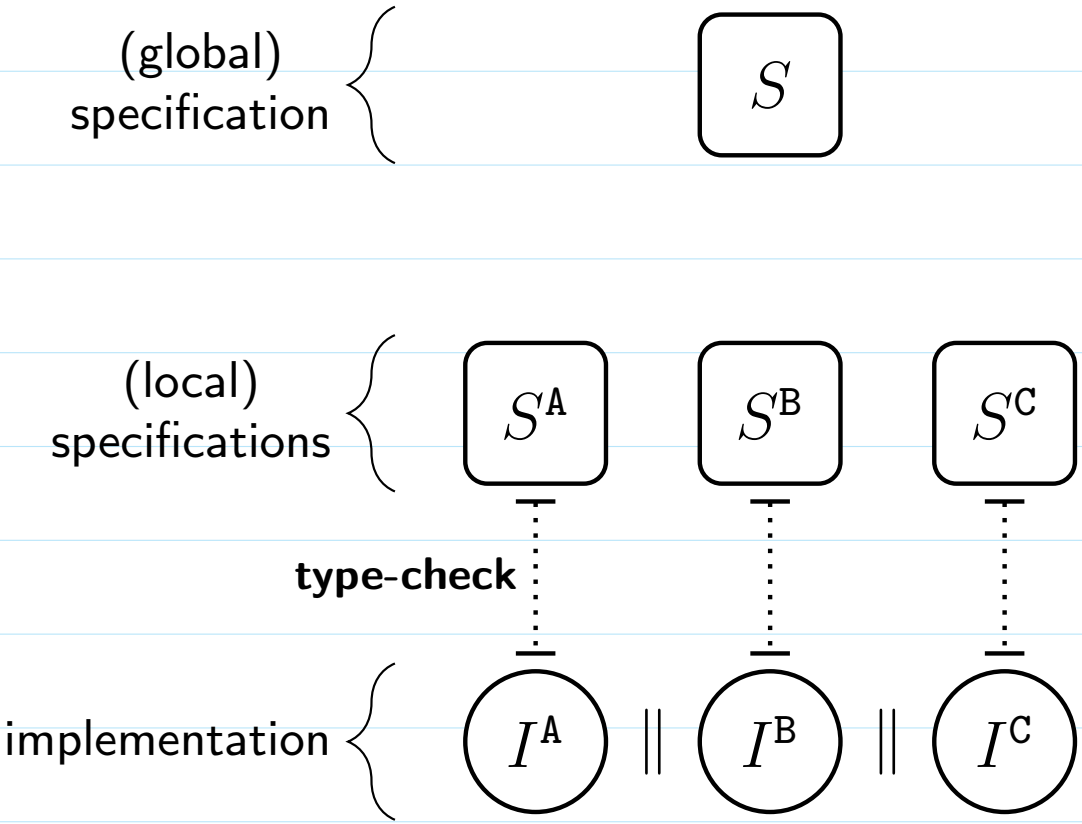
(But, expressiveness is an issue)



Many **specifications** cannot be decomposed (modulo $\equiv$)

Many **implementations** cannot be type-checked (conservative)

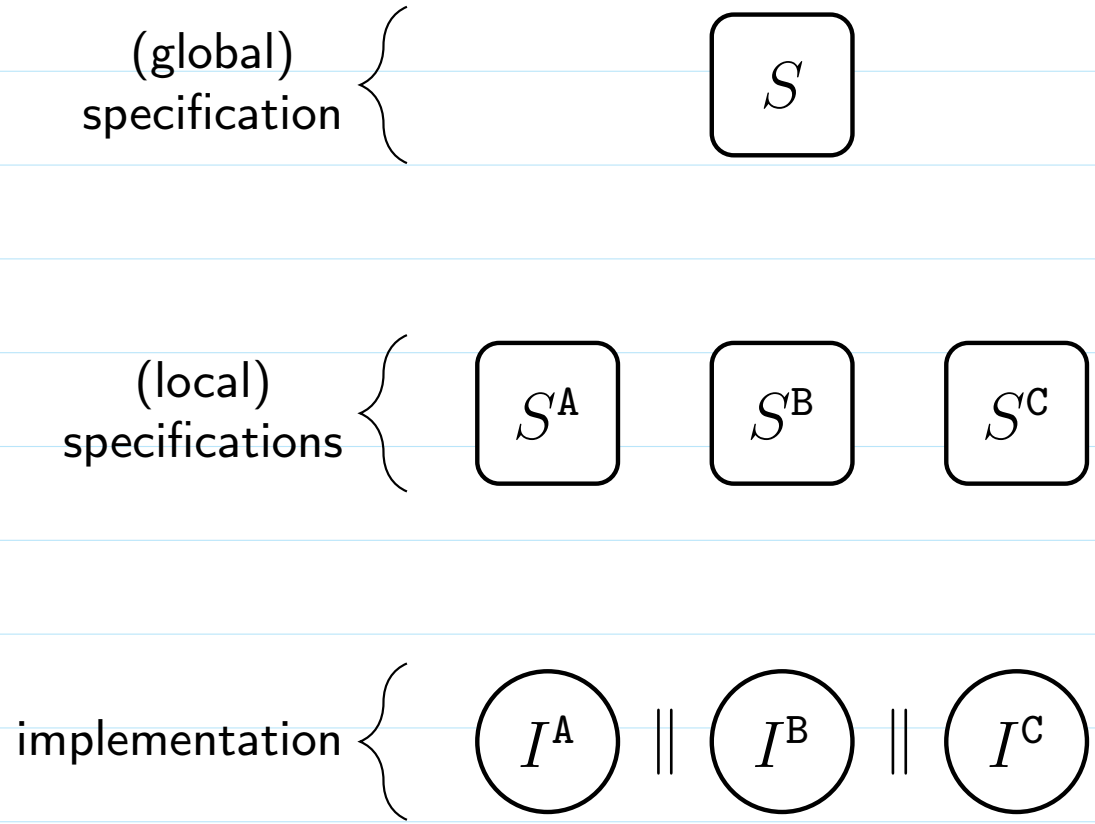
This work: Improve expressiveness



~~Many specifications cannot be decomposed (modulo =)~~

Many **implementations** cannot be type-checked (conservative)

This work: Improve expressiveness



~~Many specifications cannot be decomposed (modulo =)~~

~~Many implementations cannot be type-checked (conservative)~~

This work: Improve expressiveness

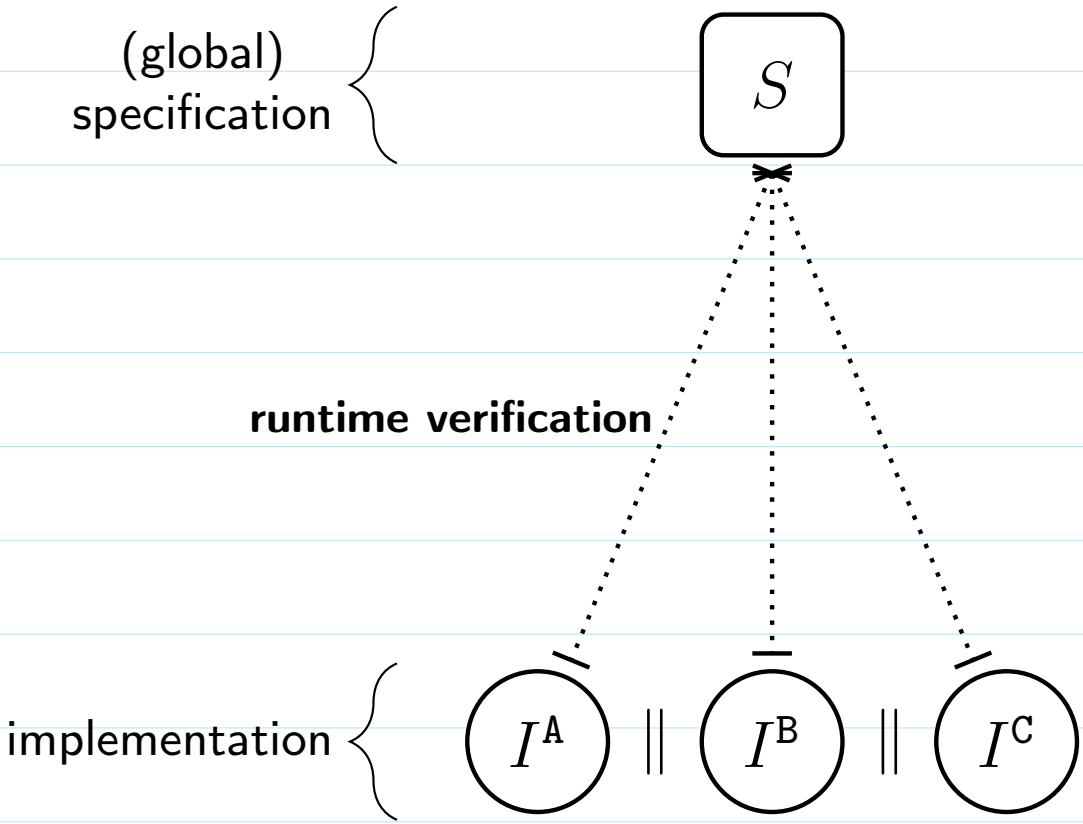
(global) specification: S

~~Many specifications cannot be decomposed (modulo =)~~

implementation: $I^{\mathtt{A}} \parallel I^{\mathtt{B}} \parallel I^{\mathtt{C}}$

~~Many implementations cannot be type-checked (conservative)~~

This work: Improve expressiveness

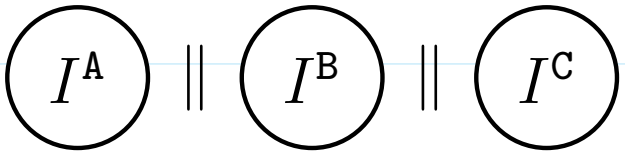


~~Many specifications cannot be decomposed (modulo =)~~

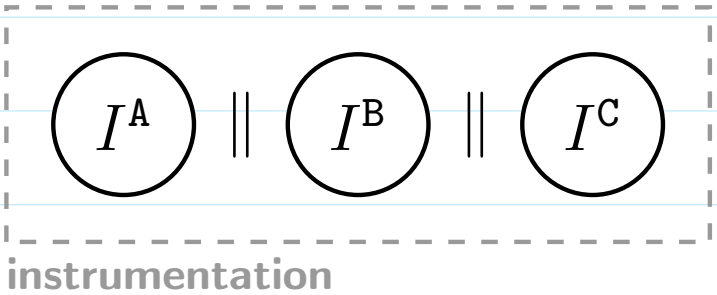
~~Many implementations cannot be type-checked (conservative)~~

