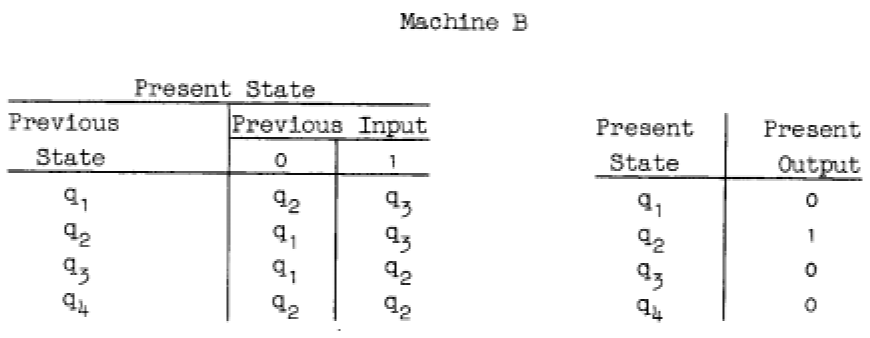
Important Programming Concepts (Even on Embedded Systems) Part V: State Machines

Jason Sachs ● January 5, 2015

Other articles in this series:

- Part I: Idempotence
- Part II: Immutability
- Part III: Volatility
- Part IV: Singletons
- Part VI: Abstraction

Oh, hell, this article just had to be about state machines, didn't it? State machines! Those damned little circles and arrows and *q*'s.



Machine B

Present State		
Previous State	Previous Input 0	Previous Input 1
q_1	q_2	q_3
q_2	q_1	q_3
q_3	q_1	q_2
q_4	q_2	q_2

Present State	Present Output
q_1	0
q_2	1
q_3	0
q_4	0

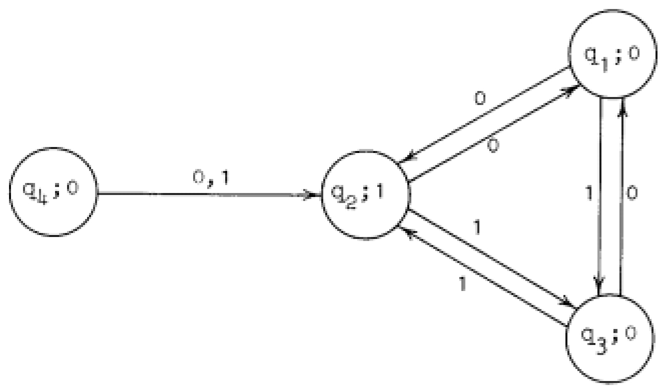


FIGURE 4. Transition Diagram of Machine B

Yeah, I know you don't like them. They bring back bad memories from University, those Mealy and Moore machines with their state transition tables, the ones you had to write up as part of that stupid traffic light project. You'd rather be learning Haskell or Rust or Ruby or Go or Swift, or the newest version of jQuery or D3 or Raphaël or Node.js or something cool and NEW!

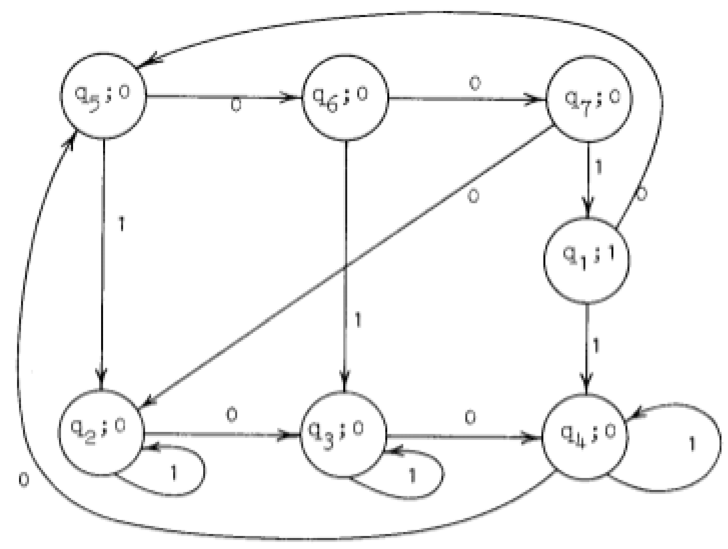
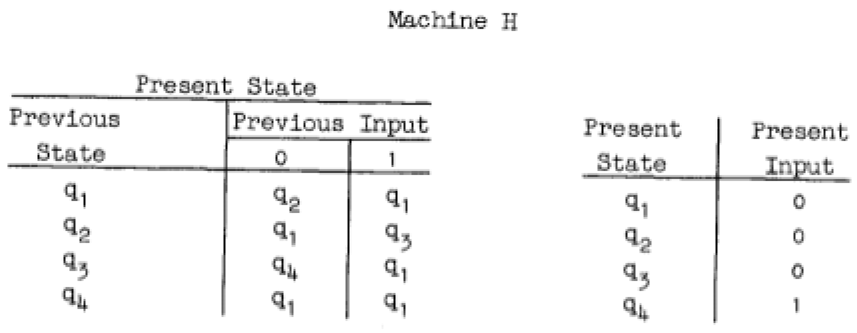


