CMU 15-671 Models of Software Systems Fall 1997

State Machine Basics

Garlan & Wing Handout 5 15 September 1997

In four lectures this term we will be studying general concepts associated with state machines. This handout covers some of the basic terms. We see in Section 6 how we can use set notation and predicate logic to describe infinite state machines in a succinct and precise manner.

1 Why State Machines?

A state machine is a simple mathematical model. It is a fundamental and ubiquitous model in Computer Science. A computer is nothing but a state machine. It has registers and memory (state) which contain values that change over time as its operations are executed (state transitions). A programming language is a way to describe a class of state machines. A program written in that programming language is a description of a state machine. The execution of that program corresponds to an execution of the state machine it describes.

A software system is a very, very complicated state machine. One thing that makes it complicated is its size: there is usually an infinite number of states, an infinite number of state transitions, an infinite number of executions, and each execution may possibly be infinite (not terminate). If everything were finite and not very large, we could probably reason about the behavior of the software system entirely in our heads. But since the real world is not so small and manageable, we need to find ways to model, describe, and reason about these large things in terms of a finite and small(er) number of small things.

One purpose of this course is to tell you about some of these ways of managing complexity. There are three themes we will visit and revisit: *notation*, *abstraction*, and *modularization*. With the appropriate notation, using abstraction and modularization (composition and decomposition) techniques, you will be able to model and reason about complex software systems. But remember, as with any mathematical model, you will be able to discuss only those things that that model models. If a state machine doesn't let you model the cost of the system, then you won't be able to reason about how expensive or cheap it will be to build.

2 Some Metacomments

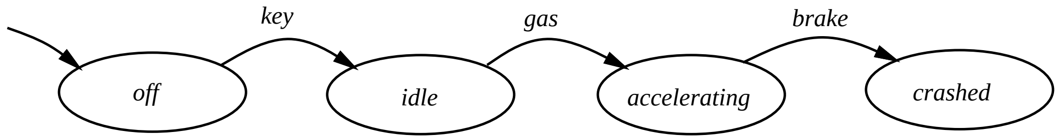
- In these four lectures and their associated handouts, I will emphasize concepts, not notation. I will use standard mathematical notation (as you have seen in the past few lectures) for defining terms. I will also introduce a little notation to make state machine descriptions easier to read. Later in the course we will be covering a few *specific* notations that are useful for describing *specific* classes of state machines. For example, Z is useful for describing sequential state machines; CSP, concurrent ones.

- At some times in lecture and in these handouts, I will be excessively pedantic and precise. At others I will be intentionally sloppy because the details are either unimportant or tedious. Of course the hard part is knowing when you may be sloppy and when you must be precise. You'll acquire this sensibility over time with practice.

- There is no one accepted model of state machines that is used as a standard for all of Computer Science or Software Engineering. Rather, each domain, discipline, area, etc., defines its own variation that is appropriate for the class of problems at hand. Thus, I am making a noble attempt in this handout to present a single model that is simple and abstract enough that will then allow me to present many of the common variations.

3 A Simple Example

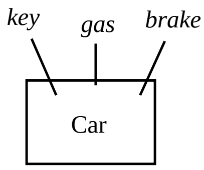
Let's start with a simple example of a car and model it as a state machine:



Car's State Transition Diagram

This Car has a very short lifetime. It starts out in the initial state where it is *off*. When you perform the action of turning the *key* to start the car, it moves into the *idle* state. After you apply some *gas*, it moves into the *accelerating* state. If you apply the *brake*, it dies, ending up in the *crashed* state.

You can think of my Car as a black box whose *interface* to the outside world is a set of observable states (ovals above) and a set of actions (arrows above). Sometimes I'll want to focus on just the actions and thus depict the Car's interface as follows:



Car's Interface

Imagine stuffing the Car's state transition diagram inside the box. In Section 7.1 I will discuss interfaces more.

Suppose in this Car example, I turn the key and I am unlucky: the car won't start. To model the possibility that the car goes from the *off* state to more than one possible next state (*off* and *idle*), I need to model *nondeterministic* behavior. Nondeterminism comes up naturally in the real world when you cannot predict what the next state of a machine will be given some event or action. Perhaps it's due to an internal choice made by the machine. For example, choosing an element from a set is a nondeterministic action; you know you're going to get some element of the set back, but you don't know which one.

