State Machines

Michael Brockway

February 5, 2018

A control system example

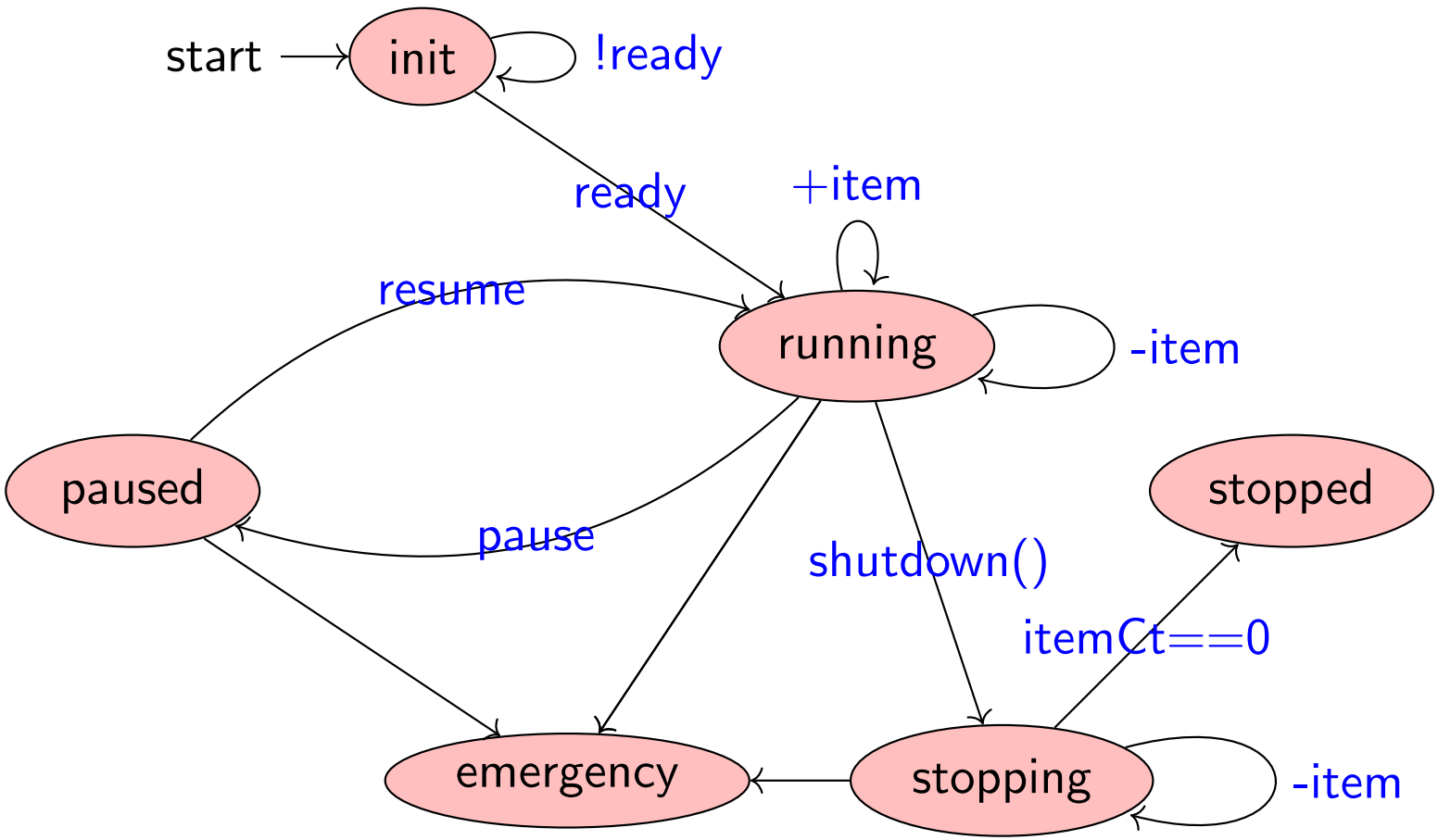
Suppose you have to write software for an industrial system. A simple example is one comprising a conveyor belt, a robot arm that picks items up from an input area and puts then on the the belt, and another robot arm that removes them from the far end of the belt.