FIGURE 8. Transition Diagram of Machine F_3

State machines are like your great-aunt’s pantyhose, old and tired and tattered, the ones she keeps hanging onto for sentimental reasons, from the 1960’s. Not sexy.



Machine H

	Present State	
Previous State	Previous Input	
	0	1
q_1	q_2	q_1
q_2	q_1	q_3
q_3	q_4	q_1
q_4	q_1	q_1

Present State	Present Input
q_1	0
q_2	0
q_3	0
q_4	1

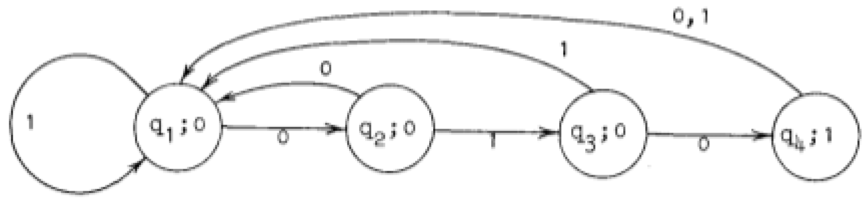


FIGURE 9. Transition Diagram of Machine H

I mean, that’s just not what you had in mind when you learned programming; you wanted to do great things with algorithms, and move vast quantities of information with only your mind!

So then why are we talking about state machines?

The Tale of a Young Programmer: Fantasy and Enlightenment

Picture this:

Your name is Ryan, you're a nondescript white American male, you just graduated from college, and already you have a JOB! A programming job! You never would have thought it would turn out this way! For your freshman year, you thought you'd major in economics, but once you actually started getting into it, things turned ugly and you wanted to get away from it as fast as possible, so you declared a major in history, because that was kind of fun looking up information about old stuff, but you had trouble seeing how you were going to swing it in the real world with a history degree. And then you accidentally stumbled upon your roommate's coursework; he was a CS major learning Javascript, and somehow this programming stuff caught your interest, and it was exciting! Farewell history, hello CS! Woohoo! Your roommate got a flashy job with a startup doing something with web scale databases. You, on the other hand, were worried about being called out as a history-major-in-CS's clothing. But somehow you got this job. A JOB! With an actual paycheck!

It's with a company that designs electronics for high-end furniture. You're working on this motorized couch, with drink holders and a built-in beer cooler and reclining seats that tilt up and down. The couches are $5000+ but they're controlled by a little chip that costs less than 50 cents, so you're learning all about embedded programming. Your manager is a guy named Vic, and he must be in his late 40's, with salt-and-pepper hair. He reminds you somewhat of your uncle Victor, who was about the same age when he died a few years ago from a heart attack. So you think of your manager as Uncle Vic.

For your first project, you learned how to debounce a switch. Uncle Vic showed you how to do it in hardware with a capacitor and a Schmitt trigger inverter. Then another coworker taught you how to debounce a switch in software, using a timer interrupt.

And now it's time to do some real work! AWESOME! The lab has 6 prototypes, each of which has all sorts of wires coming out everywhere, and you'll get to work on one in a few days.

Uncle Vic tells you you're going to write a program that controls the tilt motor. There's an up button and a down button. The only tricky part is that when the person switches from down to up, or up to down, you have to wait 100 milliseconds so the motor doesn't burn out. He says you can do it any way you want, for now, though he warns you that you might have to rewrite it later. "But you should probably use a state machine," he says. "It'll save you a lot of frustration."