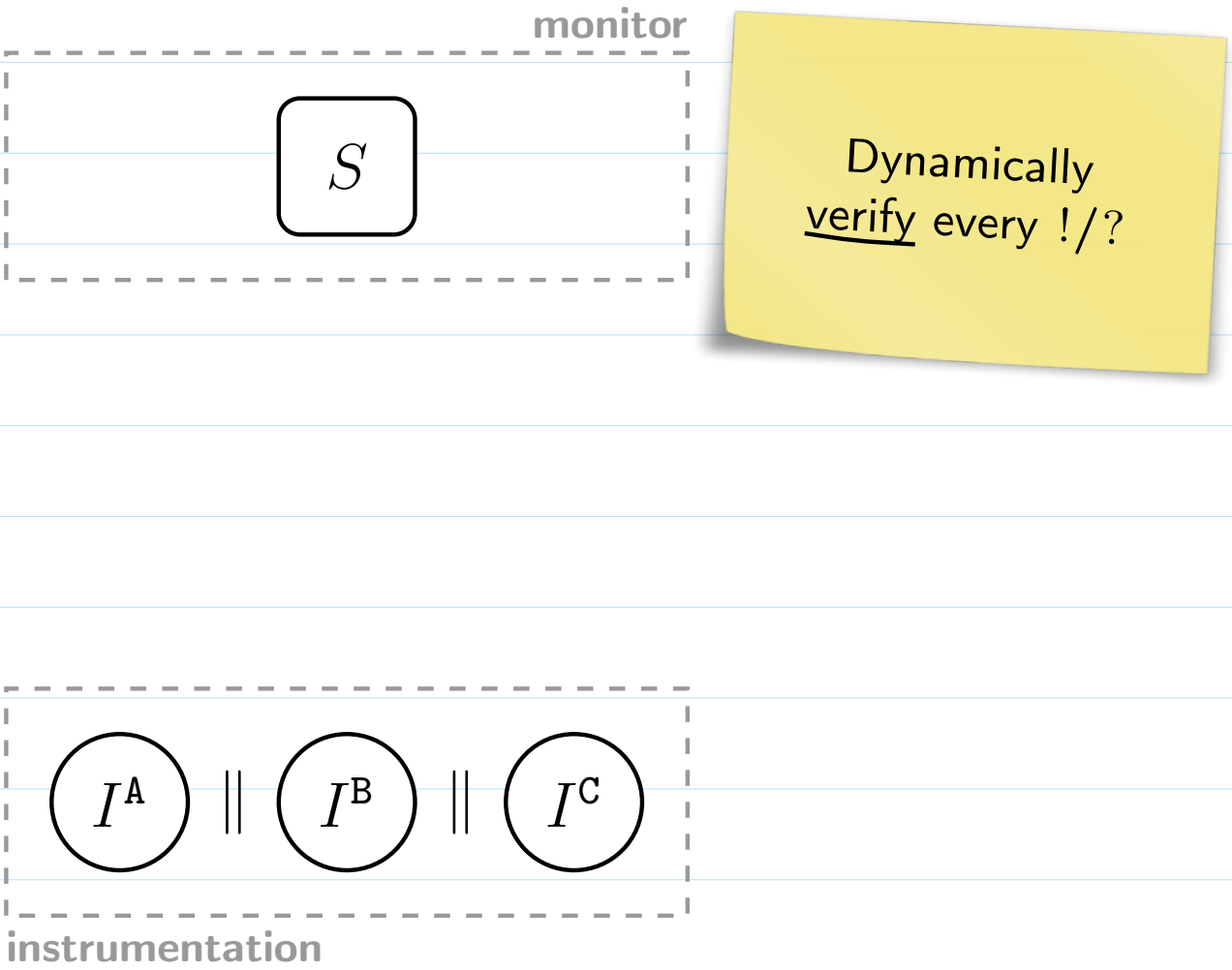
This work: Improve expressiveness

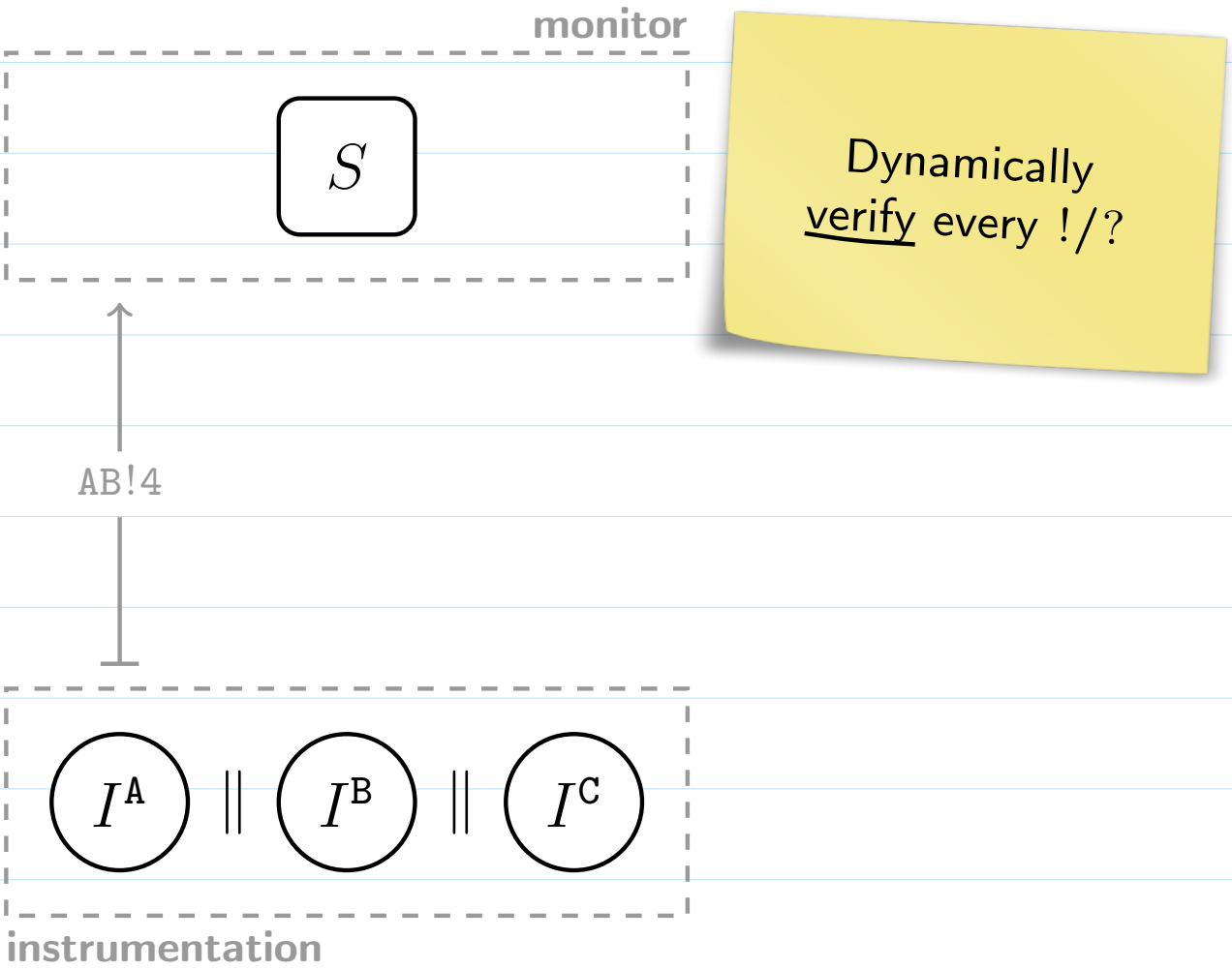

S

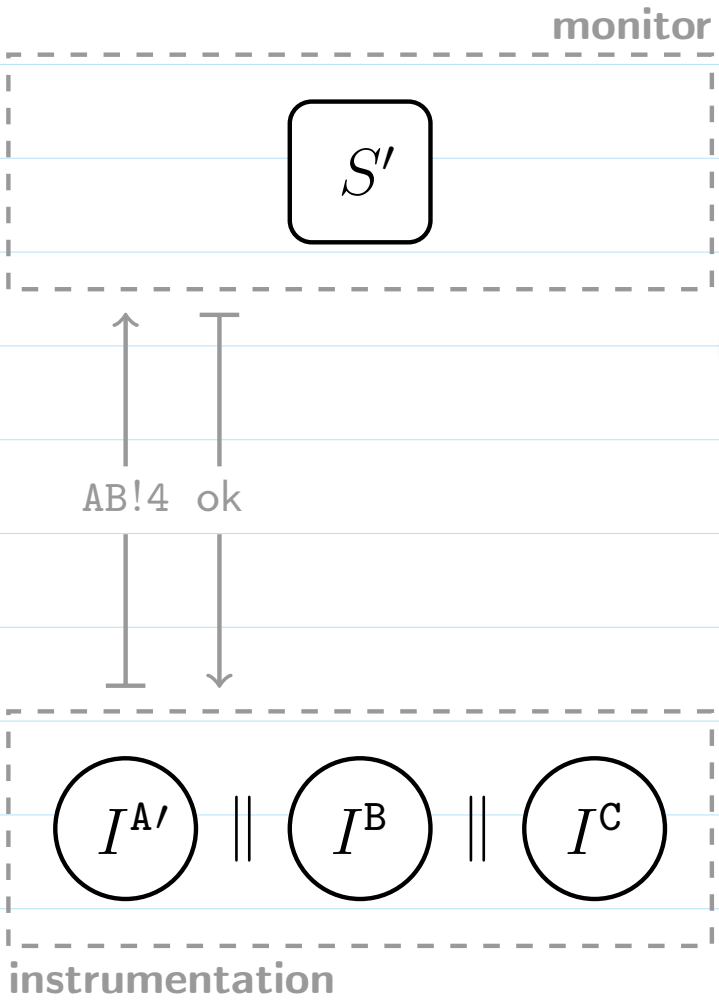

$I^{\mathtt{A}}$
$\|$
$I^{\mathtt{B}}$
$\|$
$I^{\mathtt{C}}$

Dynamically observe every **send**/**recv**



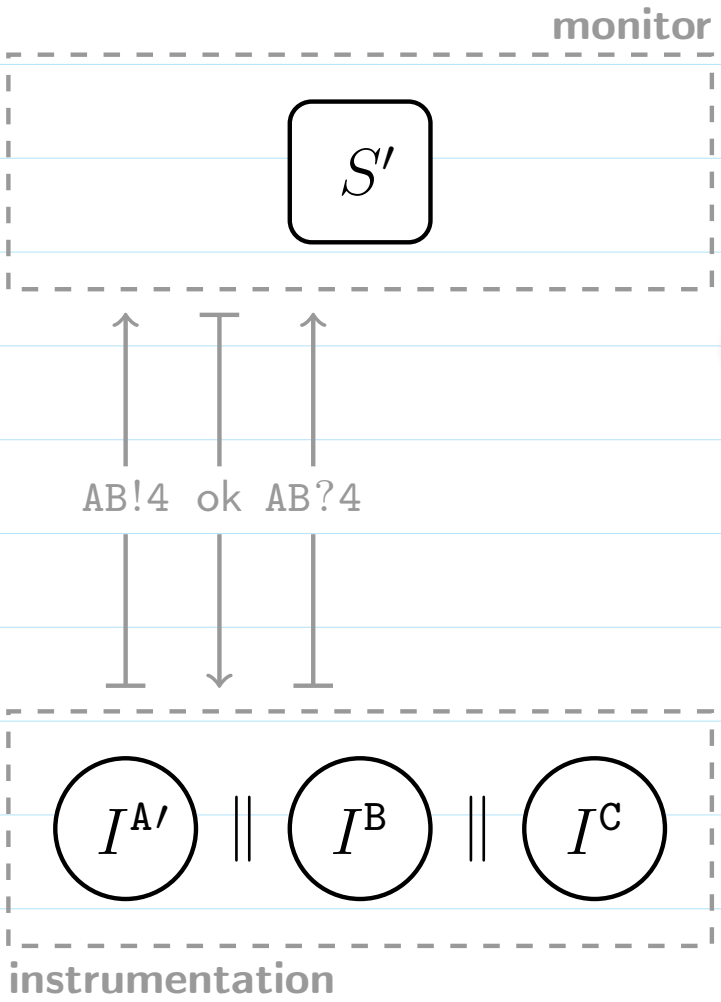

monitor
S
Dynamically verify every !/?
Dynamically observe every **send**/**recv**
$I^{\mathtt{A}} \parallel I^{\mathtt{B}} \parallel I^{\mathtt{C}}$
instrumentation


monitor
S
Dynamically verify every !/?
AB!4
