Concurrency

Concurrency and Distributed Systems

January 2023

Contents

Parallel composition

We can describe a concurrent combination of two processes using the parallel operator `[ || ]`.

This operator takes four arguments. The two outer arguments are processes. The two inner arguments are sets of events.

The operator is commutative.

compare: external choice

Interpretation: Parallel

If `P` and `Q` are process-valued expressions, and `aP` and `aQ` are set-valued expressions, then

```
P [ aP || aQ ] Q
```

is a process that behaves as a combination of `P` and `Q` in which

- events from `aP` are allowed only when `P` allows them
- events from `aQ` are allowed only when `Q` allows them

we call `aP` the *alphabet* of `P`

Step law: Parallel

If

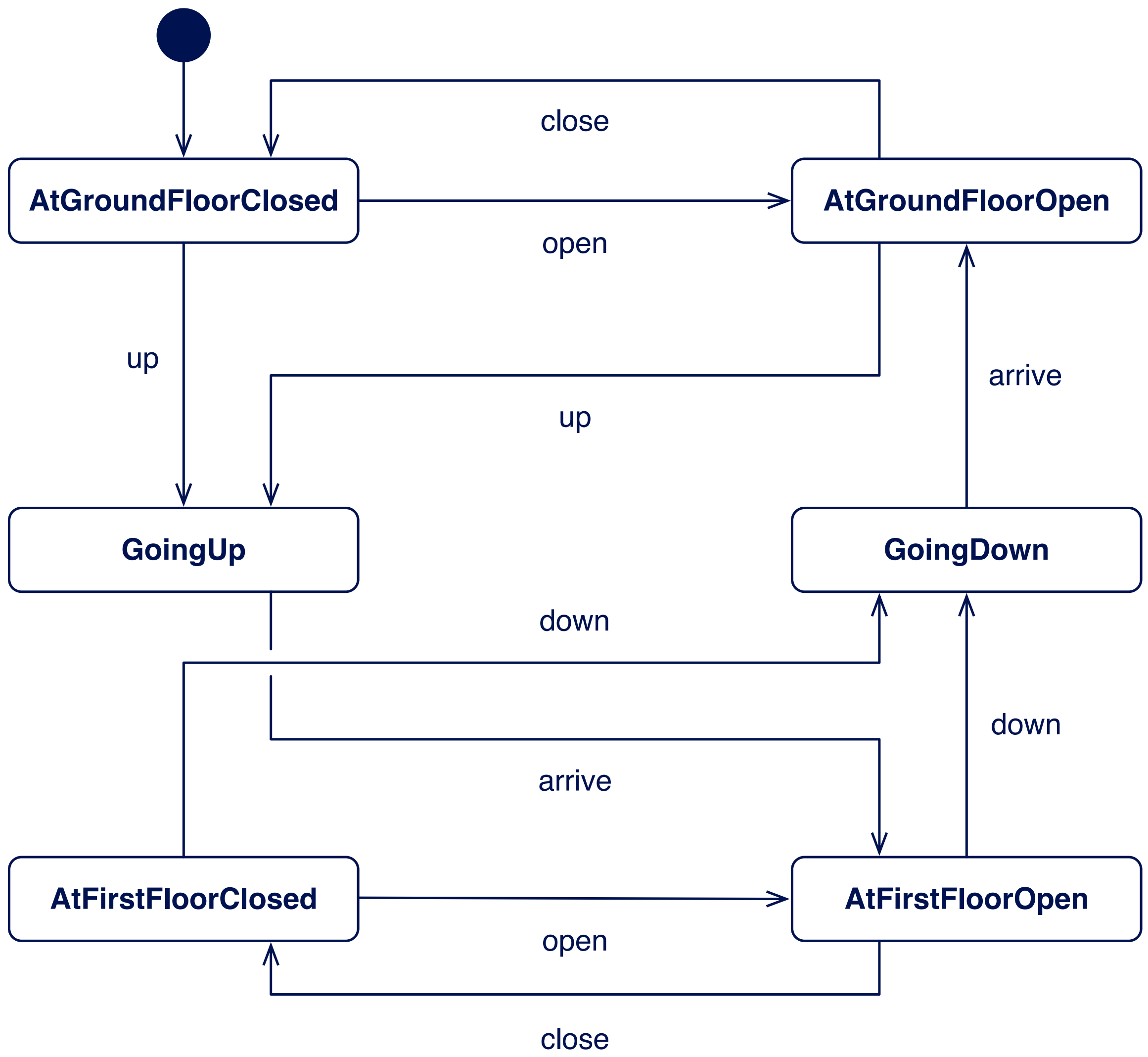
```
P = [] e : A @ e -> P(e)
Q = [] e : B @ e -> Q(e)
```

