Processes

Concurrency and Distributed Systems

November 2023

Contents

Processes

process, *n.*

1. That which goes on or is carried on; a continuous action, or series of actions or events; […]

   1604 W. Shakespeare *Hamlet* III. iii. 29 Behinde the Arras I'le conuay my selfe To heare the processe.

   ⋮

8. *Computing.* An instance of the execution of a program in a multitasking operating system, typically in an environment that protects it from other processes.

   2009 S. Pugh *Wicked Cool Ruby Scripts* iii. 51 If a process has been expending CPU cycles for more than a reasonable amount of time, then it's time to kill the process.

Modelling

modelling, *n.*

2. The devising or use of abstract or mathematical models.

1965 C. H. Springer et al. *Advanced Methods & Models* i. 4
The best way to untangle the confusion which many people have about mathematical modeling as a method for solving important business problems is to untangle the whole idea of model building as a way of thinking about the world we live in.

Model

model, *v.*

2. To construct a model of theory of the structure of.

1667 J. MILTON *Paradise Lost* VIII. 79 When they come to model Heav'n And calculate the Starrs.

⋮

10. *transitive.* To devise a (usually mathematical) model or simplified description of (a phenomenon, system, etc.)

1960 *Jrnl. Royal Statist. Soc.* B. **22** 242 In order to obtain complete predictive accuracy the model would be no simpler than what was being modelled, and would not be a model.

Oxford English Dictionary

Process modelling

In this course, we will learn how to model or describe processes—whether these are business processes, scientific workflows, or processes in an operating system—in terms of actions that may be performed or events that may occur.

Our descriptions will be in terms of possibilities rather than probabilities, allowing us to answer questions such as:

- could this sequence of events ever occur?
- is this action always available?
- will this workflow complete successfully?

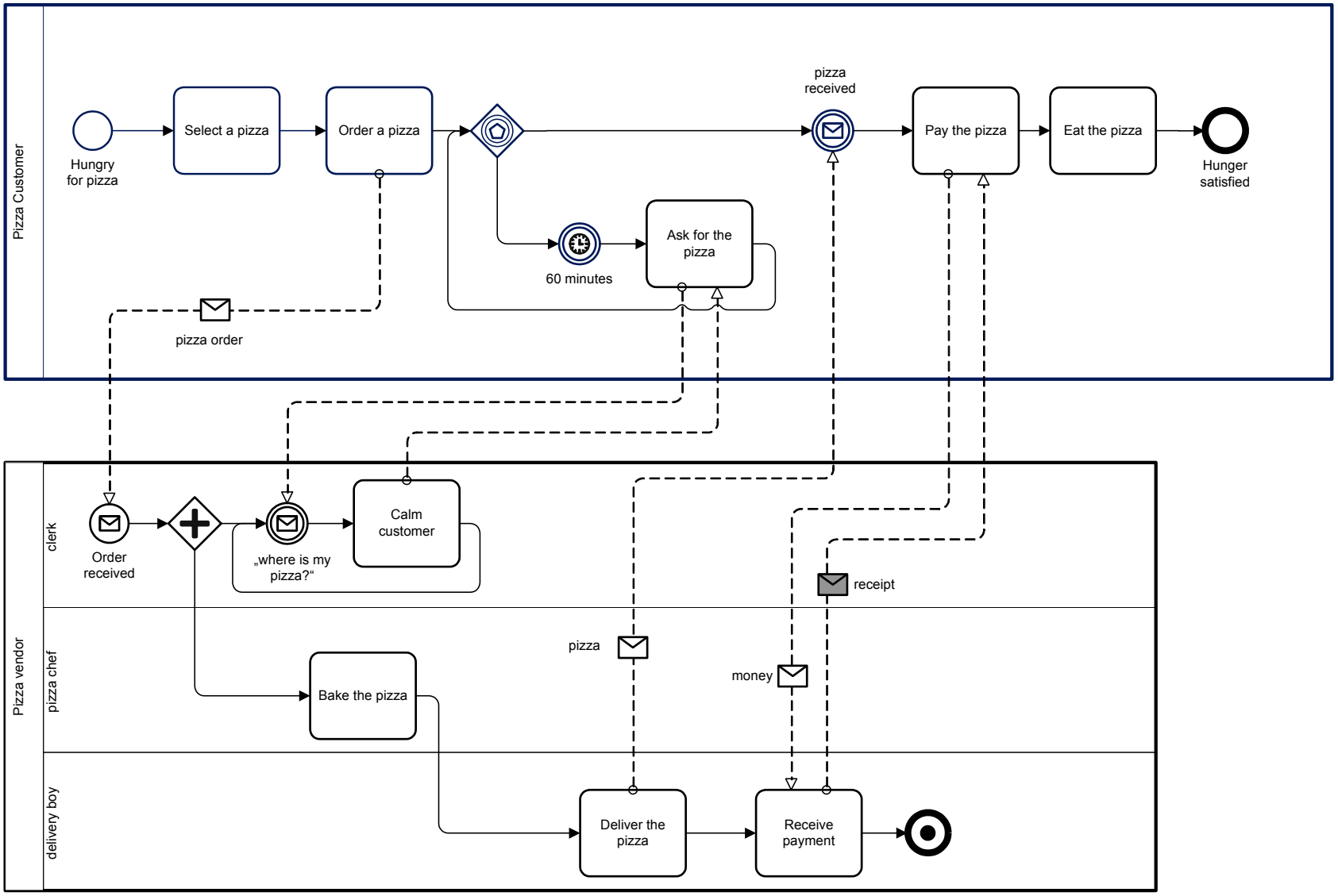
Modelling languages

Many modelling languages are graphical.

