

# Refinement

**Concurrency and Distributed Systems**

**January 2023**

# Contents

- **Nondeterminism**
- **Testing**
- **Refinement**
- **Compositionality**

## Nondeterminism

**nondeterminism, *n.***

1. *Philos.* The doctrine that human action is free and not determined by previous events.
2. *Math. and Computing.* A mode of computation in which, at certain points, there is a choice of ways to proceed which cannot be predicted in advance.

Oxford English Dictionary

## Nondeterminism

We may wish to allow for two or more different patterns of behaviour, each of which would be regarded as satisfactory.

Alternatively, we may wish to describe a design at a higher level of **abstraction**, focussing upon only some of the events.

In either case, the resulting process description may be **nondeterministic**.

or it may not be...it depends upon how the information we are not considering affects the occurrence and availability of the events in our description

## Semantics

If  $\langle a, b \rangle$  is a trace of  $P$  then, after the event  $a$  has occurred, that process **may** allow  $b$ .

If  $(\langle a \rangle, \{b\})$  is a failure of  $P$  then, after the event  $a$  has occurred, that process **may** refuse  $b$ .

A **nondeterministic** process is one that, after some trace, may allow some event but may also refuse that same event.

## Example

EXT =

a -> b -> STOP

[]

c -> d -> STOP

**is deterministic**

INT =

a -> b -> STOP

|~|

c -> d -> STOP

**is nondeterministic**

## Example

```

failures(INT) = {
  (<>, {}), (<>, {a}), (<>, {b}), (<>, {c}), (<>, {d}), (<>, {a, b}),
    , (<>, {a, d}), (<>, {b, c}), (<>, {b, d}), (<>, {c, d}),
    , (<>, {a, b, d}), , (<>, {b, c, d}),
  (<a>, {}), (<a>, {a}), (<a>, {c}), (<a>, {d}), (<a>, {a, c}),
  (<a>, {a, d}), (<a>, {c, d}), (<a>, {a, c, d}), (<a, b>, {}),
  (<a, b>, {a}), (<a, b>, {b}), (<a, b>, {c}), (<a, b>, {d}),
  (<a, b>, {a, b}), (<a, b>, {a, c}), (<a, b>, {a, d}), (<a, b>, {b, c}),
  (<a, b>, {b, d}), (<a, b>, {c, d}), (<a, b>, {a, b, c}), (<a, b>, {a, b, d}),
  (<a, b>, {a, c, d}), (<a, b>, {b, c, d}), (<a, b>, {a, b, c, d}), (<c>, {}),
  (<c>, {a}), (<a>, {b}), (<a>, {c}), (<c>, {a, b}), (<c>, {a, c}),
  (<c>, {b, c}), (<c>, {a, b, c}), (<c, d>, {}), (<c, d>, {a}), (<c, d>, {b}),
  (<c, d>, {c}), (<c, d>, {d}), (<c, d>, {a, b}), (<c, d>, {a, c}), (<c, d>,
  {a, d}), (<c, d>, {b, c}), (<c, d>, {b, d}), (<c, d>, {c, d}),
  (<c, d>, {a, b, c}), (<c, d>, {a, b, d}), (<c, d>, {a, c, d}), (<c, d>, {b, c, d}),
  (<c, d>, {a, b, c, d}) }

