Hiding

Concurrency and Distributed Systems

November 2023

Contents

Abstraction

abstraction, *n.*

1. The action of taking something away; the action or process of withdrawing or removing something from a larger quantity or whole.

2. The process of isolating properties or characteristics common to a number of diverse objects, events, etc., without reference to the peculiar properties of particular examples or instances.

(Oxford English Dictionary)

Aside

abstraction-monger, *n.* *depreciative*

a scholar, thinker, etc., who shows a preference for abstract concepts over practical judgements or empirical facts

1834 S. T. Coleridge *Marginalia* II. 426 …a favourite word with the Alexandrine Abstraction-mongers.

1856 R. A. Vaughan *Hours with Mystics* II. viii. viii. 97 His philosophy is never that of the abstraction-monger.

1985 S. Lukes tr. H. Bergson in *Emile Durkheim* i. ii. 52, I have always thought that he would be an abstraction-monger.

Internal events

If a parallel combination of processes is used to describe a system, then some of the events shared between these processes may not represent external transactions.

To obtain a description of the behaviour of the system at its external interface, we may find it useful to remove these events from consideration, taking into account that

- these internal events may occur as soon as all of the processes involved are ready
- we cannot constrain their occurrence, nor do we see when they occur

Example

We may wish to describe the behaviour of process `Array(4)` not in terms of the whole alphabet

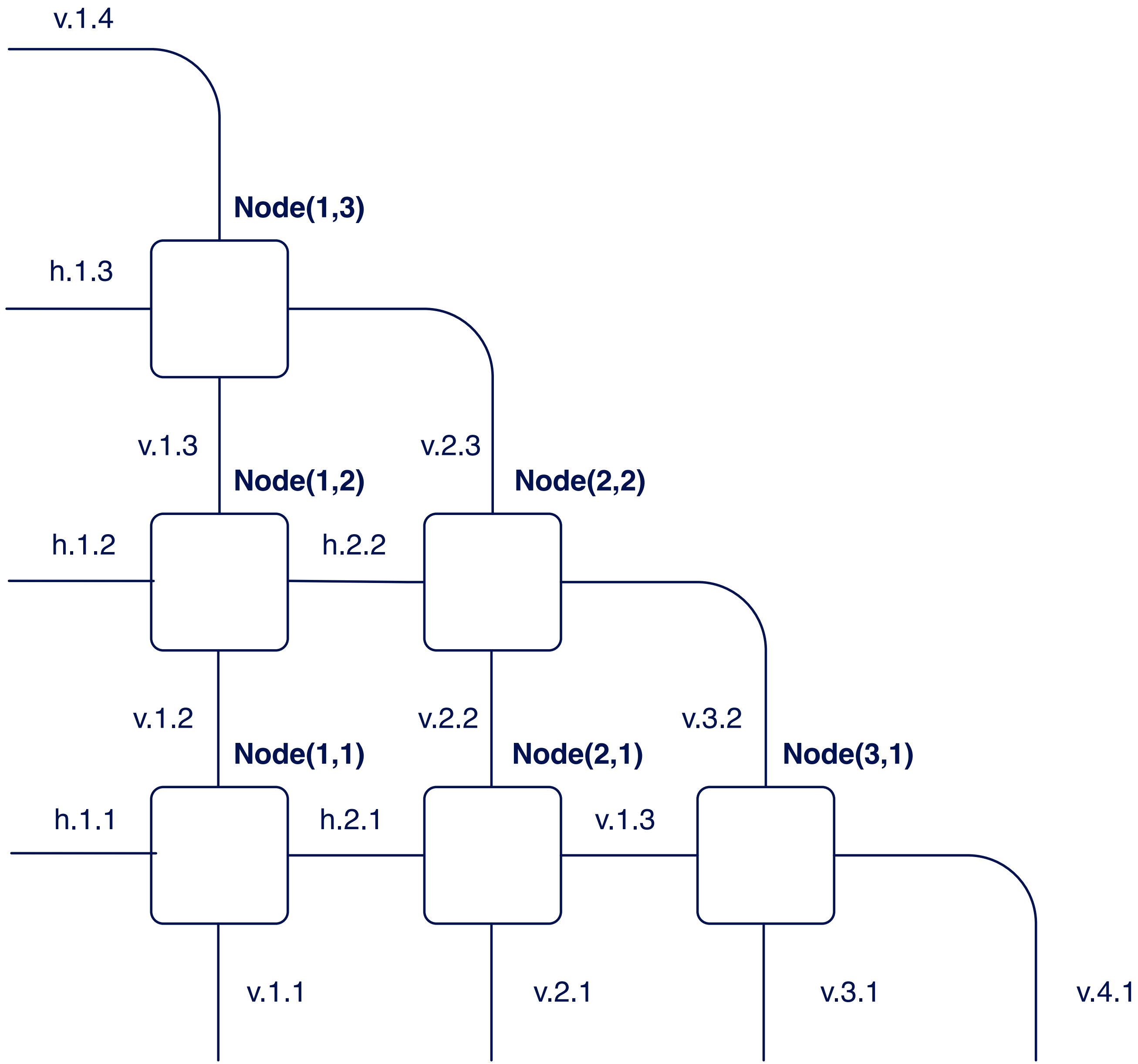
```
aArray = {| v.i.j, h.i.j | i <- {1..4}, j <- {1..4} |}
```

