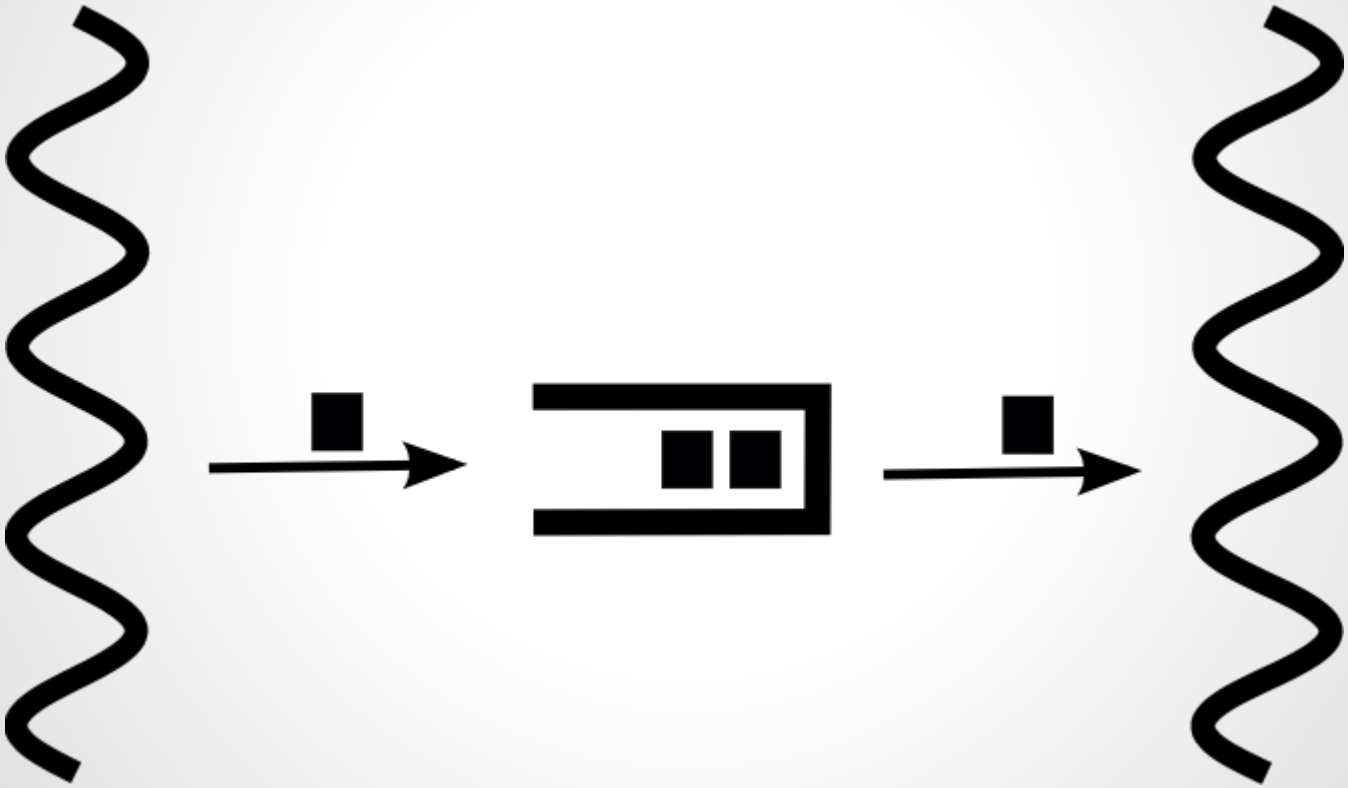
A core.async Debugging Toolkit

@david_mcneil

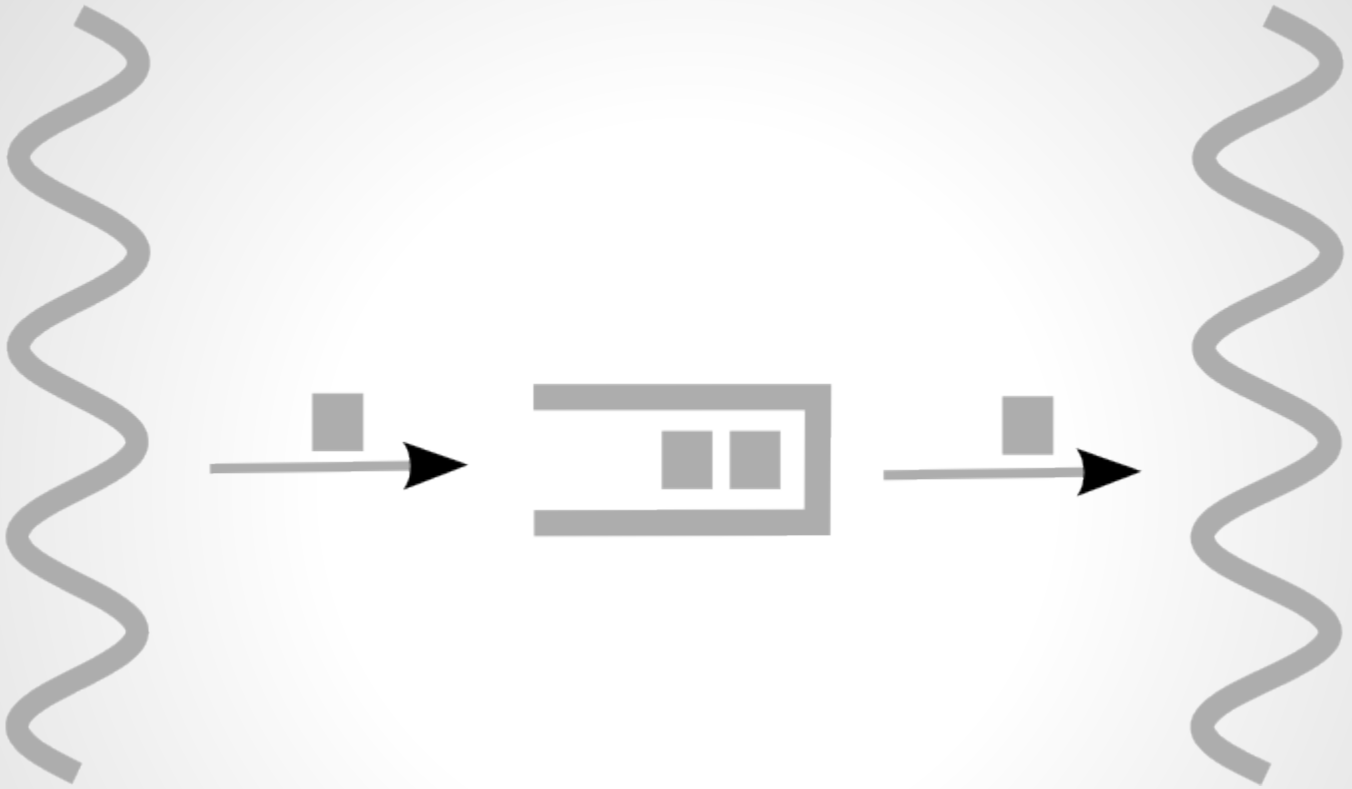