Dynamically observe every send/recv
$I^{\mathtt{A}} \parallel I^{\mathtt{B}} \parallel I^{\mathtt{C}}$
instrumentation



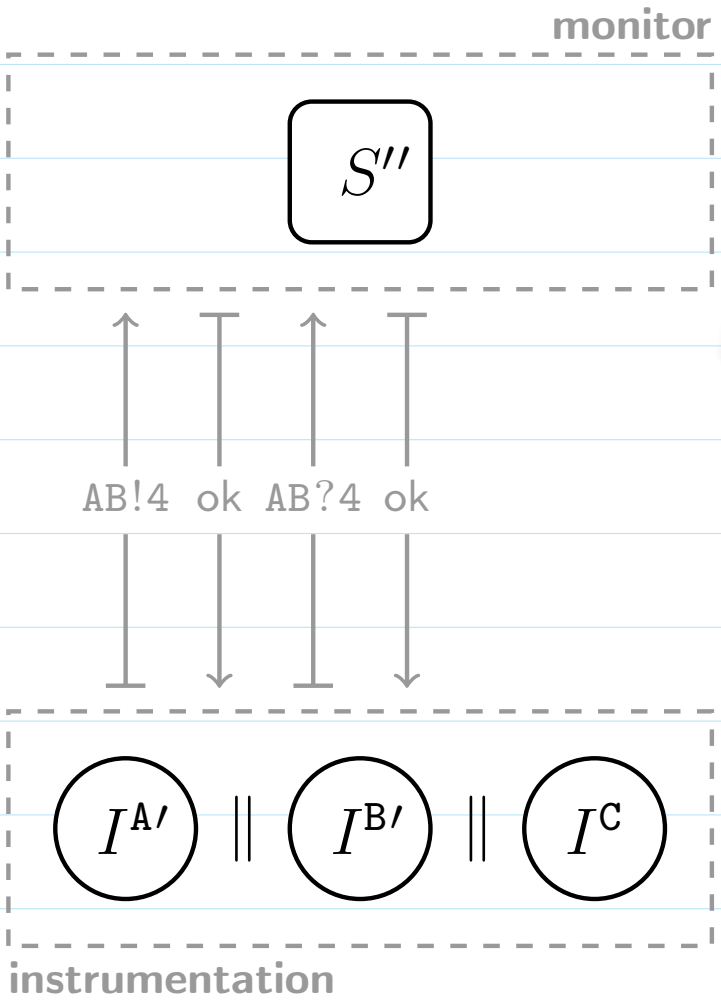
Dynamically verify every !/?

Dynamically observe every **send/recv**



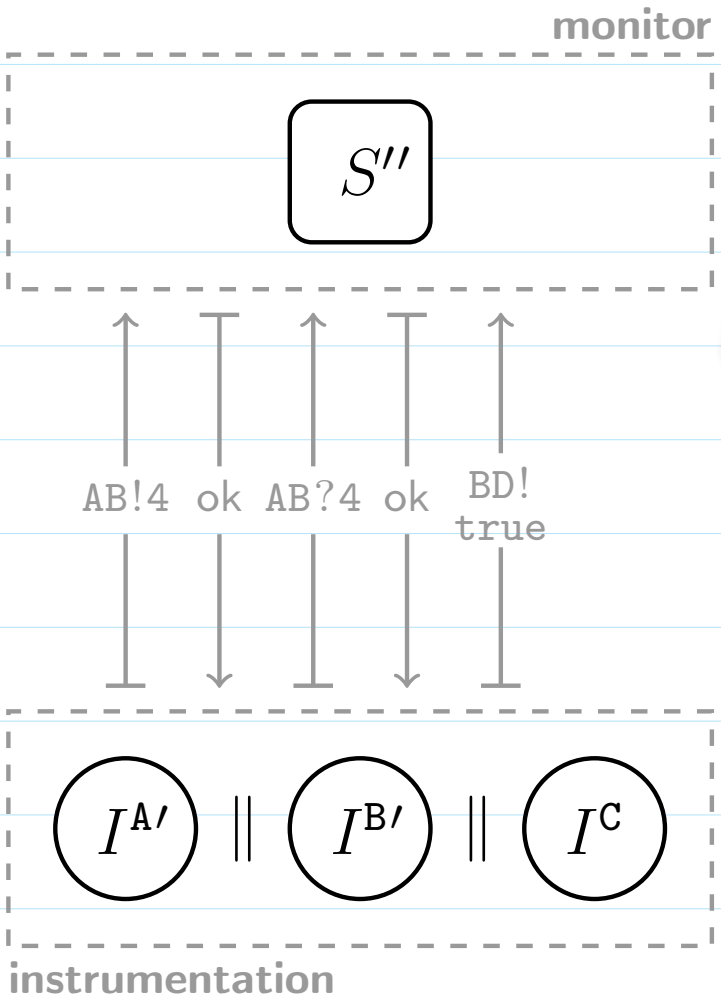
Dynamically verify every !/?

Dynamically observe every **send**/**recv**



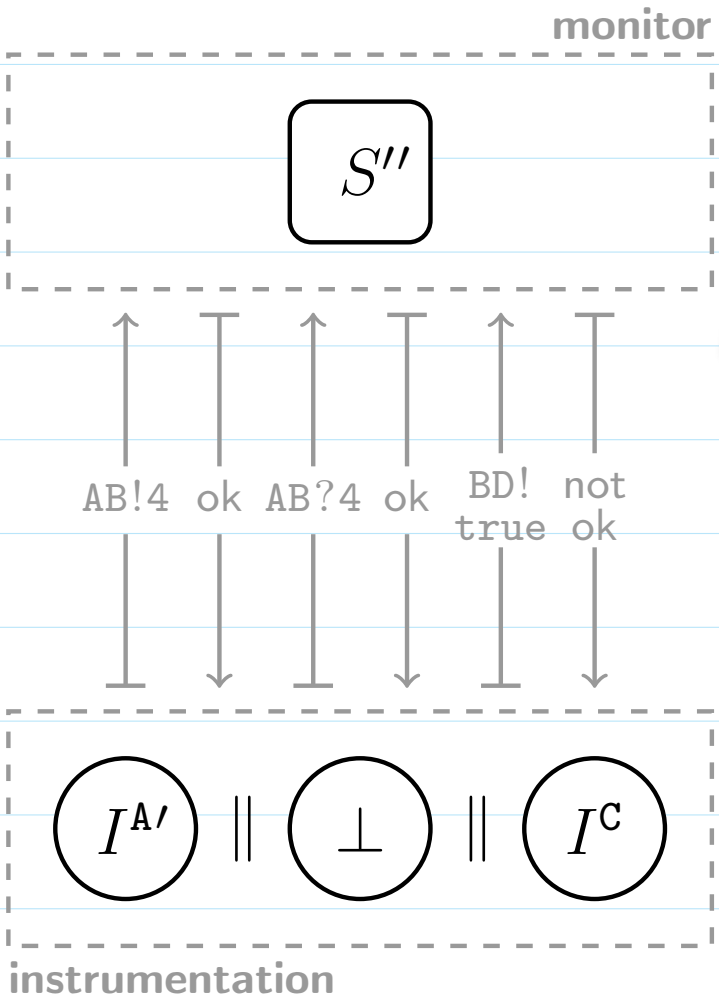
Dynamically verify every !/?

Dynamically observe every **send/recv**



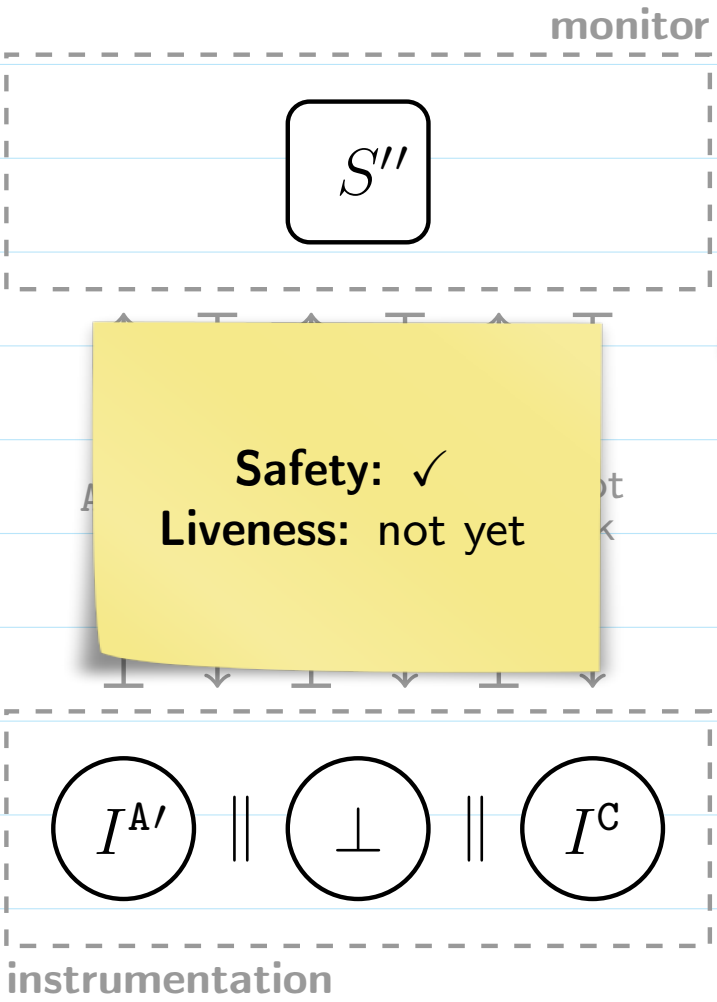
Dynamically verify every !/?