with `A < aP` and `B < aQ` then

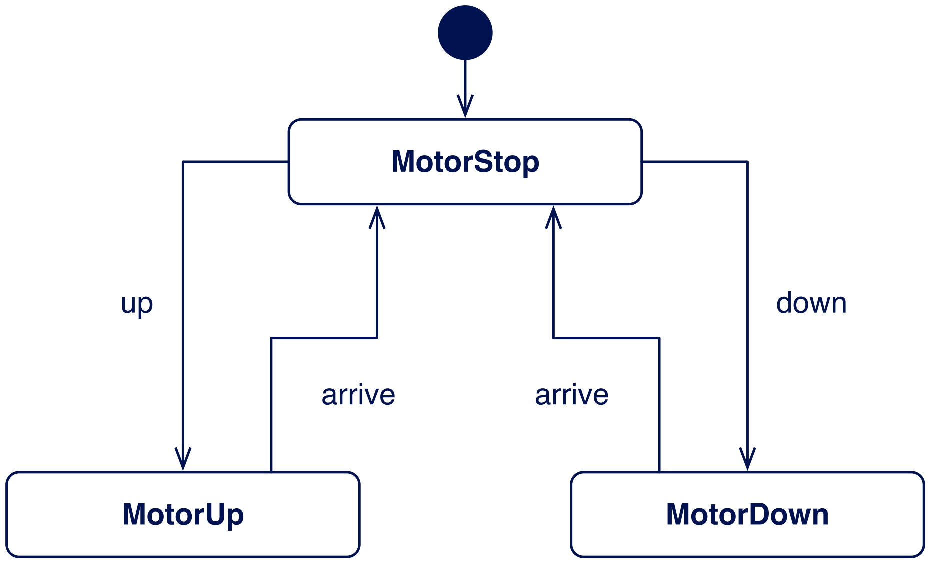
```
P [ aP || aQ ] Q =
  ( [] e : diff(A,aQ) @ e -> P(e) [ aP || aQ ] Q )
  []
  ( [] e : diff(B,aP) @ e -> P [ aP || aQ ] Q(e) )
  []
  ( [] e : inter(A,B) @ e -> P(e) [ aP || aQ ] Q(e)
```

compare: external choice

Example


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

Example



Example

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

LiftController =
  AtGroundFloorClosed

AtGroundFloorClosed =
  open -> AtGroundFloorOpen
  []
  up -> GoingUp

AtGroundFloorOpen =
  close -> AtGroundFloorClosed
  []
  up -> GoingUp
```

Example

