An oversimplified history of CSP

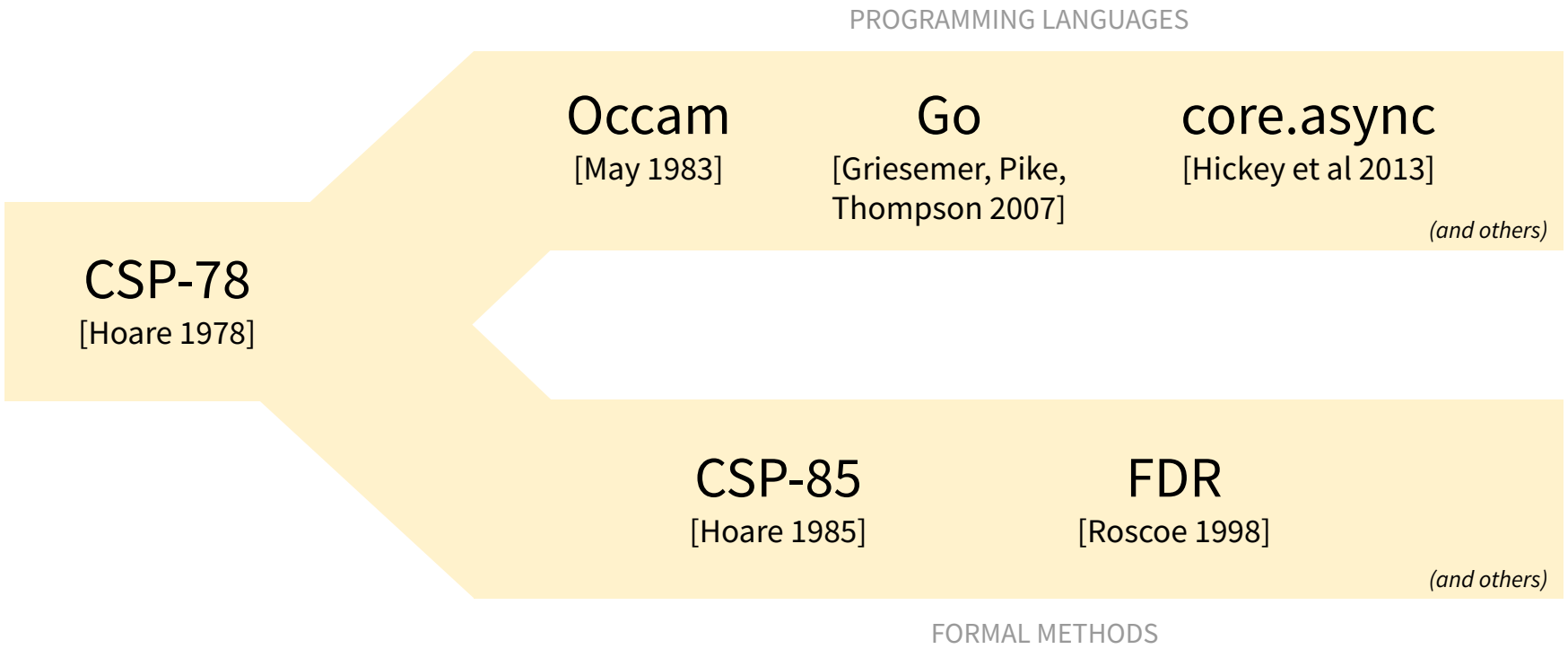
Papers We Love Boston • October 25th, 2016