Dynamically observe every **send/recv**



Dynamically verify every !/?

Dynamically observe every **send**/**recv**



Dynamically verify every !/?

Safety: ✓
Liveness: not yet

Dynamically observe every **send**/**recv**

Contributions: [TACAS'20]

- Practice
  - Specification language (Clojure library for monitors)
  - Implementation language (Clojure library for instr.)
  - Non-trivial examples and benchmarks

- Theory
  - Core specification calculus (global MPST "on steroids")
  - Core implementation calculus (mini-Clojure with channels)

Clojure..!?

(= Lisp on the JVM)

Shared memory
concurrency,
with channels

Clojure..!?

(= Lisp on the JVM)

Dynamically typed
(vs. static; good fit with runtime verification)

Shared memory
concurrency,
with channels

Clojure..!?

(= Lisp on the JVM)

Dynamically typed
(vs. static; good fit with runtime verification)

Powerful **macros**
(seamless specification–implementation experience)

Shared memory
concurrency,
with channels

Clojure..!?

(= Lisp on the JVM)

Dynamically typed
(vs. static; good fit with runtime verification)

Powerful **macros**
(seamless specification–implementation experience)

Shared memory
concurrency,
with channels

Clojure..!?

(= Lisp on the JVM)

"ease of dev." >
"performance"
(not a systems language)
[State of Clojure 2019]

Dynamically typed
(vs. static; good fit with runtime verification)

Powerful **macros**
(seamless specification–implementation experience)

Shared memory
concurrency,
with channels

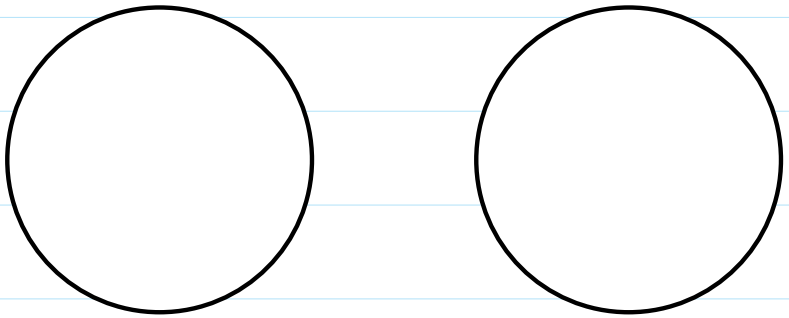
Clojure..!?

(= Lisp on the JVM)

"ease of dev." >
"performance"
(not a systems language)
[State of Clojure 2019]

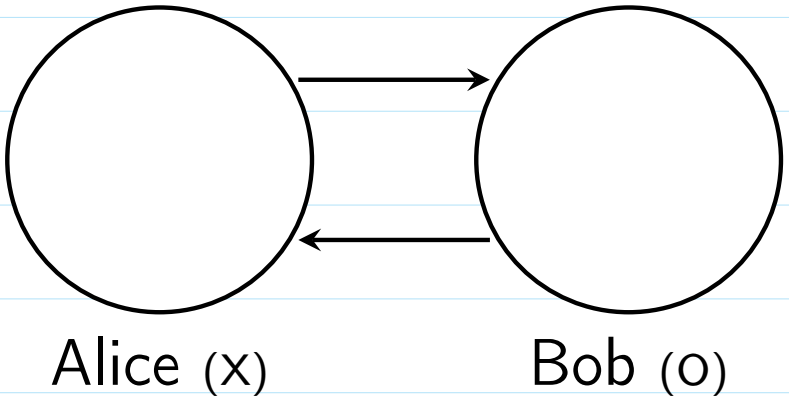
7th **"most loved"**
(cf. Go, C#, Scala, Java)
[SO Dev. Survey 2019]

Tic–Tac–Toe

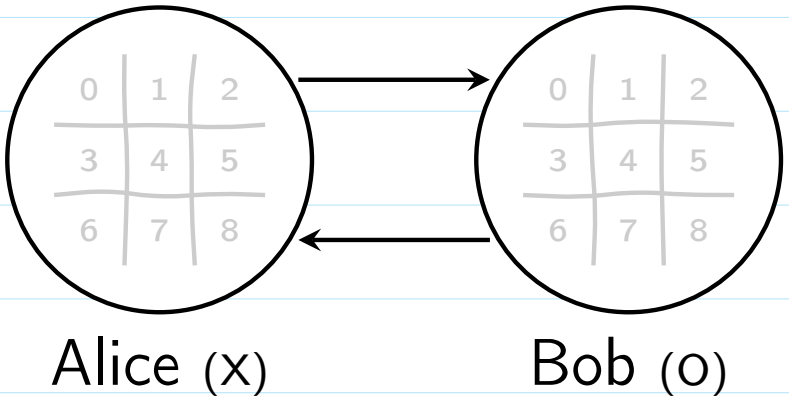


Alice (x) Bob (o)