September 2014

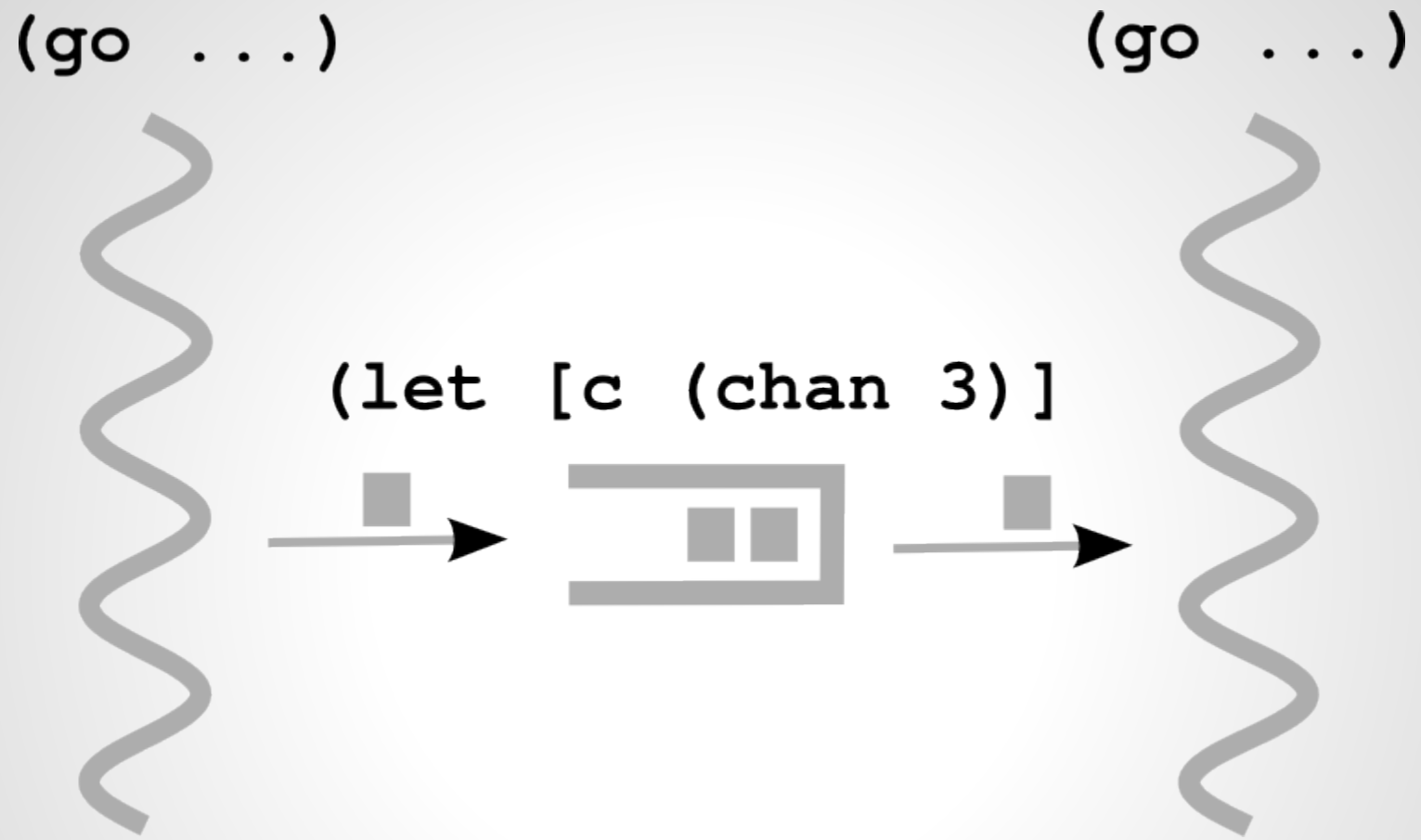
core.async

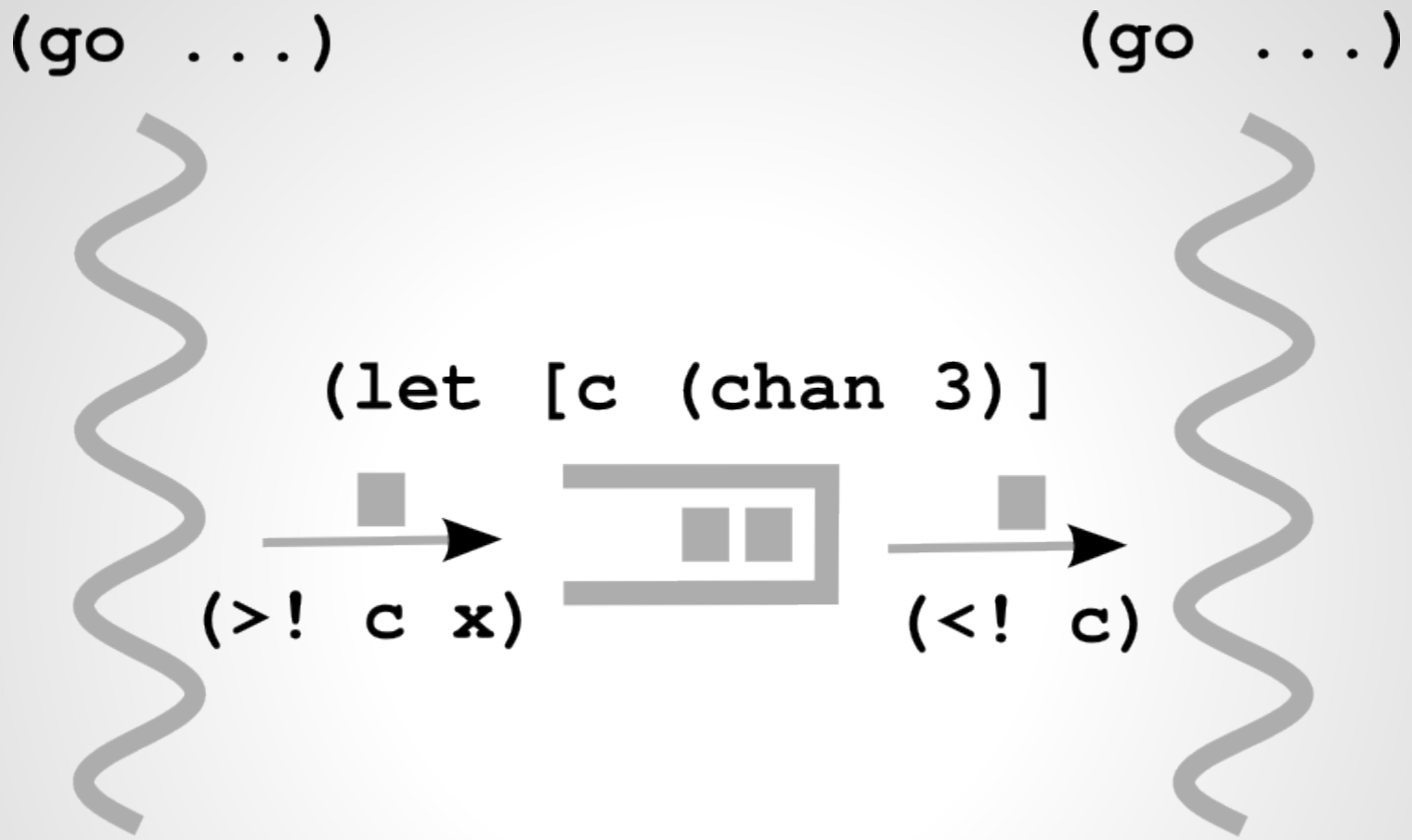


(go . . .)