```
AtFirstFloorClosed =
  open -> AtFirstFloorOpen
  []
  down -> GoingDown

AtFirstFloorOpen =
  close -> AtFirstFloorClosed
  []
  down -> GoingDown

GoingUp = arrive -> AtFirstFloorOpen

GoingDown = arrive -> AtGroundFloorOpen
```

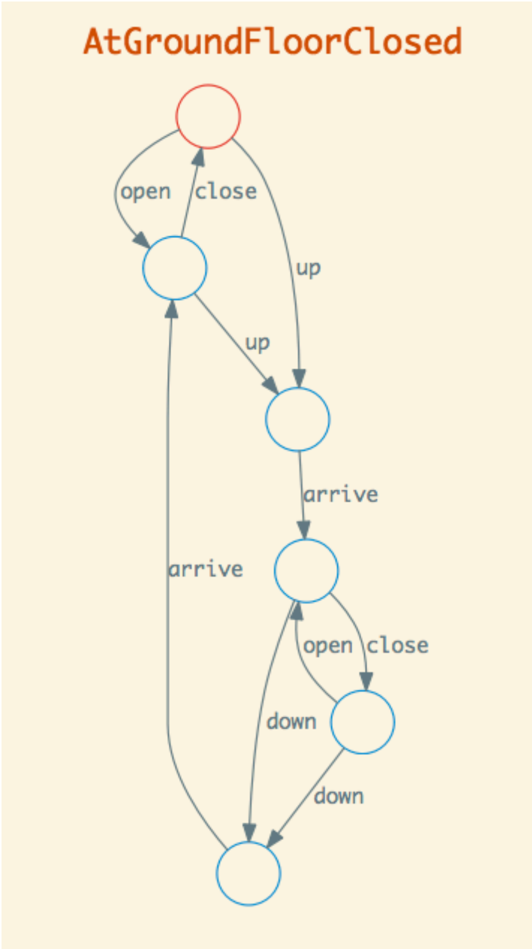
Example

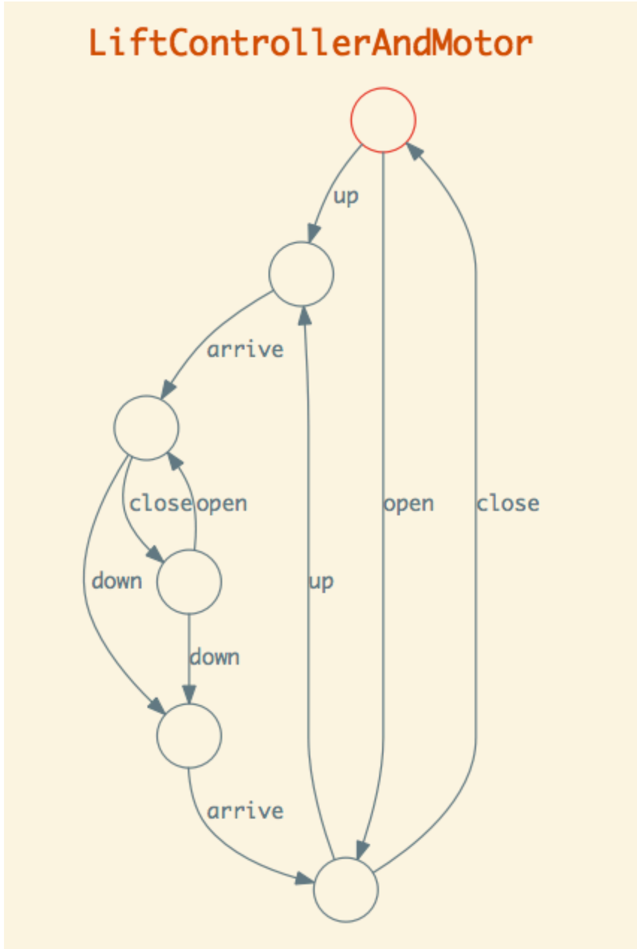
```
aLiftMotor = { up, down, arrive }

LiftMotor =
  let
    MotorStop =
      up -> MotorUp
      []
      down -> MotorDown