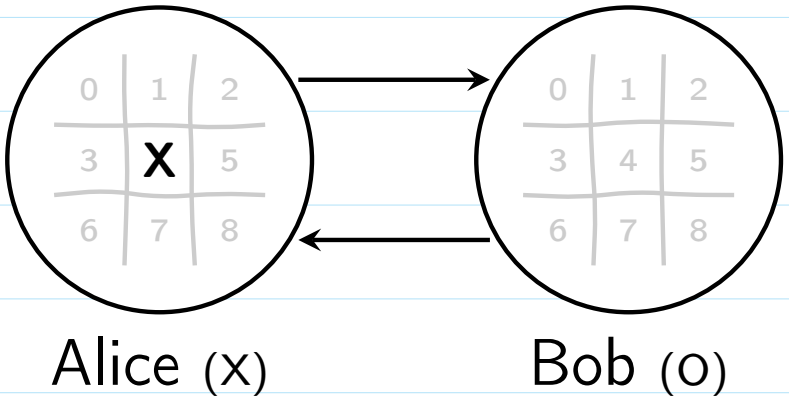
Tic–Tac–Toe



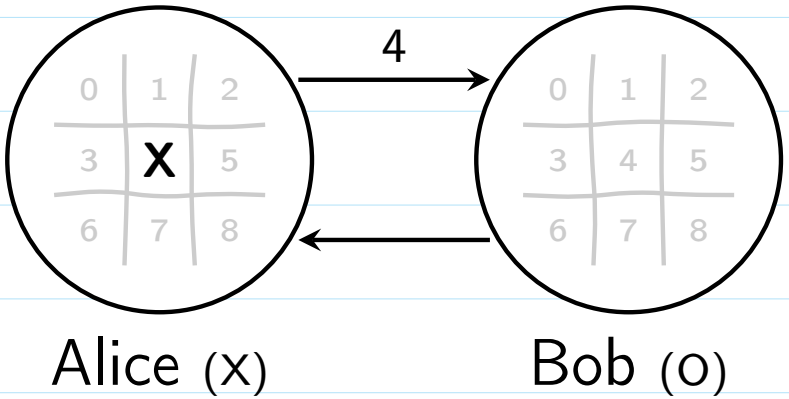
Tic–Tac–Toe



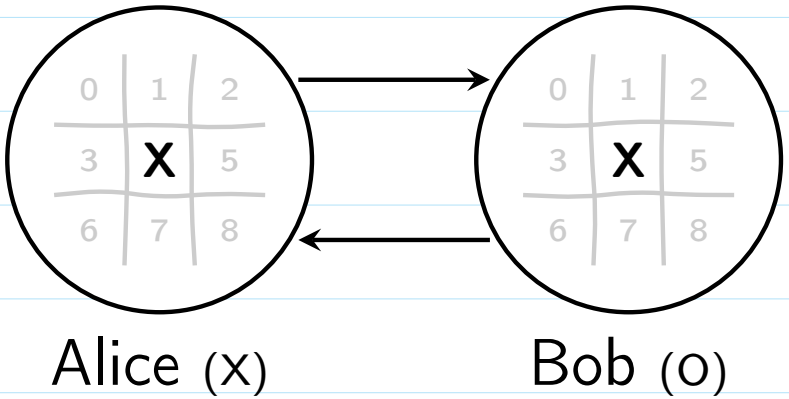
Tic–Tac–Toe



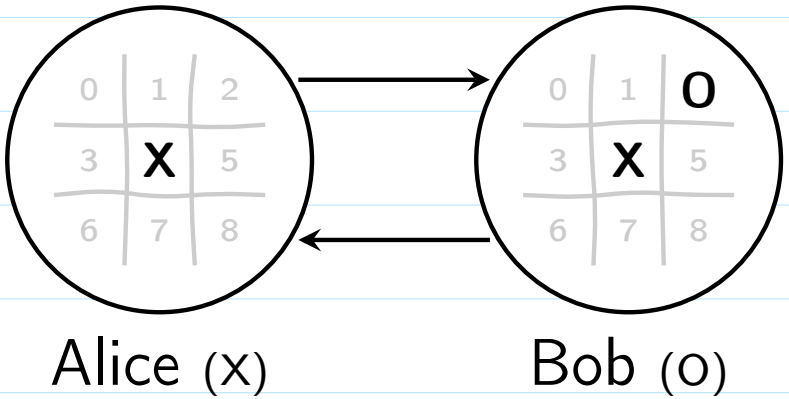
Tic–Tac–Toe



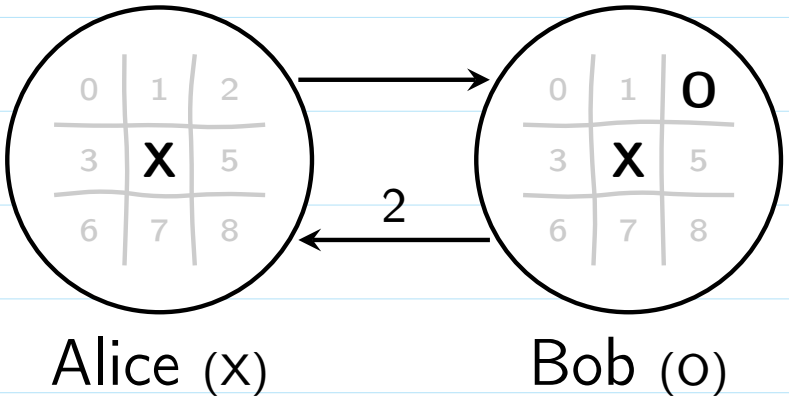
Tic–Tac–Toe



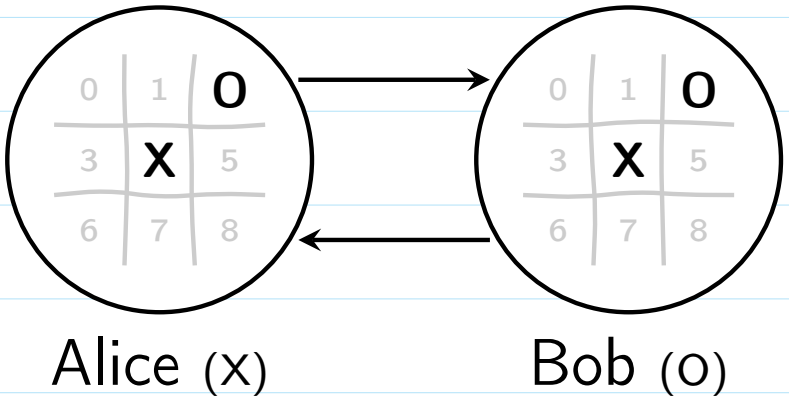
Tic–Tac–Toe



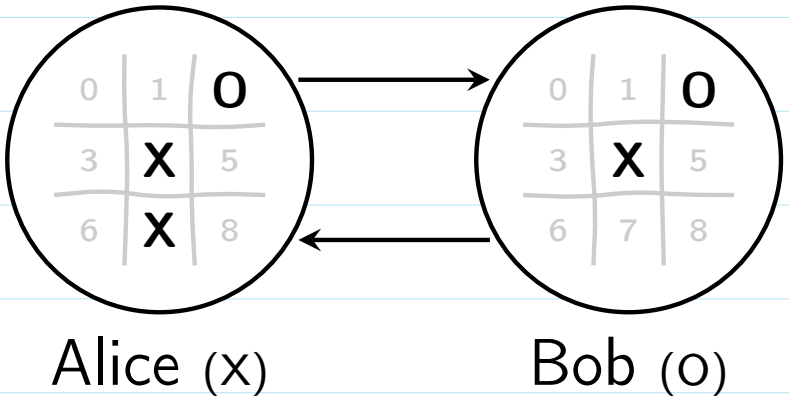
Tic–Tac–Toe



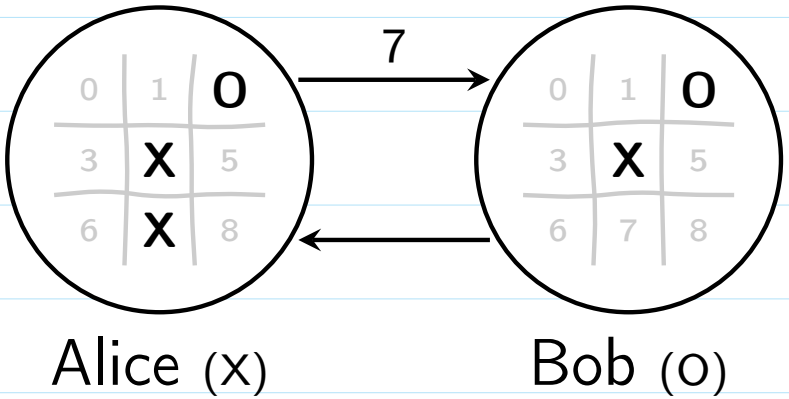
Tic–Tac–Toe



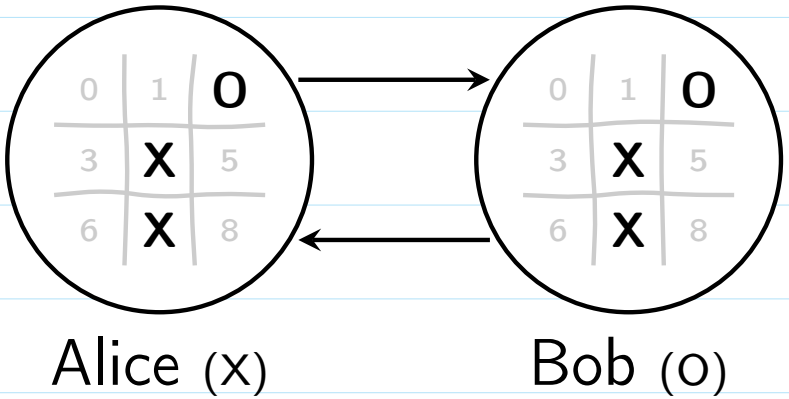
Tic–Tac–Toe



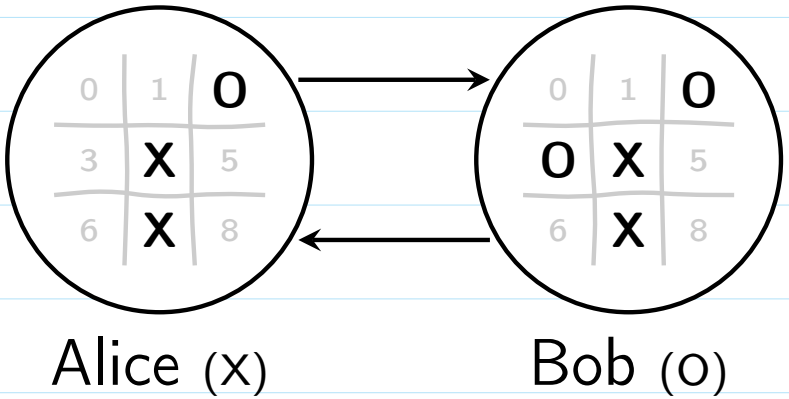
Tic–Tac–Toe



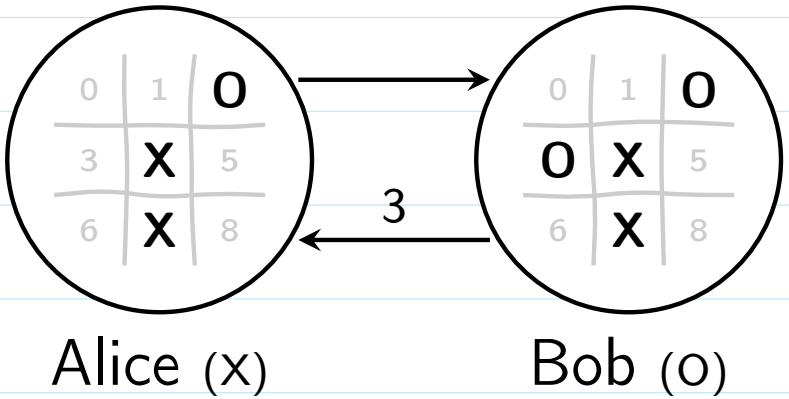
Tic–Tac–Toe



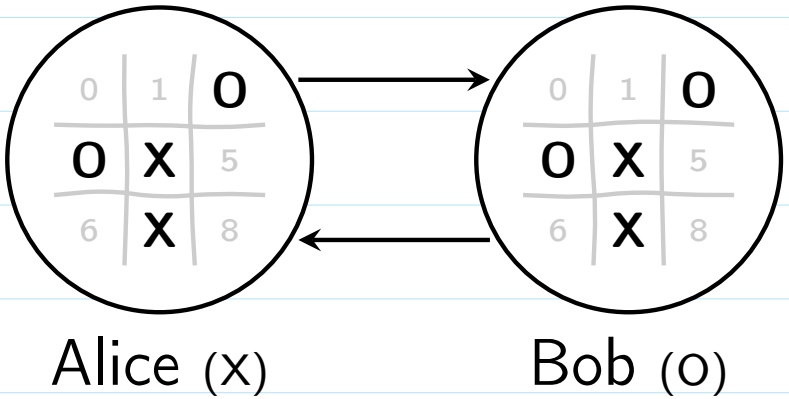
Tic–Tac–Toe



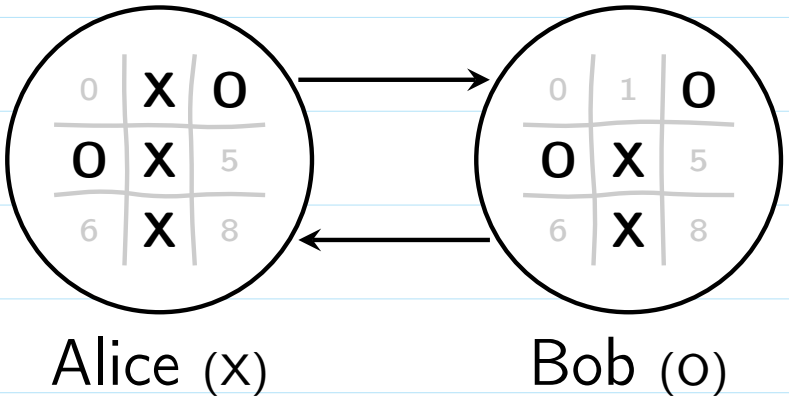
Tic–Tac–Toe



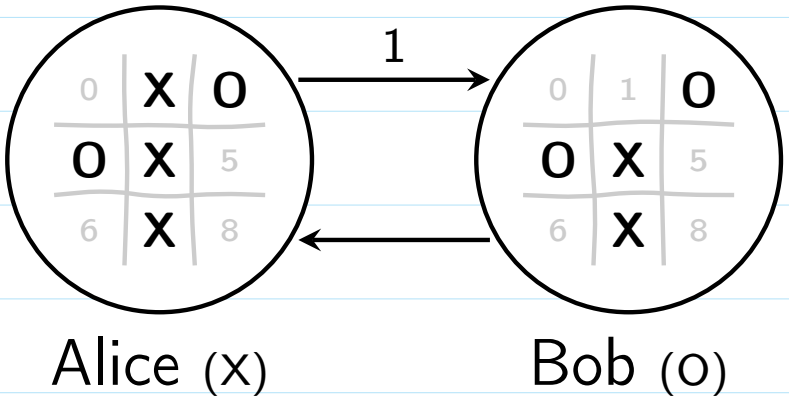
Tic–Tac–Toe



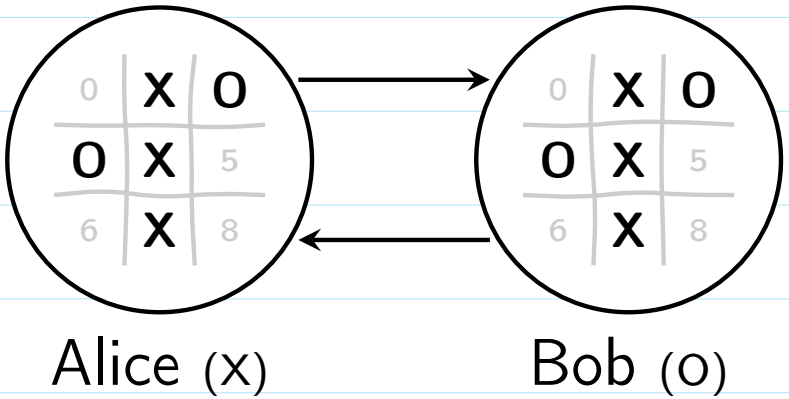
Tic–Tac–Toe



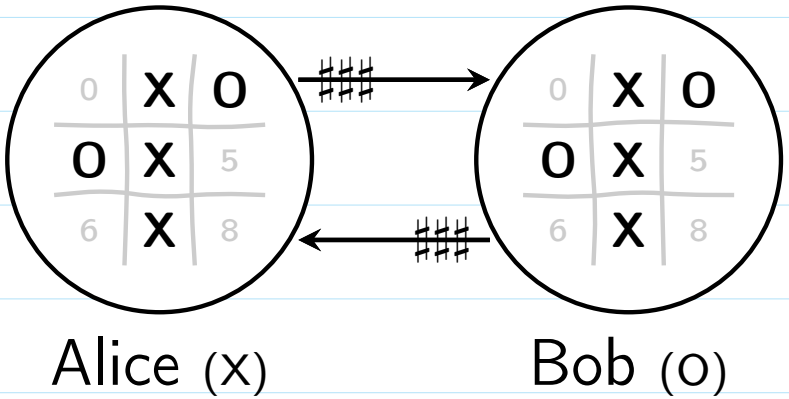
Tic–Tac–Toe



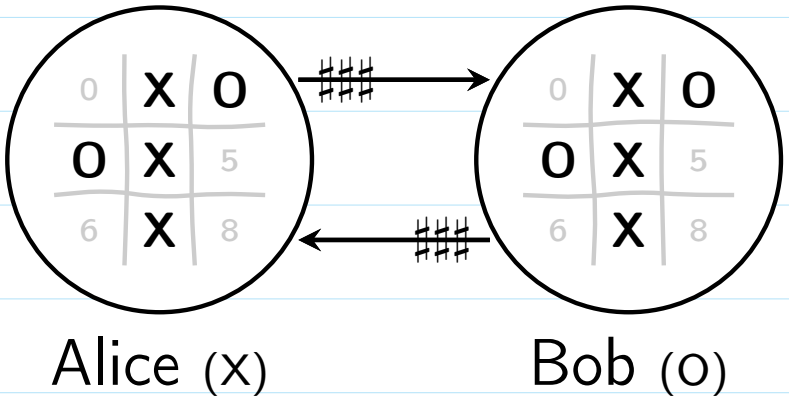
Tic–Tac–Toe



Tic–Tac–Toe



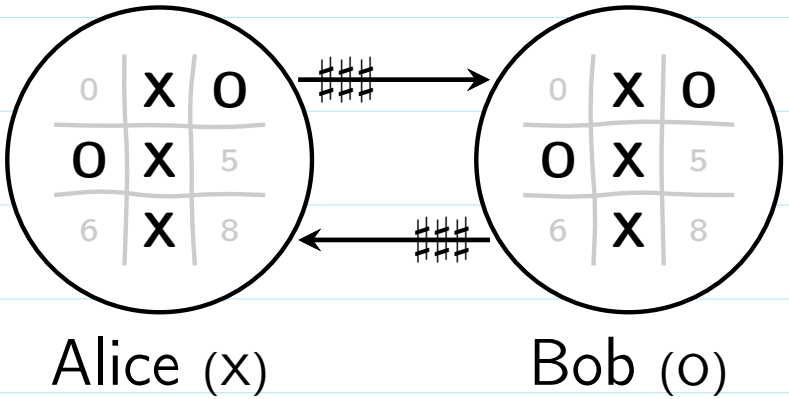
Tic–Tac–Toe



Tic–Tac–Toe

```
(def a (role "alice"))
(def b (role "bob"))
```

Specification

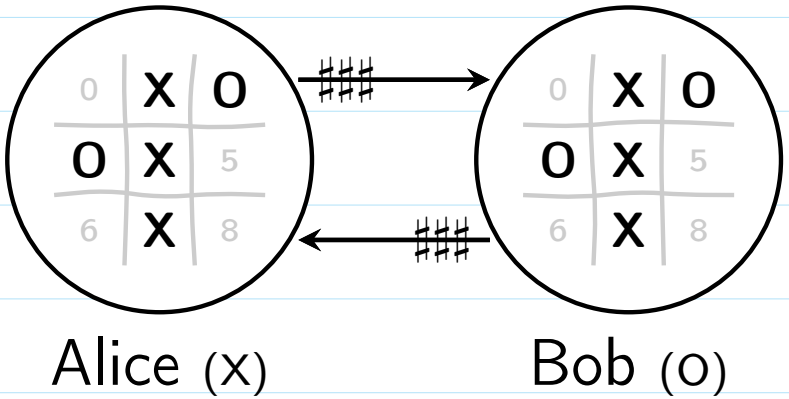


Tic–Tac–Toe

```
(def a (role "alice"))
(def b (role "bob"))

(def ttt-cl (dsl
  (par (### a b)
       (### b a))))
```

Specification



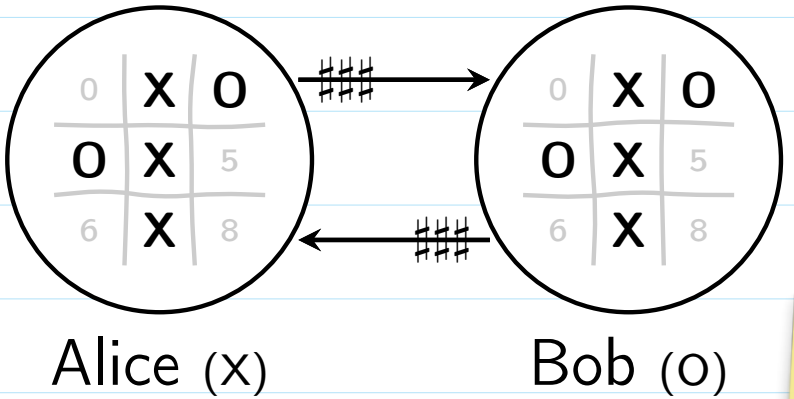
Tic–Tac–Toe

```
(def a (role "alice"))
(def b (role "bob"))

(def ttt-cl (dsl
  (par (### a b)
       (### b a))))
```

```
(def ttt (dsl (fix :X
  [(--> a b Long)
   (alt (ins ttt-cl)
        [(--> b a Long)
         (alt (ins ttt-cl)
              (fix :X))])])))
```

Specification



Tic–Tac–Toe

Not supported by existing MPST tools (because: value-dependent control flow)

```
(def a (role "alice"))
(def b (role "bob"))

(def ttt-cl (dsl
  (par (### a b)
       (### b a))))
```

```
(def ttt (dsl (fix :X
  [(--> a b Long)
   (alt (ins ttt-cl)
        [(--> b a Long)
         (alt (ins ttt-cl)
              (fix :X))])])))
```

Specification

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```clojure
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```clojure
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```clojure
(require '[clojure.core.async :refer :all])

(def c1 (chan 1)) (def c2 (chan 1))
```

Implementation: Channels

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```
(require '[clojure.core.async :refer :all])

(def c1 (chan 1)) (def c2 (chan 1))
```

Implementation: Channels

```
(thread



  (close! c1))
```

Implementation: Alice

Implementation: Bob

```clojure
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```clojure
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```clojure
(require '[clojure.core.async :refer :all])

(def c1 (chan 1)) (def c2 (chan 1))
```

Implementation: Channels

```clojure
(thread
  (loop [g g0]
            (recur g))))))
  (close! c1))
```

Implementation: Alice

Implementation: Bob

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```
(require '[clojure.core.async :refer :all])

(def c1 (chan 1)) (def c2 (chan 1))
