Variations on State Machines

Garlan & Wing Handout 6 17 September 1997

The state machine model presented in the previous handout is not suitable, appropriate, or natural for modeling all systems. In this handout we look at different variations of the basic model I've presented so far.[1]

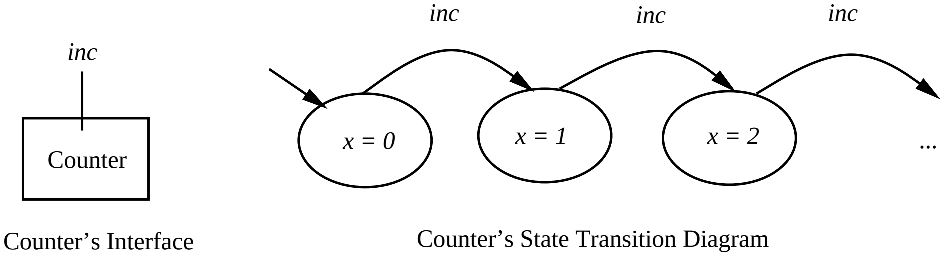
Which model you choose to use depends on what you are trying to model. You certainly want to choose one that allows you to state as precisely and concisely those things you care about. Some models may make distinctions that you don't care about; some may make assumptions that don't fit your problem. But sometimes which you choose is just a matter of taste. When you decide what model to use you should understand why you are choosing one over another.

Given a state machine, $M = (S, I, A, \delta)$, in the first three sections, I am going to refine some of M's components. First I'm going to give more structure to states in S (Section 1), then to actions in A (Section 2), and then generalize the functionality of δ (Section 3). In the fourth section I show how all these things can be used together.

Finally, in the last two sections I discuss other refinements of state machine models that are often seen in practice.

1 States Map Variables to Values

Let's revisit my integer counter example.



In the diagram above, I introduced the variable x to "hold" the value of the integer counter. The notion of a state as having variables that can have values of some type should be familiar to you from your programming experience.

With respect to state machine models, what I am doing is refining the notion of what a machine state is; I am adding some internal structure so that states are more than just named entities like *2*, *off*, or *crashed*. In general, each state in S of a state machine, M, is a finite function from a finite set, *Var*, of *variables* (sometimes called "identifiers" or "names") to a possibly infinite set, *Val*, of *Values*:

$$M = (\{S : Var \twoheadrightarrow\!\!\!\!\!\!+ \ Val\}, I, A, \delta)$$

[1] Unlike in the last handout, I'm not going to spell out each of the components of the state machine for each of the examples given in this handout. Also, I'm going to introduce some notation that I hope is minimal and simple enough for you to be able to use in your homework problems.

Moreover, I am going to assume variables and values are *typed* much like in a programming language. Hence if the variable x is of type *int*, then I can map it to the integer value "1" but not the real value "3.14159".

Counter's set of states, S, is a set of total functions mapping x to some integer value:

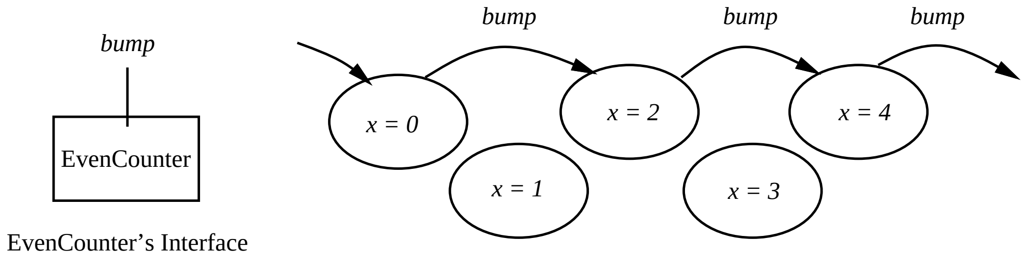
$$\{s : \{x\} \rightarrow int \mid s(x) \geq 0\}$$

Since a state is a mapping from variables to values, $s(x)$ denotes the value of the variable x in state s.