(go . . .)




(go ...)
(go ...)
(let [c (chan 3)]


(go ...)
(go ...)
(let [c (chan 3)]
(>! c x)
(<! c)

core.async example

```
(defn f []
  (let [c1 (chan)
        c2 (chan)]
    (go
      (>! c1 1))
    (go
      (>! c2 2))
    (go
      (println (+ (<! c1)
                  (<! c2))))))
(f) ;; prints 3
```

step.async

```
(let [machine ((step-machine) f)]
  (doseq [i (range 10)]
    (step machine)
    (step-wait machine))
  (pprint machine))
```

step machine state

```
{:threads           ["thread-1"
                     "thread-2"
                     "thread-3"]
 :channels          ["channel-20001"
                     "channel-20002"],
 :channel-contents {"channel-20001" []
                    "channel-20002" [2]}
 :blocked-takes    {"thread-3" ["channel-20002"]}
 :blocked-puts     {"thread-2" "channel-20002"
                    "thread-1" "channel-20001"}
 :last-action      [:put "thread-2" "channel-20002" 2]}
```

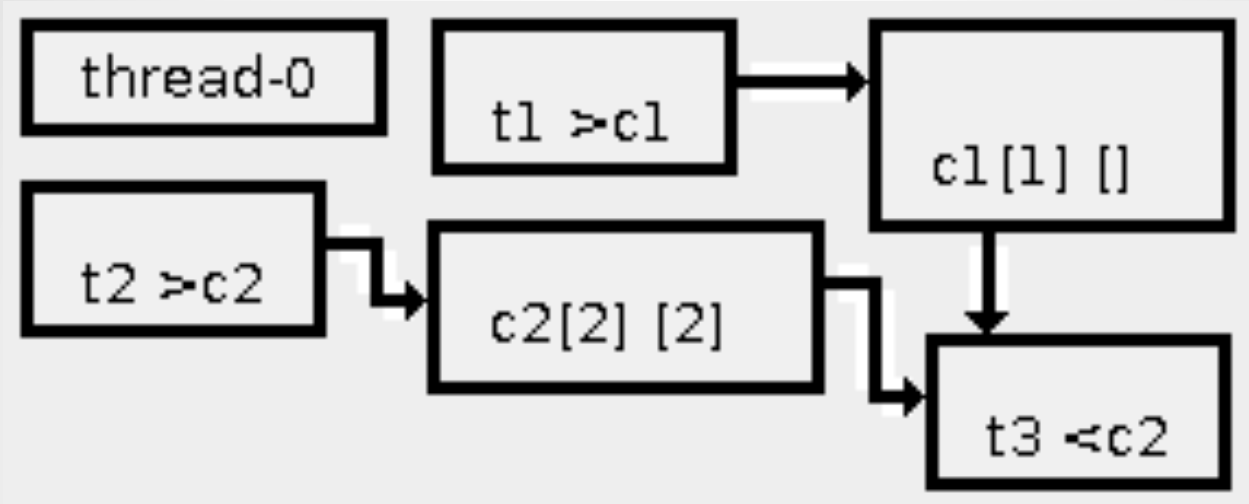
Name the step.async constructs

```
(defn f []
  (let [c1 (chan-named "c1")
        c2 (chan-named "c2")]
    (go-named "t1"
      (>! c1 1))
    (go-named "t2"
      (>! c2 2))
    (go-named "t3"
      (println (+ (<! c1)
                  (<! c2))))))
```

step.async

```
(let [m (step-machine
          :channel-history? true)
      machine (m f)]
  (state-trace machine true)
  (doseq [i (range 10)]
    (step machine)
    (step-wait machine)))
```

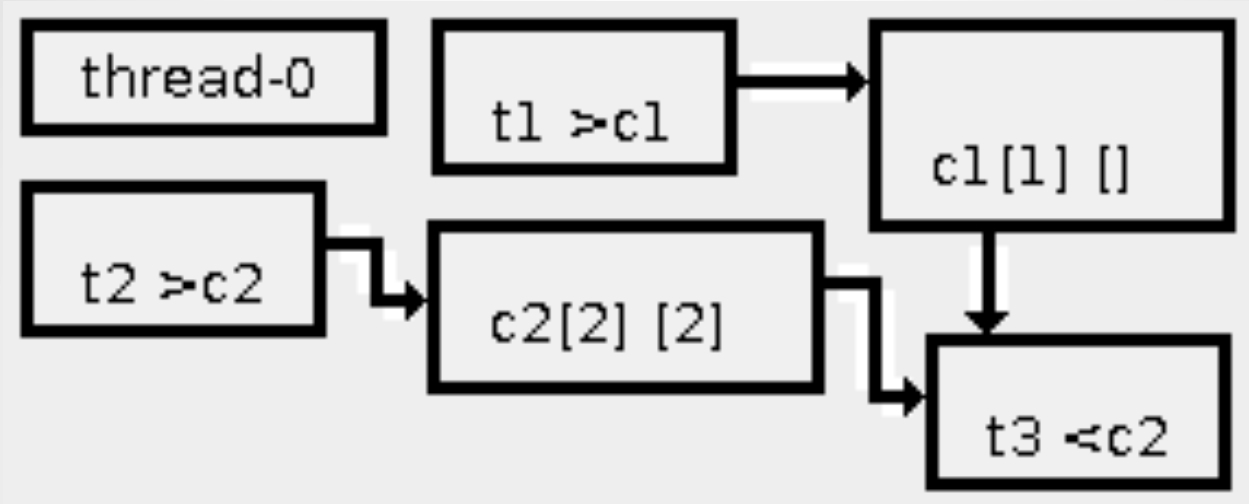
*Vijual*ize step machine state



Threads

- arrows indicate flow of items in/out of channels
- current status of thread indicated
  - "> x" putting to the given channel
  - "< x" taking from the given channel