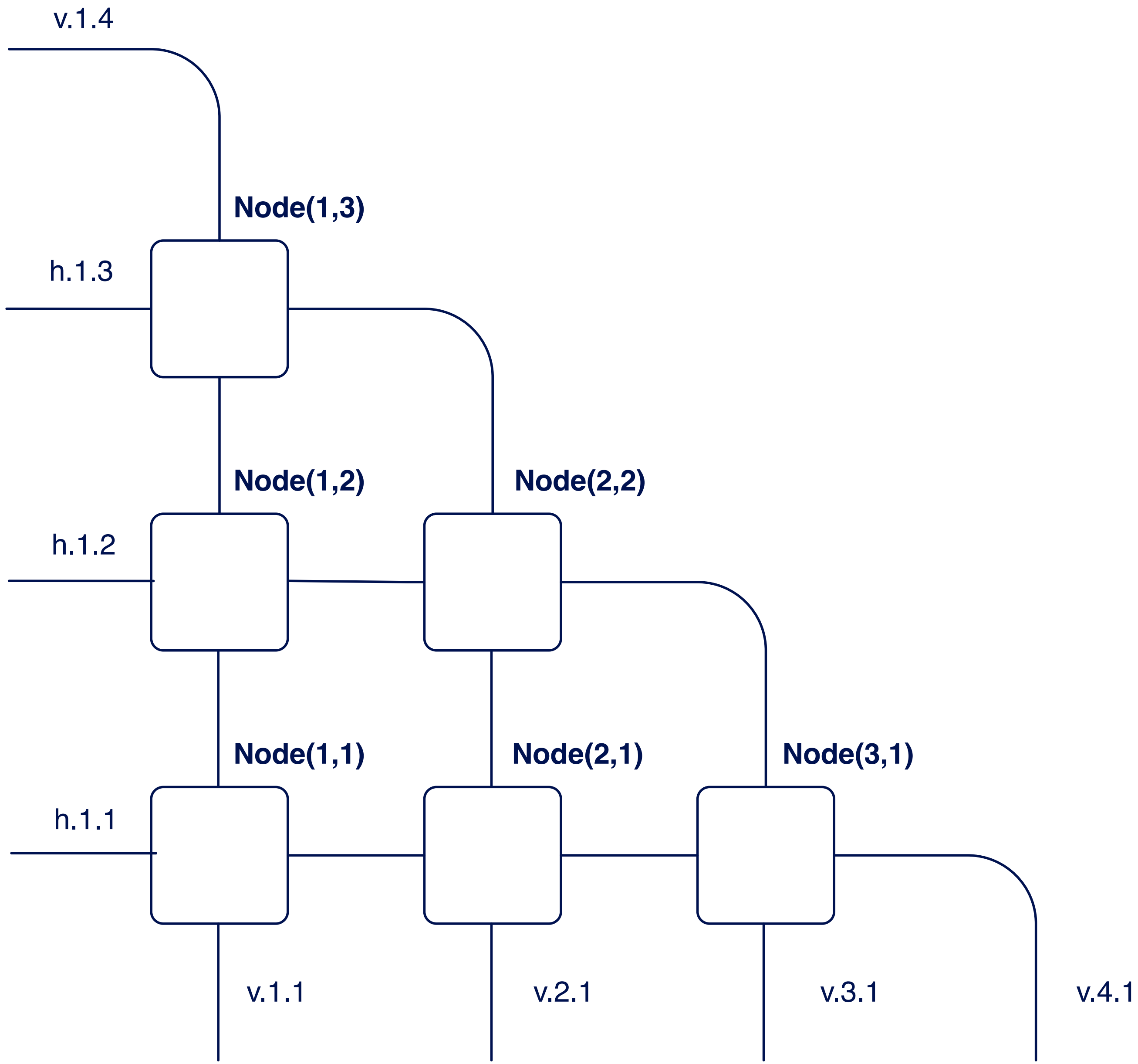
but instead in terms of the union of the two sets

```
aInput = {| h.1.1, h.1.2, h.1.3, v.1.4 |}
```

and

```
aOutput = {| v.1.1, v.2.1, v.3.1, v.4.1 |}
```


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


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

Hiding

We can construct a restricted view of a process, treating some of its alphabet as internal events, using the hiding operator \.