Suppose I want to write the state transition function for my Counter? Then, as defined earlier, let S (my set of states) be the set of functions that map the variable x to an integer value. Then similar to the SimpleCounter, I have

$$\delta_{inc} = \{(s, a, s') : (S \times \{inc\} \times S) \mid s'(x) = s(x) + 1\}$$

Now suppose I have a counter that allows state transitions only from states whose value for its state variable, x, is an even number. EvenCounter starts in the initial state $x = 0$ and whenever I *bump* its state, I get to the next even number:



Part of EvenCounter's State Transition Diagram

Let EvenCounter's set of states, S, be the same as for Counter. EvenCounter's transition function δ is:

$$\delta_{bump} = \{(s, a, s') : (S \times \{bump\} \times S) \mid even(s(x)) \wedge s'(x) = s(x) + 2\}$$

where we assume the predicate *even* has been defined appropriately. (Notice that some states in EvenCounter are unreachable. Which ones?)

Unfortunately writing the state transition function as predicates over sets of pairs of states and writing $s(x)$ and/or $s'(x)$ whenever I want to refer to the value of a state variable becomes pretty unwieldy quickly. By introducing two keywords, I will write the state transition function for each action in a more readable notation. Here's what I write for Counter's *inc* action:

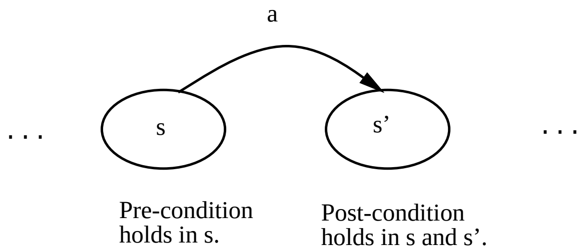
inc
  pre *true*
  post $x' = x + 1$

and for EvenCounter's *bump* action:

bump
  pre $even(x)$
  post $x' = x + 2$

The first line, which I'll call the *header*, gives the name of the action whose state transition behavior I describe in the subsequent two lines. The second line gives a *pre-condition*, which is just a predicate, and the third line gives a *post-condition*, which is just another predicate. The interpretation of the pre- and post-conditions is: In order for the state transition to occur from the state s to the state s' the pre-condition must hold in s; after the state transition occurs, then the post-condition must hold in s'. The state transition cannot occur if the pre-condition is not met. Post-conditions in general need to talk about the values of state variables in both the state before the state transition occurs (the "pre-state") and the state after it occurs (the "post-state"). I use an unprimed variable to denote the value of the variable in the pre-state and a primed variable to denote its value in the post-state. So, x' really stands for $s'(x)$; x, for $s(x)$.

Here's how to visualize what the pre- and post-conditions capture:



For action a to occur the pre-condition must hold in s. If a occurs, the post-condition must hold in s and s'.

In the Counter example, the *inc* action has the trivial pre-condition, "true." This means that the *inc* action can be performed in any state in S. EvenCounter's *bump* action has a non-trivial pre-condition. Another typical non-trivial pre-condition is requiring that a *pop* action not be performed on an empty stack. We'll see other examples of non-trivial pre-conditions later. *Inc*'s post-condition says that the value of the integer counter is increased by one from its previous value; *bump*'s post-condition says that the value is increased by two.

In general, for a given $M = (S, I, A, \delta)$, the template[2] I'm using to describe δ_{action} is

$$\begin{array}{l} \textit{action} \\ \quad \textbf{pre } \Phi(v) \\ \quad \textbf{post } \Psi(v, v') \end{array}$$

where *action* is in A, and Φ and Ψ are (state) predicates over a vector, v, of state variables. The above template stands for the following part of the definition of the state transition function, δ:

$$\delta_{action} = \{(s, a, s') : S \times \{action\} \times S \mid \Phi[s(v)/v] \wedge \Psi[s'(v)/v', s(v)/v]\}$$

In English this says that the precondition (Φ) has to hold in the pre-state *and* the post-condition (Ψ) has to hold in the pre/post-states[3].