The leading example is the Unified Modeling Language from the Object Management Group (OMG), a commercial standards organisation, with its state diagram and activity diagram notations.

OMG are also responsible for BPMN: the Business Process Modeling Notation. This may be seen as a more complicated version of the activity diagram notation.

BPMN


Pizza Customer
Hungry for pizza
Select a pizza
Order a pizza
pizza received
Pay the pizza
Eat the pizza
Hunger satisfied
60 minutes
Ask for the pizza
pizza order
Pizza vendor
clerk
pizza chef
delivery boy
Order received
„where is my pizza?“
Calm customer
receipt
pizza
money
Bake the pizza
Deliver the pizza
Receive payment

BPMN Charter

“A standard Business Process Model and Notation (BPMN) will provide businesses with the capability of understanding their internal business procedures in a graphical notation and will give organizations the ability to communicate these procedures in a standard manner. […]”

Purposes

- a standard notation for sketching aspects of design
- a programming notation for scripting workflows
- a specification language:
  - for testing properties of a design
  - for proving correctness at a higher level of abstraction

(a formal specification language)

Edsger W. Dijkstra

Testing shows the presence, not the absence of bugs.

> NATO Software Engineering Conference, Rome 1969
>
> http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF

The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.

> 1972 Turing Award Lecture

A picture may be worth a thousand words, a formula is worth a thousand pictures.

> EWD1239: A first exploration of effective reasoning

Our course

We will:

- understand what is required of a process modelling notation—graphical or otherwise—if it is to be used as a formal specification language
- understand how we might use a process modelling language with a formal semantics to express, test, or prove properties of processes

Requirements

We need a process modelling language with a logical or mathematical semantics: mapping each term in the language to a unique mathematical meaning.

The language, and its semantics, needs to be compositional: the meaning of a term can be calculated from the meanings of its component parts.

Finally, the semantics needs to support abstraction. We wish to be able to construct and compare descriptions of the same system at different levels of abstraction.

Why?

- consistency: we shouldn't be making contradictory statements about our descriptions
- scalability: we shouldn't have to describe everything as one big process; we should be able to put descriptions together
- assurance: we should be able to test or prove that the statements we make are correct*

*a test will involve a description of the same system at a higher level of abstraction

Semantics

semantics, *n.*

1. The meaning of signs; the interpretation or description of such meaning; […]

   ⋮

5. *Computing.* The meaning of the strings in a programming language.

   1964 *IEEE Trans. Electronic Computers* **13** 343/2 A compiler and a description of the machine for which it compiles is a complete and formal description of the syntax (i.e., grammar) and semantics (i.e., meaning).

Sequential Semantics

We can give a semantics to a sequential, imperative programming notation using Hoare logic (1969).

For example, the assignment `a := E` will result in a state satisfying predicate `P` if and only if it starts from a state satisfying `P[E/a]`: that is, one in which `P` is true for `E` in place of `a`.

```
{ P[E/a] }  a := E  { P }
```

for any predicate `P`, expression `E`, and variable `a`.

Example

The assignment `a := b` will leave the system in a state in which `a` is strictly greater than `3` if and only if it starts from a state in which `b` is strictly greater than `3`.

```
{ b > 3 }  a := b  { a > 3 }
```

Assertions

How can one check a large routine in the sense of making sure that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.

Alan Turing, Cambridge, June 24 1950

Concurrency Semantics

In a sequential context, the meaning of the programming statement

```
a := 1 ; b := a
```

is easily understood. After the statement has been executed, variables `a` and `b` both have the value `1`.

In a concurrent or parallel context, we have to consider the points at which other programs might change the value of `a` and/or `b`.

Concurrency

concurrency, *n.*

1. A running together in place or time.

   1597 R. Hooker *Of Lawes Eccl. Politie v*, in *Wks.* II. 121 Is it probable that God should…command concurrency of rest with extraordinary occasions of doleful events?

   ⋮

2. Accordance in operation or opinion; co-operation; consent […]

   1596 W. Warner *Albions Eng.* (rev. ed.) ix. xlix. 228 But much more Concurrancie from one to all to stop that common Sore.

Oxford English Dictionary

Example

Consider the possible effects of

```
a := 1 ; b := a
```

executing concurrently with

```
a := 2
```

Semantics

We can give a concurrency semantics to our language by mapping each term to a set of sequences of events or interactions.

This is called a trace semantics.

The trace semantics of a process is the set of all sequence of events that it can participate in.