The three components have their own control software but a *supervisory unit* needs to coordinate their actions. In particular,

- When starting the system up, it needs to know when each of the other components is ready to act;
- When shutting the system down it needs to know there are no items part-way through the system;
- It needs to be able to tell the input robot there is an item to pick up;
- It needs to know when the output robot has removed an item from the belt;
- It needs to be able to pause the system and resume operation;
- It needs to halt the system in case of emergency.

A control system example

To design software for this, think of the system being in one of a set of *states*:



A control system example

The states are

- init running paused stopping stopped emergency
- init is the *initial* state.

Transitions between states are denoted by labelled arrows

- The labels may denote *events* or *actions* or *conditions*
- !ready means the components have not all notified that they are ready; once they have (ready event) the system goes to running state.
- +item means an item has entered the system; -item means an item has been delivered by the system;
- itemCt==0 means all items have been delivered; there are none left in the system, so the system can transfer to the stopped state. Counter `itemCt` is programmed to keep track of this.

A control system example

In general, the behaviour you have to program is dependent on the state of the system. There you tend to write such constructs as

```
if (state == init} {
  if (allComponentsReady())   { state = idle; }
}
else if (state == running)  {
  if (pause())                  { state = paused; }
  else if (itemArrived())       { itemCt++; }
  else if (itemDelivered())     { itemCt--; }
  else if (shutdown())          { state = stopping; }
}
else if (state == stopping) {
    if (itemCt==0)              { state = stopped; }
    else if (itemDelivered()) { itemCt--;  }
}
else  ...
```

A control system example

Another coding style is to use the `switch` construct ...

```
switch (state) {
  case init:
    if (allComponentsReady())   { state = idle; }
    break;
  case running:
    if (pause())                { state = paused; }
    else if (itemArrived())     { itemCt++; }
    else if (itemDelivered())   { itemCt--; }
    else if (shutdown())        { state = stopping; }
    break;
  case stopping:
    if (itemCt==0)              { state = stopped; }
    else if (itemDelivered()) { itemCt--;  }
    break;
  .... (more cases)
}
```

Note that labels on transitions to state emergency have been omitted from the diagram to save clutter. There would be cases correpsonding to these too.

The 'philosophy' of state machines such as this is that

- the system in some particular state will always try to move via a transition so another state.
- If it cannot that sytem is *deadlocked*.

The transition labels indicate *events* that trigger change of state and/or an update of state variables.

- This might be an external event:
  - an input from a sensor or the network;
  - an interrupt
- ... or just some predicate on the state variable (eg `itemCt == 0`) becoming `true`.