Visualize step machine state



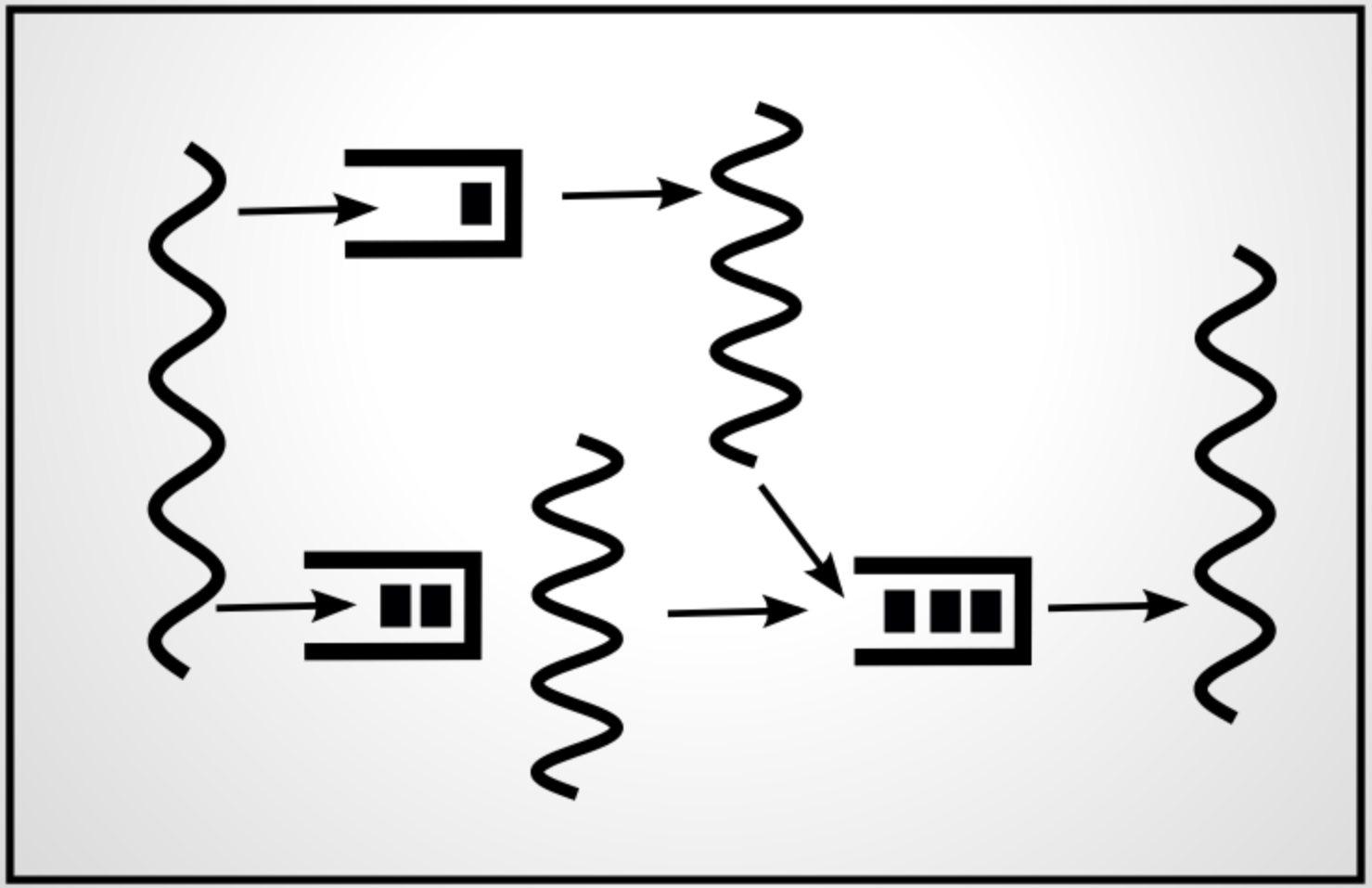
Channels

- first vector shows history of items through channel
- second vector shows current items in channel buffer