4 State Machines: Definitions of Basic Concepts

Definition 1 *A* **state machine** M *is a quadruple,* (S, I, A, δ)*, where*

- S *is a finite or infinite set of* **states**,
- $I \subseteq S$ *is finite set of* **initial states**,
- A *is a finite set of* **actions**, *and*
- $\delta \subseteq S \times A \times S$ *is a* **state transition relation**.

If S is finite, then M is a *finite state machine.*

I is sometimes defined so that it can be an infinite subset of S (when of course S is infinite). A is sometimes called the *alphabet* of M. Elements in A are sometimes called *events* or *operations*. In other models A may be infinite. Sometimes δ is defined to be a function, $S \times A \twoheadrightarrow S$, rather than a relation.

However, by our having δ be a relation, we can more easily model nondeterminism. Recall that it's equivalent to view the type of δ as $S \times A \longleftrightarrow S$; thus, given a state and an action, we can move to possibly more than one next state.

(Aside: the action component, a, of a triple, (s, a, s'), in δ, is sometimes just viewed as the *label* for the state transition from s to s'. Thus, sometimes these kinds of state machines are called *labeled state transition systems.*)

Applying the above definition of a state machine, we have for the Car:

Car = (
$\{off, idle, accelerating, crashed\}$,
$\{off\}$,
$\{key, gas, brake\}$,
$\{(off, key, idle), (idle, gas, accelerating), (accelerating, brake, crashed)\}$
).

4.1 Concepts

Let M be a state machine (S, I, A, δ).

Definition 2 *Each triple,* (s, a, s'), *in* δ *of* M *is a* **step** *of* M.

Definition 3 *An* **execution fragment** *is a finite or infinite sequence* $\langle s_0, a_1, s_1, a_2, \ldots, \rangle$ *of alternating states and actions such that for all* i (s_i, a_{i+1}, s_{i+1}) *is a step of M.*

Definition 4 *An* **execution** *is an execution fragment starting with an initial state of M (and ending in a state if finite).*

Definition 5 *A state is* **reachable** *if it is a final state of a finite execution.*

There are two reasonable ways to define what the *behavior* of a state machine is. One way ("event-based" or "action-based") says what is observable are a machine's actions; the other ("state-based") says what is observable are a machine's states. Which way you prefer is philosophical. Here are two alternative definitions of what a trace is:

Definition 6 *(Event-based) A* **trace** *is the sequence of actions of an execution.*

Definition 7 *(State-based) A* **trace** *is the sequence of states of an execution or is the sequence,* $\langle s_i \rangle$, *for each* $s_i \in I$.

Finally, we can define what the behavior of a machine is.

Definition 8 *The* **behavior** *of a machine M (Beh(M)) is the set of all traces of M.*

Behaviors are *prefix-closed*, which means, for a given behavior, B: (1) The empty trace is in $Beh(M)$; and (2) if a trace is in B then any prefix of that trace is in B.

In other work on state machine models, behaviors do not have or are not assumed to have the prefix-closure property.

4.2 Revisiting My Car

Consider the Car machine.

- $\langle idle, gas, accelerating \rangle$ is a step of Car; $\langle idle, brake, crashed \rangle$ is not.
- $\langle idle, gas, accelerating, brake, crashed \rangle$ is an execution fragment of Car.
- $\langle off, key, idle \rangle$ and $\langle off, key, idle, gas, accelerating, brake, crashed \rangle$ are executions, but $\langle accelerating, brake, crashed \rangle$ and $\langle accelerating, gas, idle \rangle$ are not.
- All states in Car are reachable.
- In an event-based viewpoint $\langle key, gas \rangle$ and $\langle key, gas, brake \rangle$ are traces of Car but $\langle gas, brake \rangle$ and $\langle gas, key \rangle$ are not.
- In a state-based viewpoint $\langle off \rangle$ and $\langle off, idle, accelerating, crashed \rangle$ are traces of Car but $\langle idle, accelerating \rangle$ and $\langle accelerating, idle \rangle$ are not.

Finally, using the event-based definition of trace, we have

$$Beh(Car) = \{\langle\rangle, \langle key \rangle, \langle key, gas \rangle, \langle key, gas, brake \rangle\}.$$

Using the state-based definition, we have

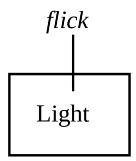
$$Beh(Car) = \{\langle\rangle, \langle off \rangle, \langle off, idle \rangle, \langle off, idle, accelerating \rangle, \langle off, idle, accelerating, crashed \rangle\}.$$

In both cases the empty sequence is a member of the behavior.