    MotorUp = arrive -> MotorStop

    MotorDown = arrive -> MotorStop
  within
    MotorStop
```

Example

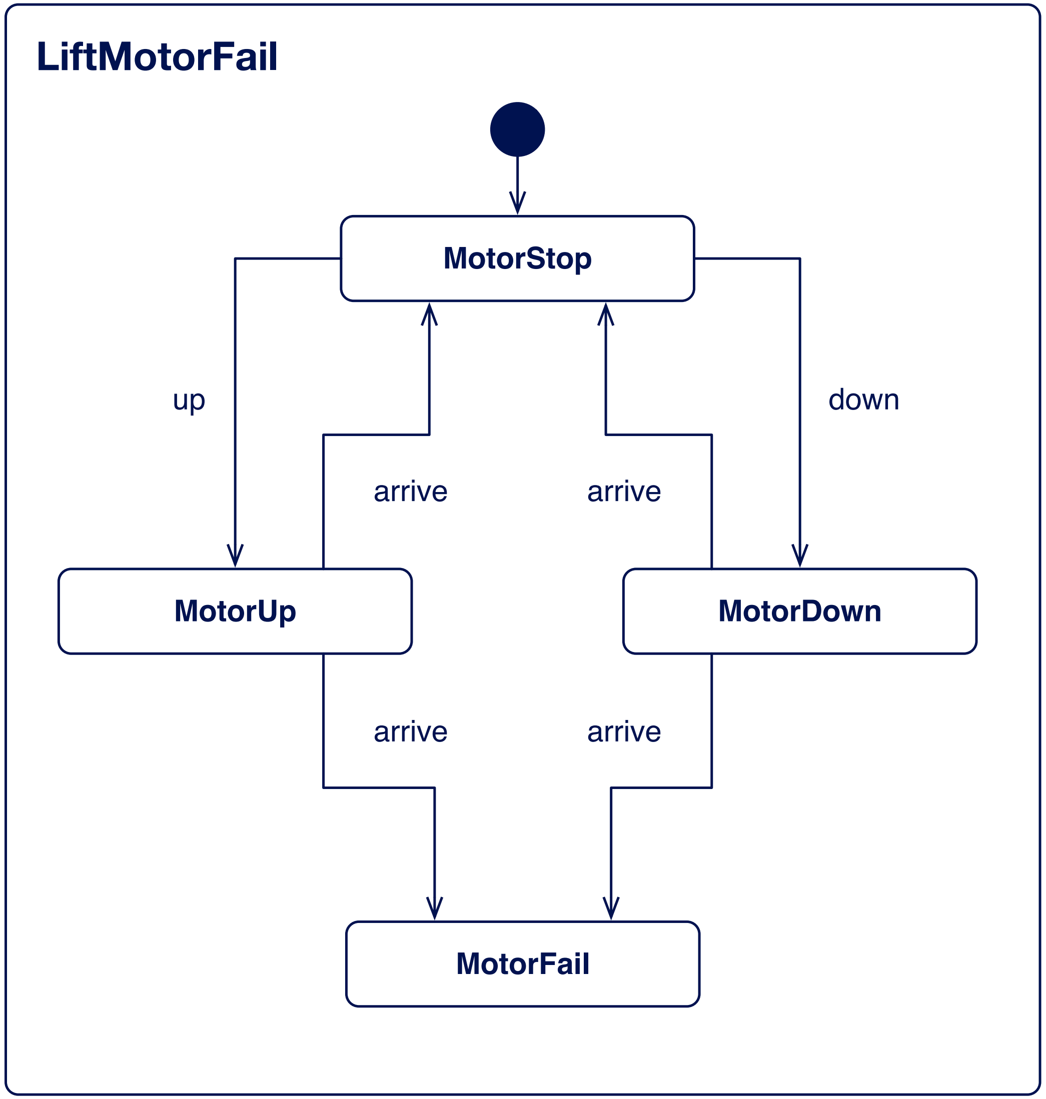
```
LiftController [aLiftController || aLiftMotor] LiftMotor

= AtGroundFloorClosed
  [aLiftController || aLiftMotor]
  MotorStop

= open ->
   (AtGroundFloorOpen
    [aLiftController || aLiftMotor]
    MotorStop)
  []
  up ->
   (GoingUp
    [aLiftController || aLiftMotor]
    MotorUp)
```

Example


AtGroundFloorClosed
open
close
up
up
arrive
arrive
open
close
down
down


LiftControllerAndMotor
up
arrive
close
open
open
close
down
up
down
arrive

Example


LiftMotorFail
MotorStop
up
down
arrive
arrive
MotorUp
MotorDown
arrive
arrive
MotorFail

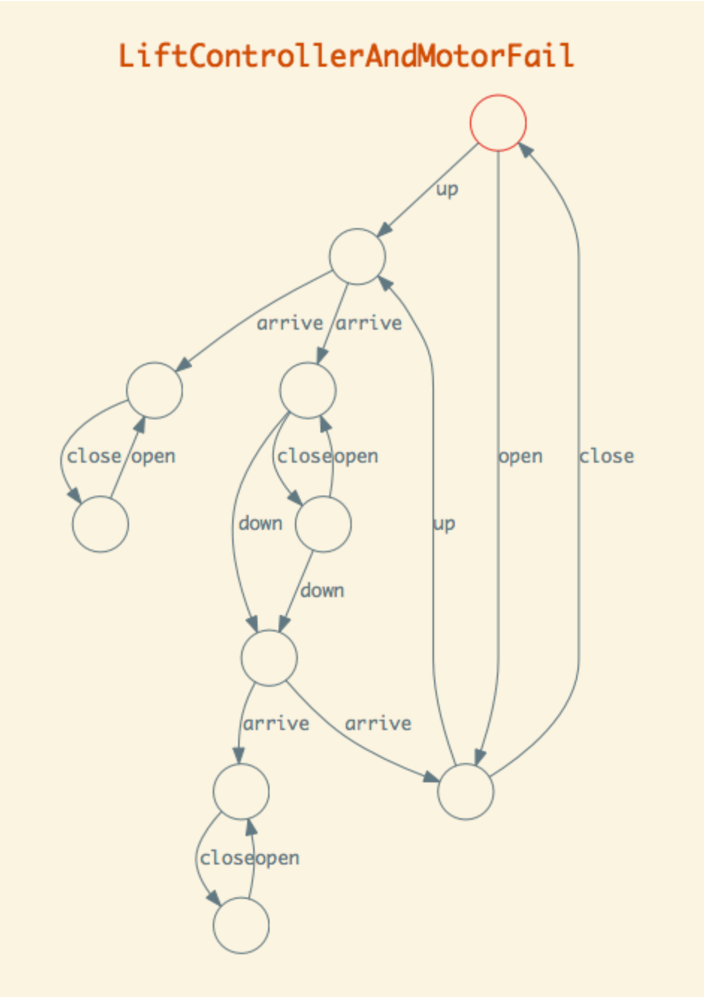
Example