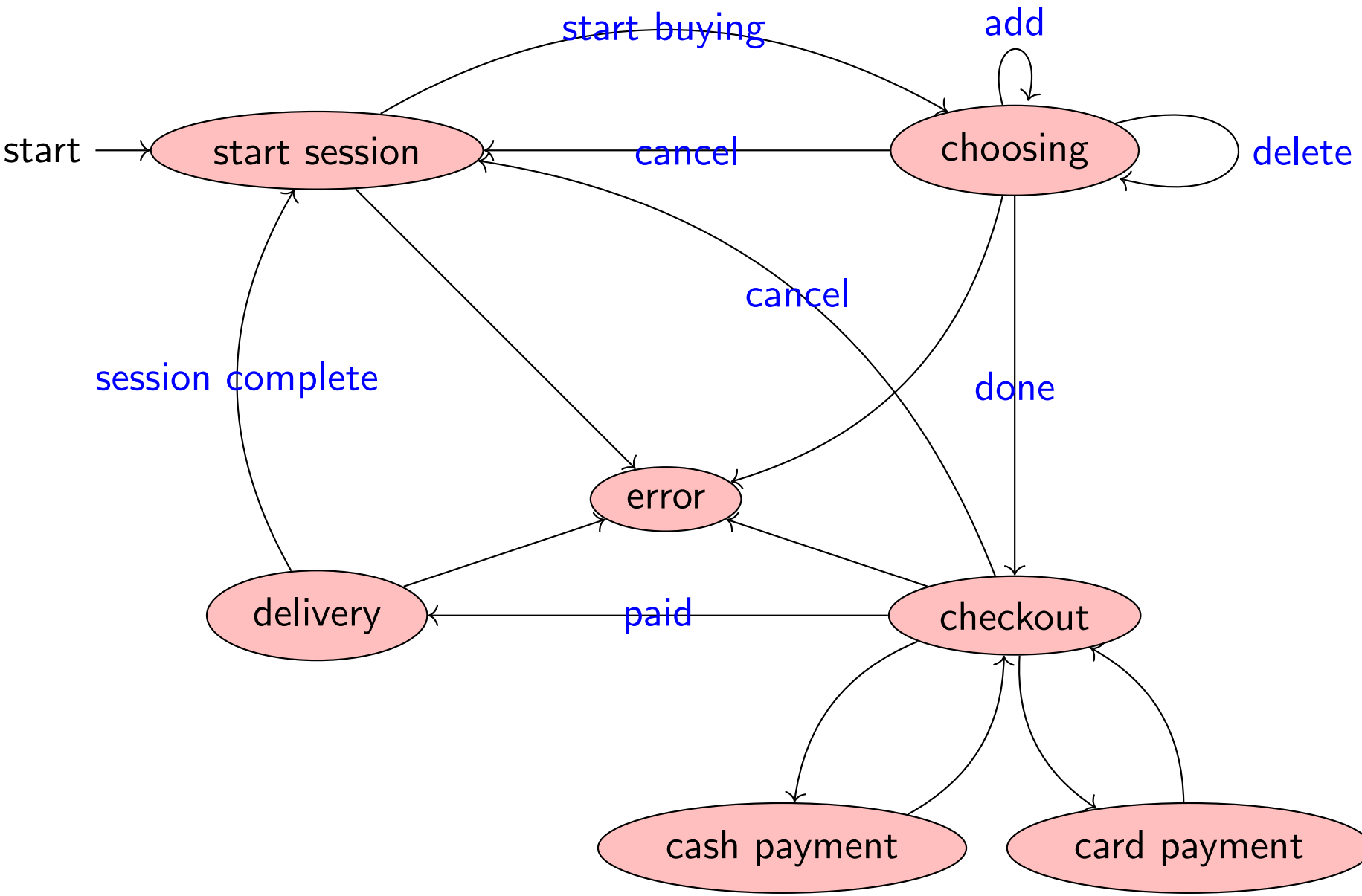
Vending machine example

Another informal example is a vending machine which

- allows the customer to choose quantities of each of a number of wares,
- will accept payment by cash or card, and
- dispense the goods at the conclusion of payment.

If you were developing the software for this machine you might find a state machine like the following useful.

Vending machine example



Vending machine example

Exercises for you -

- Describe the significance of each of the states.
- Describe the significance of each of the transition labels.
- Suggest labels for the transitions between ‘checkout’ and ‘cash payment’, ‘card payment’ and describe their significance.
- Write pseudocode for (some of) the diagram.
- Can you extend the state machine to include accepting coins/notes? giving change?

More formal state machines

This way of depicting 'computational logic' has been around for a while.

A related idea, *flow charts* [wiki] have been around since the 1950s or longer.

- The basic flow-chart is very simple: a start-point, end- or exit-points, rectangles denoting activities and diamond-shaped boxes denoting decisions.
- Program logic was depicted on flow charts in the early years but this became unwieldy very quickly.

A more sophisticated formal construct, the *finite state automaton* (or *finite state machine*, FSA or FSM if you like TLAs[1]) has been very useful in theoretical computing science, and continues to be.