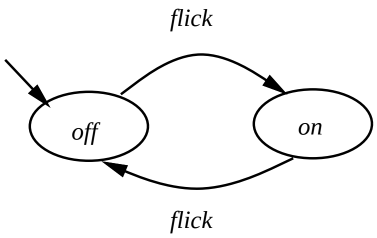
5 Infinite Executions and Infinite Behavior

The Car has a finite behavior of finite executions. In general, a state machine can have infinite executions and infinite behavior. An infinite execution is an infinite sequence of alternating states and actions. An infinite behavior is an infinite set of executions. Note that elements in an infinite behavior might all be finite.

Consider a simple light switch whose interface and state transition diagram are shown below:



Light's Interface



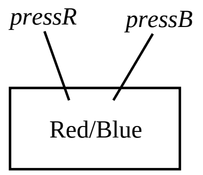
Light's State Transition Diagram

Light $= (\{off, on\}, \{off\}, \{flick\}, \{(off, flick, on), (on, flick, off)\})$. Some executions of Light are:

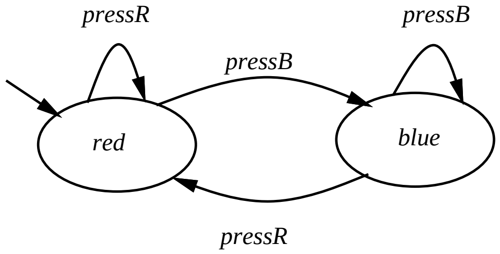
$\langle off, flick, on \rangle$
$\langle off, flick, on, flick, off \rangle$
$\langle off, flick, on, flick, off, flick, on \rangle$
$\ldots$
$\langle off, flick, on, flick, off, flick, on, flick, off, \ldots, \rangle$

There are an infinite number of finite executions and the last execution listed above is infinite. Thus *Beh(Light)* is an infinite set of finite and infinite traces.

Here is an example of a state machine with more than one infinite trace:



Red/Blue Light's Interface



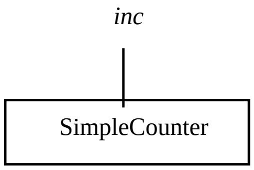
Red/Blue Light's State Transition Diagram

One of the infinite traces (event-based) is the infinite sequence of pressR actions. What are some others?

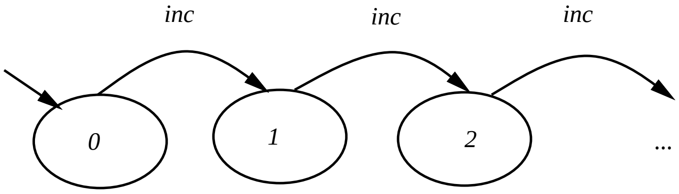
6 Infinite States and Infinite State Transitions

In all the examples so far, you've seen only a finite number of states, and hence, a finite number of state transitions, i.e., S and δ were finite. I have been able to draw state transition diagrams for these state machines in their entirety. For software systems in general we must deal with an infinite number of states and hence an infinite number of state transitions. Their state transition diagrams are impossible to draw out completely (at least in the way I've been drawing them).

The simplest example is an integer counter (which is like a ticking clock), initialized at 0.



SimpleCounter's Interface



SimpleCounter's State Transition Diagram

As soon as need to model a system that deals with any domain with an infinite set of values, e.g., integers, then I admit the possibility of an infinite number of states. Can you see now why most programs, let alone software systems, are infinite state machines?

My SimpleCounter example has just one action, *inc*. It has an infinite number of state transitions because there is an infinite number of states over which the state transition relation, δ, is defined—because there is an infinite number of values that the SimpleCounter can take.

Now, suppose I want to write a description of the state machine using the notation you've seen so far. I would write it something like this:

SimpleCounter = (
$\{0, 1, 2, \ldots\}$,
$\{0\}$,
$\{inc\}$,
$\{(0, inc, 1), (1, inc, 2), (2, inc, 3), \ldots\}$
).