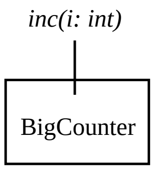
Other interpretations of pre/post-condition specifications are possible. I'm just giving you one reasonable one here.[4] For example, another common one is where the conjunction used above is replaced by an implication. Under this interpretation, which is used in Z and Larch, if the pre-condition does not hold and you try to do the action then anything can happen, i.e., "all bets are off." You could end up in an unexpected state, an error state, or an undefined state.

2 Actions

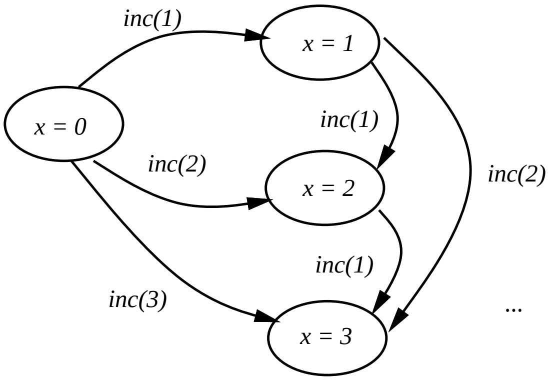
2.1 Actions with Arguments

Now suppose I want to let my integer counter's *inc* action take an integer argument. We see it's even more difficult to draw BigCounter's state transition diagram, only part of which is shown here:

[2] I will be elaborating on this template throughout this handout.
[3] Recall that the post-condition is defined over two states.
[4] This one also is consistent with the discussion (in the State Machine Basics handout) about actions that cannot occur.



BigCounter's Interface



Part of BigCounter's State Transition Diagram

It's much easier and more concise to write the state transition function for *inc* as follows:

inc(i: int)
  pre *true*
  post $x' = x + i$

I've extended the header in my specification to include a list of input arguments (and their types). I've intentionally chosen syntax to look like programming language notation.

The technical term for what I have done is *lambda abstraction*. Using a single template I've actually defined an infinite set of functions, one for each integer i. Instead of defining separate actions $inc_1, inc_2, \ldots$ I've defined a family of actions $inc(i)$.

According to the above specification, there is nothing preventing the input integer argument that I hand to *inc* from being negative. Suppose I want my counter to always increase in value, never decrease? I can capture this by strengthening the pre-condition:

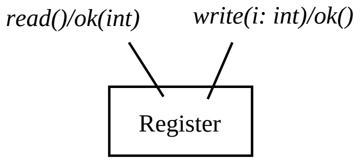
inc(i: int)
  pre $i > 0$
  post $x' = x + i$

I'll call this new machine FatCounter. It's an example of a state machine with an action that has a non-trivial pre-condition.

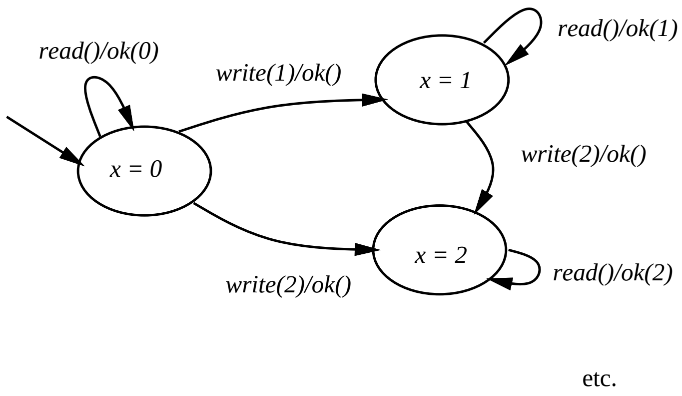
2.2 Actions with Results

Sometimes actions produce results of interest to the external observer. When I query my checking account balance, I expect the result to be displayed on the ATM screen or printed on a piece of paper.