[1] three-letter acronyms

Finite-state automata

A finite-state automaton is based on a finite set A of symbols or *letters*, called its *alphabet*. It is a construct (Q, R, q_0, F) where

- Q is a (finite, non-empty) set of *locations* or *states*;
- q_0 is a particular ($q_0 \in Q$) *initial state*;
- R is a set of *edges* between pairs of states, labelled with letters from A. A typical edge is something like $q_1 \xrightarrow{a} q_2$ where $a \in A$ and $q_1, q_2 \in Q$.
- $F \subseteq Q$. F is a particular subset of states, *final* states.

We have seen two (almost) examples already: the control system model and the vending machine model. Here, Q is the set of states, q_0 the intial state, A the set of events or actions that label the arrows, and R the labelled arrows themselves.

For more (similar) examples see the [wiki]

Finite-state automata

A *run* of an automaton is a sequence of locations/states starting with the initial state q_0 and with each state in the sequence connected to the previous state by an edge labelled by an element of A. Thus A, for 'alphabet' could also stand for 'actions'. An automaton is

- *deterministic* if from any state, there is at most one transition available to a 'next state, and
- *non-deterministic* otherise: from some states there is a choicde of 'next' actions.

A early use of finite-state automata in theoretical computing science was to define a *language* over an alphabet, A.

- Take a string of symbols from A, $a_1 a_2 a_3 ... a_n$ and present this as 'input' to an automaton (Q, R, q_0, F).
- Let the automaton run, 'consuming' a letter each time it takes a transition:
- ... so the run will look something like $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \xrightarrow{a_4} ... \xrightarrow{a_n} q_n$.

Finite-state automata

If a run is possible in which $q_n \in F$, that is, it ends up in a state in the special subset F, we say the automaton *accepts* the string.

An early theoretical result was if A is the 'ordinary' alphabet, then the strings of letters from A accepted by FSMs are exactly the *regular expressions*.

Interestingly, this is the same whether we restrict to deterministic automata, or allow non-deterministic ones too. We can also allow runs to include transitions with an 'empty' label, but we still get just the requlars expressions.