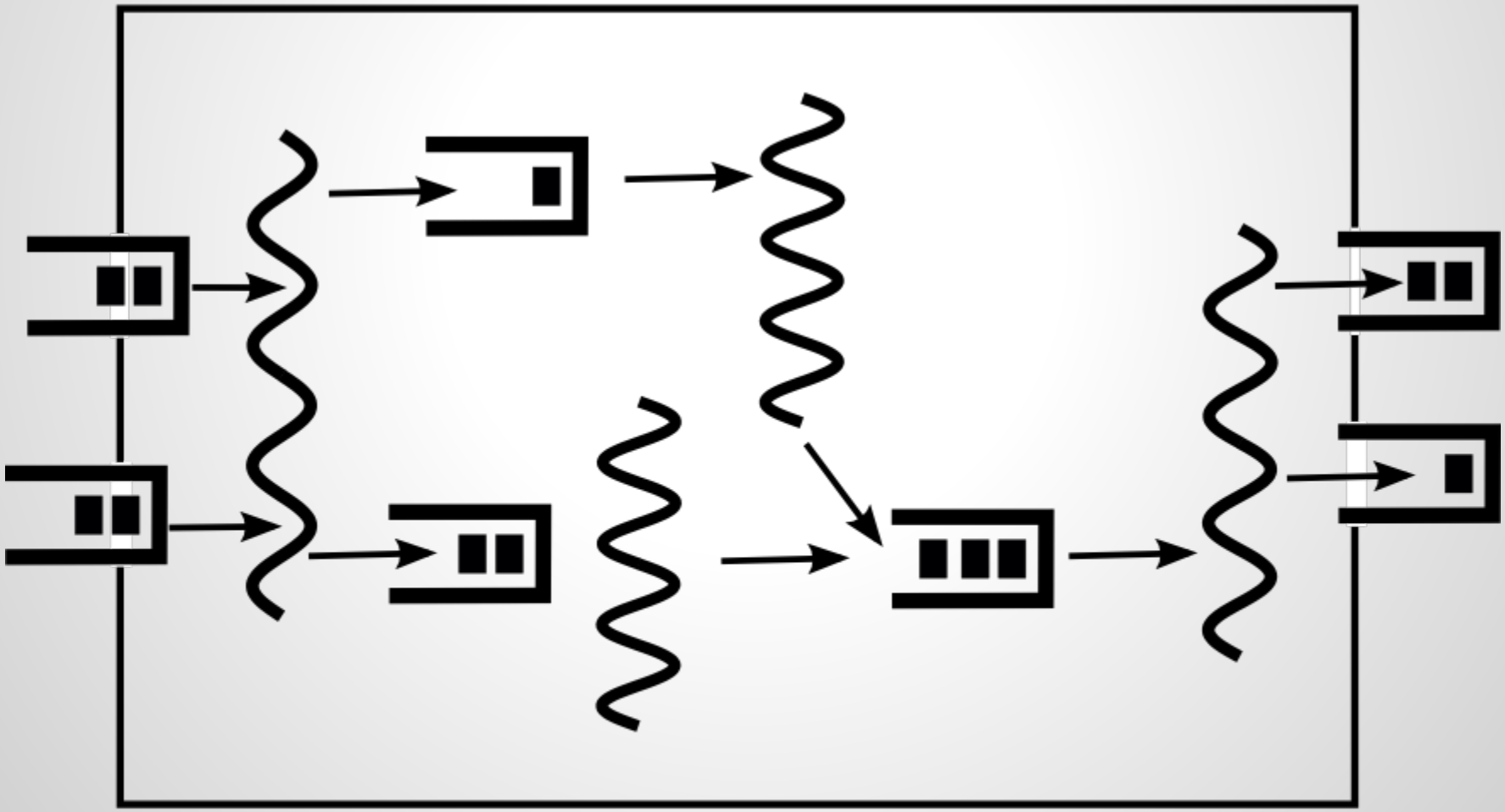
Machines

Photo credit: D J Shin

async machine



async machine with input/output channels



Two ways of stepping

- allow the machine to proceed by one "increment"

```
(step machine)
```

Two ways of stepping

- allow the machine to proceed by one "increment"

```
(step machine)
```

- allow the machine to run until it winds down

```
(step-all machine)
```

Waiting for steps

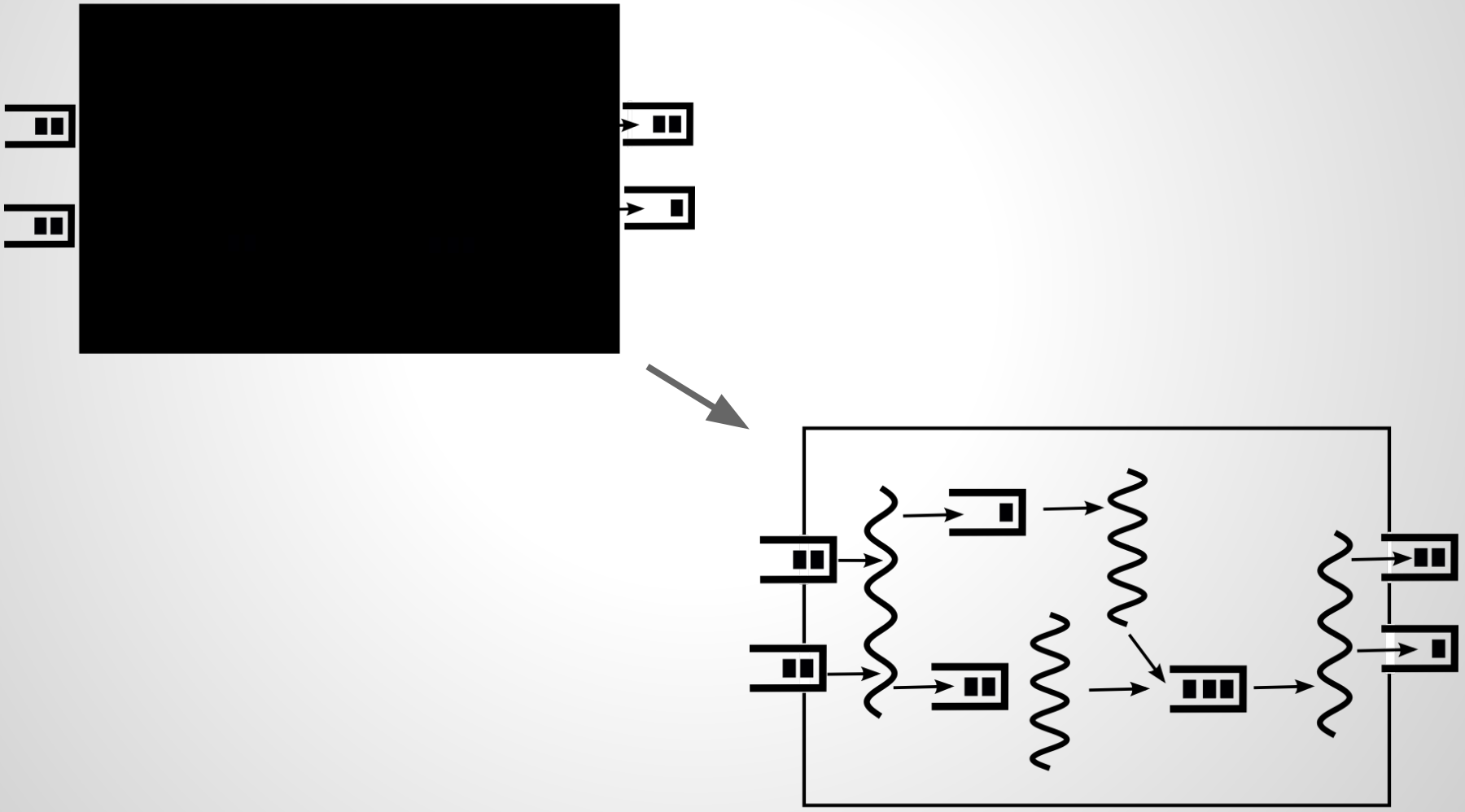
- allow the machine to proceed by one "increment"

```
(step machine)
(step-wait machine)
```

- allow the machine to run until it winds down

```
(step-all machine)
(quiesce-wait machine)
```

Transparent Execution



Transparent Execution

- Tracks named go threads

Transparent Execution

- Tracks named go threads
- Tracks named channels

Transparent Execution

- Tracks named go threads
- Tracks named channels
- Optionally captures history

Transparent Execution

- Tracks named go threads
- Tracks named channels
- Optionally captures history
- Captures async machine state as data to be processed or visualized

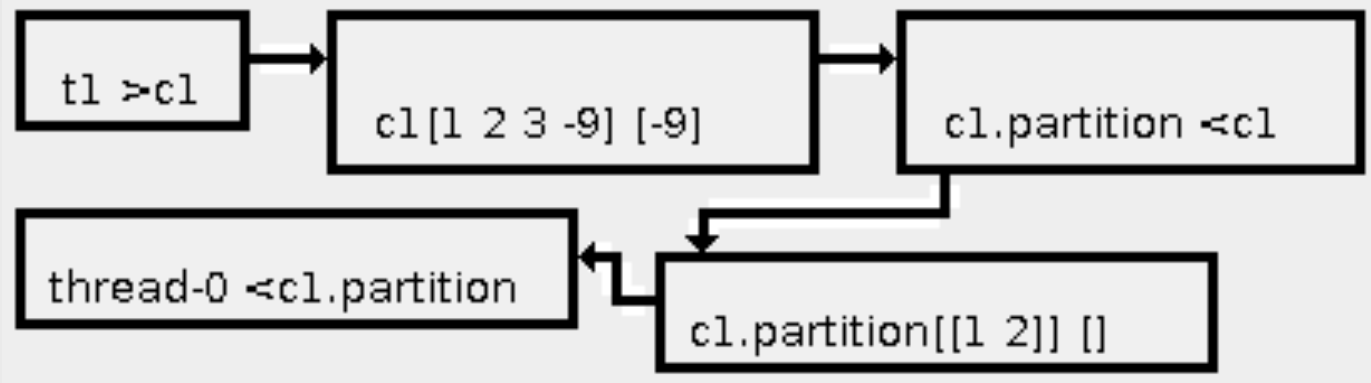
Supported async operators

```
map merge into take unique partition
partition-by
map< map>
mapcat< mapcat>
pipe split
reduce
onto-chan to-chan
mult tap untap untap-all
mix admix unmix unmix-all toggle solo-mode
pub sub unsub unsub-all
filter> remove> filter< remove<
```

partition example

```
(fn []
  (let [c1 (chan-named "c1" 10)
        c2 (partition 2 c1)]
   (go-named "t1"
             (>! c1 1)
             (>! c1 2)
             (>! c1 3)
             (>! c1 -9))
   [(<!! c2)
    (<!! c2)]))

;; [[1 2] [3 -9]]
```

partition example



Other features

- Calls between core.async and step.async

Other features

- Calls between core.async and step.async
- Composing async functions in a step-machine