```

## Example

```

failures(EXT) = {
  (<>, {}),
  (<>, {b}),
  (<>, {d}),
  (<>, {b, d}),
  (<a>, {}),
  (<a>, {a}),
  (<a>, {c}),
  (<a>, {d}),
  (<a>, {a, c}),
  (<a>, {a, d}),
  (<a>, {c, d}),
  (<a>, {a, c, d}),
  (<a, b>, {}),
  (<a, b>, {a}),
  (<a, b>, {b}),
  (<a, b>, {c}),
  (<a, b>, {d}),
  (<a, b>, {a, b}),
  (<a, b>, {a, c}),
  (<a, b>, {a, d}),
  (<a, b>, {b, c}),
  (<a, b>, {b, d}),
  (<a, b>, {c, d}),
  (<a, b>, {a, b, c}),
  (<a, b>, {a, b, d}),
  (<a, b>, {a, c, d}),
  (<a, b>, {b, c, d}),
  (<a, b>, {a, b, c, d}),
  (<c>, {}),
  (<c>, {a}),
  (<c>, {b}),
  (<c>, {c}),
  (<c>, {a, b}),
  (<c>, {a, c}),
  (<c>, {b, c}),
  (<c>, {a, b, c}),
  (<c, d>, {}),
  (<c, d>, {a}),
  (<c, d>, {b}),
  (<c, d>, {c}),
  (<c, d>, {d}),
  (<c, d>, {a, b}),
  (<c, d>, {a, c}),
  (<c, d>, {a, d}),
  (<c, d>, {b, c}),
  (<c, d>, {b, d}),
  (<c, d>, {c, d}),
  (<c, d>, {a, b, c}),
  (<c, d>, {a, b, d}),
  (<c, d>, {a, c, d}),
  (<c, d>, {b, c, d}),
  (<c, d>, {a, b, c, d}) }

```



## Question

**Internal choice is sometimes referred to as ‘nondeterministic choice’.**

**In general, a process of the form  $P \mid \sim \mid Q$  will indeed be nondeterministic, although this isn’t always the case.**

**Can you give a simple example?**

## Question

**Similarly, an external choice between two processes need not be deterministic.**

**Can you give a simple example?**

**nondeterminism is a semantic property**

## Determinism checking

**The tool allows us to check whether a process  $P$  is deterministic, using the following assertion:**

**`assert P : [deterministic[F]]`**

**This assertion will use the failures model  $F$ .**

## Example

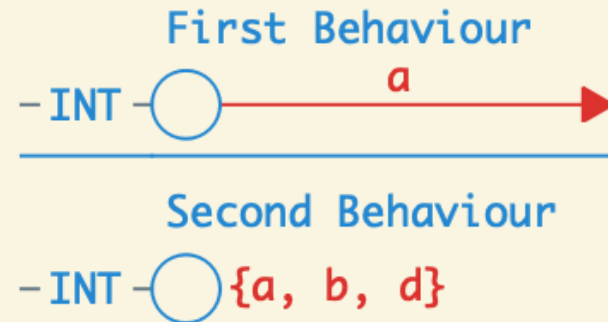
assert EXT : [deterministic[F]]

assert INT : [deterministic[F]]

**The second assertion will fail.**

## Example

### Determinism Counterexample



## Using different models

If we omit the model parameter  $[F]$ , or replace it with the default  $[FD]$ , then the check will use the failures-divergences model instead. It will then check for divergences at the same time.

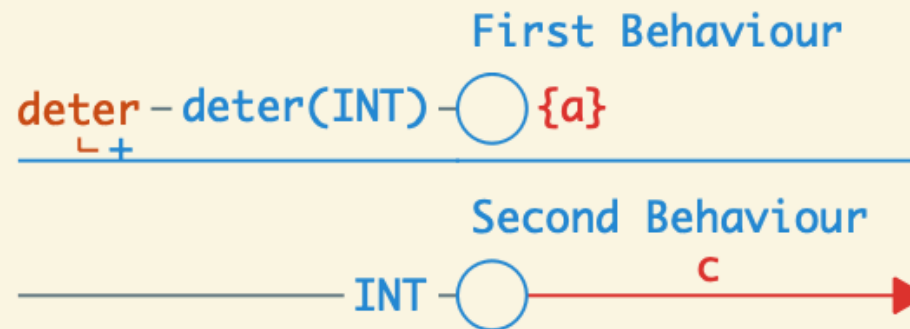
We may prefer to check for this separately, using the assertion

`assert P : [divergence free]`

The algorithm employed for the failures-divergences model is more efficient, but the explanation provided is less accessible.