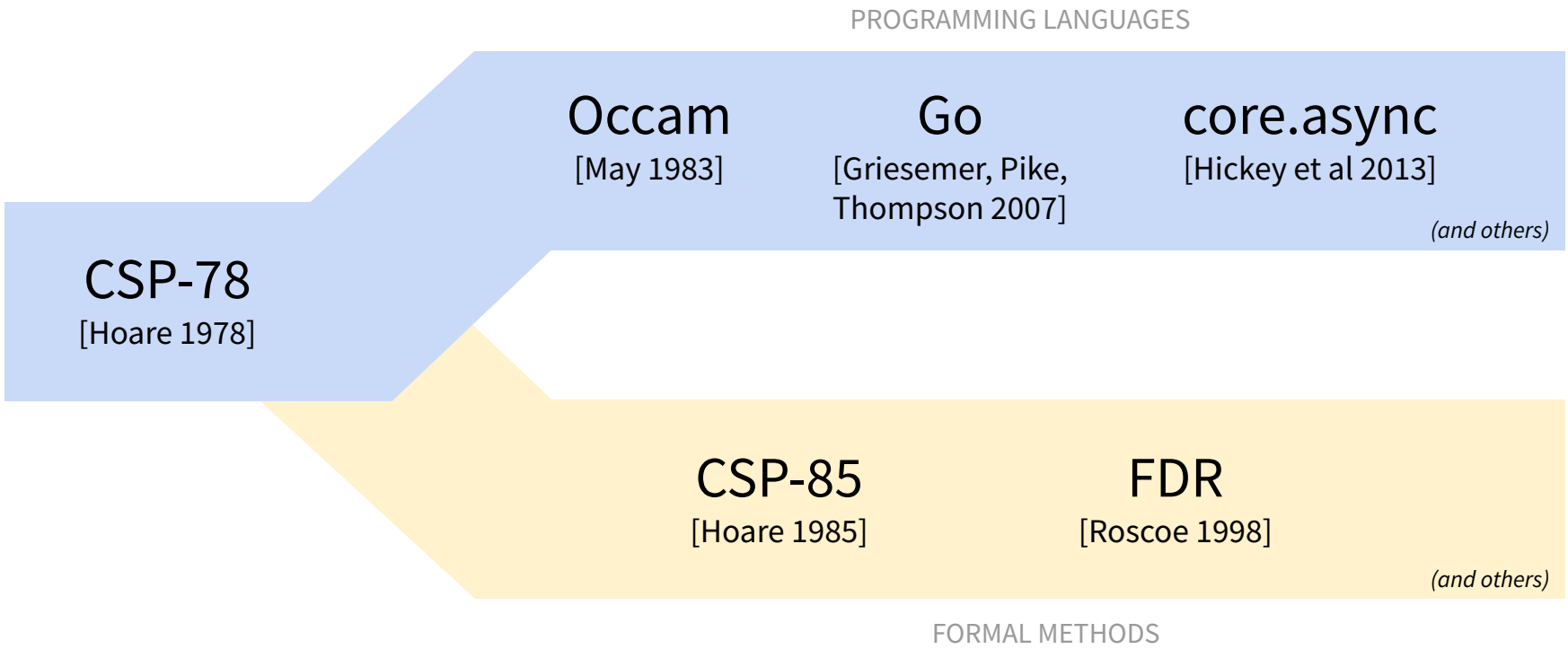
Douglas Creager • @dcreager

CAR Hoare. "Communicating Sequential Processes", CACM 21(8), 1978.
CAR Hoare. Communicating Sequential Processes. Prentice Hall, 1985.
AW Roscoe. The theory and practice of concurrency. Prentice Hall, 1998.

There are two versions of CSP.

They have less in common with each other than you might think!


PROGRAMMING LANGUAGES
Occam
[May 1983]
Go
[Griesemer, Pike, Thompson 2007]
core.async
[Hickey et al 2013]
(and others)
CSP-78
[Hoare 1978]
CSP-85
[Hoare 1985]
FDR
[Roscoe 1998]
(and others)
FORMAL METHODS


PROGRAMMING LANGUAGES
Occam
[May 1983]
Go
[Griesemer, Pike, Thompson 2007]
core.async
[Hickey et al 2013]
(and others)
CSP-78
[Hoare 1978]
CSP-85
[Hoare 1985]
FDR
[Roscoe 1998]
(and others)
FORMAL METHODS


Original CSP

CSP (1978)

Programming Techniques
S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

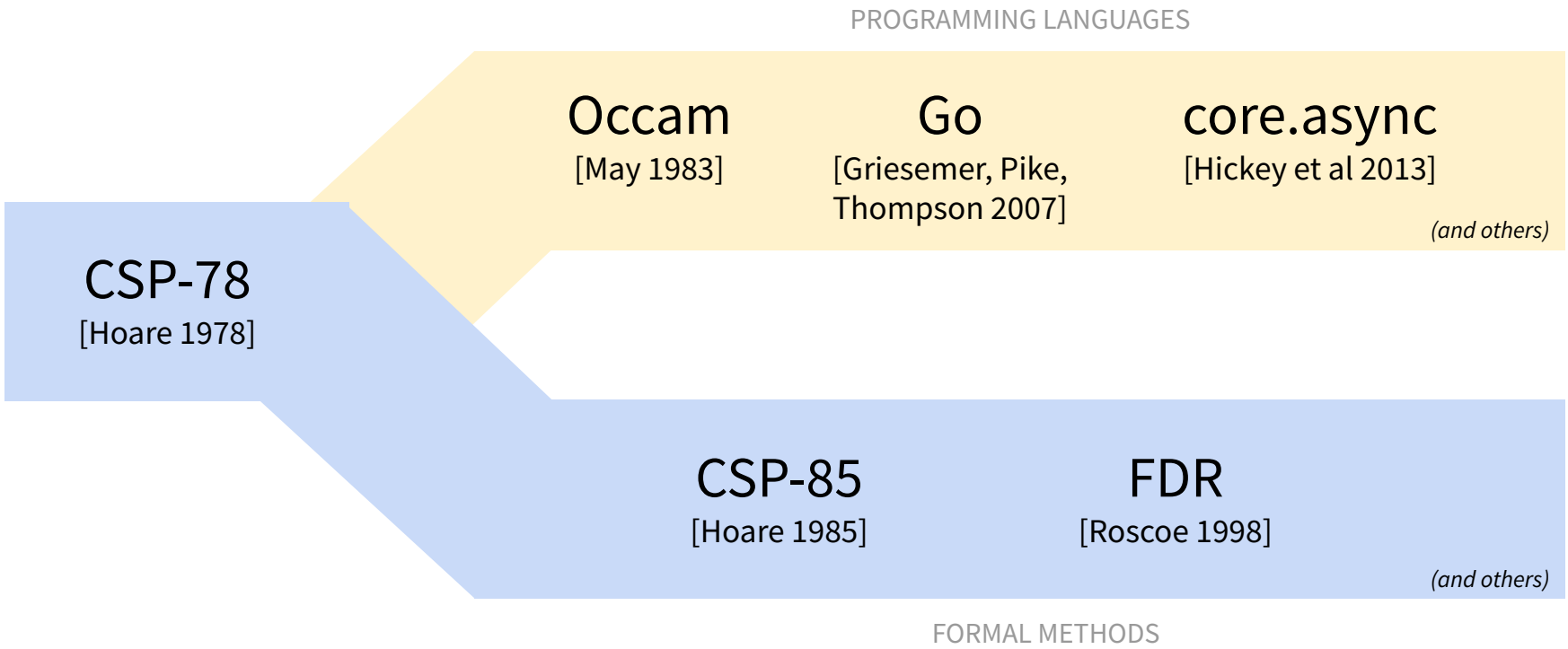
This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays
CR Categories: 4.20, 4.22, 4.32