The left-hand argument is a process-valued expression. The right-hand argument is a set-valued expression.

Interpretation: hiding

If `P` is a process and `H` is a set of events, then

```
P \ H
```

is a process that behaves as `P` under the assumption that every event in the set `H` is an internal transaction.

Algebra

```
  (P \ H) \ I = (P \ I) \ H

(P |~| Q) \ H = (P \ H) |~| (Q \ H)
```

Step law: hiding

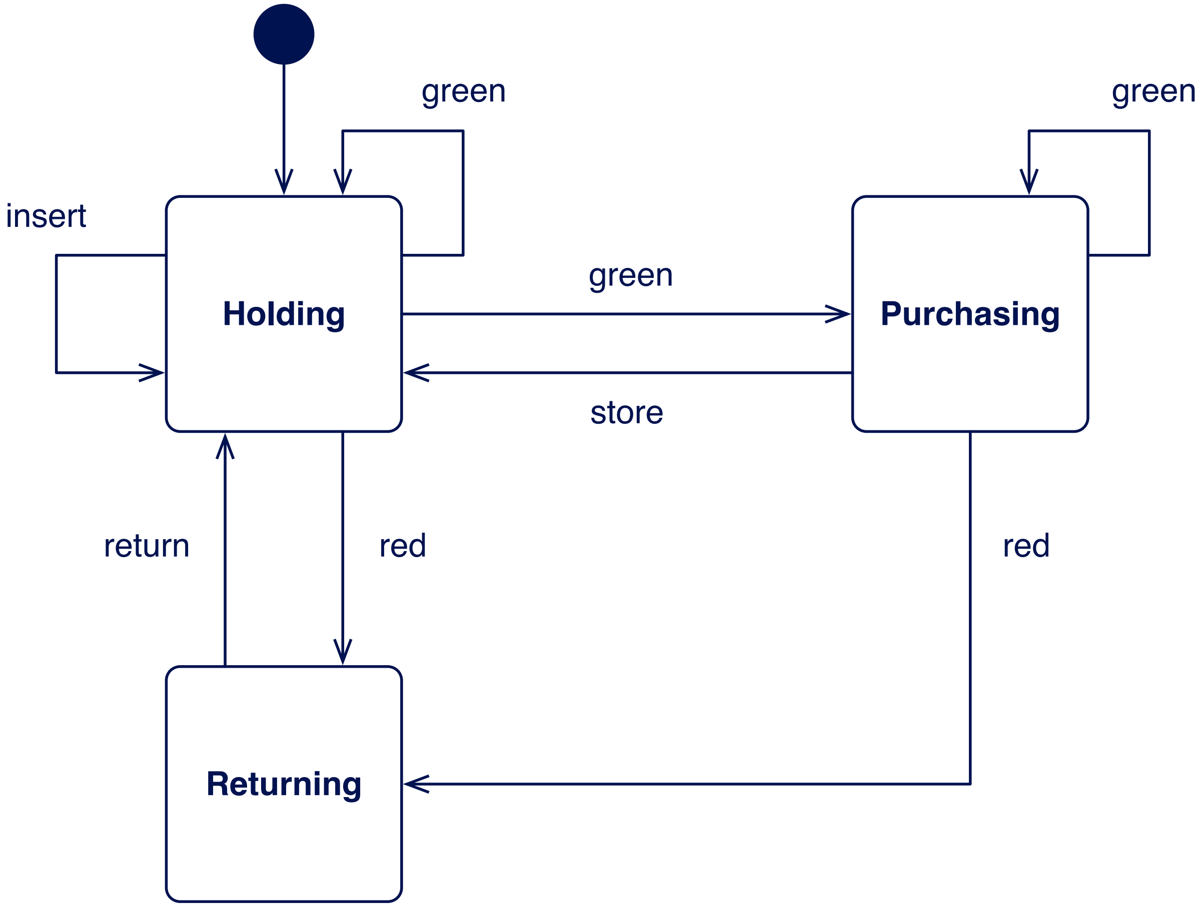
If

```
P = [] e : A @ e -> P(e)
```

then

```
P \ H = ( ( [] e : diff(A,H) @ e -> (P(e) \ H) )
          []
          ( |~| h : inter(A,H) @ (P(h) \ H) ) )
        |~|
        ( |~| h : inter(A,H) @ (P(h) \ H) )
```

