Communication

Concurrency and Distributed Systems

January 2023

Contents

Channels

Parameters in event names can serve two purposes:

- to represent data being passed;
- to indicate an event performed by a process instance.

They are written after a dot: `c.v`

Parameter expressions can take values from built-in or user-defined datatypes.

Built-in datatypes include sets and sequences.

compare: process parameters

Declarations

If `c` is a name and `T` is a type-valued expression, then

```
channel c : T
```

introduces the set of all events of the form `c.t`, where `t` is a value of type `T`.

We may introduce more than one channel in the same declaration: for example,

```
channel c, d : T
```

would introduce two channels `c` and `d`, both of type `T`.

Built-in datatypes

The built-in type `Int` represents the set of positive and negative integers: $-2^{31} + 1 \mathinner{\ldotp\ldotp} 2^{31} - 1$.

The built-in type `Bool` represents the set of Boolean values: `True` and `False`.

Example

The following declaration introduces a channel `insert` that can carry any integer value

```
channel insert : Int
```

User-defined datatypes

The declaration

```
datatype NewType = name1 | name2 | name3
```

introduces

- a new datatype `NewType`
- three constants, `name1`, `name2`, and `name3`, as values of that datatype

Constructors

The declaration

```
datatype NewType = name1 | name2 | label.OtherType
```

introduces

- a new datatype `NewType`
- two constants, `name1` and `name2`, as values of that datatype
- a set of values of the form `label.t`, where `t` is any value of the datatype `Type`

types may be recursive – `Type` may be `NewType`

Example

```
datatype Tree = Leaf.Int | Node.Int.Tree.Tree
```

```
Leaf.3 :: Tree
```

```
Node.5.(Leaf.3).(Leaf.5) :: Tree
```

More components

An event on a channel may have several components. The definition

```
channel c : T . U
```

introduces a set of events

```
{ c.t.u | t <- T, u <- U }
```

Example

```
Floor = {0..2}
datatype Person = pA | pB | pC

channel call : Floor . Person
```

Productions

If we wish to refer to the set of all events associated with a channel, then we may use the productions operator. If `c` is a channel, then

```
{| c |}
```

is the set of all events `c.x` matching the declaration of `c`.

More generally, if `c` is a channel with multiple components, then

```
{| c.t |}
```

is the set of all events `c.t.x` matching the declaration.

Example

```
{| floorRequest |} =
  { floorRequest.0.pA, floorRequest.0.pB,
    floorRequest.0.pC, floorRequest.0.pD,
    floorRequest.1.pA, floorRequest.1.pB,
    floorRequest.1.pC, floorRequest.3.pD,
    ... }
```

Input and output

If two or more processes share a channel then, for each component:

- some processes may allow any value
- some processes may insist on exactly one value

Input and output

If the transaction described by the channel occurs, then for each component

- all participating processes agree on the value chosen
- their future behaviour may depend upon that value

Input

If the transaction described by the channel occurs, then a process allowing any value for some component

- now has a value for that component to work with
- a value arising from the behaviour of the other components

Input

We may use an indexed external choice to describe a process that allows any value for a component.

If `c` is a channel declared as

```
channel c : V
```

and `P` is a process parameterised by values of type `V`, then

```
[] v : V @ c.v -> P(v)
```

allows any event of the form `c.v`, and may be referred to as input on channel `c`.

Example

```
LiftController = ...

  call?f?p -> ...
```

Input

We may use `?` to indicate an indexed external choice for a particular component of a channel.

If `c` is a channel declared as

```
channel c : V
```

and `P` is a process parameterised by values of type `V`, then

```
c?v -> P(v)
```

allows any event of the form `c.v`, and may be referred to as input on channel `c`.

if we want to allow only `c.v` for some subset `W` of `V`, then we can write `c?v:W -> P(v)`

Output

We may use `!` to indicate insistence upon a particular value for a particular component:

If `c` is a channel declared as

```
channel c : V
```

and `P` is a process, and `E` is an expression with value `val` of type `V`

```
c!E -> P
```

will allow only the event `c.val`, and may be referred to as output on channel `c`.