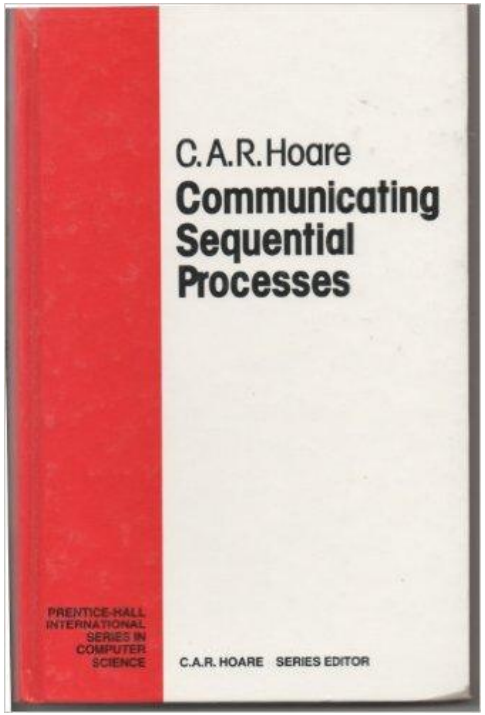
```
[SIEVE(i:1..100)::
   p,mp:integer;
   SIEVE(i − 1)?p;
   print!p;
   mp := p; comment mp is a multiple of p;
*[m:integer; SIEVE(i − 1)?m →
      *[m > mp → mp := mp + p];
      [m = mp → skip
      []m < mp → SIEVE(i + 1)!m
]     ]
||SIEVE(0)::print!2; n:integer; n := 3;
        *[n < 10000 → SIEVE(1)!n; n := n + 2]
||SIEVE(101)::*[n:integer;SIEVE(100)?n → print!n]
||print::*[(i:0..101) n:integer; SIEVE(i)?n → ...]
]
```

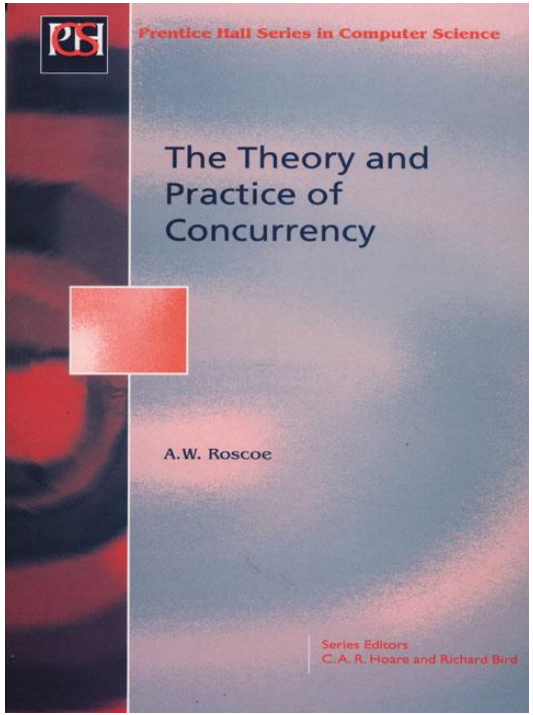
Note: (1) This beautiful solution was contributed by David Gries. (2) It is algorithmically similar to the program developed in [7, pp. 27–32].