Here's a Register with *read* and *write* actions. *Read* returns the value of the register; *write* takes an argument and modifies the register's state. Its initial value is the integer value 0.



Register's Interface



Part of the Register's State Transition Diagram

Here are the specifications of the actions:

read()/ok(int)
  pre *true*
  post $result = x$

write(i: int)/ok()
  pre *true*
  post $x' = i$

The first thing to notice is that I've introduced the word "ok" in the header. I've done this for two reasons. The first is that I want to set myself up so that I can have a convenient way to distinguish normal termination from exceptional termination of a procedure, a feature supported by most advanced programming languages. More on this later. The second is a trivial point: For symmetry, I prefer writing *read()/ok(1)* instead of *read()/1*. Think of the instance of an action as a procedure call. Then the state transition labeled *read()/ok(1)* corresponds to calling the *read* procedure and getting the integer 1 back.

What I am doing is adding more structure to actions. In general, a state transition is an *action instance*, which is a pair of an *invocation event* and *response event*. An *invocation event* is the name of the action plus the values of its input arguments; a *response event* is the name of the termination condition (e.g., ok) plus the value of its result.

An execution of a state machine is a sequence of alternating states and action instances. Some executions of the Register machine are:

$\langle x = 0, write(1)/ok(), x = 1, read()/ok(1), x = 1\rangle$
$\langle x = 0, write(1)/ok(), x = 1, read()/ok(1), x = 1, read()/ok(1), x = 1,$
$\quad write(5)/ok(), x = 5, read()/ok(5), x = 5\rangle$
$\langle x = 0, write(1)/ok(), x = 1, write(7)/ok(), x = 7, write(9001)/ok(), x = 9001\rangle$

For the above executions, I have the following (event-based) traces:

$\langle write(1)/ok(), read()/ok(1)\rangle$
$\langle write(1)/ok(), read()/ok(1), read()/ok(1), write(5)/ok(), read()/ok(5)\rangle$
$\langle write(1)/ok(), write(7)/ok(), write(9001)/ok()\rangle$

What would I have for a state-based definition of trace?

The second thing to notice in the above specification is that I've given in parentheses the type of the result, if any, of each action. The *read* action returns an *int* value; *write* doesn't return anything.

The third thing to notice is that in the post-condition I use a special reserved word *result* to stand for the return value. This trick works fine as long as an action produces only one result. It generalizes in the obvious way in case an action produces more than one result.

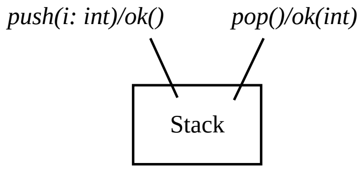
Technical Aside 1: There is a subtle difference between an action, which is a member of the finite set A, and an action instance, which is a member of the possibly infinite set of state transitions, as defined by δ. This difference is analogous in programming to the difference between the definition (declaration) of a procedure and a call (invocation) of it.

Technical Aside 2: There are state machine models that treat invocation events and response events as separate kinds of actions. Invocation events could be viewed as *input actions*; response events, as *output actions* (see the handout on State Machine Basics.) These models are mainly used for modeling classes of concurrent and distributed systems. For now, there is no compelling reason to treat them separately.

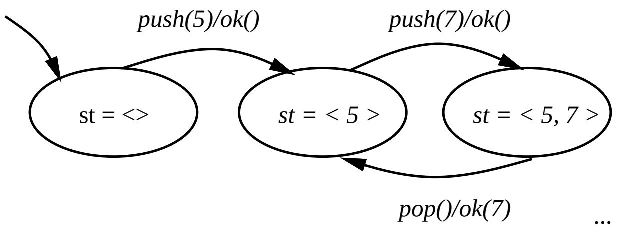
2.3 Actions that Terminate Exceptionally

Many advanced programming languages support exception handling and thus I should be able to specify the interface of a program that can raise exceptions.