Example

```
traces[a := 1] = { <>, <write.a.1> }

traces[b := a] =
  { <>, <read.a.1>, <read.a.2>,
    <read.a.1, write.b.1>, <read.a.2, write.b.2> }

traces[a := 2] = { <>, <write.a.2> }

traces[Memory(a)] =
  { <>, <write.a.0>, <write.a.1>, <write.a.2>,
        <write.a.3> ...
    <write.a.1, read.a.1>, ...
    <write.a.2, read.a.2>, ...
    <write.a.1, write.a.2, read.a.2>, ...
    <write.a.2, write.a.1, read.a.1>, ... }
```

Example

The parallel composition of the three sequential components—

```
(a := 1 ; b := a) || a := 2 || Memory(a)
```

—has the following trace semantics:

```
{ <>, <write.a.1>, <write.a.2>,
  <write.a.1, read.a.1>,
  <write.a.2, read.a.2>,
  <write.a.1, write.a.2, read.a.2>,
  <write.a.2, write.a.1, read.a.1>,
  <write.a.1, write.a.2, read.a.2, write.b.2>,
  <write.a.2, write.a.1, read.a.1, write.b.1> }
```

State diagrams

The UML includes a state diagram notation inspired by David Harel's StateCharts language.

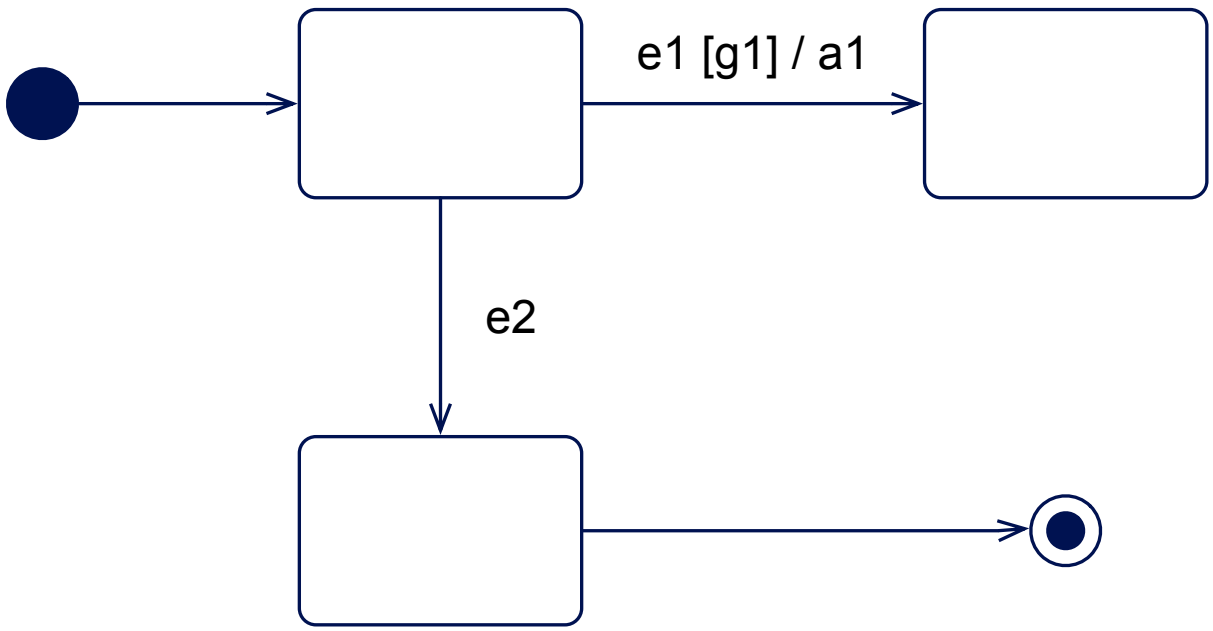
It is considerably simpler* than BPMN, and we will use it—in the following slides—as an example of a graphical process modelling language.

*in terms of syntax, at least; neither language has a simple or satisfactory semantics