## Example

### Determinism Counterexample



## Testing

**We can establish properties of processes by checking to see whether a particular trace or failure is included in the semantics.**

**This can help to confirm that a process has the intended meaning, whether we are using that process as a specification, an assumption, or as part of a design.**



## May

If  $\langle a, b \rangle$  is a trace of  $P$  then, after the event  $a$  has occurred, that process **may** allow  $b$ .

If  $(\langle a \rangle, \{b\})$  is a failure of  $P$  then, after the event  $a$  has occurred, that process **may** refuse  $b$ .

At least one of the two premisses must be true. A process must either allow or refuse each event.

## Must

If  $\langle a, b \rangle$  is **not** a trace of  $P$  then, after the event  $a$  has occurred, that process **must** refuse  $b$ .

If  $(\langle a \rangle, \{b\})$  is **not** a failure of  $P$  then, after the event  $a$  has occurred, that process **must** allow  $b$ .

Both premisses may be false, in which case the process is nondeterministic.

## I can haz trace

The tool allows us to check whether a trace  $tr$  is a trace of process  $P$  using the following assertion:

`assert P : [has trace[T]]: tr`

## I can haz failure?

**There is no assertion that can be used to check for the presence of a specific failure in the semantics of a process.**

**However, we can check whether a trace is deterministically available: that is, none of the events could also have been refused along the way.**

```
assert P : [has trace[F]]: tr
```

## Example

assert EXT : [has trace[T]]: <a, b>

assert INT : [has trace[T]]: <a, b>

assert EXT : [has trace[F]]: <a, b>

assert INT : [has trace[F]]: <a, b>

**The fourth assertion will fail.**

## Example

### Acceptance Counterexample

Specification Behaviour  
—Unknown—○ Allowed Refusals:  
                   $\{\{b, c, d\}\}$

---

Implementation Behaviour  
——INT—○  $\{a, b, d\}$

## Comparing processes

**Most of the time, however, we will wish to compare two processes—to test one process against another.**

**If each process has a finite number of states, then it will be possible to check all of the possible behaviours with a single test.**

**even if the sets of traces and failures are infinite**

## Refinement

If every behaviour of process  $P$  is also a behaviour of process  $Q$ , then we say that  $P$  is a refinement of  $Q$ .

We write this as:

$$Q \ [M] \ P$$

where  $M$  is one of our semantic models:

- $T$  traces
- $F$  failures
- $FD$  failures-divergences

every behaviour of  $P$  is allowed by  $Q$



## Example

**failures(EXT) is a subset of failures(INT), and so**

**INT  $\sqsubseteq$  EXT**

**as we move from right to left, we are refining, eliminating behaviours, and—for the failures model—increasing determinism**

## Definition

**refinement, *n.***

1. **The refining of a substance or product; esp. the removing of impurities or unwanted elements.**
2. **The improvement, modification, or clarification of a faculty, product, mechanism.**

## Trace refinement

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

$$P \sqsubseteq_T Q$$

to indicate that  $P$  is trace-refined by  $Q$  or—equally—that  $Q$  is a trace refinement of  $P$ .

Trace refinement is simply the removal of possible traces.

in the traces model, we cannot see nondeterminism

## Trace refinement (restated)

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

$$P \sqsubseteq T Q$$

if and only if every trace of  $Q$  is also a trace of  $P$ .

In moving from  $P$  to  $Q$ , we have simply removed some\* of the traces from  $P$ .

\*or possibly none

## Example

(a -> b -> STOP [] c -> d -> STOP)

[T=

(a -> b -> STOP)

## Example

$a \rightarrow b \rightarrow \text{STOP}$

$[T =$

$a \rightarrow \text{STOP}$

## Using trace refinement

If Spec describes a set of good traces, then the refinement

$\text{Spec} \sqsubseteq \text{Impl}$

holds if and only if every trace of Impl is good.

## Safety

Trace refinement allows us to check what are sometimes called **safety properties**.