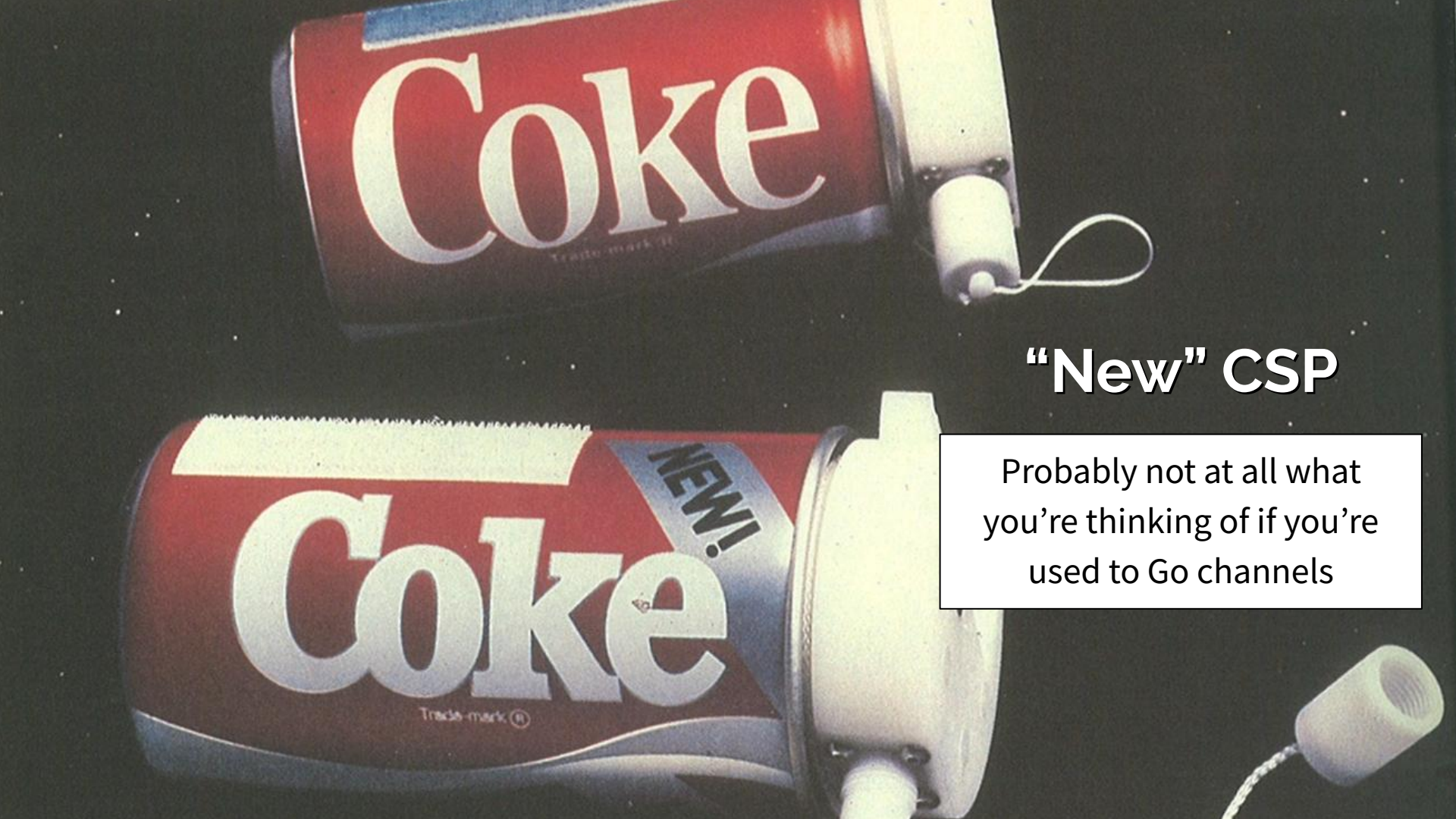
See Adian Cockcroft's PWL SF presentation for more!


PROGRAMMING LANGUAGES
Occam
[May 1983]
Go
[Griesemer, Pike, Thompson 2007]
core.async
[Hickey et al 2013]
(and others)
CSP-78
[Hoare 1978]
CSP-85
[Hoare 1985]
FDR
[Roscoe 1998]
(and others)
FORMAL METHODS

Books We Love?







“New” CSP

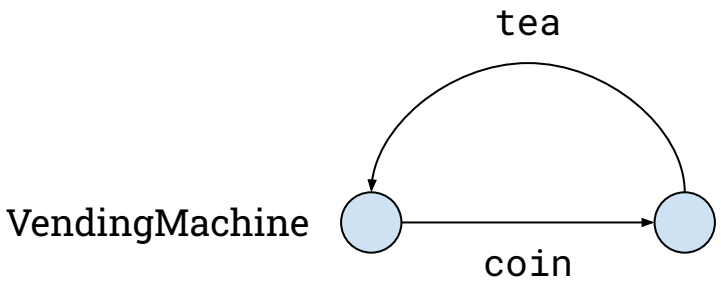
Probably not at all what you’re thinking of if you’re used to Go channels

CSP (1985)