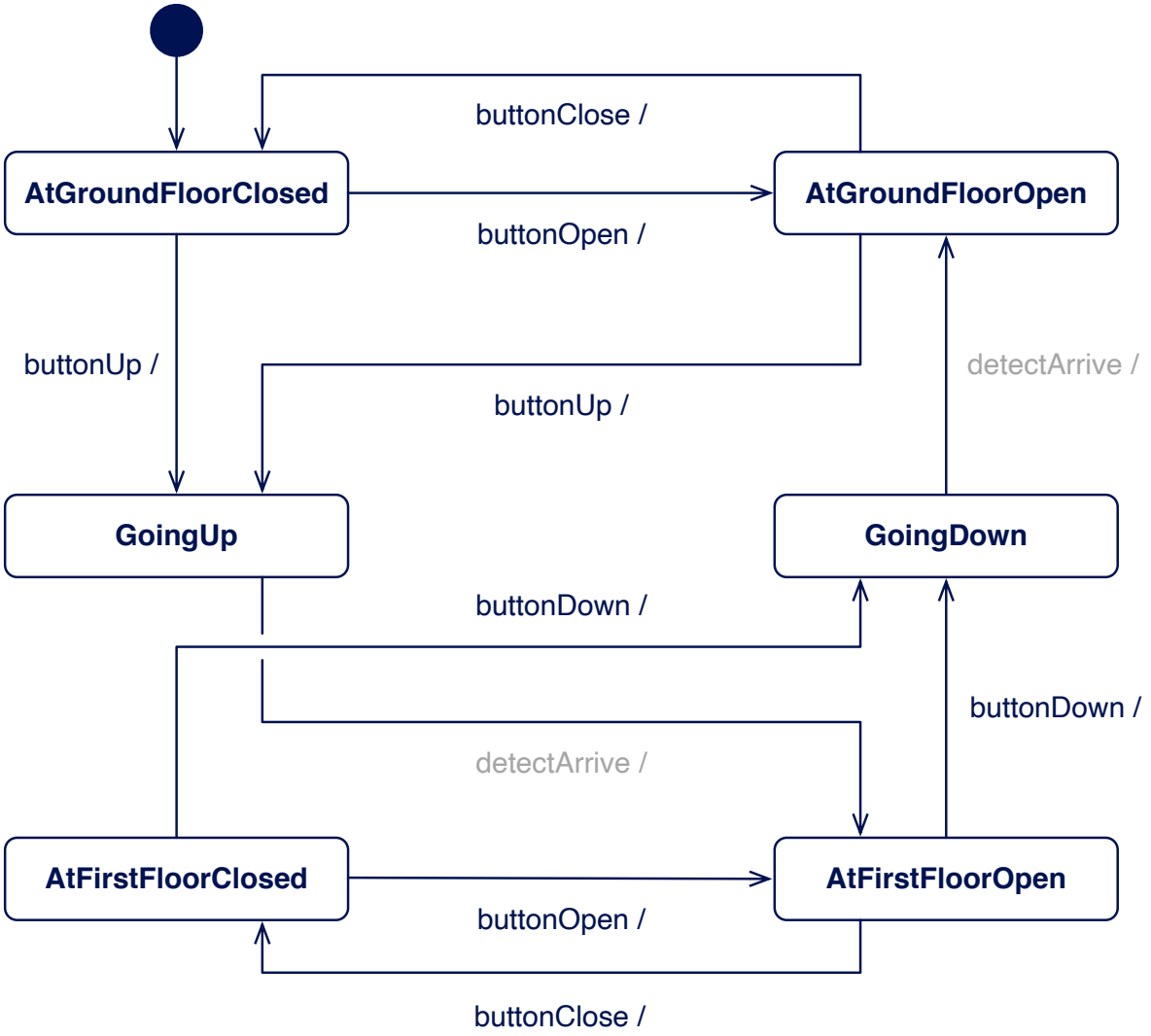
Notation

- Each box represents a state. Actually, a set of states, or a region of the state space.
- A filled circle is used to represent the initial state; a filled circle within a circle is used to represent a final state.
- Each line represents a transition between states; each transition can be
  - triggered by an event `e`;
  - constrained by a boolean-valued guard `[g]`; and
  - associated with one or more actions `a`.