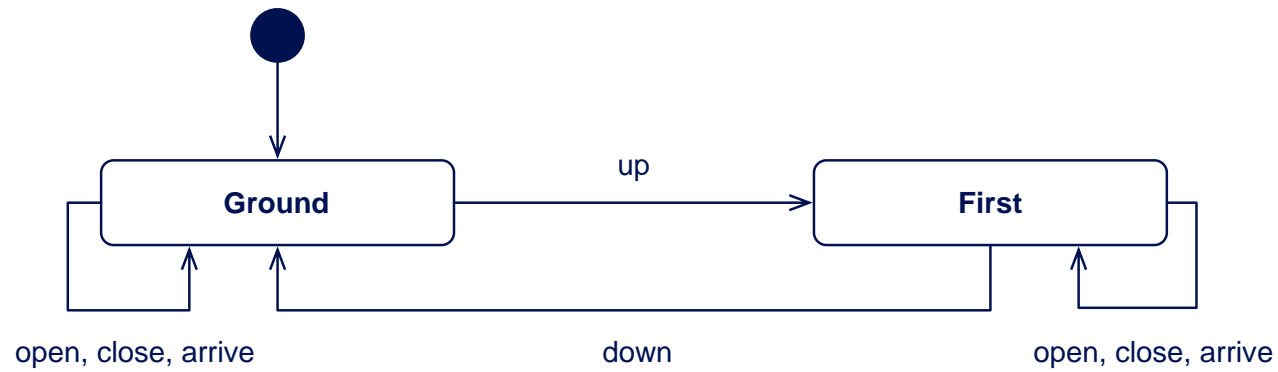
These are assertions that ‘nothing bad happens’. For our process models, this can be interpreted as

- no transaction takes place when it shouldn’t – or, equally
- every trace is a good trace

just because a system satisfies this kind of ‘safety property’, it doesn’t mean that it is ‘safe’ in other senses of the word: refusing to perform a transaction might also be ‘a bad thing’ that happens.



## Example



focussing on the ups and downs

## Example

Spec =

let

Ground =

up -> First

[]

( [] e : {open, close, arrive} @ e -> Ground )

First =

down -> Ground

[]

( [] e : {open, close, arrive} @ e -> First )

within

Ground

## Example

**We can use the tool to check that this specification is satisfied by our lift controller design:**

```
assert Spec [T= LiftController
```

**what goes up must come down before it goes up again**

## Example

Spec [T= STOP

**STOP will satisfy any safety property**

**a lift that never moves is safe, in this respect**

## Trace equivalence

For any processes  $P$  and  $Q$ , if

$$P \sqsubseteq T Q$$

and

$$Q \sqsubseteq T P$$

then

$$\text{traces}(P) = \text{traces}(Q)$$

this does not mean that  $P = Q$  unless...

## Determinism

**If  $P$  and  $Q$  are both deterministic, then**

$$\text{traces}(P) = \text{traces}(Q)$$

**is enough to show that  $P = Q$ .**

## Example

If P and Q are defined by

$$P = a \rightarrow b \rightarrow P \quad [] \quad c \rightarrow d \rightarrow P$$
$$Q = c \rightarrow d \rightarrow Q \quad [] \quad a \rightarrow b \rightarrow Q$$

then the four assertions

$$\text{assert } P : [\text{deterministic}]$$
$$\text{assert } Q : [\text{deterministic}]$$
$$\text{assert } P \quad [T = Q]$$
$$\text{assert } Q \quad [T = P]$$

would be enough to confirm that  $P = Q$ .

## Failures refinement

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

$$P \sqsubseteq_F Q$$

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

In the failures semantics, we know which failures correspond to nondeterministic choices, and failures refinement is precisely the removal of nondeterminism.



## Failures refinement

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

$$P \sqsubseteq F Q$$

if and only if every failure of  $Q$  is also a failure of  $P$ .

In moving from  $P$  to  $Q$ , we may have removed some of the failures, but only those that correspond to one half of a nondeterministic choice.