Example

```
datatype MotorInstruction = up | down | stop

channel motor : MotorInstruction

LiftController =
  ...

  motor!down -> ...
```

Output

For a single component, `!` behaves exactly as (`.`).

However, for a channel with multiple components, the expression

```
c?x.y -> ...
```

denotes a process that will allow any value of `x` and any value of `y`: that is, an external choice over pairs of values.

If we wish to describe a process that will allow any value of one component but will insist upon a particular value for the next, then we need to use `!`.

```
c?x!y -> ...
```

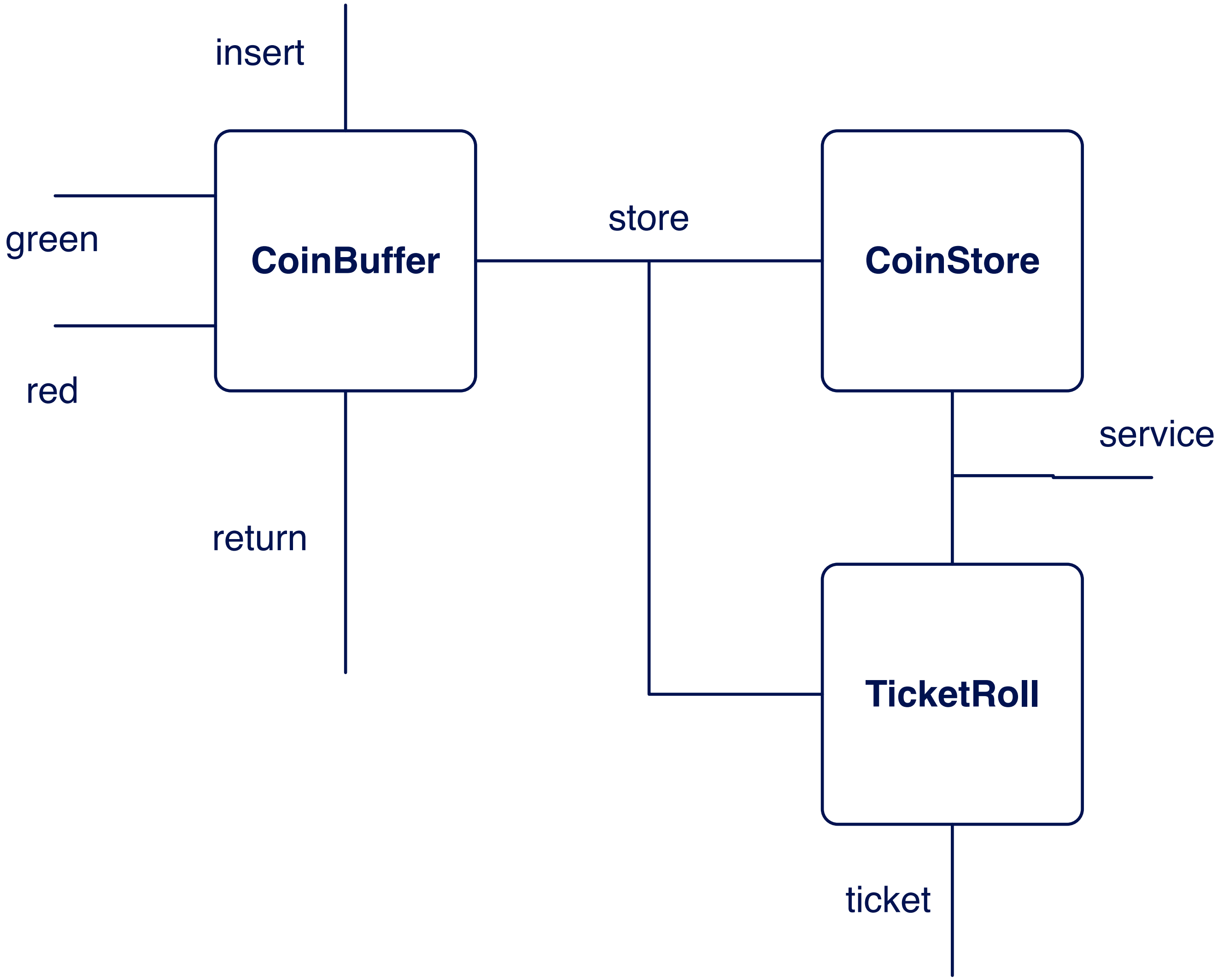
Example

```
Person(pA) = ...

  call?f!pA -> ...
```