Example



Example: lift controller

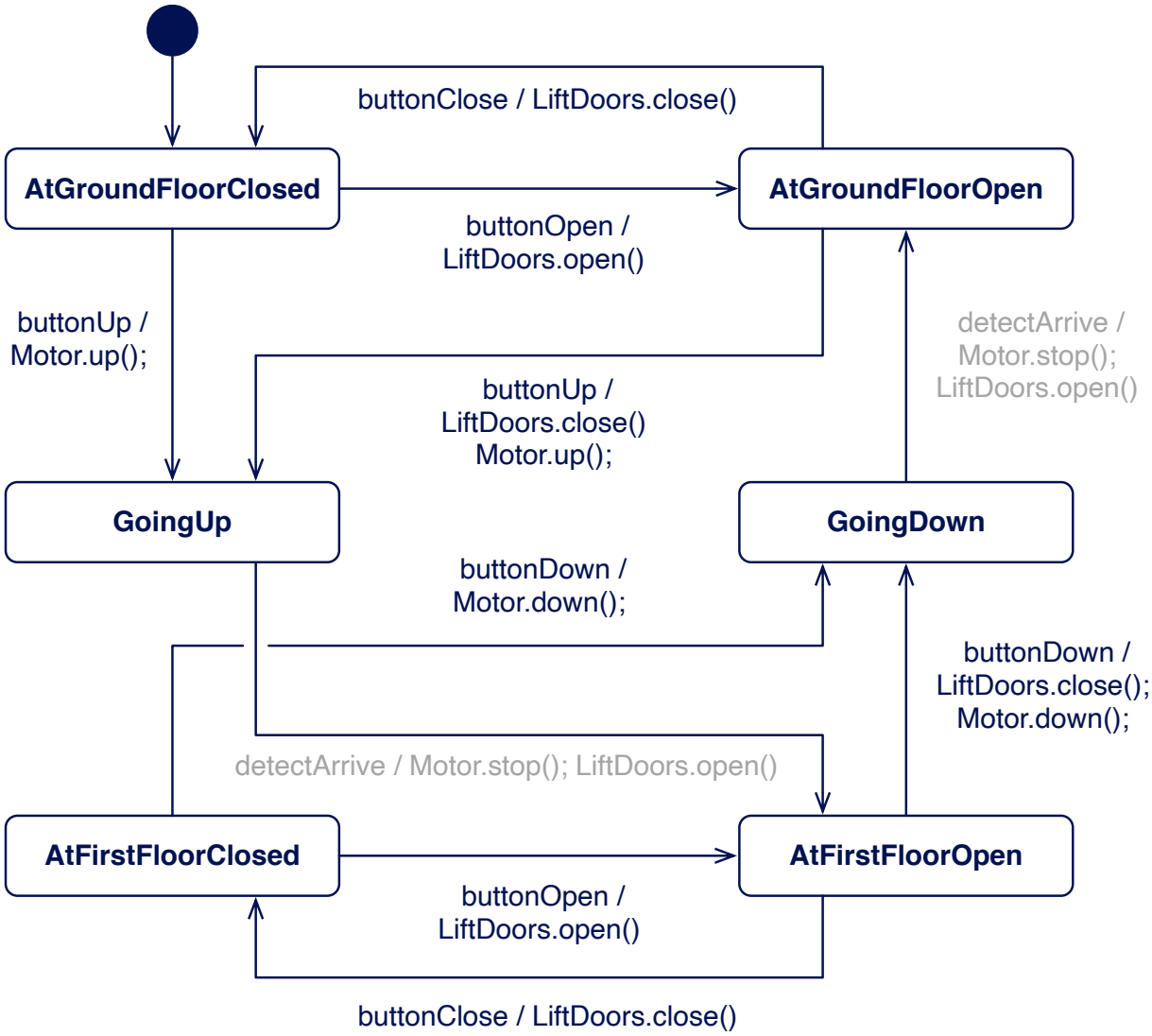


Actions

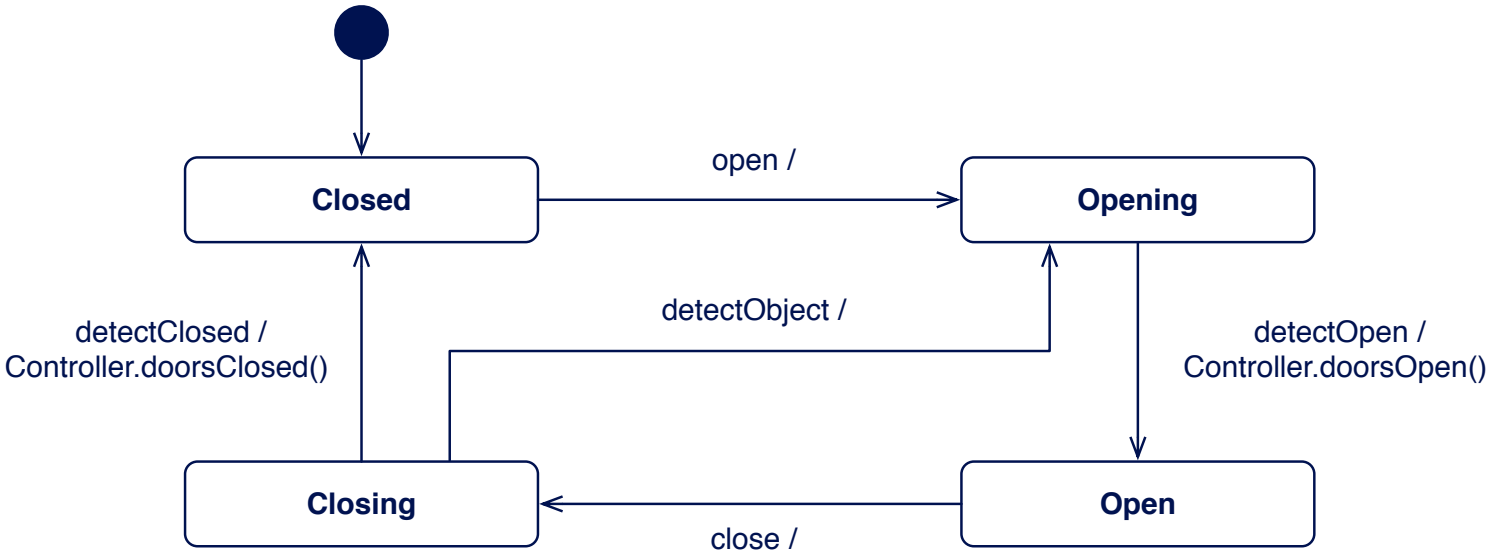
Once triggered, the actions of a transition must be completed before the ‘state machine’ can process another event.

An action may send an event to another ‘state machine’, described by another state diagram.