## Example

$(a \rightarrow b \rightarrow \text{STOP} \mid \sim \mid c \rightarrow d \rightarrow \text{STOP})$

$[F=$

$a \rightarrow b \rightarrow \text{STOP}$

## Example

$(a \rightarrow b \rightarrow \text{STOP} \quad [] \quad c \rightarrow d \rightarrow \text{STOP})$

**is not failures-refined by**

$a \rightarrow b \rightarrow \text{STOP}$

## Using failures refinement

If Spec describes a set of good failures, then the refinement

$\text{Spec} \sqsubseteq_{\text{F}} \text{Impl}$

holds if and only if every failure of Impl is good.

## Using failures refinement

If Spec describes a set of good failures, then the refinement

$\text{Spec} \sqsubseteq_{\text{F}} \text{Impl}$

holds if and only if every trace allowed by Impl is good, and every set of events that may be refused by Impl is good.

good = included as a (possibly nondeterministic) option in Spec

## Liveness

Failures refinement allows us to check what are sometimes called **liveness properties**.

These are assertions that ‘something good happens eventually’.  
For our process models, this can be interpreted as

- no set of transactions is refused when it shouldn't be – or, equally
- no additional requirement is imposed, in terms of other transactions that would have to happen first

## Example

**In the case of the `LiftController` design, we may wish to insist that the lift doors can be opened at any time.**

```
assert
```

```
Spec1 =
```

```
  let
```

```
    CanOpen = open -> CanOpen
```

```
  within
```

```
    CanOpen
```

