State

Concurrency and Distributed Systems

November 2023

Contents

- State
- Parameters
- Indexed choice
- Algebra

State

A sequential process can also be described in terms of the effect of each transaction upon a set of variables that characterise the state of the process.

In graphical terms, this allows us to collapse the description of the process, perhaps even to a single state.

state-based modelling

Variables

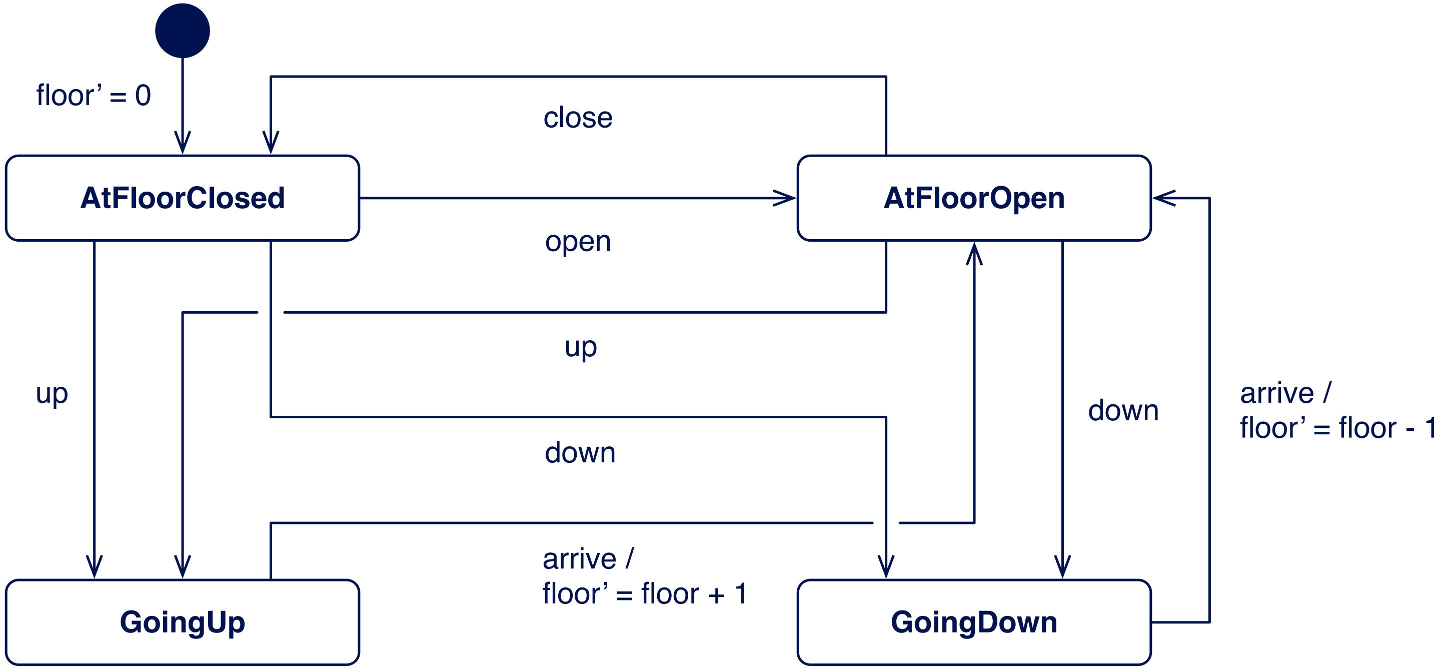
The action language used with UML state diagrams is typically imperative.

We might prefer use a declarative notation instead, writing a predicate to describe the relationship between the state before and the state after a transaction.

For example, the value of a variable `x` after a transaction could be written `x'`: that is, decorated with a single closing quote.

some notations decorate the before value instead: e.g. x_0

Example


floor’ = 0
close
AtFloorClosed
AtFloorOpen
open
up
up
down
down
arrive /
floor’ = floor - 1
arrive /
floor’ = floor + 1
GoingUp
GoingDown

Parameters

Parameters in process names can serve two purposes:

- to represent state information;
- to identify an instance of a generic process.