Example: lift controller



Example: lift doors



Asynchrony

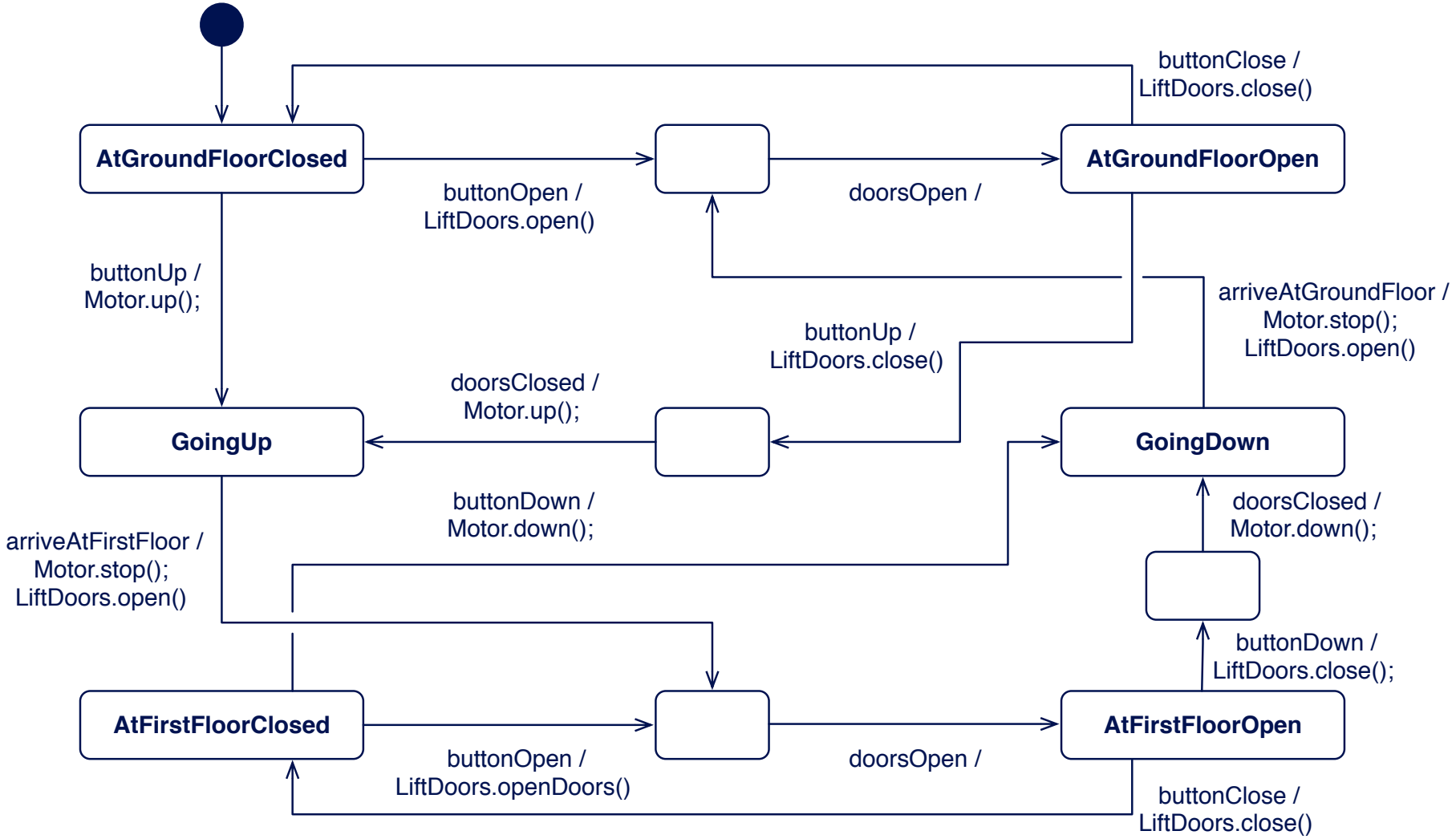
That event may be accepted even if there is no transition for it to trigger (it could be simply discarded).

The sender will not know whether a transition has been triggered unless the receiving machine sends another event back.

Asynchronous communication may cause an ‘explosion’ in the number of states we need to consider.

> **asynchrony** *n.* absence or lack of concurrence in time
> (Merriam Webster)