Consider the following Stack machine:



Stack's Interface



Part of Stack's State Transition Diagram

with *push* and *pop* specified as follows:

$$\begin{array}{l} push(i\text{: } int)/ok() \\ \quad \textbf{pre } true \\ \quad \textbf{post } st' = st \frown \langle i \rangle \end{array}$$

$$\begin{array}{l} pop()/ok(int) \\ \quad \textbf{pre } st \neq \langle\rangle \\ \quad \textbf{post } st = st' \frown \langle result \rangle \end{array}$$

Here's how I specify a more robust interface to Stack that allows *pop* to raise the exception *empty* if I try to perform the *pop* action on an empty stack (*push* stays the same):

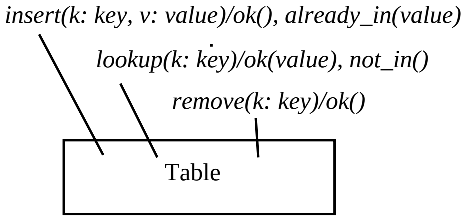
$$\begin{array}{l} pop()/ok(int),\ empty() \\ \quad \textbf{pre } true \\ \quad \textbf{post } st \neq \langle\rangle \Rightarrow (st = st' \frown \langle result \rangle \wedge terminates = ok) \wedge \\ \qquad\qquad st = \langle\rangle \Rightarrow (st = st' \wedge terminates = empty) \end{array}$$

The first thing to notice is the addition of the name of the exceptional termination condition, *empty*, in the header. For each termination condition (normal and exceptional), I allow some kind of result to be returned; here *empty* does not return any result.

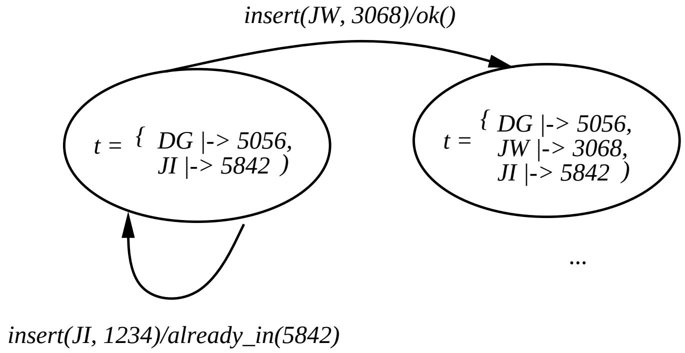
The second thing to notice is the special reserved word, *terminates*, which I introduced to hold the value of the termination condition ("ok" for normal termination and one of the exceptions listed in the header for an exceptional termination).

From a software engineering perspective, there's usually a close correlation between pre-conditions and exceptions. It's common to transform a "check" in the pre-condition to be a "check" in the post-condition. From your programming experience, this is the same as placing the responsibility on the callee rather than the caller of a procedure. With a pre-condition it's the caller's responsibility to check that the state of the system satisfies the pre-condition before calling the procedure; with an exception in lieu of the pre-condition, it's the callee's responsibility by performing a (run-time) check and raise an exception in case the state of the system violates the condition.

Here's an example where upon exceptional termination an interesting value is returned. Consider a Table machine, which stores keys and values. The state variable, t, stores the CMU telephone extensions of the 15-671 staff members:



Table's Interface



Part of Table's State Transition Diagram

We can model the state of the Table as a function from keys to values [5]. The *insert* action returns an exception *already_in* if there already exists a value associated with the key for which you are trying to insert a particular key-value pair, (k, v), and if so, it returns the current value bound to that key, k:

insert(k: key, v: value)/ok(), already_in(value)

pre *true*

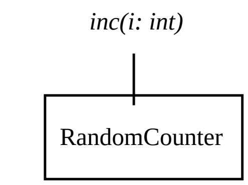
post $k \notin dom\ t \Rightarrow (t' = t \cup \{k \mapsto v\} \wedge terminates = ok) \wedge$