They are written in parentheses: `P(x,y)`

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

Built-in data types include sets and sequences.

Process state

Where a parameter x is used to represent state information, it may take a new value at each recursive invocation of the process name.

This new value may be determined by an expression involving x, together with any other variables that are in scope.

Example

```
LiftController =
  AtFloorOpen(0)

AtFloorOpen(0) =
  close -> AtFloorClosed(0)
  []
  up -> GoingUp(0)

AtFloorClosed(0) =
  open -> AtFloorOpen(0)
  []
  up -> GoingUp(0)

GoingUp(0) =
  arrive -> AtFloorOpen(1)
```

Example

```
AtFloorOpen(1) =
  close -> AtFloorClosed(1)
  []
  up -> GoingUp(1)
  []
  down -> GoingDown(1)

AtFloorClosed(1) =
  open -> AtFloorOpen(1)
  []
  up -> GoingUp(1)
  []
  down -> GoingDown(1)
```

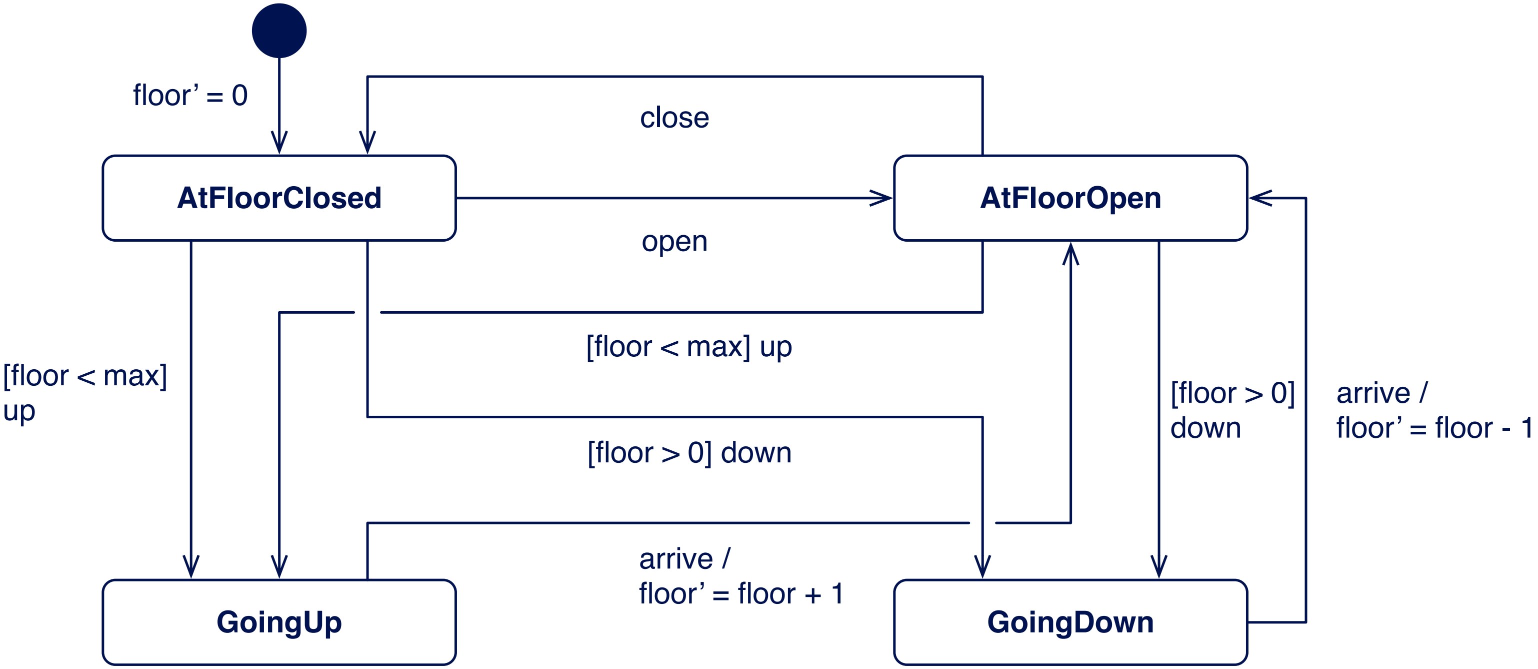
Example

```
GoingUp(1) =
  arrive -> AtFloorOpen(2)

GoingDown(1) =
  arrive -> AtFloorOpen(0)

...
```