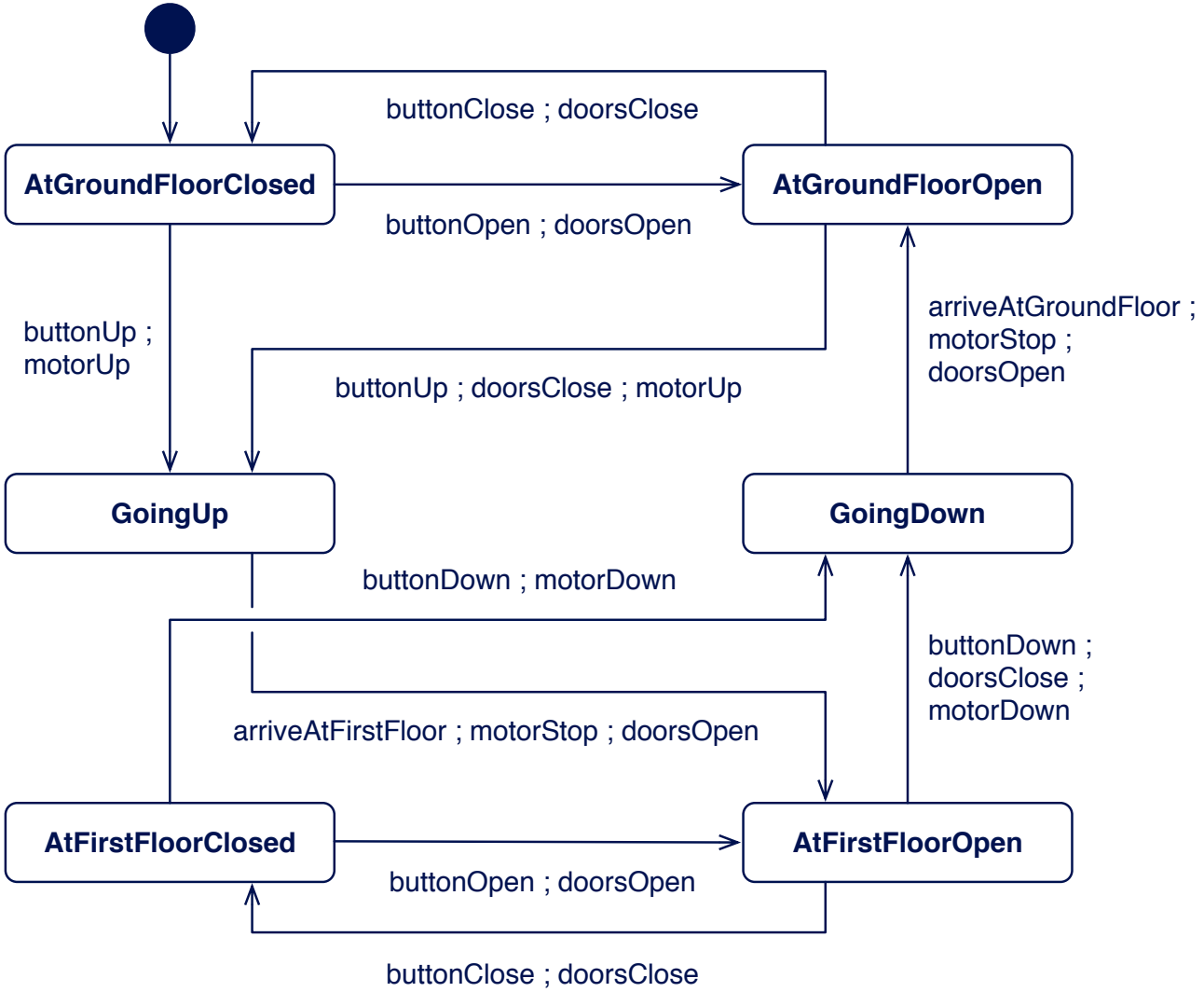
Example: lift controller with acknowledgement



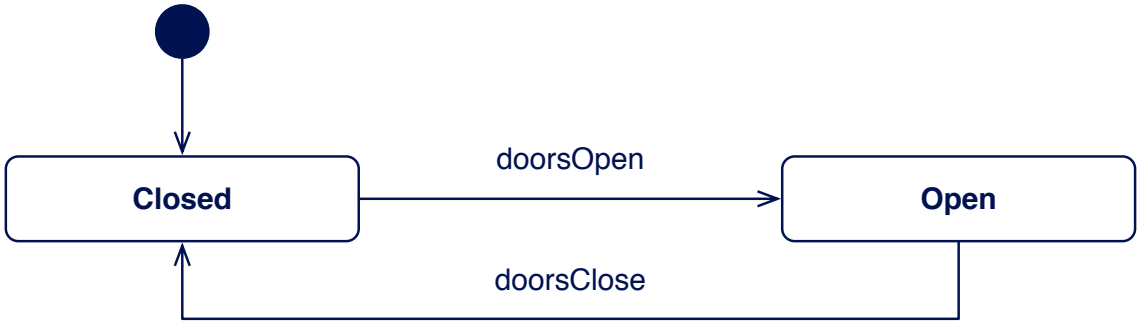
Synchronisation

We may improve upon the state diagram notation by representing the sending of a message and the receipt of an acknowledgement as a single, abstract event: a synchronisation.