```

Implementation: Channels

```
(thread
  (loop [g g0]
    (let [i (get-blank g)
          g (set g i x)]
      (recur g))))))
  (close! c1))
```

Implementation: Alice

Implementation: Bob

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```
(require '[clojure.core.async :refer :all])

(def c1 (chan 1)) (def c2 (chan 1))
```

Implementation: Channels

```
(thread
  (loop [g g0]
    (let [i (get-blank g)
          g (set g i x)]
      (>!! c1 i)



            (recur g))))))
  (close! c1))
```

Implementation: Alice

Implementation: Bob

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```
(require '[clojure.core.async :refer :all])

(def c1 (chan 1)) (def c2 (chan 1))
```

Implementation: Channels

```
(thread
  (loop [g g0]
    (let [i (get-blank g)
          g (set g i x)]
      (>!! c1 i)
      (if (not-final? g)
            (recur g))))))
  (close! c1))
```

Implementation: Alice

Implementation: Bob

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```
(require '[clojure.core.async :refer :all])

(def c1 (chan 1)) (def c2 (chan 1))
```

Implementation: Channels

```
(thread
  (loop [g g0]
    (let [i (get-blank g)
          g (set g i x)]
      (>!! c1 i)
      (if (not-final? g)
        (let [i (<!! c2)
              g (set g i o)]

          (recur g))))))
  (close! c1))
```

Implementation: Alice

Implementation: Bob

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```
(require '[clojure.core.async :refer :all])

(def c1 (chan 1)) (def c2 (chan 1))
```

Implementation: Channels

```
(thread
  (loop [g g0]
    (let [i (get-blank g)
          g (set g i x)]
      (>!! c1 i)
      (if (not-final? g)
        (let [i (<!! c2)
              g (set g i o)]
          (if (not-final? g)
            (recur g))))))
  (close! c1))
```

Implementation: Alice

Implementation: Bob

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```
(require '[clojure.core.async :refer :all])

(def c1 (chan 1)) (def c2 (chan 1))
```

Implementation: Channels

```
(thread
  (loop [g g0]
    (let [i (get-blank g)
          g (set g i x)]
      (>!! c1 i)
      (if (not-final? g)
        (let [i (<!! c2)
              g (set g i o)]
          (if (not-final? g)
            (recur g))))))
  (close! c1))
```

Implementation: Alice

```
(thread
  (loop [g g0]
    (let [i (<!! c1)
          g (set g i x)]
      (if (not-final? g)
        (let [i (get-blank g)
              g (set g i o)]
          (>!! c2 i)
          (if (not-final? g)
            (recur g))))))
  (close! c2))
```

Implementation: Bob

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```
(require '[discourje.core.async :refer :all])

(def c1 (chan 1)) (def c2 (chan 1))
```

Implementation: Channels and instrumentation/monitor

```
(thread
  (loop [g g0]
    (let [i (get-blank g)
          g (set g i x)]
      (>!! c1 i)
      (if (not-final? g)
        (let [i (<!! c2)
              g (set g i o)]
          (if (not-final? g)
            (recur g))))))
  (close! c1))
```

Implementation: Alice