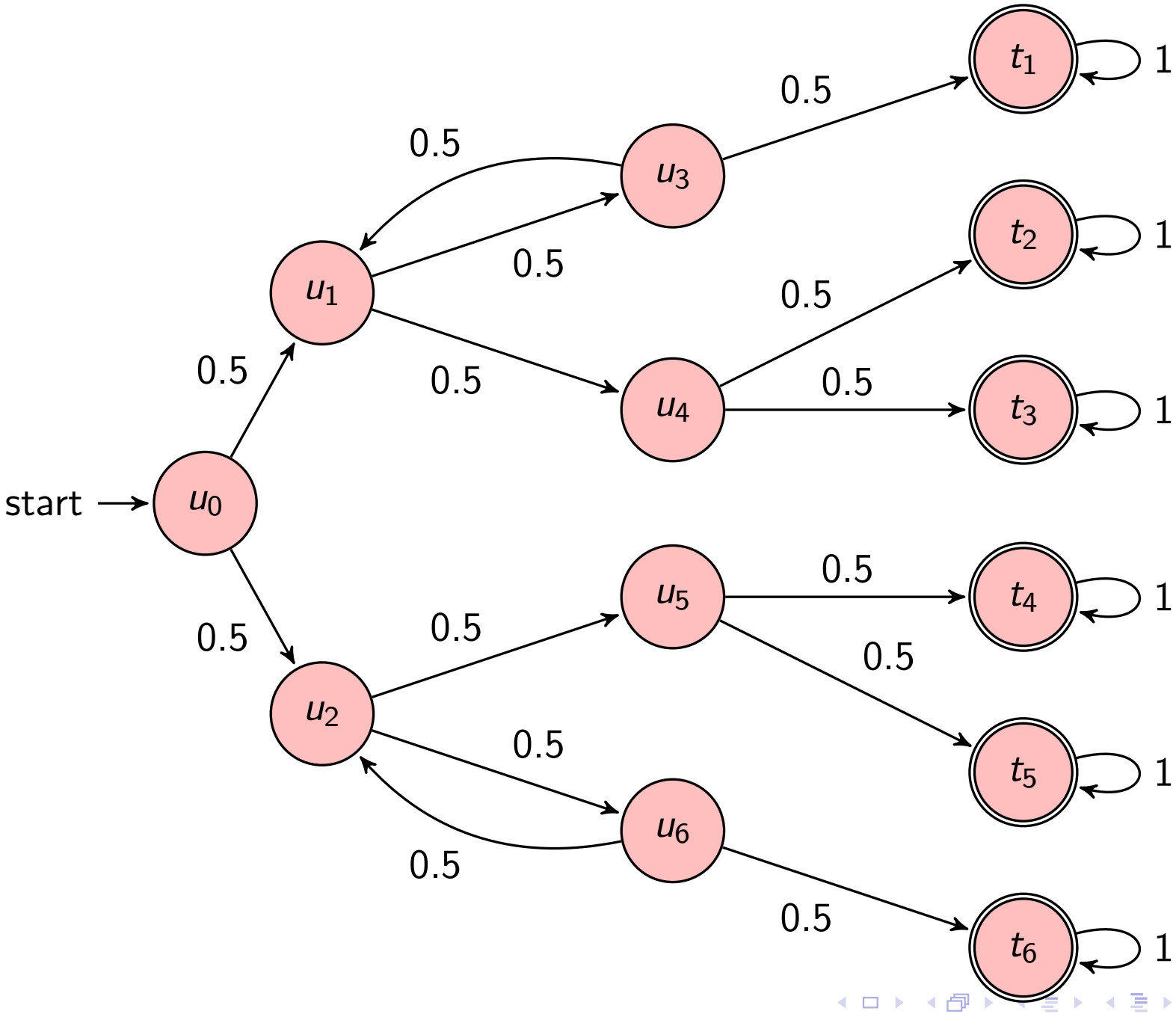
Timed automata

In one of our final-year modules, embedded system specification and design, we employ this further development of the finitate-state automaton, in which

- the automaton has a set of 'stop-clocks' which 'tick' at a regular rate during a run.
- A state/location can be specified to accessible only if the time on a clock is $\leq$ some value, and
- the transitions/edges are decorated with
  - *guards* which allow the transition only if some test on the clocks is satisfied, and also with
  - *actions* with reset clocks and perform other updates to system variables when the transtion is taken during a run.

This turns out to be a powerful technique for modelling and testing the behaviour of real-time

Probabilistic automata



Probabilistic automata

Another interesting devlopement of the basic finite automaton is a state machine in which the transitions between states/locations are decorated with *probabilities*.

- Each state has a set of forward transitions with probablities adding up to 1.
- In a run, the transition with probability p is selected, with probability p.

In the example above (due to Knuth and Yao), each state has two transitions outward and a run takes one transition or the other with a 50-50 chance - a toss of a coin.

- The probability of reaching state t_2 is $(\frac{1}{2})^3+(\frac{1}{2})^5+(\frac{1}{2})^7+(\frac{1}{2})^9+\ldots = (\frac{1}{2})^3(1+\frac{1}{4}+\frac{1}{16}+\frac{1}{64}+\ldots) = \frac{1}{8}\times\frac{4}{3} = \frac{1}{6}$
- In fact the same calculation applies to all the t_n states.

A roll of a 6-sided die is simulated with coin tosses!

State diagrams and the state design pattern

Not surprisingly, UML has a version of state machines: see, for instance, agilemodeling.com

- These *state charts* can be hierarchical, as in figure two on the Agile Modelling page: a state can be made up of a 'miniature' state machine.

Lastly, again another object-oriented view of state machines is given by the *state design pattern*: see, for example the State pattern wiki

- This design pattern really formalises the system for coding a state-driven system like the examples we began with.
- Formally, there is a choice of functionalities dependent on state. In C this used be done with an array of pointers to functions indexed by the state. In a modern OO language like java, the different (polymorphic) functionalities would be in classes implementing a common interface and the state would determine a particular implementation.