Guards in state machines

Just as in UML state machines, we may prefix a transaction with a Boolean expression in square brackets to indicate that the transaction should be available only if the corresponding condition is true.

Example


floor’ = 0
close
AtFloorClosed
AtFloorOpen
open
[floor < max]
up
[floor < max] up
[floor > 0]
down
arrive /
floor’ = floor - 1
[floor > 0] down
arrive /
floor’ = floor + 1
GoingUp
GoingDown

Conditional

If `B` is a Boolean-valued expression, and `E` and `F` are expressions of the same type, then

```
  if B then E else F
```

is an expression that takes the value of `E` if `B` evaluates to `true` and that of `F` if `B` evaluates to `false`.

There are no side-effects associated with the evaluation.

this is not an imperative language

Guards in CSP

If `B` is a Boolean-valued expression, and `P` is a process-valued expression, then

```
B & P
```

takes the value of `P` if `B` evaluates to `true` and that of `STOP` if `B` evaluates to `false`.

```
B & P = if B then P else STOP
```

Stop and external choice

Using the failures semantics, we can check that for any process P

```
P [] STOP = P
```

that is, the inclusion of STOP makes no difference to an external choice.

we can use the semantic equations to check this, or the step laws—see below

Guards and external choice

This allows us to write

```
B & P [] C & Q
```

instead of

```
if (B and C) then
  P [] Q
else if B then
  P
else if C then
  Q
else
  STOP
```

Example

```
LiftController(max) =
  AtFloorOpen(0)

AtFloorOpen(n) =
  close -> AtFloorClosed(n)
  []
  (n < max) & up -> GoingUp(n)
  []
  (n > 0) & down -> GoingDown(n)
```

Example

```
AtFloorClosed(n) =
  open -> AtFloorOpen(n)
  []
  (n < max) & up -> GoingUp(n)
  []
  (n > 0) & down -> GoingDown(n)

GoingUp(n) =
  arrive -> AtFloorOpen(n + 1)

GoingDown(n) =
  arrive -> AtFloorOpen(n - 1)
```

Algebra

If `P` and `Q` are process-valued expressions such that

```
P = Q
```

then we can substitute `P` for `Q`, or vice versa, in any other expression.

substitutivity

Example

```
LiftController =
  AtFloorOpen(0)

AtFloorOpen(n) =
  close -> AtFloorClosed(n)
  []
  (n < max) & up -> arrive -> AtFloorOpen(n+1)
  []
  (n > 0) & down -> arrive -> AtFloorOpen(n-1)
```

Example

```
AtFloorClosed(n) =
  open -> AtFloorOpen(n)
  []
  (n < max) & up -> arrive -> AtFloorOpen(n+1)
  []
  (n > 0) & down -> arrive -> AtFloorOpen(n-1)
```

Local definitions

If D is a set of defining equations and E is an expression, then

```
  let D within E
```

is an expression that takes the value of E in the (additional) context provided by the equations D.

This mechanism allows us to use a name more than once, provided that we do so in two different sets of local, defining equations.

It allows us also to make clear the entry point into a set of mutually-recursive equations.