Example: lift controller with synchronisations



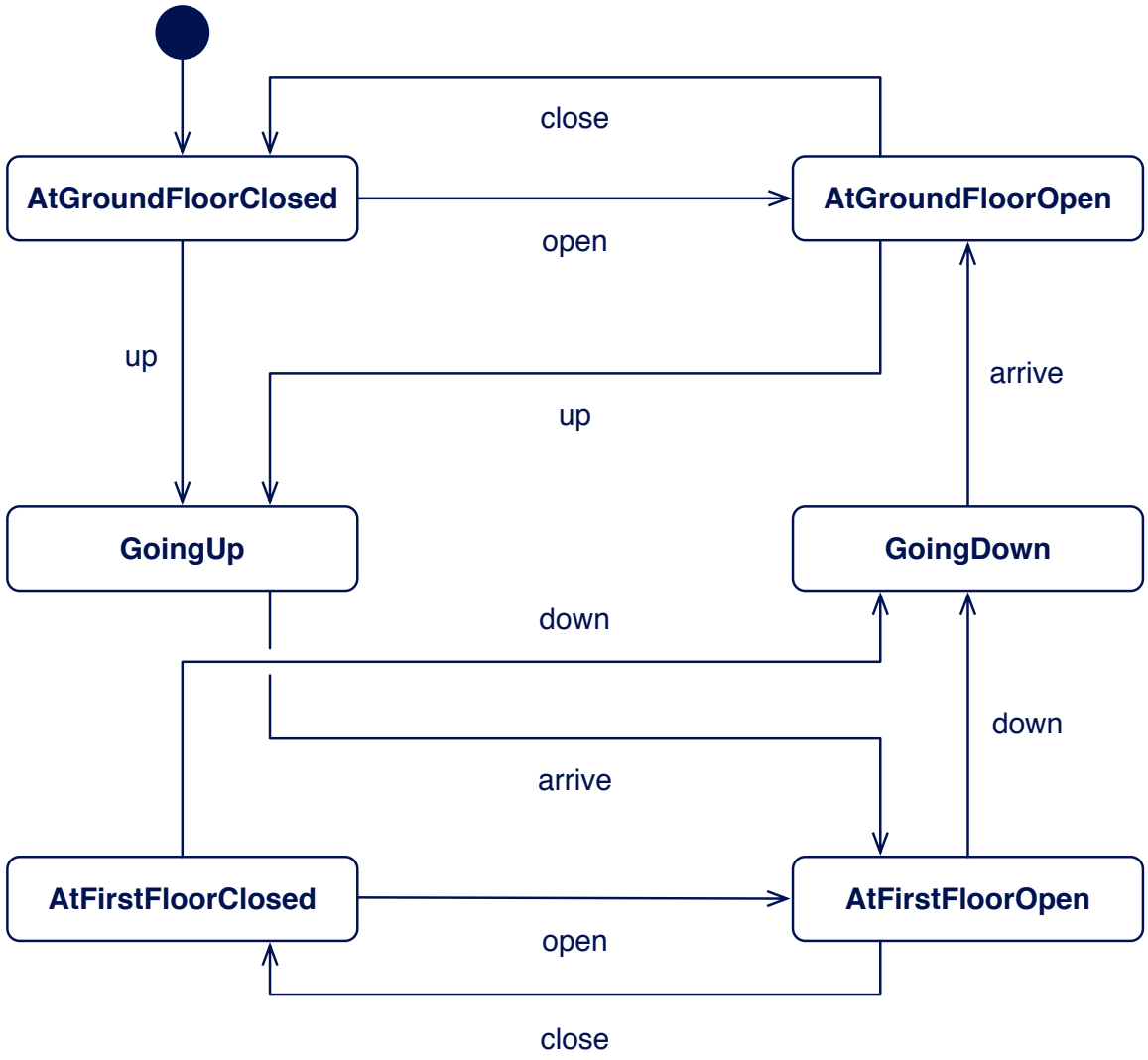
Example: lift doors with synchronisations



Transactions

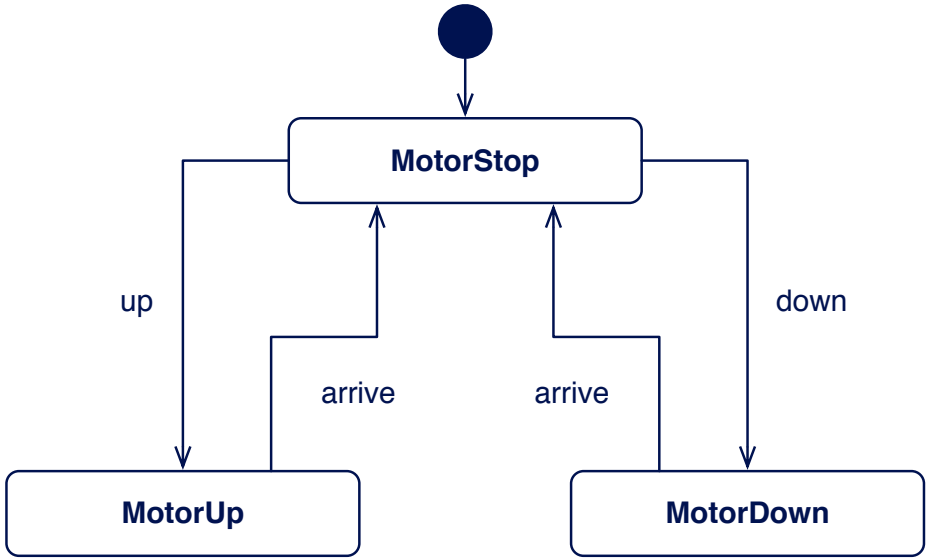
We can go further, and have any number of processes as parties to the abstract event, synchronisation, or transaction.

Example: lift controller with transactions



Example: lift doors with transactions

Example: lift motor with transactions



Transactions

In database world, a transaction is a single, 'logical' operation on data. It has the following characteristics:

- atomicity — if it happens, it happens completely
- consistency — if it happens, it happens for every party
- isolation — if two transactions can happen at the same time, they can be seen as happening in either order
- durability — if it has happened, it has happened

in our process modelling language, every event will be a transaction

Extensibility

There is a no limit to the number of parties to a given transaction.

Working with transactions, we can always add another diagram, another process, to say more about:

- when a transaction may occur, and
- what consequences it may have.

Abstraction

Our modelling language will be used to describe processes that correspond to:

- designs that we might implement;
- assumptions about the context in which they operate; and
- tests that they should pass, or specifications that they should satisfy.

Different levels of abstraction will be involved, and our processes may be nondeterministic.

Beyond diagrams

Pictures are useful for illustrating specific aspects of behaviour, within or across components.

They may be used also to describe simple, sequential components or specifications.

To explore the semantics of parallel composition and abstraction, we will switch to a textual notation*.

*or, if you like, mathematical formulae

Summary

- Processes
- Semantics
- State diagrams
- Transactions

Index