```
(thread
  (loop [g g0]
    (let [i (<!! c1)
          g (set g i x)]
      (if (not-final? g)
        (let [i (get-blank g)
              g (set g i o)]
          (>!! c2 i)
          (if (not-final? g)
            (recur g))))))
  (close! c2))
```

Implementation: Bob

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```
(require '[discourje.core.async :refer :all])
(def m (moni (spec ttt)))
(def c1 (chan 1)) (def c2 (chan 1))
```

Implementation: Channels and instrumentation/monitor

```
(thread
  (loop [g g0]
    (let [i (get-blank g)
          g (set g i x)]
      (>!! c1 i)
      (if (not-final? g)
        (let [i (<!! c2)
              g (set g i o)]
          (if (not-final? g)
            (recur g))))))
  (close! c1))
```

Implementation: Alice

```
(thread
  (loop [g g0]
    (let [i (<!! c1)
          g (set g i x)]
      (if (not-final? g)
        (let [i (get-blank g)
              g (set g i o)]
          (>!! c2 i)
          (if (not-final? g)
            (recur g))))))
  (close! c2))
```

Implementation: Bob

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]     ...)
(defn set        [g i v] ...)
(defn not-final? [g]     ...)
```

Implementation: Tic–Tac–Toe concepts

```
(require '[discourje.core.async :refer :all])
(def m (moni (spec ttt)))
(def c1 (chan 1 a b m)) (def c2 (chan 1 b a m))
```

Implementation: Channels and instrumentation/monitor

```
(thread
  (loop [g g0]
    (let [i (get-blank g)
          g (set g i x)]
      (>!! c1 i)
      (if (not-final? g)
        (let [i (<!! c2)
              g (set g i o)]
          (if (not-final? g)
            (recur g))))))
  (close! c1))
```

Implementation: Alice

```
(thread
  (loop [g g0]
    (let [i (<!! c1)
          g (set g i x)]
      (if (not-final? g)
        (let [i (get-blank g)
              g (set g i o)]
          (>!! c2 i)
          (if (not-final? g)
            (recur g))))))
  (close! c2))
```

Implementation: Bob

```
(def blank " ") (def x "x") (def o "o")

(def g0 [blank blank blank
         blank blank blank
         blank blank blank])
```

```
(defn get-blank  [g]      ...)
(defn set        [g i v]  ...)
(defn not-final
```

Implementation: Tic–Tac–Toe concepts

Actually, there's an **unsafe execution..!**

```
(require '[discourje.core.async :refer :all])
(def m (moni (spec ttt)))
(def c1 (chan 1 a b m)) (def c2 (chan 1 b a m
```

Implementation: Channels and instrumentation/monitor

```
(thread
  (loop [g g0]
    (let [i (get-blank g)
          g (set g i x)]
      (>!! c1 i)
      (if (not-final? g)
        (let [i (<!! c2)
              g (set g i o)]
          (if (not-final? g)
            (recur g))))))
  (close! c1))
```

Implementation: Alice

```
(thread
  (loop [g g0]
    (let [i (<!! c1)
          g (set g i x)]
      (if (not-final? g)
        (let [i (get-blank g)
              g (set g i o)]
          (>!! c2 i)
          (if (not-final? g)
            (recur g))))))
  (close! c2))
```

Implementation: Bob

```
(def m (role "master"))
(def w (role "worker"))
```

Indexed roles
and **parametrised**
specifications

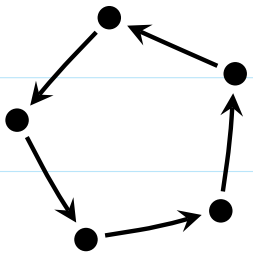
Roles

Indexed roles
and **parametrised**
specifications

```
(def m (role "master"))
(def w (role "worker"))
```

Roles

```
(def ring (dsl :k :t
  [(rep seq [:i (range (- :k 1))]
     (--> (w :i) (w (+ :i 1)) :t))
   (--> (w (- :k 1)) (w 0) :t)])
```



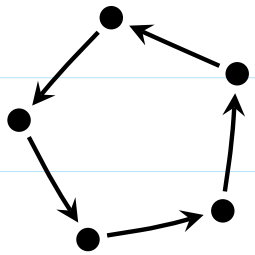
Specification: Ring

Indexed roles
and **parametrised**
specifications

```
(def m (role "master"))
(def w (role "worker"))
```

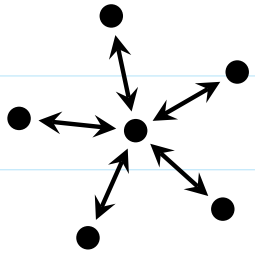
Roles

```
(def ring (dsl :k :t
  [(rep seq [:i (range (- :k 1))]
     (--> (w :i) (w (+ :i 1)) :t))
   (--> (w (- :k 1)) (w 0) :t)]))
```



Specification: Ring