The problem is what about those two occurrences of ...? Here, the pattern is clear and we can probably rely on sharing the same intuition as to what goes in those .... In general, we need a way to describe more complex sets. We would like to find a way to characterize an infinite set of things in terms of a finite string of symbols.

Fortunately, predicate logic provides just the notation we need. To characterize the set of all non-negative integers, I can write:

$$\{x : int \mid x \geq 0\}$$

I can even characterize the set of initial states using a predicate:

$$\{x : int \mid x = 0\}$$

I've taken care of the first occurrence of ..., but what about the second? I can define the state transition relation, δ, as a set of triples, (s, a, s'), for which the pair of states, s, s', satisfies a given predicate. I can do this for each action in A and then take the union of these sets. For example, I can define the set of triples, (s, a, s'), that *inc* contributes to δ as follows:

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

where $S = \{x : int \mid x \geq 0\}$, as defined above. Since my SimpleCounter has only one action, $\delta = \delta_{inc}$. In general,

$$\delta = \bigcup_{\mathrm{a} \in \mathrm{A}} \delta_a$$

Notice the power of set notation and predicate logic. Using a few symbols I am able to write out the state transition relation which is defined on an infinite set of states. The critical thing is that because I have a *finite* set of actions, I can define a finite number of δ_a, one for each action; in so doing, I am describing the state transition relation, δ, which is a possibly infinite set of state transitions (because I may have to define it over an infinite number of states).[1]

To be pedantic, I can put this all together and rewrite my description of SimpleCounter as follows:

SimpleCounter = $\langle$
$\{x : int \mid x \geq 0\}$,
$\{x : int \mid x = 0\}$,
$\{inc\}$,
$\{(s, a, s') : \{x : int \mid x \geq 0\} \times \{inc\} \times \{x : int \mid x \geq 0\} \mid s' = s + 1\}$
$\rangle$.

Technical Aside: The observant reader will notice that I'm using the state *name* for two purposes: to *name* the state and to give the *value* of the SimpleCounter in that state. In the next handout we will refine the structure of states that will allow me to make a distinction between names and values.

7 Notes

7.1 Environment and Interfaces

A system doesn't live in isolation. It interacts with its *environment*. When we model a system as a state machine we are modeling the *interface* the system has with its environment. Later in the course when we discuss concurrency we will model the environment itself as a state machine and then discuss the interactions between a system and its environment as the behavior of the composition of two state machines. For now (and for the first part of this course) we will focus our attention on modeling the system and understanding

[1] I apologize for belaboring these points.

a system's behavior through its state machine model. For now, you can pretend that you are the system's environment.

Intuitively the behavior of a state machine captures what the environment *observes* of the system modeled by the machine. Many state machine models differ on what "observes" mean. (That's one reason why I introduced an event-based view and a state-based view.) When you identify what a system's sets of states and actions are you are defining what its *observable behavior* is. So, when you design a system, especially if it is supposed to be put together with some other system, it is critical that you identify its interface. A rule of thumb to use when you're trying to nail down a system's interface and you're not sure whether something belongs in the interface or not, is to ask yourself, "Can I observe it?" If so, then "it" has to be modeled somehow; "it" is part of the system's interface.

7.1.1 Input and Output Actions

There are two kinds of interactions between an environment and a system: either the environment might do something to the system to cause a state change (or obtain information about its current state) or the system might do something (or produce something of interest) to the environment and cause a change in the environment. In the light switch example, you can *flick* the light. That is the only action you can do to the light. In the red and blue light example, there are two things you can do: *pressR* and *pressB*. On the other hand, consider a digital clock that displays the time in hours and minutes. Every time a minute passes, the clock's display changes; you can observe each of those state transitions. (Aside: The clock is an example of why one might prefer a state-based view of a trace; we could argue that what we really observe is the clock's state (the display), not the state transitions.)

One way to model this dichotomy of actions is to separate the set of actions into a set of *input actions* and a set of *output actions*. Intuitively, input actions correspond to those things the environment does to the system (like pressing buttons and flicking switches); output actions correspond to those things the system does to the environment (like an ATM handing out cash or a vending machine dispensing candy).