```
LiftMotorFail =
  let
    MotorStop =
      up -> MotorUp [] down -> MotorDown

    MotorUp =
      arrive -> MotorStop [] arrive -> MotorFail

    MotorDown =
      arrive -> MotorStop [] arrive -> MotorFail

    MotorFail = STOP
  within
    MotorStop
```

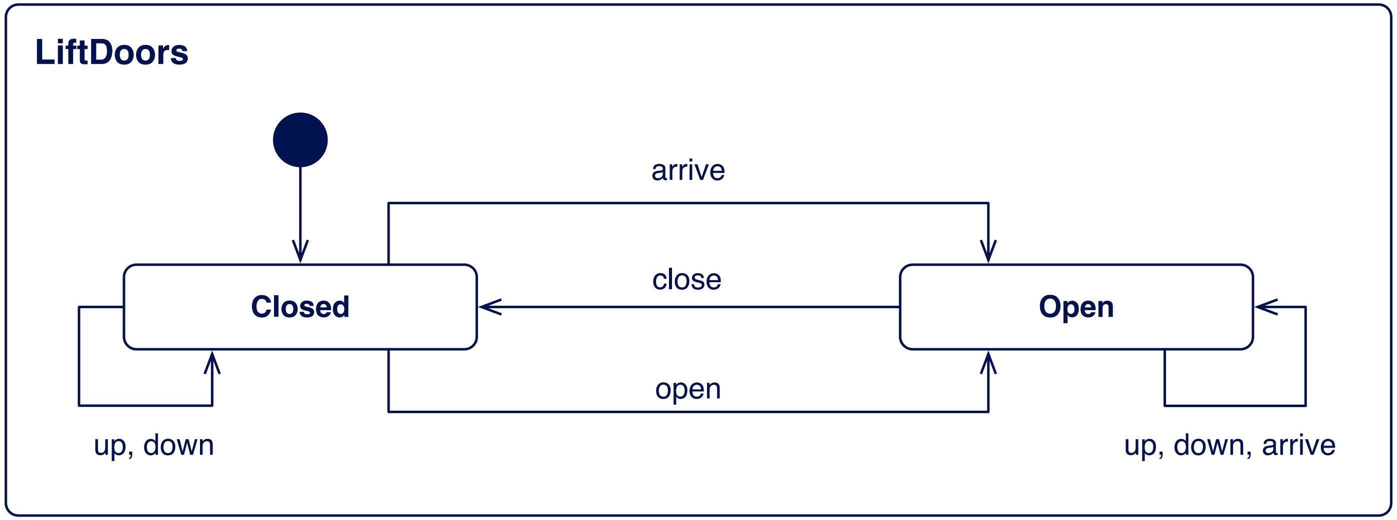
Example