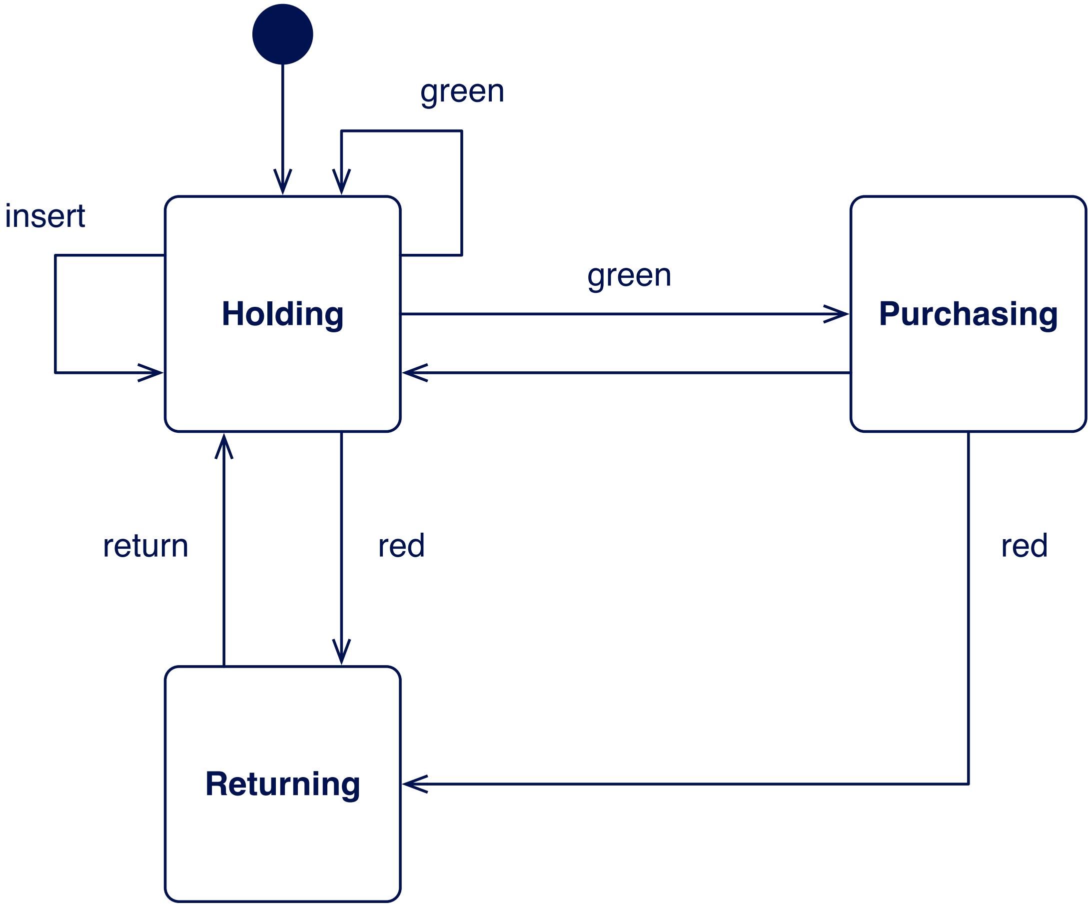
Example


green
green
insert
green
Holding
Purchasing
store
return
red
red
Returning

Example

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

Example


green
insert
Holding
green
Purchasing
return
red
red
Returning

Example

```
Purchasing(n,v) \ {| store |} =
  ( ( green -> Purchasing(n,v) \ {| store |}
      []
      red -> Returning(v) \ {| store |} )
    |~|
    Holding(0,0) \ {| store |} )
  |~|
  Holding(0,0) \ {| store |}
```

Divergence

A recursive process is well-defined only if every recursive invocation in its definition is guarded by at least one event.

If a guarding event is hidden, then the process in question may be completely undefined.

Example

```
Purchasing(n,v) \ { green } =
  Purchasing(n,v) \ { green }
  |~|
  ( Purchasing(n,v) \ { green }
    |~|
    ( red -> Returning(v) \ { green }
      []
      store!n -> Holding(0,0) \ { green } ) )
```

Traces: hiding

```
traces(P \ A) = { hide(tr,A) | tr <- traces(P) }
```

where

```
hide(<>,A) = <>
hide(<x>^s,A) =
  if member(x,A) then
    hide(s,A)
  else
    <x>^hide(s,A)
```

Failures: hiding

```
failures(P \ A) =
  union( { (hide(tr,A),ref) |
              (tr,union(ref,A)) <- failures(P) },
         { (d,ref) |
              ref <- Set(Event), d <- divergences(P \ A) }
```

Failures-divergences refinement

If `P` and `Q` are processes, then we write

```
P [FD= Q
```

to indicate that `P` is failures-divergences-refined by `Q` or – equally – that `Q` is a failures-divergences refinement of `P`.

Failures-divergences refinement is precisely the removal of nondeterminism, while accounting properly for definedness.