Example

```
channel insert : Coin
channel green, red, ticket
channel store, return : {0..400}
channel service

Coin = {10,20,50,100}
price = 100
bufferCap = 4
storeCap = 8
rollSize = 2
```

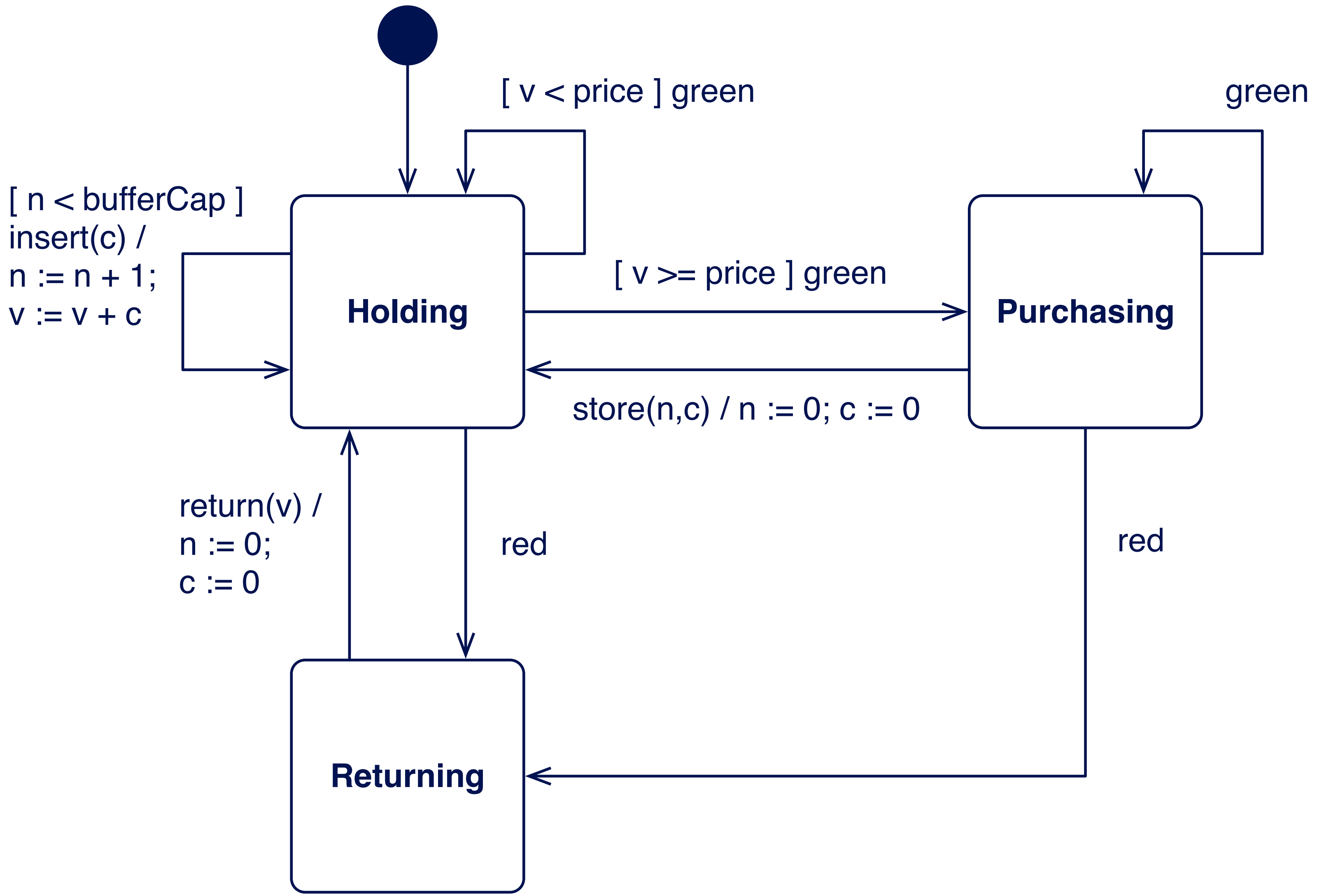
we write `0..400` rather than `Int`...