```
LiftController
[aLiftController || aLiftMotor]
LiftMotorFail

= open -> ...
  []
  up ->
    (arrive ->
       ...
       )
```

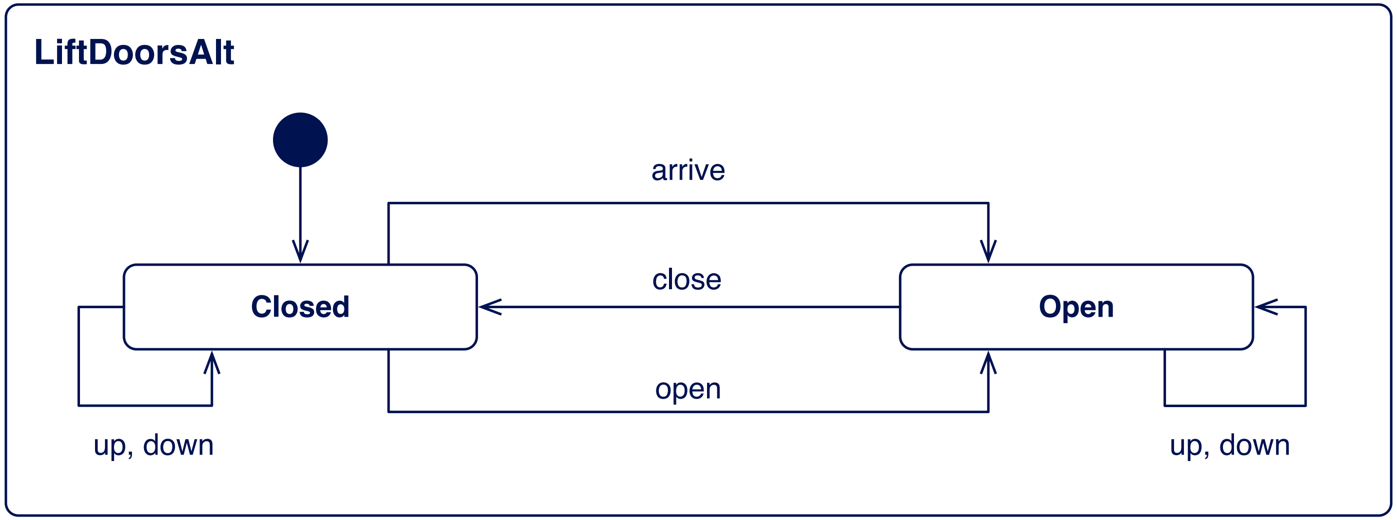
Example


LiftControllerAndMotorFail
up
arrive
arrive
close
open
close
open
open
close
down
up
down
arrive
arrive
close
open

Example


LiftDoors
arrive
close
Closed
Open
open
up, down
up, down, arrive

Example



Example

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

LiftDoors =
  let
    Closed =
      ( [] e : {open,arrive} @ e -> Open )
      []
      ( [] e : {up,down} @ e -> Closed )

    Open =
      ( [] e : {up,down,arrive} @ e -> Open )
      []
      ( close -> Closed )
  within
    Closed
```

Example

```
LiftDoorsAlt =
  let
    Closed =
      ( [] e : {open,arrive} @ e -> Open )
      []
      ( [] e : {up,down} @ e -> Closed )

    Open =
      ( [] e : {up,down} @ e -> Open )
      []
      ( close -> Closed )
  within
    Closed
```

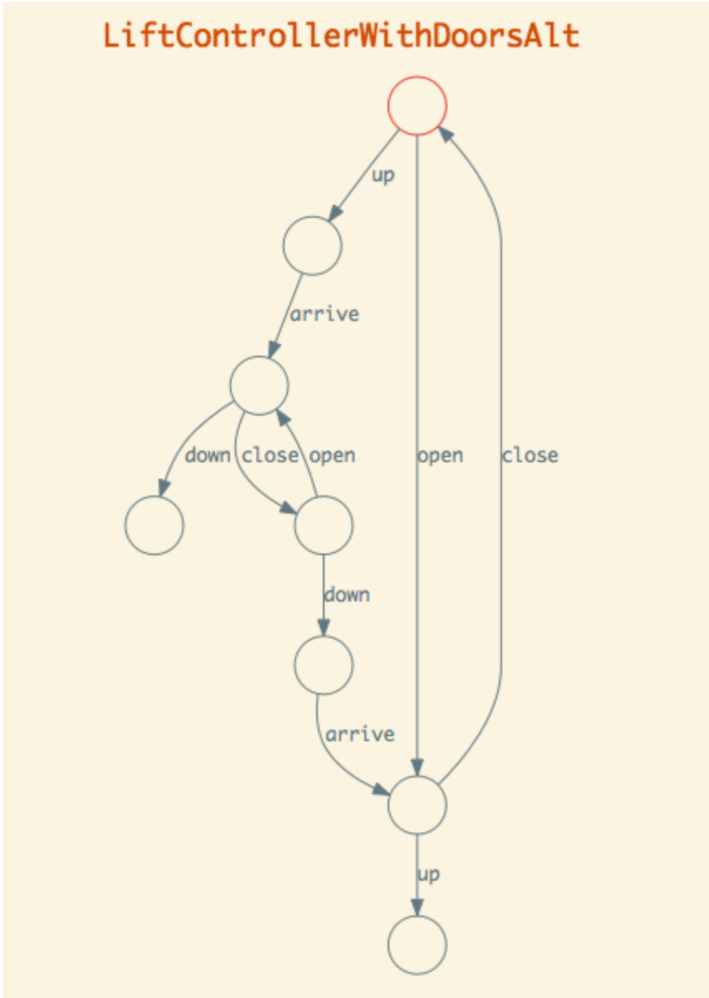
Example

```
LiftController [ aLiftController || aLiftDoors ] LiftDoors
[F=
LiftController

LiftController
[F=
LiftController [ aLiftController || aLiftDoors ] LiftDoors

LiftController [ aLiftController || aLiftDoors ] LiftDoorsAlt
=
?
```

```
LiftController
[ aLiftController || aLiftDoors ]
LiftDoorsAlt
=
AtGroundFloorClosed
[ aLiftController || aLiftDoors ]
Open
=
open -> ... [] up -> ...
[ aLiftController || aLiftDoors ]
close -> ... [] up -> ... [] down -> ...
=
up ->
 (GoingUp
  [ aLiftController || aLiftDoors ]
  Open)
=
```

```
=
up ->
 (arrive -> ...
  [ aLiftController || aLiftDoors ]
  close -> ... [] up -> ... [] down -> ...
  )
=
up -> STOP
```