$k \in dom\ t \Rightarrow (t' = t \wedge terminates = already\_in \wedge result = t(k))$

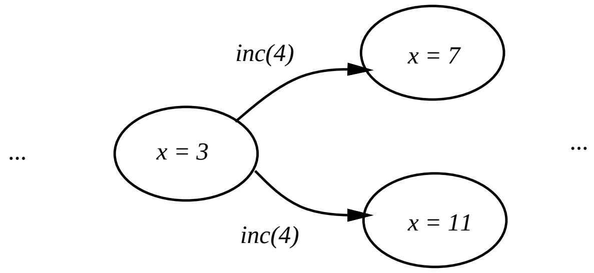
3 Nondeterminism

So far, δ has been a function, that is, for each state, s, and action, a, δ mapped us to at most one next state. Suppose I have a RandomCounter machine with an *inc* action that takes an integer argument:

[5] I leave it to you to formalize this state machine. You will see something similar when you get to the Birthday Book example in Z.



RandomCounter's Interface



Part of RandomCounter's State Transition Diagram

Here is the specification of *inc*:

$$\begin{array}{l} inc(i\text{: } int) \\ \qquad \textbf{pre } i > 0 \\ \qquad \textbf{post } x' = x + i \lor x' = x + 2i \end{array}$$

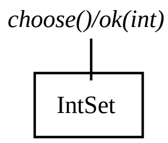
According to the specification, *inc* can either increment the counter's value by the value of its argument i or by twice that value. Thus, there are two possible states that you might end up in after doing the *inc* action given some integer argument. Since there is more than one, δ needs to map to a set of states[6]:

$$\delta(x = 3, inc(4)) = \{x = 7, x = 11)\}$$

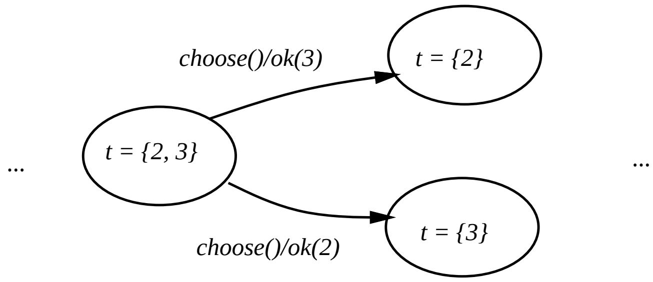
As an observer, you don't know which state transition will occur when you feed *inc* the integer 4. You must be prepared to deal with either possibility. The choice of which post-state is taken is made by the machine itself. Notice that the nondeterminism shows up in the specification of *inc* in the use of logical disjunction in the post-condition.

4 Putting Everything So Far Together

Suppose I have an integer set, t, and in its interface is a *choose* action that doesn't take any arguments and removes and returns an element from t.



IntSet's Interface



Part of IntSet's State Transition Diagram

where *choose* is specified as follows:

$$\begin{array}{l} choose()/ok(int) \\ \qquad \textbf{pre } t \neq \emptyset \\ \qquad \textbf{post } result \in t \land t' = t \setminus \{result\} \end{array}$$

[6] To make my point, I am viewing δ as a function from (state, action) pairs to a set of states, rather than as a relation on (state, action, state) triples.

The nondeterminism shows up in the specification for *choose* in the use of the set membership operator ($\in$) in the post-condition. I don't know which element of t will be returned; I know only that some element will be returned.