7.1.2 Abstraction

In the Light example I chose not to make a distinction between flicking the light switch up or flicking it down. In both cases I simply named the action of flicking the switch "flick." I made a choice to *abstract* from the direction of flicking the switch. (This abstraction gives the implementor of my Light model the freedom to implement the light switch with a button that pops in and out rather than with a lever that moves up and down.) When you are modeling a system you will often be faced with such design decisions. If you are faced with the problem of whether something should be modeled at a particular level of abstraction, the question you should be asking yourself is "Is this level of detail relevant to this level of abstraction?" or more precisely "Is this distinction observable by the environment?" If the answer is "no," i.e., the observer has no way of telling two things apart or you as the system designer don't want to provide the observer a way of telling two things apart, then you should abstract from the difference between the two things. For example, suppose I have an apple, an orange, and an eggplant. I might decide that I do not want an observer to tell the difference between the apple and the orange, but only the difference between fruits (apple or orange) and vegetables (eggplant).

As another example, think of my Car. By the way I chose to model it, you (as the environment) don't get to see all its states or all its state transitions. For example, to go from the *idle* state to the *accelerating* state I may actually have shifted gears, say from first to second and second to third and so on, before getting to the *accelerating* state. It was my choice to *abstract* from some of its states (e.g., being in third gear) and state transitions (shifting from second to third). In making my choice of what to reveal to the observer, I hid those states and state transitions from the observer because they were irrelevant. The only information you have about the Car is what I reveal to you. These are design decisions as a modeler that I made.

Some state machine models allow you to make a distinction between *external actions* and *internal actions*. External actions are part of the system's interface and are observable by the system's environment. Internal actions are hidden and not observable.

7.1.3 Actions Revisited

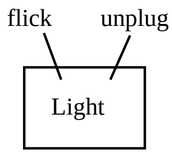
By combining the two points made in the previous two subsections we see that we can divide a set of actions into external and internal actions. We can further divide a set of external actions into input and output actions. Different models of state machines may or may not make these distinctions. For example, the I/O automata model developed by Nancy Lynch and her students at MIT makes both distinctions.

7.2 A Subtle Point: Actions That Can't Happen

There are two ways actions cannot happen. Either an action is simply not part of the system's interface; or it is, but no state transition occurs if you try to perform it. For example, when you get a pull-down menu on your Mac, only those actions listed are part of the interface. It's simply impossible for you to try to do an action that is not listed. But also, any action that is "grayed out" indicates that nothing will happen if you select that action: no state transition occurs.

Consider the Light example. Since "unplug" is not in the set of actions, there is no way I can even think of doing such an action. The point is that the only actions that may possibly happen are those that are explicitly given in the state machine's set of actions, A.

However, suppose "unplug" were added to Light's set of actions and *I keep everything else the same*. In particular, its state transitions are still defined only for the action "flick." What happens if I try to unplug the light?



An Unpluggable Light?

There are four reasonable interpretations:

1. Nothing happens.
2. It's undefined. That is, using functional notation, $\delta(\mathit{off}, \mathit{unplug}) = \perp$ and $\delta(\mathit{on}, \mathit{unplug}) = \perp$. ($\perp$, read as "bottom," is the mathematical symbol for "undefined.")
3. It's an error. Chaos can occur (core dumped, machine crashes).
4. It can't happen.

We will take the fourth interpretation because we can model the first three explicitly if that's the behavior we want. In the first case, we would define the state transition function such that for each state, $s \in \{\mathit{off}, \mathit{on}\}$, (s, unplug, s). In the state transition diagram, for each state, we would draw an arrow from it to itself and label the arrow with the action "unplug." In the second case, we would introduce a special state called $\perp$. In our state transition diagram we would simply draw an arrow from the *off* state to $\perp$ and an arrow from the *on* state to $\perp$. Both arrows would be labeled with the action "unplug." In the third case, we would introduce a state called "error" and draw arrows similar to those in the second case.

Notice that "bottom" means mathematically undefined, which is very different from "error." "Bottom" is like trying to divide by zero in Mathematics or where you end up if you find yourself in an infinite loop in Computer Science. "Error" is simply a way to denote an error has occurred. In Computer Science we make a distinction between being in an infinite loop (non-termination) and being in a (terminating) "bad" state (like core dumped) so it makes sense that we make this distinction in Software Engineering too.

You might ask why we would include an action like "unplug" if we can't do anything with it. One answer is that we're setting ourselves up so that we can put state machines together. Suppose we combine Unpluggable Light with a machine for which there is a state transition defined on the action "unplug"; then we may be able to make sense of doing the unplug action on the combined machine. You'll hear more on this later, when we get to concurrency.