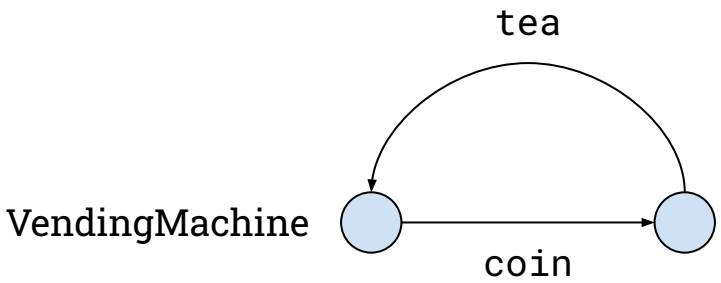
- CSP is a ***formal method***
- CSP is a ***process calculus***
- Describe your system as a ***process***
- Process is entirely defined by what ***events*** it can perform, and when
  - Internal state is ***not*** part of the definition!
  - Contrast with TLA+, Z, B method, etc.


The vending machine
British Style
KO CHA KA DEN
紅茶花伝
ROYAL
MILK TEA
ロイヤル ミルクティー
ウバ茶100%使用
牛乳たっぷり

Labeled transition system (LTS)



Traces



<coin>
<coin, tea>
<coin, tea, coin>
<coin, tea, coin, tea>
⋮

<tea>?

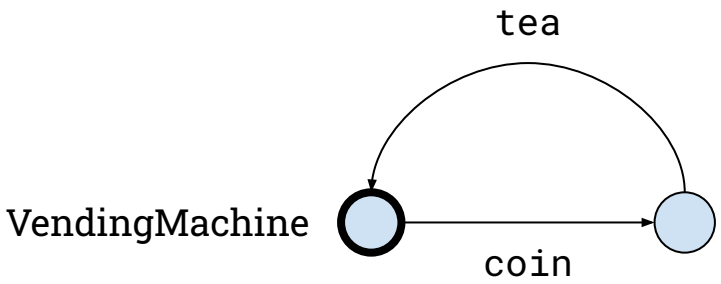
The environment

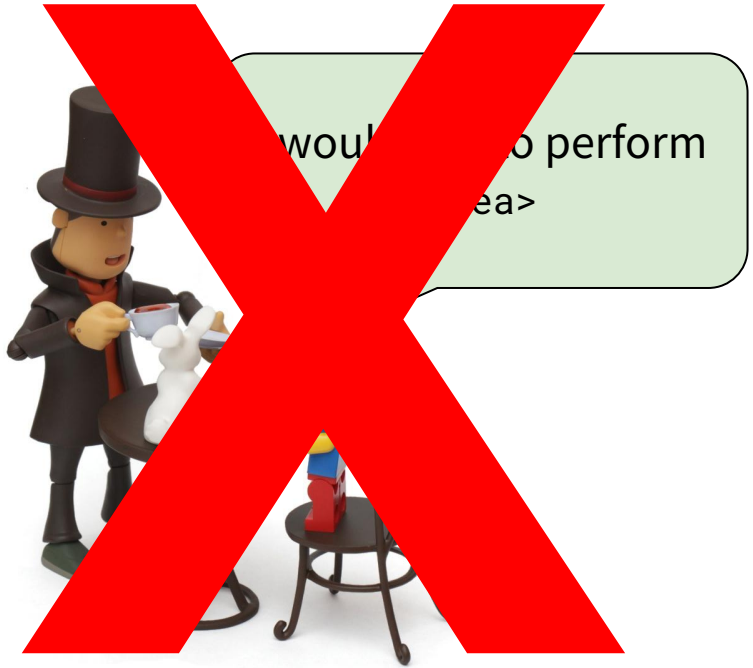
- Processes don’t do anything on their own
- We only care how they interact with something else — the ***environment***
  - Sometimes another process (via the *parallelism* operators)
  - Or you, exploring the process!

Rendezvous model

Sure, whatever

Rendezvous model





DENIED!

“Events”

Not message passing!

No sender, no receiver!

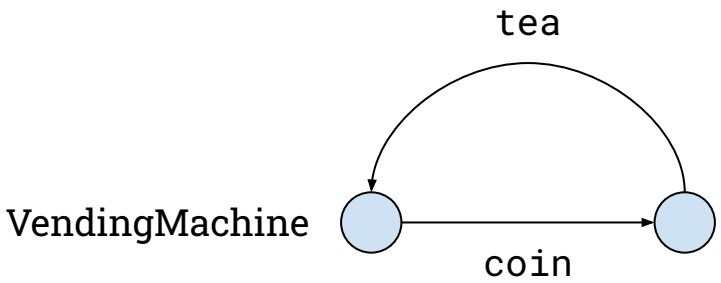
What does “`coin`” even mean?

To `VendingMachine`, it means that the machine detected that a coin was inserted into the coin slot.

To me (the environment), it means I inserted a coin into the coin slot.