Other features

- Calls between core.async and step.async
- Composing async functions in a step-machine
- `alts!` to take from many channels
  - putting to many channels *not* supported

Other features

- Calls between core.async and step.async
- Composing async functions in a step-machine
- `alts!` to take from many channels
  - putting to many channels *not* supported
- sliding and dropping buffers

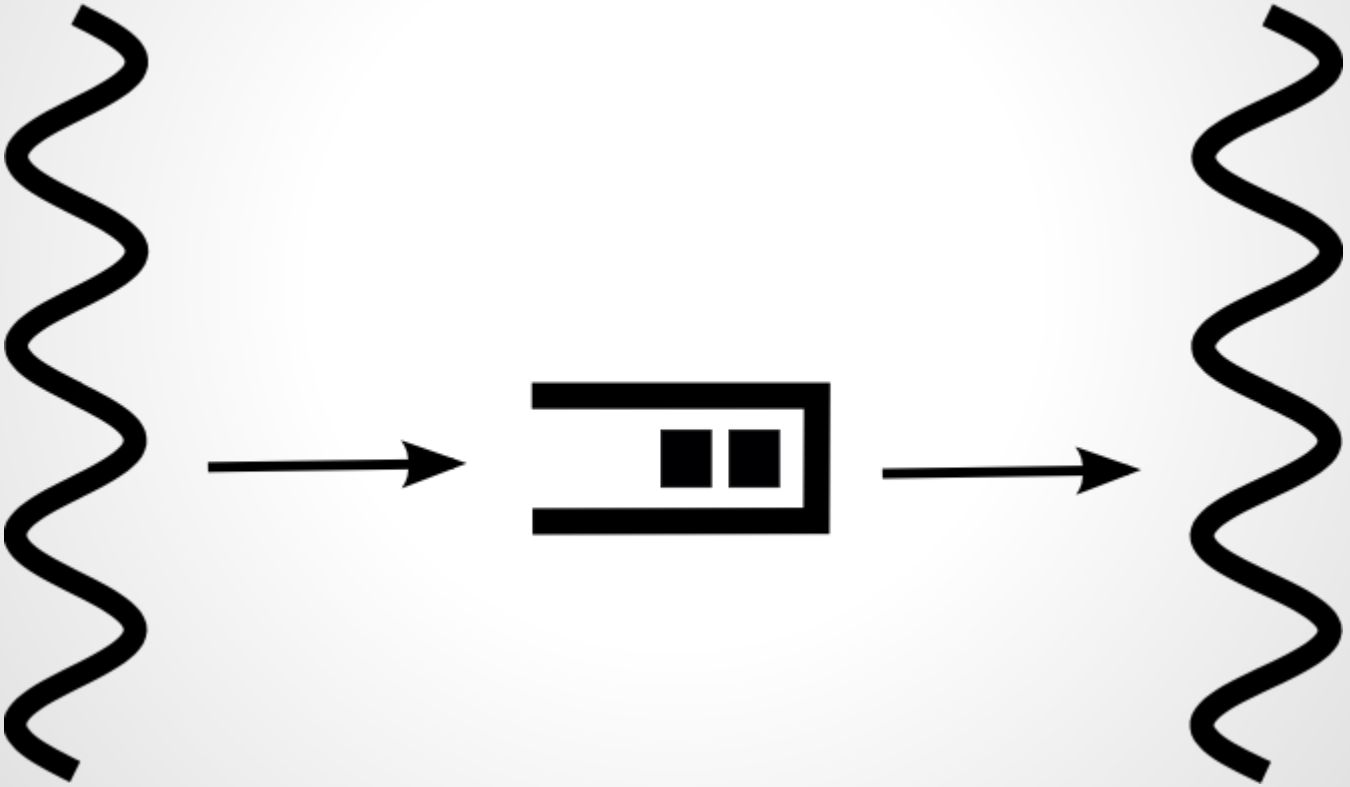
Other features

- Calls between core.async and step.async
- Composing async functions in a step-machine
- `alts!` to take from many channels
  - putting to many channels *not* supported
- sliding and dropping buffers
- conditional breakpoints

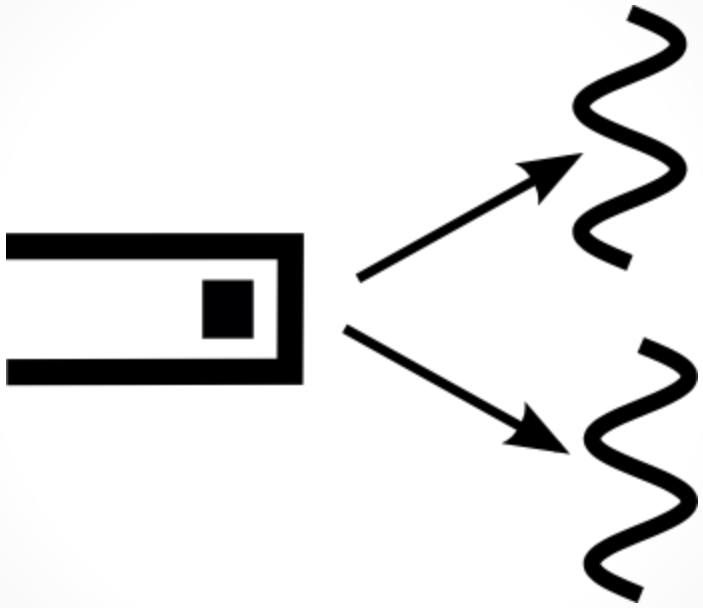
```
(set-breakpoint machine
  (fn [machine-state]
    (not (empty? (:blocked-takes machine-state)))))
```

Transparent & Deterministic Execution

core.async non-determinism



core.async "scheduling"



step.async pluggable scheduler

- deterministic scheduler
  - choose in a repeatable, systematic way
  - default scheduler

step.async pluggable scheduler

- deterministic scheduler
  - choose in a repeatable, systematic way
  - default scheduler

- random scheduler
  - accepts a random seed to reproduce "random" executions

```
(step-machine :rand-seed 123456789)
```