```
(def star (dsl :k :t
  [(rep par [:i (range :k)]
     [(--> m (w :i) :t)
      (--> (w :i) m :t)])]))
```



Specification: Star

Sales pitch:

Sales pitch:

- Guarantees
  - Safety: If a send/receive happens, then it is allowed
  - Freedom of data races (if pure Clojure)

Sales pitch:

- Guarantees
  - Safety: If a send/receive happens, then it is allowed
  - Freedom of data races (if pure Clojure)

- Ergonomics
  - Expressive specification language (beyond existing MPST)
  - Discourje syntax $\subseteq$ Clojure syntax (easy to start using)
  - Discourje editor/IDE/compiler $=$
    Clojure editor/IDE/compiler (no new tools)

Sales pitch:

What about **performance..?**

- Guarantees
  - Safety: If a send/receive happens, then it is allowed
  - Freedom of data races (if pure Clojure)

- Ergonomics
  - Expressive specification language (beyond existing MPST)
  - Discourje syntax $\subseteq$ Clojure syntax (easy to start using)
  - Discourje editor/IDE/compiler $=$ Clojure editor/IDE/compiler (no new tools)

NAS Parallel Benchmarks

NAS Parallel Benchmarks

Computational **fluid dynamics** kernels:
CG, FT, IS, MG

NAS Parallel Benchmarks

Computational **fluid dynamics** kernels: CG, FT, IS, MG

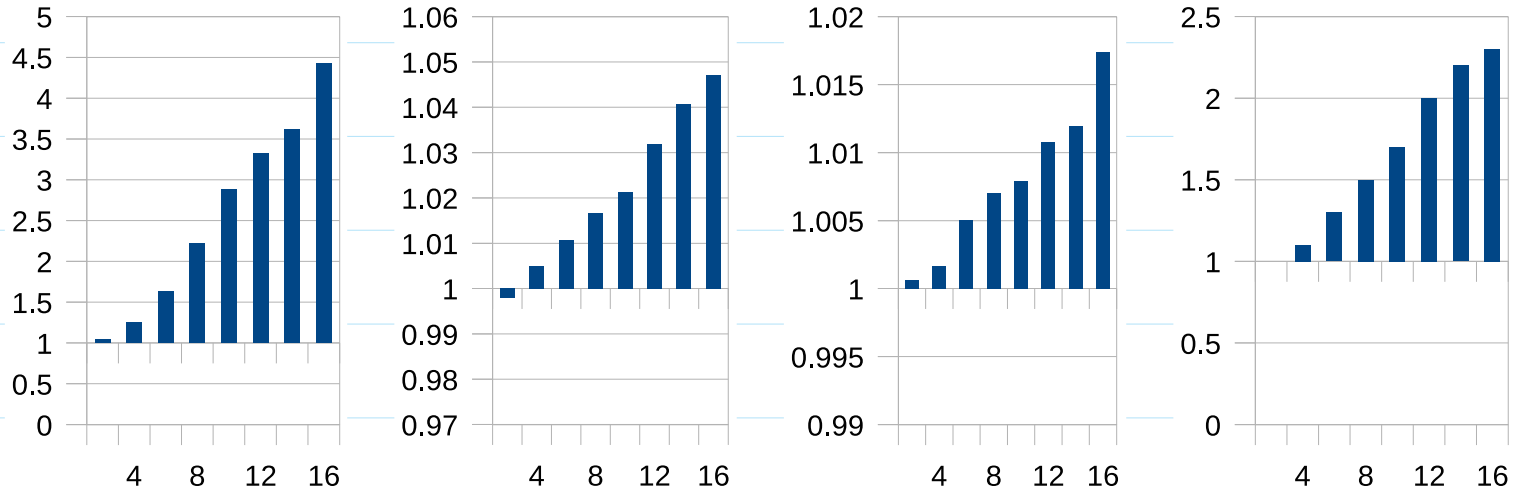
NASA's **reference** implementations **vs. Discourje** versions (using Java interop)

NAS Parallel Benchmarks

Computational **fluid dynamics** kernels: CG, FT, IS, MG

NASA's **reference** implementations **vs. Discourje** versions (using Java interop)

Hardware: 16 cores at 2.1 GHz (no hyper-threading); 96 GB memory



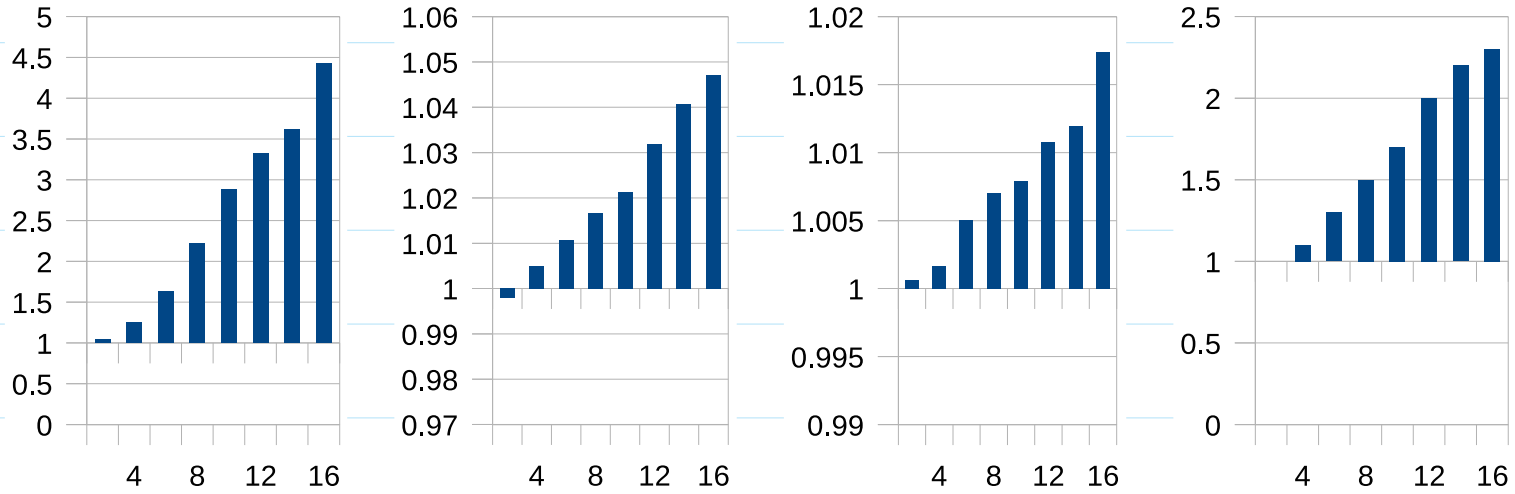
x-axis: #threads y-axis: Discourje slowdown

NAS Parallel Benchmarks

Computational **fluid dynamics** kernels: CG, FT, IS, MG

NASA's **reference** implementations **vs. Discourje** versions (using Java interop)

Hardware: 16 cores at 2.1 GHz (no hyper-threading); 96 GB memory



x-axis: #threads y-axis: Discourje slowdown

NAS Parallel Benchmarks

Conclusions:

- Overhead can be $<5\%$ in real concurrent programs
- Usages: **(i)** testing/debugging; **(ii)** fail-safe (in production..!)

Contributions: [TACAS'20]

- Practice
  - Specification language (Clojure library for monitors)
  - Implementation language (Clojure library for instr.)
  - Non-trivial examples and benchmarks

- Theory
  - Core specification calculus (global types "on steroids")
  - Core implementation calculus (mini-Clojure with channels)

Syntax of **specification calculus:**

$$
\begin{aligned}
S \;::=\; & 1 \;\mid\; p \twoheadrightarrow q : f \;\mid\; pq?v \;\mid\; p \not\twoheadrightarrow q \;\mid \\
& S_1 + S_2 \;\mid\; S_1 \cdot S_2 \;\mid\; S_1 \parallel S_2 \;\mid\; \mathrm{fix}\, X\, S \;\mid\; X \\
& \bigotimes\nolimits^{+}_{n \le x \le n'} S \;\mid\; \bigotimes\nolimits^{\cdot}_{n \le x \le n'} S \;\mid\; \bigotimes\nolimits^{\parallel}_{n \le x \le n'} S
\end{aligned}
$$

Syntax of **specification calculus:**

$$
\begin{aligned}
S \;::=\; & 1 \;\mid\; p \rightarrowtriangle q : f \;\mid\; pq?v \;\mid\; p \not\rightarrowtriangle q \;\mid \\
& S_1 + S_2 \;\mid\; S_1 \cdot S_2 \;\mid\; S_1 \parallel S_2 \;\mid\; \mathrm{fix}\, X\, S \;\mid\; X \\
& \bigotimes\nolimits^{+}_{n \le x \le n'} S \;\mid\; \bigotimes\nolimits^{\cdot}_{n \le x \le n'} S \;\mid\; \bigotimes\nolimits^{\parallel}_{n \le x \le n'} S
\end{aligned}
$$

Semantics:

$$
S \xrightarrow{\beta} S' \qquad\qquad \beta \in \{pq!v, pq?v, pq\sharp\}
$$

Syntax of **implementation calculus:**

$$
\begin{aligned}
v \;&::=\; \mathbf{nil} \mid \ell \mid (\mathbf{fn}\, x\, I) \mid \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid \texttt{1} \mid \cdots \\
I \;&::=\; (I_1\, I_2) \mid x \mid (\mathbf{def}\, x\, I) \mid (\mathbf{let}\, x\, I_1\, I_2) \mid \\
&\quad\; (\mathbf{loop}\, x\, I_1\, I_2) \mid (\mathbf{recur}\, I) \mid (\mathbf{if}\, I_1\, I_2\, I_3) \mid \\
&\quad\; (\mathbf{send}\, I_1\, I_2) \mid (\mathbf{recv}\, I) \mid (\mathbf{close}\, I) \mid \\
&\quad\; (\mathbf{chan}\, I) \mid I_1 \cdot I_2 \mid I_1 \parallel I_2
\end{aligned}
$$

Semantics:

$$
(I, \mathcal{E}, \mathcal{H}) \xrightarrow{\alpha} (I', \mathcal{E}', \mathcal{H}') \qquad \alpha \in \{\ell!v, \ell?v, \ell\sharp, \tau\}
$$

Runtime verification:

$(I, \mathcal{E}, \mathcal{H})$

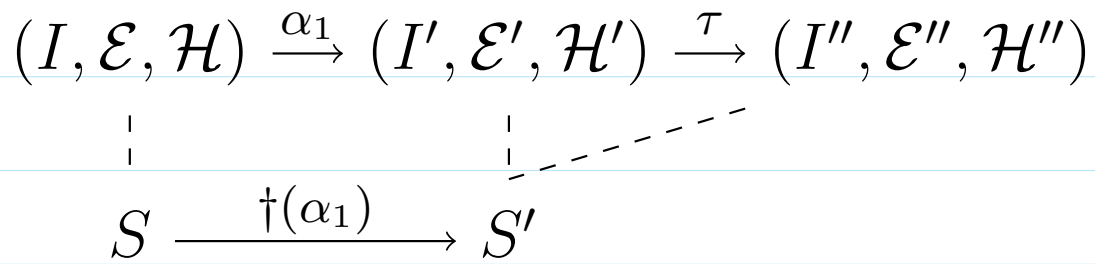
$\vdots$

S

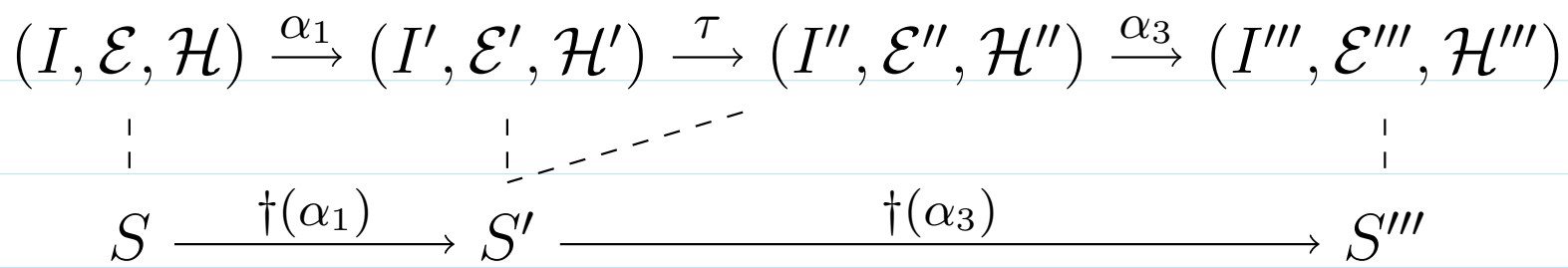
Runtime verification:

$$\begin{array}{ccc} (I, \mathcal{E}, \mathcal{H}) & \xrightarrow{\alpha_1} & (I', \mathcal{E}', \mathcal{H}') \\ \vdots & & \vdots \\ S & \xrightarrow{\dagger(\alpha_1)} & S' \end{array}$$

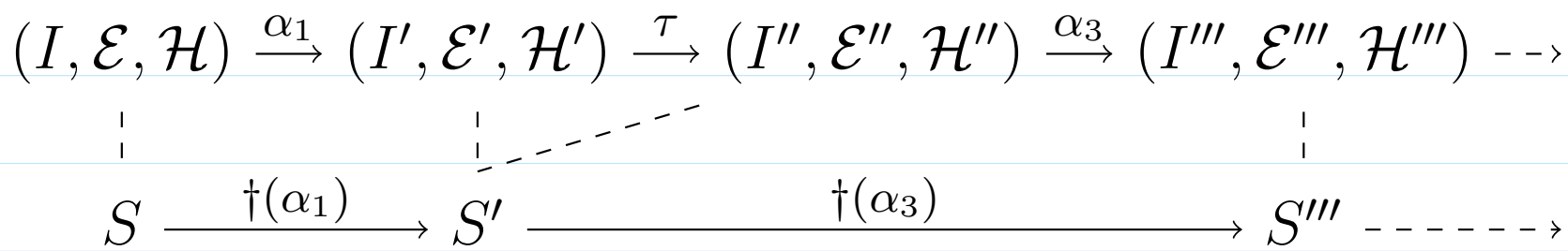
Runtime verification:



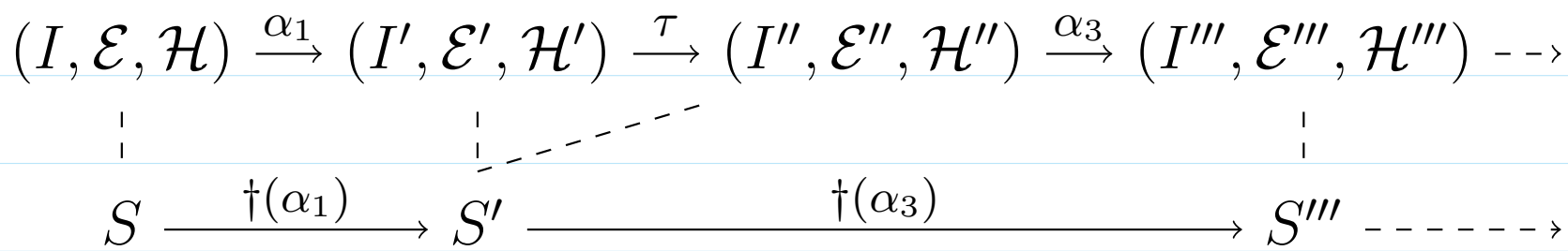
Runtime verification:



Runtime verification:



Runtime verification:



$\dagger$: heap locations $\rightarrow$ sender–receiver pairs

(e.g., $\dagger(\texttt{0x2A}) = \texttt{AB}$) (cf. (**chan** 1 a b m))

Related work:

- Hybrid MPST: combination of static type-checking and dynamic monitoring (decomposition-based)

  [e.g.: Bocchi et al., TCS 669; Neykova et al., FAC 29]

- MPST without decomposition (not fully automated)

  [e.g.: López et al., OOPSLA'15]

- Formal techniques for Clojure (no concurrency)

  [e.g.: Bonnaire-Sergeant et al., ESOP'16]

Takeaways:

- Concurrency remains hard
- Runtime verification for MPST, without decomposition, is an interesting alternative (notably: improved expressiveness)
- Paper and artifact with more details:
  - `doi:10.1007/978-3-030-45190-5_15`
  - `https://github.com/discourje`

Takeaways:

- Concurrency remains hard
- Runtime verification for MPST, without decomposition, is an interesting alternative (notably: improved expressiveness)
- Paper and artifact with more details:
  - `doi:10.1007/978-3-030-45190-5_15`
  - `https://github.com/discourje`