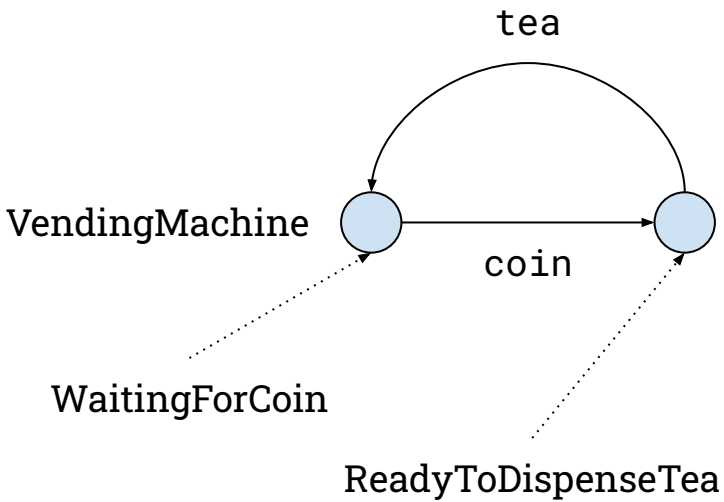
The rendezvous model means those two interpretations are inseparable!

In CSP! (take one)




```
channel coin
channel tea

VendingMachine =
    coin → tea → VendingMachine
```

In CSP! (take two)




```
channel coin
channel tea

WaitingForCoin =
    coin → ReadyToDispenseTea
ReadyToDispenseTea =
    tea → WaitingForCoin

VendingMachine = WaitingForCoin
```

Follow along at github.com/dcreager/csp-models


Coffee!
BRASILIA
Mléko
illy

Tea and coffee

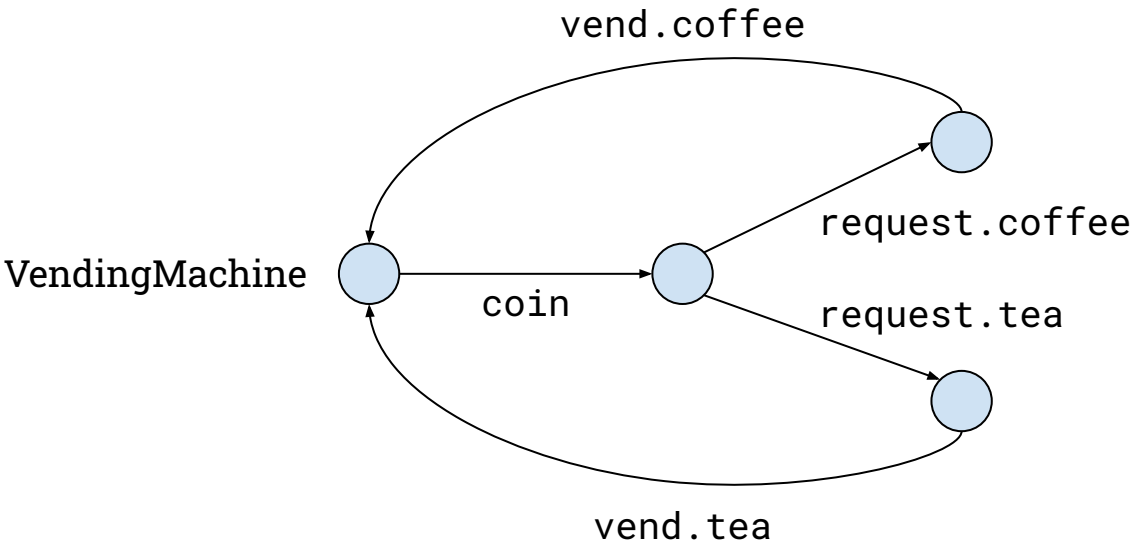
```
datatype Product = tea | coffee
channel coin
channel request : Product
channel vend : Product

WaitingForCoin = coin → WaitingForRequest
WaitingForRequest =
    request!tea → ReadyToDispenseTea
        □
    request!coffee → ReadyToDispenseCoffee
ReadyToDispenseTea = vend!tea → WaitingForCoin
ReadyToDispenseCoffee = vend!coffee → WaitingForCoin

VendingMachine = WaitingForCoin
```

Multiple beverages

Events that carry data

“External” choice

Tea and coffee



<coin, request.tea, vend.coffee>

not allowed!


DEMO!

Safety checks

Do you want to manually list all of
the traces that you want to check?

NO!

Of course not, that’s what computers are for!

Refinement

Refinement

- Describe your system (“Impl”)
- Describe the property you want your system to have (“Spec”)

(Both using CSP!)

- Perform a ***refinement check*** (“Spec $\sqsubseteq$ Impl”)
  - Everything that Impl does, Spec must also be able to do
  - i.e., Everything that Impl does is allowed by Spec

- CSP has several ***semantic models***
  - which define what “allowed by” means

Traces refinement

“The machine should always dispense the beverage that the customer asked for.”

```
AlwaysVendsRequestedProduct = let
    NoProductRequested = request?product → ProductRequested(product)
    ProductRequested(product) = vend!product → NoProductRequested
within
    NoProductRequested
```

```
assert AlwaysVendsRequestedProduct ⊑T VendingMachine
```


DEMO!