Latent deadlock example

```
(defn f []
  (let [c1 (chan-named "c1" 0)
        c2 (chan-named "c2" 0)]
    (go-named "w1"
              (>! c1 1)
              (<! c2))
    (go-named "w2"
              (<! c1)
              (>! c2 2)
              (<! c1)
              (println "done"))
    (go-named "w3"
              (Thread/sleep 1000)
              (>! c1 3))))

(f) ;; prints "done"
```

Use test.check to find race condition

```
  (:require [clojure.test.check.clojure-test :refer [defspec]]
            [clojure.test.check.generators :as gen]
            [clojure.test.check.properties :as prop]


(defspec test-race 100
  (prop/for-all [r gen/pos-int]
    (= :all-exited
       (let [machine ((step-machine :rand-seed r) f)]
         (step-all machine)
         (quiesce-wait machine)))))
```

Use test.check to find race condition

```
(defspec test-machine 100
  (prop/for-all [r gen/pos-int]
    (= :all-exited
       (let [machine ((step-machine :rand-seed r) f)]
         (step-all machine)
         (quiesce-wait machine)))))
```

```
(test-machine)
;; =>
{:result false, :failing-size 1, :num-tests 2, :fail [1],
:shrunk {:total-nodes-visited 1, :depth 0, :result false,
:smallest [1]}}
```

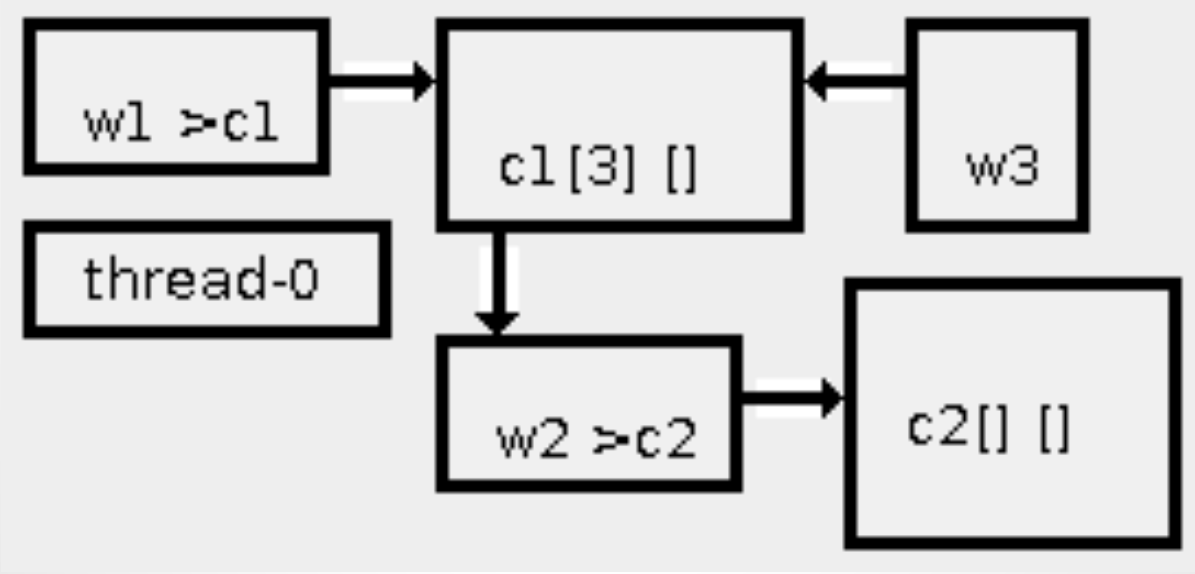
Confirm working seed

```
(let [machine ((step-machine :rand-seed 0) f)]
  (step-all machine)
  (quiesce-wait machine))

;; => :all-exited
```

Confirm failing seed

```
(let [machine ((step-machine :rand-seed 1) f)]
  (step-all machine)
  (quiesce-wait machine))

;; => :all-blocked
```

Visualize failing seed



core.async timeouts

```
(<! (timeout 200))
```

core.async timeouts

```
(get-timeouts machine)

;; => {:start-time 0,
       :duration 10,
       :timeout-name "timeout-30001",
       :timeout-id 30001,
       :thread-name "thread-1"}
```

core.async timeouts

```
(get-timeouts machine)

;; => {:start-time 0,
       :duration 10,
       :timeout-name "timeout-30001",
       :timeout-id 30001,
       :thread-name "thread-1"}

(complete-timeout machine timeout-id)
```

core.async usage

```
(ns demo
  (:require [clojure.core.async
                :refer [go chan >! <!]]))

(let [c1 (chan)
   c2 (chan)]
  (go
   (>! c1 1))
  (go
   (>! c2 2))
  (go
   (println (+ (<! c1) (<! c2)))))
```

step.async usage

```
(ns demo
  (:require [lonocloud.step.async
             :refer [go chan >! <!]]))

(let [c1 (chan)
   c2 (chan)]
  (go
   (>! c1 1))
  (go
   (>! c2 2))
  (go
   (println (+ (<! c1) (<! c2)))))
```

step.async implementation

- Alternative implementation of core.async channels & operators
  - wrapper around "go" macro

step.async implementation

- Alternative implementation of core.async channels & operators
  - wrapper around "go" macro
- core.async focuses on
  - efficiency
  - concurrency