insert
green
CoinBuffer
red
return
store
CoinStore
service
TicketRoll
ticket

Example

```
aCoinBuffer = {| insert, green, red, store, return |}

CoinBuffer =
  let
    Holding(n,v) =
      n < bufferCap &
        insert?c -> Holding(n+1,v+c)
      []
      green -> ( if v >= price then
                   Purchasing(n,v)
                 else
             Holding(n,v) )
      []
      red -> Returning(v)
```

```
    Purchasing(n,v) =
      green -> Purchasing(n,v)
      []
      red -> Returning(v)
      []
      store!n -> Holding(0,0)

    Returning(v) =
      return!v -> Holding(0,0)
  within
    Holding(0,0)
```


[v < price] green
green
[n < bufferCap]
insert(c) /
n := n + 1;
v := v + c
Holding
[v >= price] green
Purchasing
store(n,c) / n := 0; c := 0
return(v) /
n := 0;
c := 0
red
red
Returning

```
aCoinStore = {| store, service |}

CoinStore =
  let
    Holding(m) =
      store?n:{1..(storeCap-m)} -> Holding(m+n)
      []
      service -> Holding(0)
  within
    Holding(0)
```

```
aTicketRoll = {| store, service, ticket |}

TicketRoll =
  let
    Holding(t) =
      (t > 0) & store?n -> ticket -> Holding(t-1)
      []
      service -> Holding(rollSize)
  within
    Holding(rollSize)
```

```
TicketMachine =
  CoinBuffer
  [ aCoinBuffer || union(aCoinStore,aTicketRoll) ]
  ( CoinStore [ aCoinStore || aTicketRoll ] TicketRoll )

assert TicketMachine :[deadlock free]

TicketMachineWithoutService =
  TicketMachine [ aTicketMachine || {service} ] STOP

assert TicketMachineWithoutService :[deadlock free]
```

Scope

Our language of processes is declarative.

Each variable gets its value at the point at which it is declared, and retains that value for the scope of that declaration.

Example

```
Copy = in?x -> out!x -> Copy
```

Indexed parallel

If `P(x)` is a process-valued expression with parameter `x`, and `aP(x)` is a set-valued expression with the same parameter, then

```
|| x : X @ [aP(x)] P(x)
```

is a process that behaves as a combination of processes `P(x)`, one for each value of `x` in `X`, in which

- events from `aP(x)` are allowed only when `P(x)` allows them

Example

```
channel enterLift, exitLift : Floor . Person

aFloor(i) = {| enterLift.i, exitLift.i, ... |}

Floor(i) =
  let
    Holding(P) =
      enterLift!i?p:P -> ...
    ...

Floors = || i : {0..10} @ [aFloor(i)] Floor(i)
```