Example

```
LiftController =
  let
    AtFloorOpen(n) =
      close -> AtFloorClosed(n)
      []
      (n < max) & up -> arrive -> AtFloorOpen(n+1)
      []
      (n > 0) & down -> arrive -> AtFloorOpen(n-1)

    AtFloorClosed(n) =
      ...
  within
    AtFloorOpen(0)
```

Generic parameters

We may use parametrised names to introduce multiple instances of the same generic process.

The behaviour of these instances will usually depend upon the actual value of the parameter.

Example

```
LiftController(max) =
  let
    AtFloorOpen(n) =
      close -> AtFloorClosed(n)
      []
      (n < max) & up -> arrive -> AtFloorOpen(n+1)
      []
      (n > 0) & down -> arrive -> AtFloorOpen(n-1)

    AtFloorClosed(n) =
      ...
  within
    AtFloorOpen(0)
```

Indexed choice

We may define parameterised versions of the two choice operators.

These can be used to describe the availability and effect of a collection of events in a single expression.

This is useful now, but will be essential later, when we come to use collections of parameterised events (channels) in parallel compositions.

Indexed external choice

If P(x) is a process-valued expression with parameter x, then

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

is a process that is ready to behave as any one of the processes P(x) for which x is an element of the set X.

For example,

```
[] x : {1,2,3} @ P(x) = P(1) [] P(2) [] P(3)
```

The range set X may be any finite set of data values of any type, including events.

Example

```
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

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

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

`Open` and `Closed` are both well defined.

Indexed internal choice

If P(x) is a process-valued expression with parameter x, then

```
|~| x : X @ P(x)
```

is a process that could be any of the processes P(x) for which x is an element of the set X.

For example,

```
|~| x : {1,2,3} @ P(x) = P(1) |~| P(2) |~| P(3)
```

The range set X may be any non empty finite set of data values of any type, including events.

Example

The following process is always ready to allow exactly one event from the set of events A:

```
DF(A) = |~| a : A @ a -> DF(A)
```

It may refuse any combination of events from A, except for the set A itself: it can't refuse everything.

It is the most nondeterministic deadlock-free process for this set of events.

Algebra

Two processes are equivalent if they have exactly the same failures (and hence also traces) and divergences.

Our process language has algebraic properties.

Indeed, the meaning of the language can be characterised entirely by a set of algebraic laws.

substitutivity

Internal choice

```
        P |~| P = P
        P |~| Q = Q |~| P
P |~| (Q |~| R) = (P |~| Q) |~| R
```

External choice

```
       P [] P = P
       P [] Q = Q [] P
P [] (Q [] R) = (P [] Q) [] R

    P [] STOP = P
```

Distributive Laws

We say that one binary operator op1 distributes over another op2 if the result of applying op1 to the result of applying op2 is the same as that obtained by applying op1 to each operand separately, and then combining them with op2.

```
a op1 (b op2 c) = (a op1 b) op2 (a op1 c)
```

For example, in arithmetic, * distributes over +

```
a * (b + c) = (a * b) + (a * c)
```

Example

```
a -> (P |~| Q) = (a -> P) |~| (a -> Q)

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

we can use the first law to show that

```
a -> P [] a -> Q = a -> P |~| a -> Q
```

these are distributive laws

Step laws

In every case, the next step in the evolution of a process can be described as an internal choice of external choices.

A step law explains which sets of events are allowed at the next step when the operator is applied to one or more external choices.

the distributive property does the rest

Step law: external choice

If

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

then

```
P [] Q = [] e : union(A,B) @ e ->
             if member(e,diff(A,B)) then
               P(e)
             else if member(e,diff(B,A)) then
               Q(e)
             else
               P(e) |~| Q(e)
```

Example

```
          P [] STOP = P

          STOP [] Q = Q

a -> P [] a -> Q = a -> (P |~| Q)
```

Exploring behaviours

We can define step laws for all of the other operators in our language: in particular, for || and \.

With tool support, this means that we can explore the behaviours of any well-defined process, one state at a time.

Summary

- State
- Parameters
- Indexed choice
- Algebra

Index