Traces refinement

“The machine should always dispense the beverage that the customer asked for.”

```
AlwaysVendsRequestedProduct = let
    NoProductRequested = request?product → ProductRequested(product)
    ProductRequested(product) = vend!product → NoProductRequested
within
    NoProductRequested
```

~~assert AlwaysVendsRequestedProduct $\sqsubseteq_T$ VendingMachine~~

assert AlwaysVendsRequestedProduct $\sqsubseteq_T$ (VendingMachine \ {coin})


Beer

Tea and coffee and beer

```
datatype Product = tea | coffee | beer
channel coin
channel request : Product
channel vend : Product

WaitingForCoin = coin → WaitingForRequest
WaitingForRequest =
    request!tea → ReadyToDispenseTea
        □
    request!coffee → ReadyToDispenseCoffee
ReadyToDispenseTea = vend!tea → WaitingForCoin
ReadyToDispenseCoffee = vend!coffee → WaitingForCoin

VendingMachine = WaitingForCoin
```

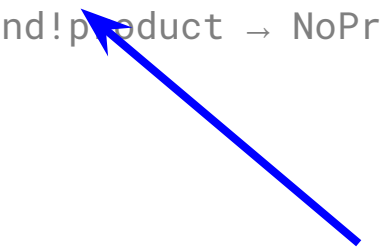
Beer!

No beer!

“We live in a universe where beer exists, but this particular vending machine doesn’t sell any.”

Traces refinement for beer

```
AlwaysVendsRequestedProduct = let
    NoProductRequested = request?product → ProductRequested(product)
    ProductRequested(product) = vend!product → NoProductRequested
within
    NoProductRequested
```



Implicit beer!

```
assert AlwaysVendsRequestedProduct ⊑T (VendingMachine \ {coin})
```

Does the refinement succeed?


SAMSUNG
DEMO!

Traces refinement for beer

```
assert AlwaysVendsRequestedProduct ⊑T (VendingMachine \ {coin})
```

Does the refinement succeed?

YES!

```
assert AlwaysVendsRequestedProduct ⊑T STOP
```

Does the refinement succeed?

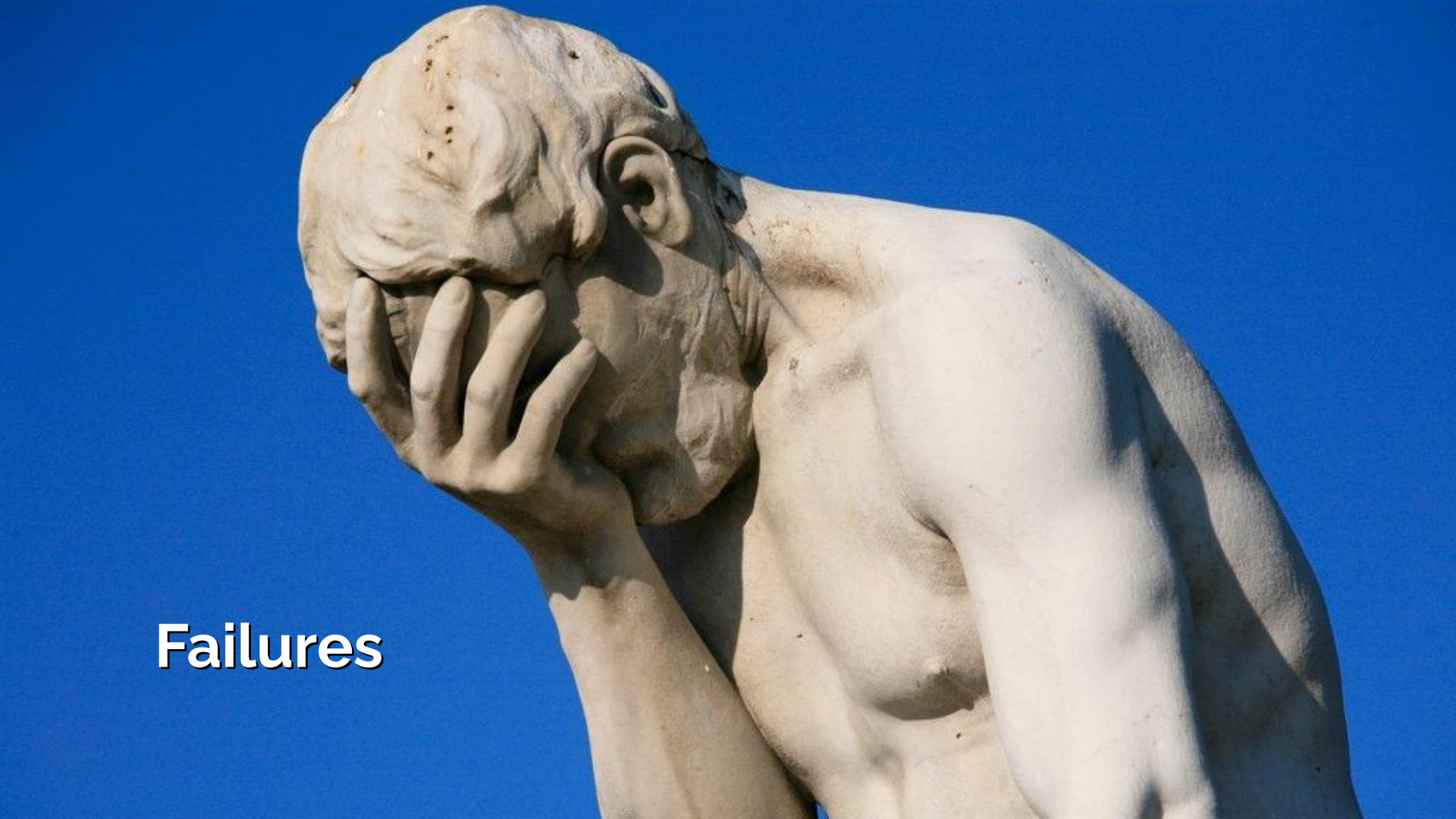
YES!