Failures-divergences refinement

If `P` and `Q` are processes, then

```
P [FD= Q
```

if and only if

- every failure of `Q` is also a failure of `P`
- every divergence of `Q` is also a divergence of `P`

Equality

If `P` and `Q` are processes, then

```
P = Q
```

if and only if

`P [FD= Q` and `Q [FD= P`

Divergence

If `P` is defined by

```
P = (a -> P) [] P
```

then `P` is immediately divergent, and not a failures-divergences refinement of any process (other than itself).

Refinement and hiding

If `P` has alphabet `aP`, and a possible refinement `Q` has alphabet `aQ`, such that `aP < aQ`, then we may wish to check whether

```
P [FD= Q \ diff(aQ,aP)
```

where < denotes the subset relation.

Why?

1. to avoid having to add all of the events of `aQ` to the specification `P`
2. to allow the tool to tackle larger state spaces on the right-hand side (hiding reduces the size of the state space)

but sometimes we end up hiding too much, and then we have to do something about it…

Example

A safety specification for `ups` and `downs`, that has to mention `open`, `close`, and `arrive`:

```
Spec =
  let
    Ground =
      up -> First
      []
    ( [] e : {open,close,arrive} @ e -> Ground )

    First =
      down -> Ground
      []
    ( [] e : {open,close,arrive} @ e -> First )
  within
    Ground
```

This is more irritating in the case of liveness:

```
Spec =
  let
    CanOpen =
      open -> Next
      []
     (STOP |~| (|~| e : {close,arrive,up,down} @ e -> Next))

    CanArriveOrClose =
     (|~| e : {arrive,close} @ e -> Next)
      []
     (STOP |~| (|~| e : {up,down} @ e -> Next))

    Next = CanOpen |~| CanArriveOrClose
  within
    Next
```

This would be easier to follow:

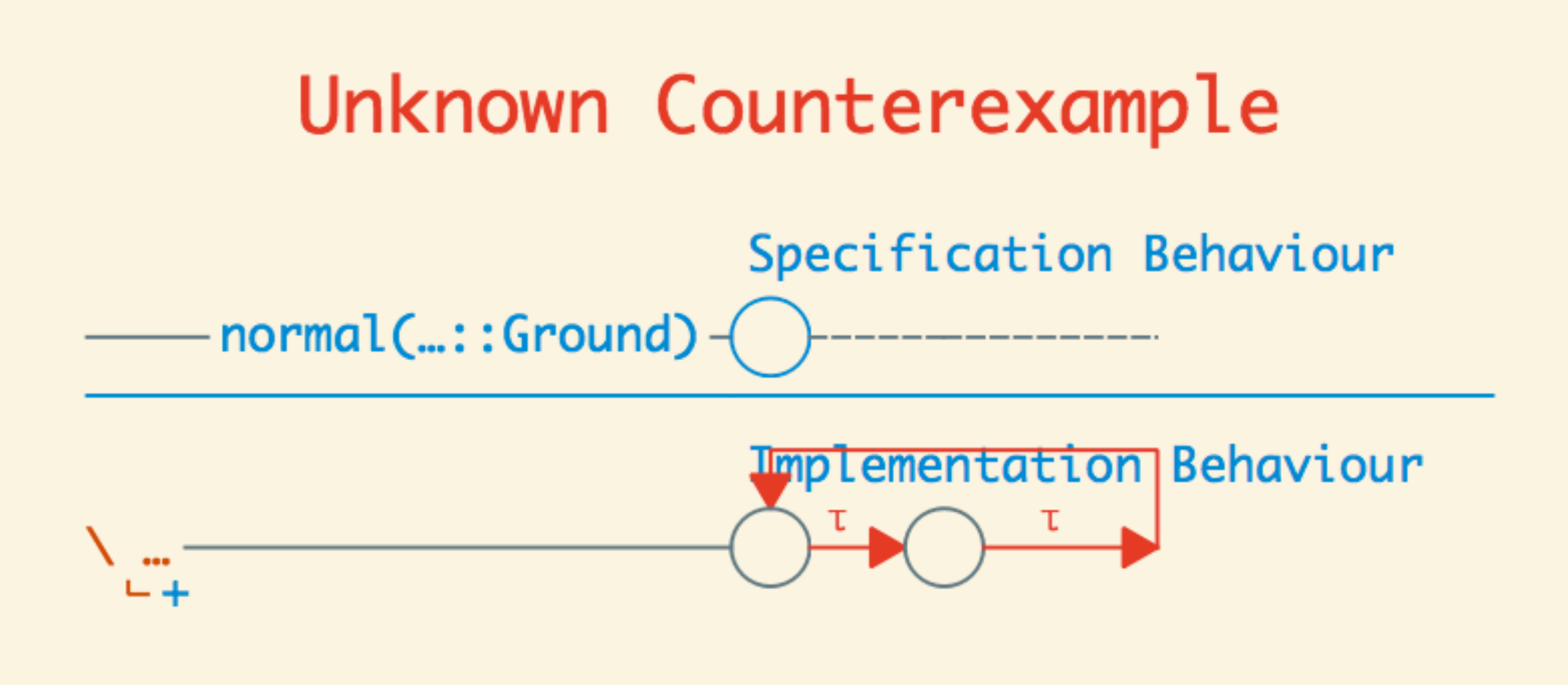
```
NewSpec =
  let
    Ground =
      up -> First

    First =
      down -> Ground
  within
   Ground
```