**However,**

```
not Spec1 [F= LiftController
```

**for two reasons!**

**(1) other things can happen; (2) it isn't true, anyway**

## Example

**If we use the tool to check the assertion**

```
assert Spec1 [F= LiftController
```

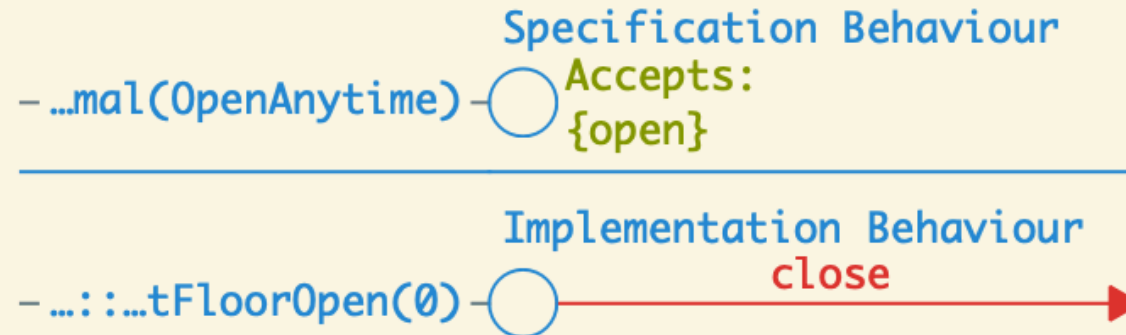
**then it will fail.**

**The specification is too strong to be satisfied**



## Example

### Trace Counterexample

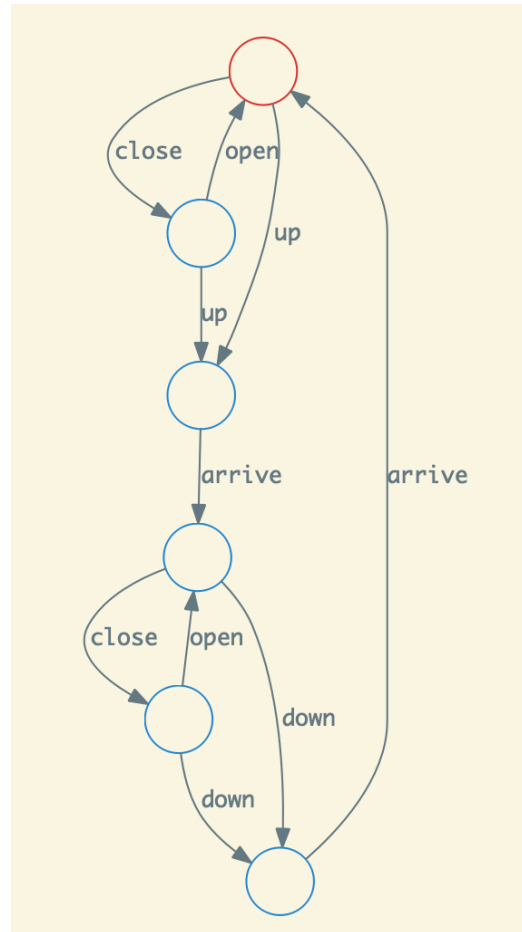


## Example

**To see what is happening, as `Li ftControl l er` has relatively few states, we may examine the state graph:**

`: graph Li ftControl l er`

# Example



## Example

In the specification, we need to allow that other events may occur.

```
OpenOrClose =
```

```
  let
```

```
    Start =
```

```
      ReadyToOpen | ~ | ReadyToClose
```

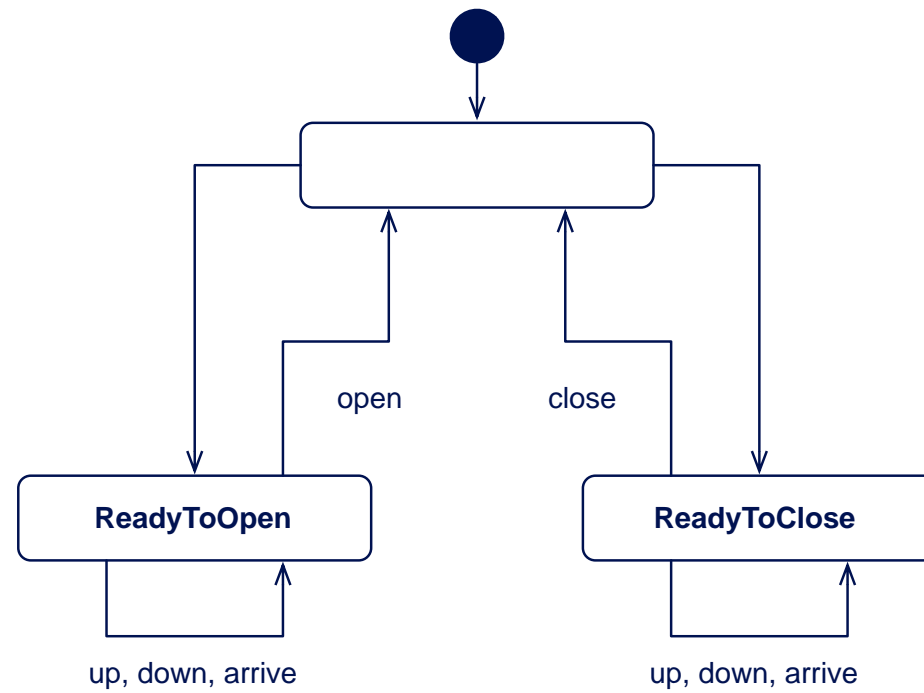
```
    ReadyToOpen =
```

```
      open -> OpenOrClose
```

```
    []
```

```
    (| ~ | e : {up, down, arrive} @ e -> ReadyToOpen)
```