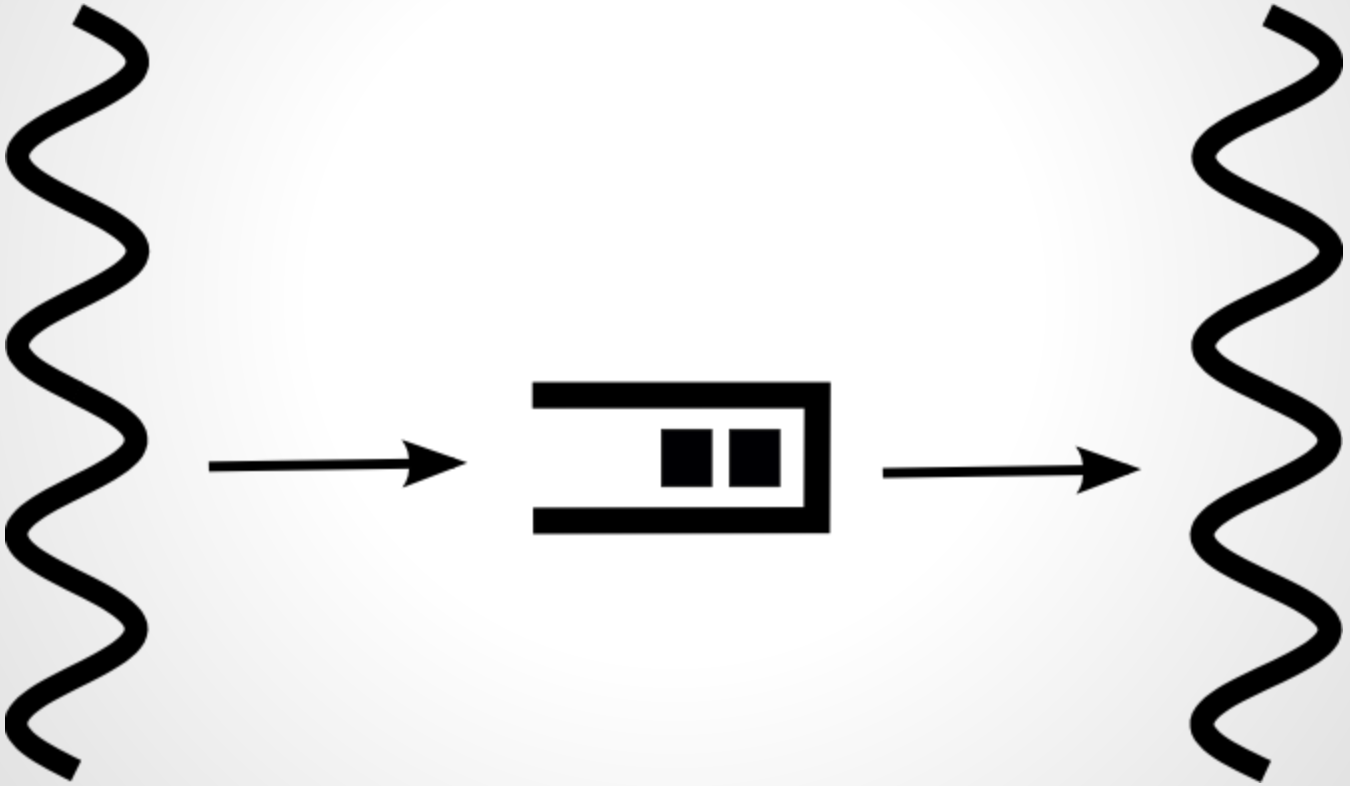
step.async implementation

- Alternative implementation of core.async channels & operators
  - wrapper around "go" macro
- core.async focuses on
  - efficiency
  - concurrency
- step.async focuses on
  - deterministic,
  - transparent, &
  - controlled execution

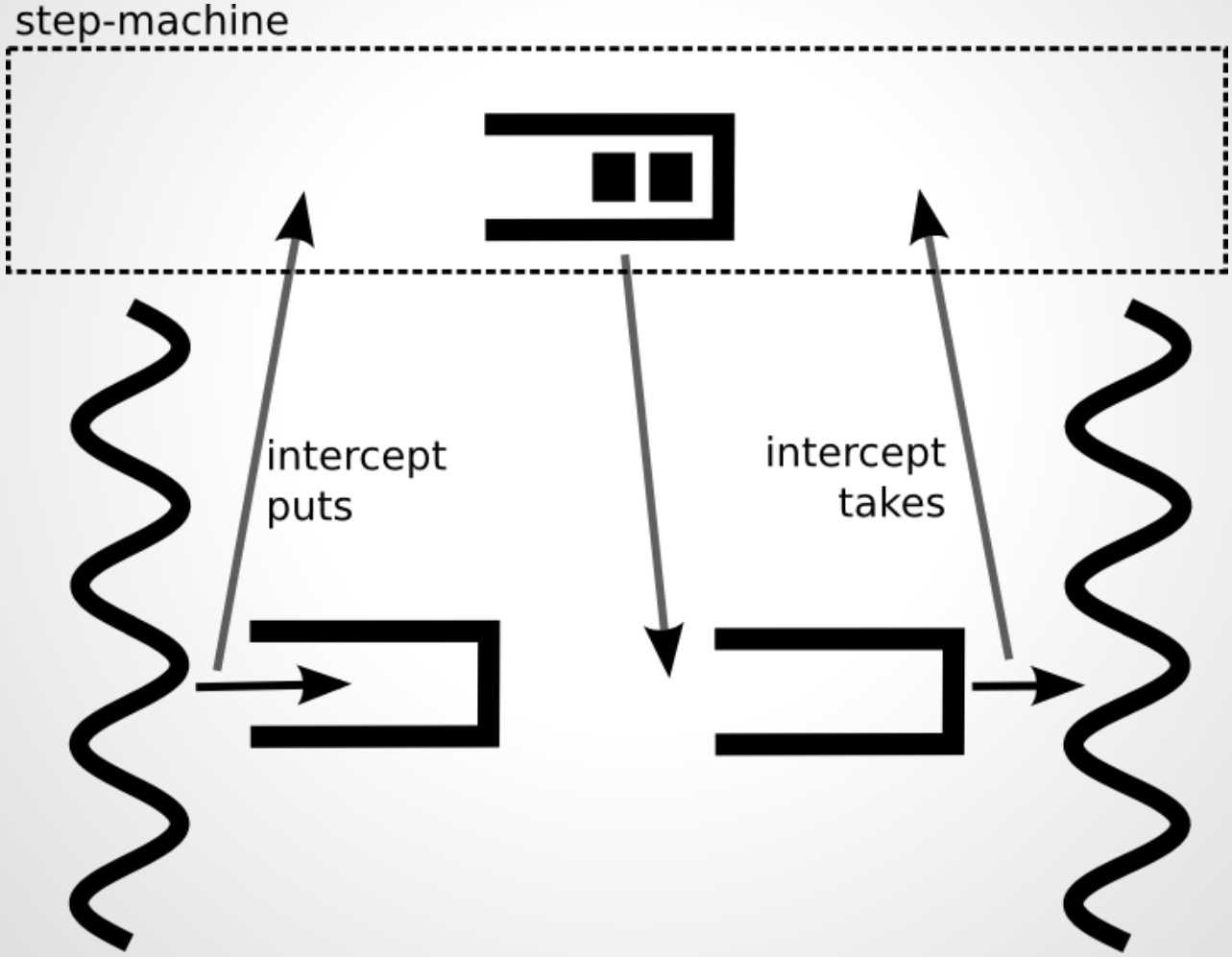
step.async implementation

```
(deftype StepMachineType [channels
                          threads
                          listeners
                          timeouts
                          log
                          result
                          config

  (…
    (dosync …)))
```

core.async threads & channels



step.async threads & channels



Stateful async machine

```
(defn accumulator [in out]
  (go-named "accumulator"
            (loop [sum 0]
              (let [v (<! in)
                    sum (+ sum v)]
                (>! out sum)
                (recur sum)))))
```

```
(let [...
      machine ((step-machine :action-history? true
                             ...
                accumulator in out)]
```

```
(async/>!! in 1)
(step-all machine)
(quiesce-wait machine)

(async/>!! in 2)
(step-all machine)
(quiesce-wait machine)

(async/>!! in 3)
(step-all machine)
(quiesce-wait machine)

;; => 1 3 6
```

Step machine backwards

```
 (let [machine (step-back machine 2)
       ...]
  ;; wait for the step back to complete
  (quiesce-wait machine)

  ;; proceed forward with an alternate input value
  (async/>!! in 4)
  (step-all machine)
  (quiesce-wait machine)

;; => 7
```

https://github.com/LonoCloud/step.async