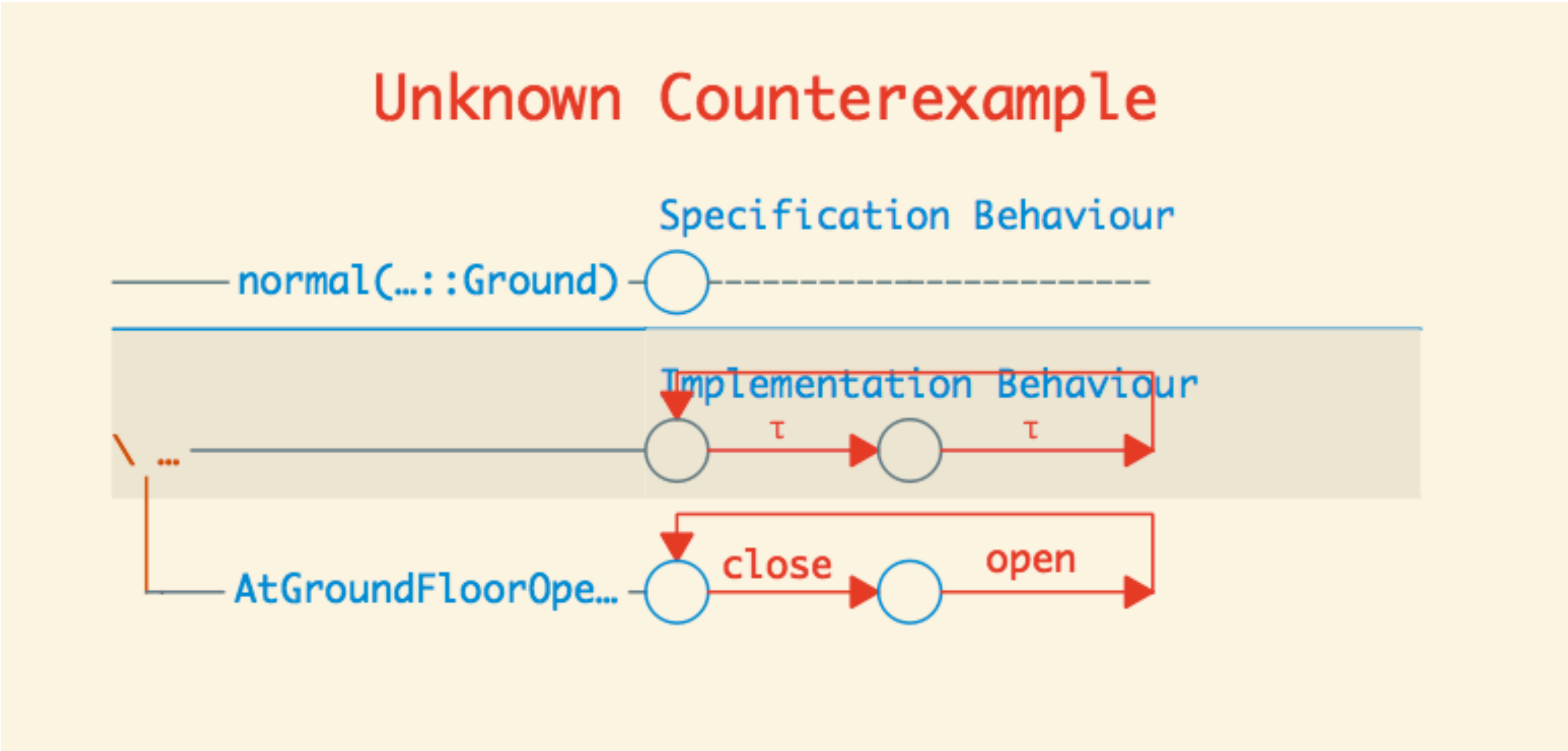
but wait...