Example

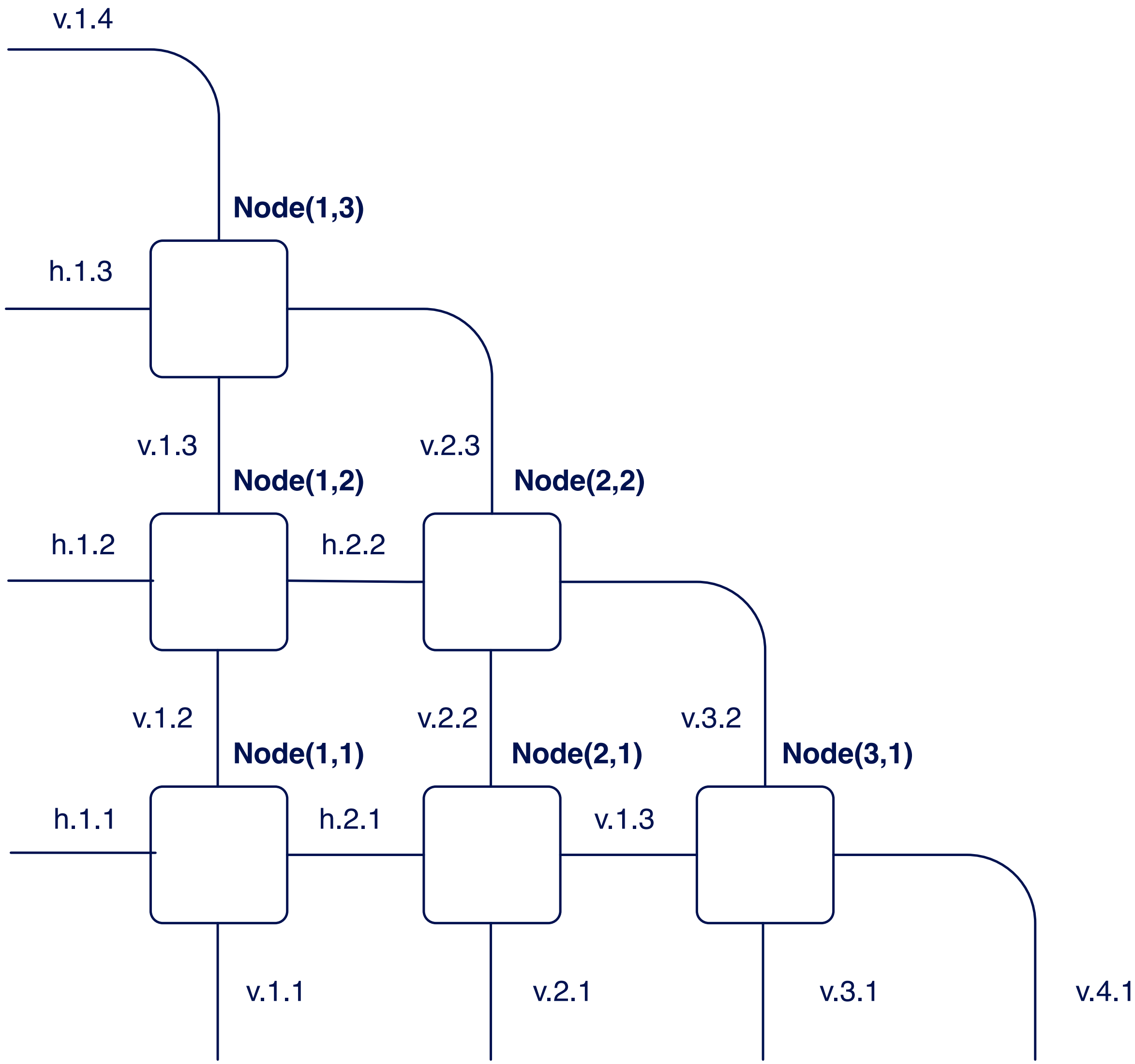
```
Node(i,j) =
  h.i.j?x -> v.i.(j+1)?y -> v.i.j!min(x,y) ->
    if i + j == N then
      v.(i+1).j!max(x,y) -> Node(i,j)
    else
      h.(i+1).j!max(x,y) -> Node(i,j)
```

original

Example

```
aNode(i,j) =
   {| h.i.j, v.i.(j+1), v.i.j,
      if i + j == N then v.(i+1).j else h.(i+1).j |}

Array(N) =
  || i : {1..N}, j : {1..N}, i+j <= N @ [aNode(i,j)] Node(i,j
```

Example


v.1.4
Node(1,3)
h.1.3
v.1.3
v.2.3
Node(1,2)
Node(2,2)
h.1.2
h.2.2
v.1.2
v.2.2
v.3.2
Node(1,1)
Node(2,1)
Node(3,1)
h.1.1
h.2.1
v.1.3
v.1.1
v.2.1
v.3.1
v.4.1