Example


LiftControllerWithDoorsAlt
up
arrive
down
close
open
open
close
down
arrive
up

Deadlock

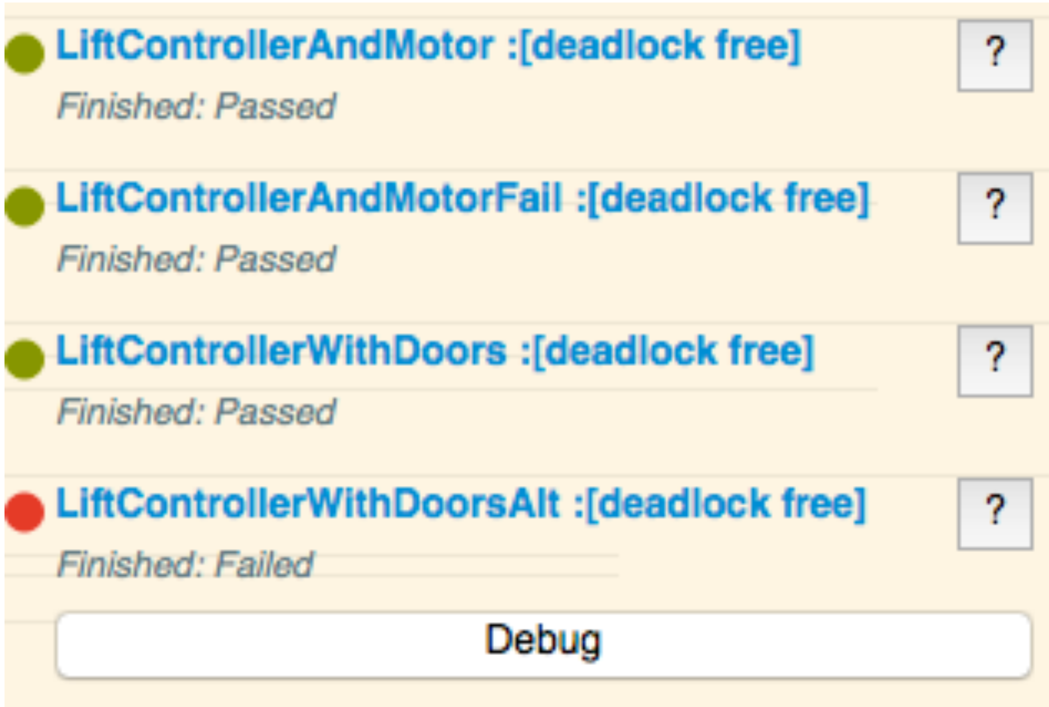
If `P` is a process-valued expression, then we may use the following assertion to check whether `P` can reach a state in which no further events are allowed:

```
assert P :[deadlock free]
```