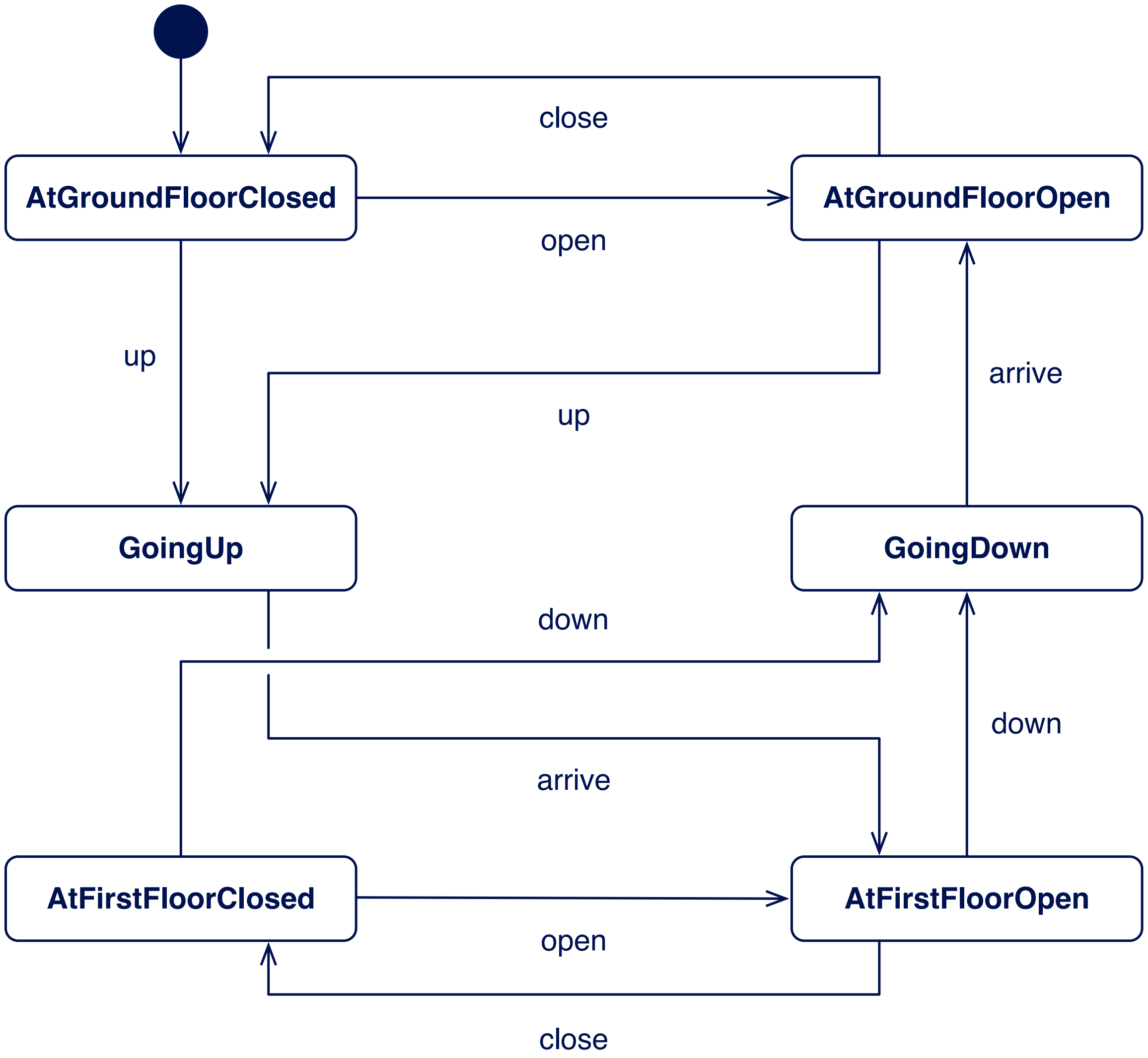
The check

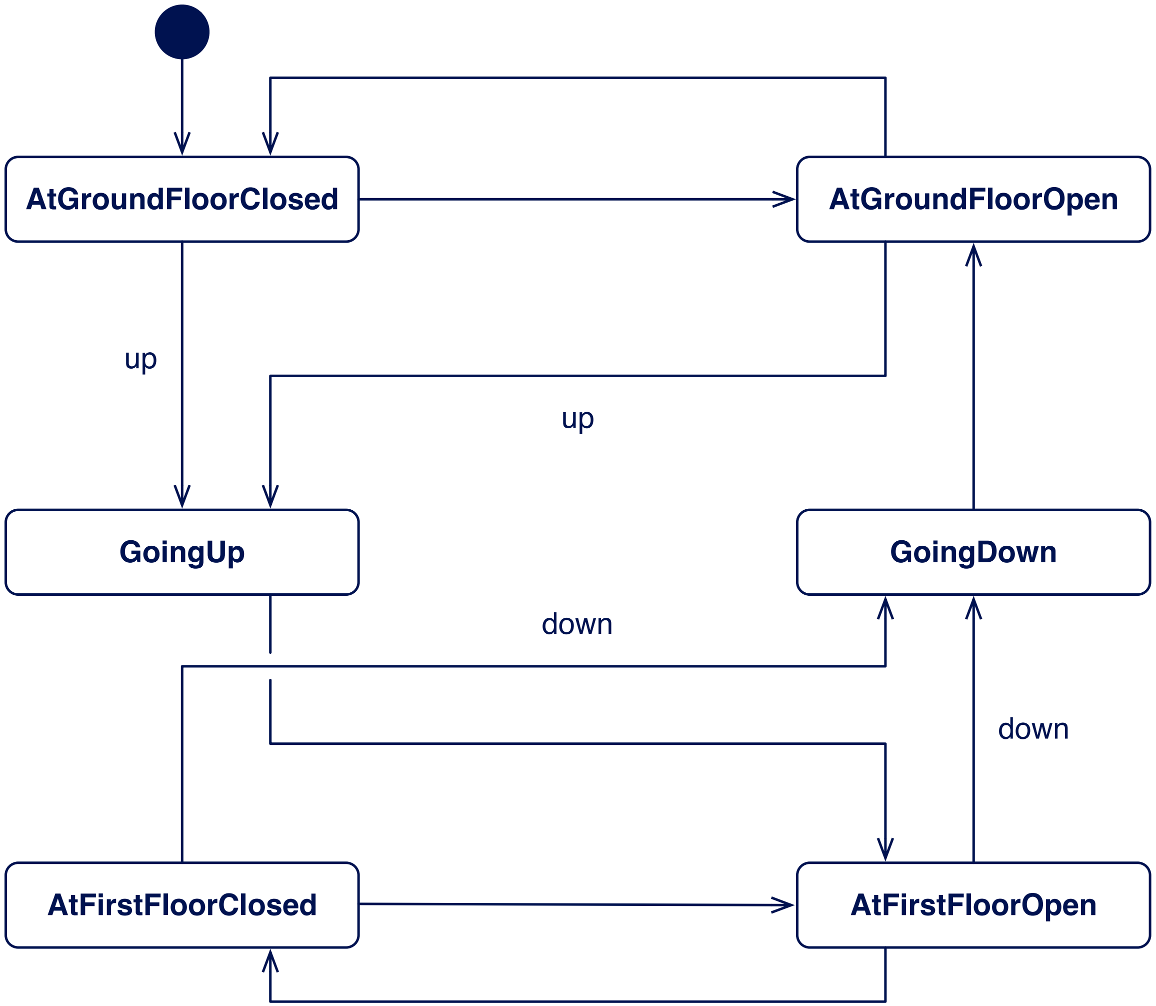
```
assert NewSpec [FD= LiftController \ {open,close,arrive}
```

will fail.


Unknown Counterexample
Specification Behaviour
normal(…::Ground)
Implementation Behaviour
τ
τ
\ …
+


Unknown Counterexample
Specification Behaviour
normal(…::Ground)
Implementation Behaviour
\ …
τ
τ
AtGroundFloorOpe…
close
open


close
AtGroundFloorClosed
AtGroundFloorOpen
open
up
up
arrive
GoingUp
GoingDown
down
down
arrive
AtFirstFloorClosed
AtFirstFloorOpen
open
close


AtGroundFloorClosed
AtGroundFloorOpen
up
up
GoingUp
GoingDown
down
down
AtFirstFloorClosed
AtFirstFloorOpen

Divergence

If `Q \ diff(aQ,aP)` diverges, and we wish to perform a failures refinement check, then we have two options:

- don't hide all of `diff(aQ,aP)`, leave some of the events as self-transitions in the specification;
- put `Q` in parallel with a limiting process `R`, with alphabet `aR < aQ`, and check

```
P [FD= (Q [aQ || aR] R) \ diff(aQ,aP)
```

```
aLiftController = {open, close, arrive, up, down}

aLimit1 = {open}
Limit1 = STOP

NewSpec [FD=
  LiftController [aLiftController || aLimit1] Limit1 \
    { arrive, open, close }
```

```
aLimit2 = {open,up,down}
Limit2 =
  let
    AllowOpen =
      open -> BlockOpen
      []
      up -> AllowOpen [] down -> AllowOpen

    BlockOpen =
      up -> AllowOpen [] down -> AllowOpen
  within
    AllowOpen

NewSpec [FD=
  LiftController [aLiftController || aLimit2] Limit2 \
    { arrive, open, close }
```

```
aLimit3 = {open,up}
Limit3 =
  let
    AllowOpen(n) =
      (n > 0) & open -> AllowOpen(n-1)
      []
      up -> AllowOpen(3)
  within
    AllowOpen(3)

NewSpec [FD=
  LiftController [aLiftController || aLimit3] Limit3 \
    { arrive, open, close }
```

Divergence check

We can check whether a process `P` is divergence-free using the assertion

```
assert P :[divergence free]
```

Summary

- Hiding
- Divergence
- Refinement and hiding

Index