# Example



## Example

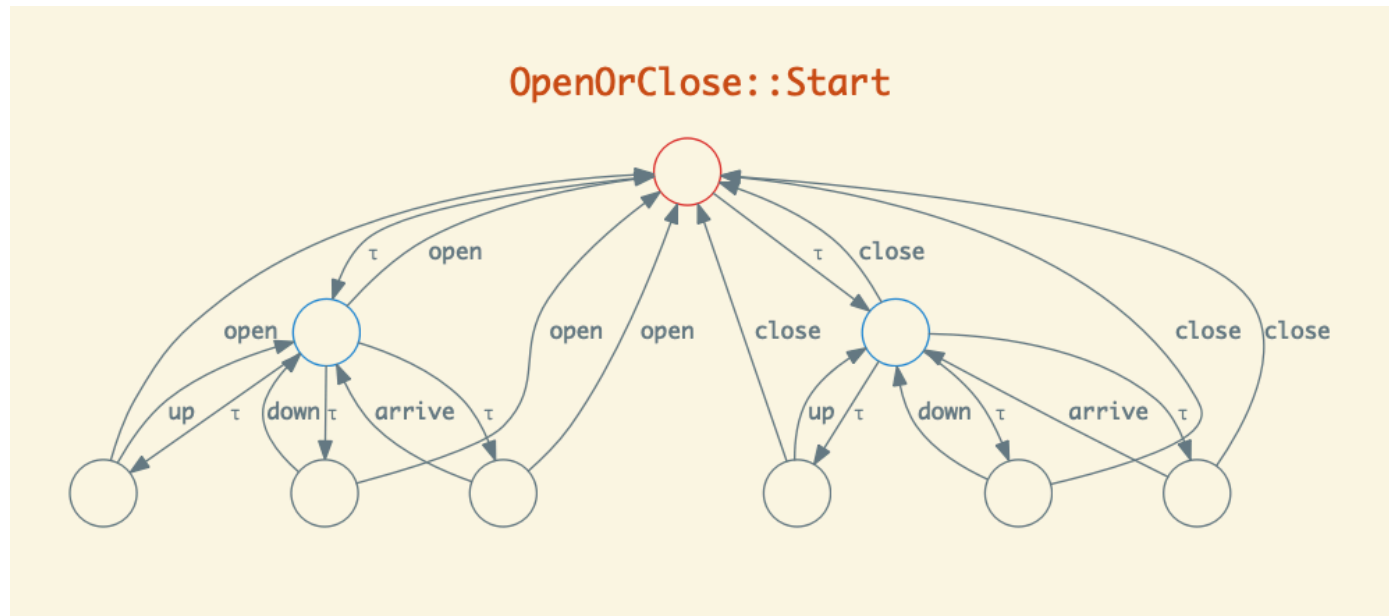
**The specification is still too strong, and the assertion**

`assert OpenOrClose [FD= LiftController`

**fails.**

**We need more nondeterminism.**

## Example



(note that silent transitions are decorated with a tau symbol)

## Example

OpenOrCloseUnlessMoving =

let

Start =

ReadyToOpen |~| ReadyToClose

ReadyToOpen =

open -> OpenOrCloseUnlessMoving

[]

(|~| e : {up, down} @ e -> Moving)



```

ReadyToClose =
  close -> OpenOrCloseUnlessMoving
[]
(|~| e : {up, down} @ e -> Moving)

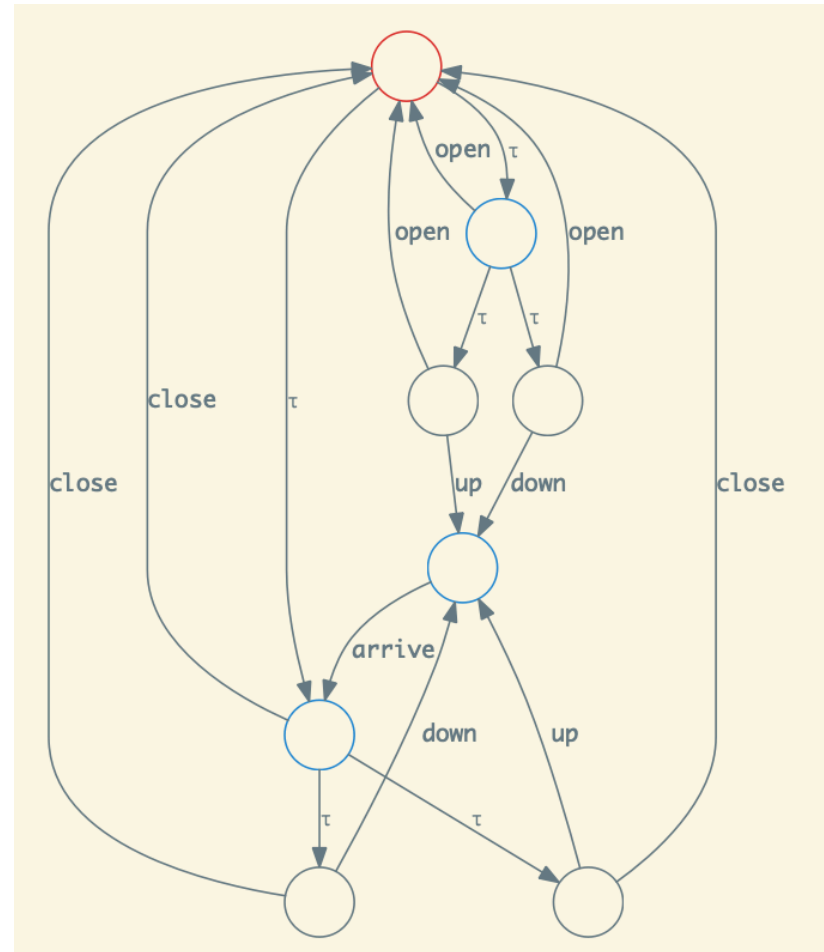
```

```

Moving = arrive -> ReadyToClose
within
Start