You might notice that the labels on the arcs in the state transition diagram above are different (by what is returned by *choose*); however, most people would still view the state transition function as nondeterministic because they would *abstract* from the actual value returned. In other words they would define δ something like this:

$$\delta(t = \{2, 3\}, choose()/ok(int)) = \{t = \{2\}, t = \{3\}\}$$

Finally, in a programming language that supports exception handling I would probably export a more robust interface for the *choose* action:

$$\begin{array}{l} \mathit{choose()/ok(int),\ empty()} \\ \quad \textbf{pre}\ \mathit{true} \\ \quad \textbf{post}\ (t \neq \emptyset) \Rightarrow (result \in t \wedge t' = t \setminus \{result\} \wedge terminates = ok) \wedge \\ \quad\quad (t = \emptyset) \Rightarrow (t' = t \wedge terminates = empty) \end{array}$$

In general, the template I use for each *action* in A of $M = (S, I, A, \delta)$ is:

$$\begin{array}{l} \mathit{action(inputs)/term_1(output_1), \ldots, term_n(output_n)} \\ \quad \textbf{pre}\ \Phi(v) \\ \quad \textbf{post}\ \Psi(v, v') \end{array}$$

where *inputs* is a list of arguments and their types, $term_i$ is the name of one of i termination conditions (including "ok") and $output_i$ is the type of the result corresponding to $term_i$. Φ and Ψ are state predicates as defined earlier. The reserved identifiers I use are:

- *ok*, used in the header, to denote normal termination,
- *result*, used in the post-condition, to denote the value returned by an action, and
- *terminates*, used in the post-condition, to denote the value of termination condition. Its value can be any of the $term_i$, including "ok," listed in the header.

I'm glossing over a number of technicalities here regarding state variables to store input arguments and the type of the result returned, depending on how an action terminates.

Finally, for simplicity, let's assume actions return only normally unless specified otherwise (by explicitly listing exceptional conditions in their headers). Under this default case, we can avoid clutter in our specifications by not always having to write "$terminates = ok$" in the post-conditions of our actions.

5 Finite State Automata

5.1 Deterministic FSA

If you ever took an automata theory course in Computer Science, you have come across state machines. A *deterministic finite state automata (FSA)* is usually defined as follows:

$$M = (S, I, F, A, \delta)$$

where S is a finite set of states; $I \subseteq S$, the set of initial states, is restricted to be a singleton set; $F \subseteq S$ is a finite set of *final* states; A is a finite set of actions, and $\delta : S \times A \nrightarrow S$ is a deterministic function.

FSA terminology	State Machine terminology
alphabet	actions
string	trace†
language L(M)	behavior Beh(M)

Here "trace†" means the trace of an execution ending in a final state. Sometimes the language of M is called *regular* or a *regular set* because it can be expressed as a regular expression using $*, \cup$, and concatenation. Just as with the behavior of a machine, the language of an FSA can be infinite, i.e., there might be an infinite number of strings accepted by M.

5.2 Nondeterministic FSA (NFSA)

The only interesting thing to say here is to refresh your knowledge about automata theory: It is always possible to turn an NFSA into a deterministic FSA; and of course, every deterministic FSA is an NFSA.

6 Finite Executions and Infinite Behavior

Recall the red and blue light example where the behavior of the light is infinite because there are an infinite number of traces associated with the light. Also, some of those traces are infinite (e.g., pressing red forever).

In some models of state machines, in particular, Communicating Sequential Processes (CSP), which we will be covering in detail later this term, infinite traces are not included in the machine's behavior. This kind of model is sometimes called the *finite-trace model.* The behavior of the machine is defined to be a possibly *infinite* set of *finite* traces.

Why is this a reasonable model? It has a simple structure which has some nice mathematical properties. Using this model may help simplify your thinking about the system; you know every trace in the behavior is finite so you don't have to worry about infinite traces. With a model that has both finite and infinite traces, whenever you prove some property about the behavior it describes, you usually have to structure your proof into two parts, one to handle the finite trace case and one to handle the infinite trace case. But perhaps the most compelling intuitive argument for this model is that you can never *see* an infinite execution; if you can't observe this behavior then why should we model it?

What are some of the disadvantages of this model? Because we cannot talk about infinite traces, the biggest disadvantage is that we cannot talk about certain properties like deadlock and fairness. To do so requires adding a lot of complicated structure to either traces or behaviors of a state machine.

We won't dwell on the advantages and disadvantages of this model here since we'll see more of it later.