Example

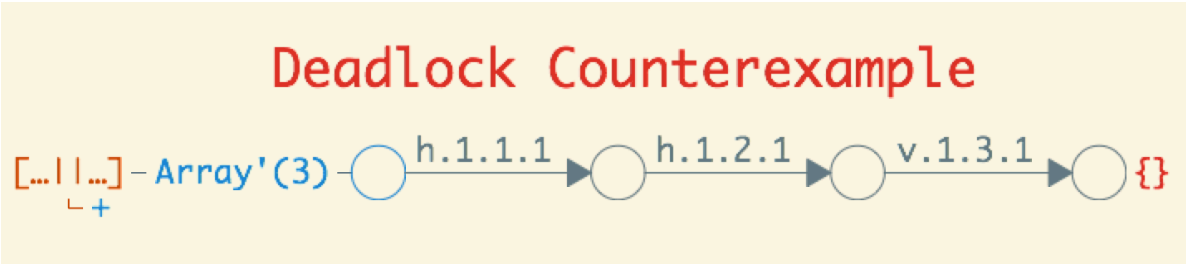
```
aInput = {| h.1.1, h.1.2, h.1.3, v.1.4 |}
Input = h.1.1!4 -> h.1.2!3 -> h.1.3!2 -> v.1.4!1 -> Input

aOutput = {| v.1.1, v.2.1, v.3.1, v.4.1 |}
Output = v.1.1!1 -> v.2.1!2 -> v.3.1!3 -> v.4.1!4 -> Output

aArray = {| v.i.j, h.i.j | i <- {1..4}, j <- {1..4} |}

ArrayIO = (Input [ aInput || aArray ] Array(4))
                 [ aArray || aOutput ] Output

assert ArrayIO :[deadlock free]
```

Example

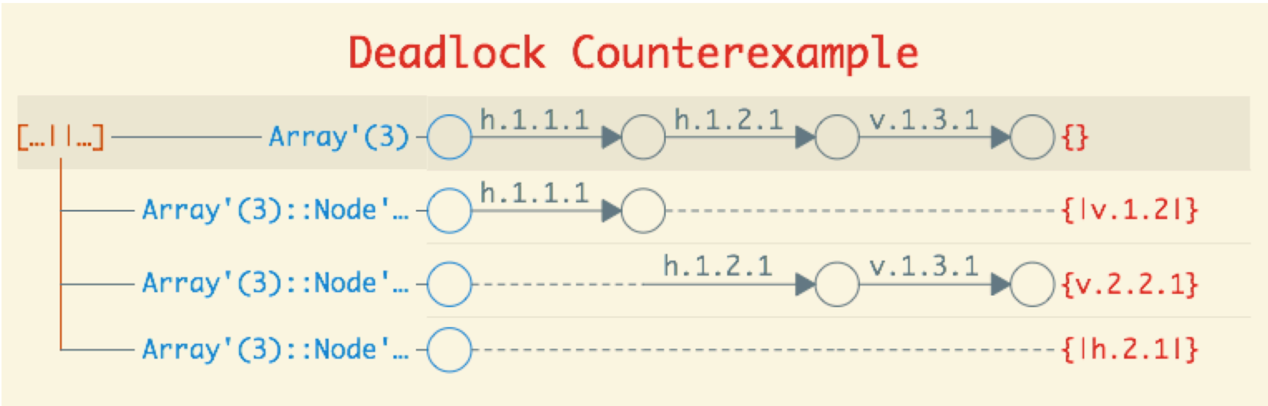
```
Node'(i,j) =
  h.i.j?x -> v.i.(j+1)?y ->
    if i + j == N then
      v.(i+1).j!max(x,y) -> v.i.j!min(x,y) -> Node'(i,j)
    else
      h.(i+1).j!max(x,y) -> v.i.j!min(x,y) -> Node'(i,j)
```

alternative

Example



Example


Deadlock Counterexample
[…||…]
Array'(3)
h.1.1.1
h.1.2.1
v.1.3.1
{}
Array'(3)::Node'…
h.1.1.1
{|v.1.2|}
Array'(3)::Node'…
h.1.2.1
v.1.3.1
{v.2.2.1}
Array'(3)::Node'…
{|h.2.1|}

Summary

- Channels
- Input and output
- Indexed parallel

Index