```

**this will do the trick—hard to draw, though!**



## Footnote: refusal sets

In the failures model, we consider whether or not a process, after allowing a particular trace, can refuse any one of a **set** of events. This allows us to properly model the behaviour of processes in a **concurrent context**.

If we were instead to consider whether or not the process might refuse each different event—one event at a time, rather than in combination—then we could build a semantics that would work for event prefix, external choice, internal choice, and parallel composition.

Such a semantics would let us down badly, however, when it came to abstraction or hiding.

With refusal sets, we get the semantics we need.

## Compositionality

If we establish that  $P$  is refined by  $Q$ , then that result remains valid in any context.

A refinement of any component produces a refinement of the system. If  $P \sqsubseteq Q$ , then

- $a \rightarrow P \sqsubseteq a \rightarrow Q$
- $P [] R \sqsubseteq Q [] R$
- $P \mid \sim \mid R \sqsubseteq Q \mid \sim \mid R$
- $P \parallel R \sqsubseteq Q \parallel R$
- $P \setminus A \sqsubseteq Q \setminus A$

## Summary

- **Nondeterminism**
- **Testing**
- **Refinement**
- **Compositionality**

# Index

2	Contents
3	Nondeterminism
4	Nondeterminism
5	Semantics
6	Example
7	Example
8	Example
9	Question
10	Question
11	Determinism checking
12	Example

- 13 **Example**
- 14 **Using different models**
- 15 **Example**
- 16 **Testing**
- 17 **May**
- 18 **Must**
- 19 **I can haz trace**
- 20 **I can haz failure?**
- 21 **Example**
- 22 **Example**
- 23 **Comparing processes**
- 24 **Refinement**
- 25 **Example**

- 26 **Definition**
- 27 **Trace refinement**
- 28 **Trace refinement (restated)**
- 29 **Example**
- 30 **Example**
- 31 **Using trace refinement**
- 32 **Safety**
- 33 **Example**
- 34 **Example**
- 35 **Example**
- 36 **Example**
- 37 **Trace equivalence**
- 38 **Determinism**



- 39 **Example**
- 40 **Failures refinement**
- 41 **Failures refinement**
- 42 **Example**
- 43 **Example**
- 44 **Using failures refinement**
- 45 **Using failures refinement**
- 46 **Liveness**
- 47 **Example**
- 48 **Example**
- 49 **Example**
- 50 **Example**
- 51 **Example**

**52 Example**

**53 Example**

**54 Example**

**55 Example**

**56 Example**

**59 Footnote: refusal sets**

**60 Compositionality**

**61 Summary**

**62 Index**