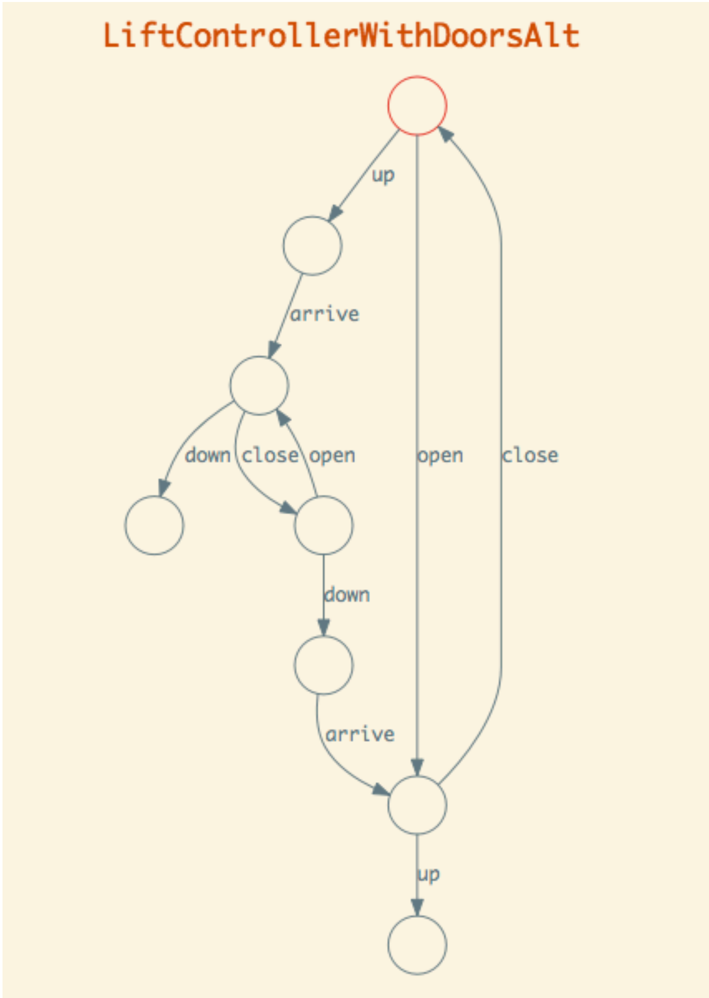
Example

```
assert LiftControllerAndMotor
  :[deadlock free]

assert LiftControllerAndMotorFail
  :[deadlock free]

assert LiftControllerWithDoors
  :[deadlock free]

assert LiftControllerWithDoorsAlt
  :[deadlock free]
```

Example


LiftControllerAndMotor :[deadlock free]
?
Finished: Passed
LiftControllerAndMotorFail :[deadlock free]
?
Finished: Passed
LiftControllerWithDoors :[deadlock free]
?
Finished: Passed
LiftControllerWithDoorsAlt :[deadlock free]
?
Finished: Failed
Debug

Example



LiftControllerWithDoorsAlt
up
arrive
down
close
open
open
close
down
arrive
up

Algebra

```
P [ aP || aP ] P = P        (if P is deterministic)

P [ aP || aQ ] Q = Q [ aQ || aP ] P

P [ aP || union(aQ,aR) ] (Q [ aQ || aR ] R)
=
(P [ aP || aQ ] Q) [ union(aP,aQ) || aR ] R
```

Distributive

```
P [ aP || aQR ] (Q |~| R)
=
(P [ aP || aQR ] Q) |~| (P [ aP || aQR ] R)
```

Example

```
aHelpful = aAwkward = { goOut, stayIn } = A

Helpful = goOut -> STOP [] stayIn -> STOP

Awkward = goOut -> STOP |~| stayIn -> STOP
```

```
Helpful [ A || A ] Helpful =
  goOut -> STOP
  []
  stayIn -> STOP

Helpful [ A || A ] Awkward =
  goOut -> STOP
  |~|
  stayIn -> STOP

Awkward [ A || A ] Awkward =
  goOut -> STOP
  |~|
  stayIn -> STOP
  |~|
  STOP
```

```
Helpful [ A || A ] Helpful =
  Helpful

Helpful [ A || A ] Awkward =
  Awkward

Awkward [ A || A ] Awkward =
  Awkward
  |~|
  STOP
```

```
External [ A || A ] External =
  External

External [ A || A ] Internal =
  Internal

Internal [ A || A ] Internal =
  Internal
  |~|
  STOP
```

Summary

- Parallel composition
- Deadlock
- Algebra

Index