“***If the customer requests beer***, the machine must vend beer.”

Traces refinement

- The traces of Impl must be a subset of the traces of Spec
  - traces(Impl) ⊆ traces(Spec)

- This only lets you check ***safety properties***
  - i.e., “this bad thing is not allowed to happen”


Failures

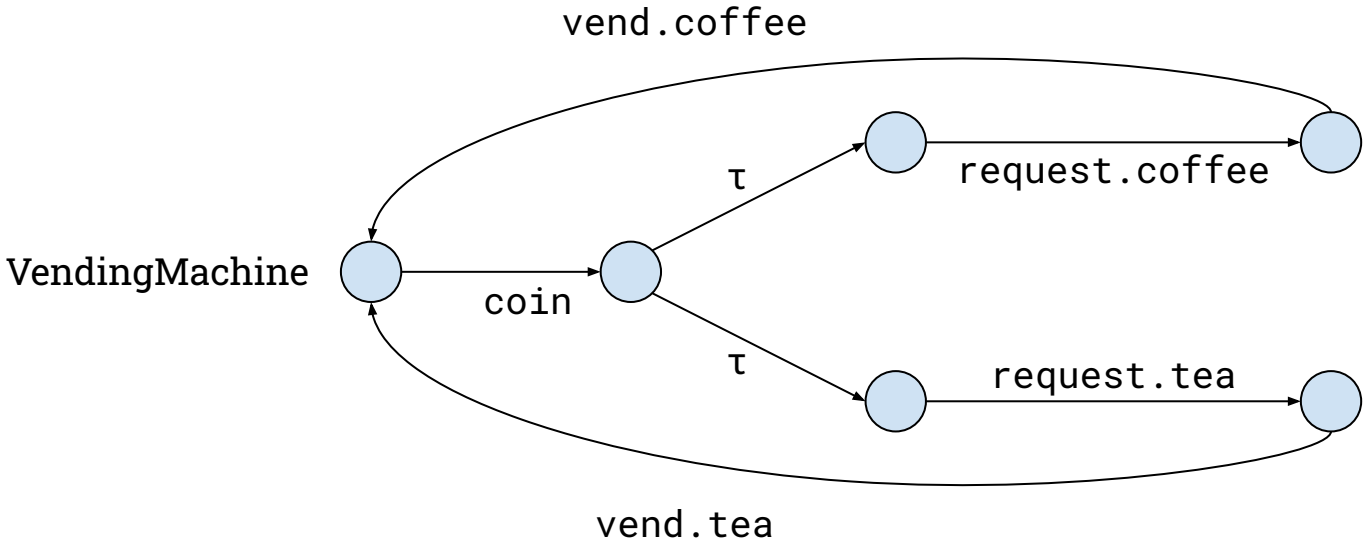
Failures

```
datatype Product = tea | coffee
channel coin
channel request : Product
channel vend : Product

WaitingForCoin = coin → WaitingForRequest
WaitingForRequest =
    request!tea → ReadyToDispenseTea
        □
    request!coffee → ReadyToDispenseCoffee
ReadyToDispenseTea = vend!tea → WaitingForCoin
ReadyToDispenseCoffee = vend!coffee → WaitingForCoin

VendingMachine = WaitingForCoin
```

Back to no beer

“Internal” choice

Failures




```
<coin>, {request.coffee}
<coin>, {request.tea}
```

Failures refinement

“The customer can choose whichever beverage they want.”

```
CustomerCanChooseAnything = request?product → CustomerCanChooseAnything
```

```
assert CustomerCanChooseAnything ⊑F VendingMachine
```

DEMO!


What's next?

We didn't cover:

- Divergences
  - i.e., infinite loops
- Parallelism operators
  - useful for modularity
  - oh and also parallelism
- Timing guarantees
  - no explicit notion of time in CSP
  - [but that can be a good thing!]
- Lots of other cool stuff!