State machines… those are those things you had to do in CS for that project, the one with the fake traffic light and crosswalk button and the timer. That didn't seem like programming, it felt icky… and old… and strange....

You've been working on the problem for a few hours, and so far your code looks like this:

```
uint16_t up_delay = 0;
uint16_t down_delay = 0;
while (true)
{
   // debounce already taken care of
   // in the button code

   // Up button pressed: first wait for delay to finish
   while (Button1_isPressed() && up_delay > 0)
   {
      wait_1msec();
      --up_delay;
   }
   // Now we can turn the motor on so it tilts the seat up
   while (Button1_isPressed())
   {
      TiltMotor_Up();
      down_delay = 100;
   }
   TiltMotor_Off();

   // Down button: first wait for delay to finish
   while (Button2_isPressed() && down_delay > 0)
   {
      wait_1msec();
      --down_delay;
   }
   // Now we can turn the motor on so it tilts the seat down
   while (Button2_isPressed()
   {
      TiltMotor_Down();
      up_delay = 100;
   }
   TiltMotor_Off();
}
```

Fairly simple. It's a sequence: you turn the tilt up when button #1 is pressed, then turn the tilt down when button #2 is pressed, waiting a bit in between, going round and round and round.

"How's the state machine going?" asks Uncle Vic when he walks by. You show him your code in progress, and he looks kind of blank-faced, mumbling that it should probably work, saying something about pathological cases. Something doesn't sit right with him.

You try it out on a breadboard, with LED's connected to the microcontroller outputs, instead of leading to real motors. You've made a few minor tweaks, but are basically following the same code you showed Uncle Vic earlier. It seems to work fine. You can't wait to try it in the lab on REAL MOTORIZED COUCHES!!!!

Uncle Vic comes by your cubicle later. You give him a demo. He's grinning. At first you feel proud, then you realize something is a bit off.

"OK, great! Now that you have that working, I want you to change your program a little bit. Let's see you control *two* motors independently. Hmm? Heh, heh."

Huh, two motors, let's see, you could expand the `while()` loop with a longer sequence… no… hmmm… ummm....

Then it dawns on you that maybe your approach isn't such a good idea.

State Machines to the Rescue! (Why Uncle Vic Is Right After All)

I first learned programming back in the early 1980's — BASIC, I'm afraid to admit, on the Timex-Sinclair 1000 and VIC-20 and Apple][and Commodore 64. And though the computers are now much better, and the languages are different (no more line numbers or `GOTO`, thank goodness), the way they teach computer programming hasn't really changed much. It's the same essential imperative method: a computer program is just something that asks the computer to do this, then that, then something else. Sequences of actions. Sometimes there are branches, and loops, but it's really just one thing happening after another.

And when things take time, we don't mind making the computer wait. It waits for us to enter our name, or open a file, or download a web page:

```
println("enter your name:");
name = input_string();
println("enter your birthday:");
bday = parse_date(input_string());
println("now I'll download your horoscope:");
webpage = download_url("http://horoscopesforthehomely.com/horoscope?name="
                + name + "&bday=" + format_date(bday));
println("downloaded! displaying...");
show_in_browser(webpage);
```

The problem with waiting is that, in our sequential view of the world, we've gone from doing one thing at a time to nothing, at least while we're waiting. So if two things need to be handled at once, like going to add more money to the parking meter while you are waiting in a long line at the post office, we can't.

If we want to cling to a sequential, imperative method of programming, but do more than one thing at a time, we have to mess around with concurrent threads, or processes, or figure out this fancy new async/await stuff.

But none of that's going to help you if you're programming on an environment without an operating system, like a 50-cent microcontroller. And anyway, state machines offer us a much simpler alternative.

State Machines: A Quick Refresher

In a nutshell, a finite state machine (we don't generally mess with the infinite kind) is an event-driven system. At any given point in time, we are in one of a predetermined set of states, and we update this to a new state based upon rules that are purely a function of the previous state and the input events.

The formal description of a finite state machine is a small medley of things: (S, s_0, I, O, T, G)

- S: a finite set of states. The machine is in one of these states at a time.
- s_0: the designated starting state.
- I: a finite set of possible inputs to the state machine.
- O: a finite set of possible outputs from the state machine.
- T: a transition function — this allows the state machine to compute the next state $s_{k+1} = T(s_k, i)$ where i represents the inputs to the state machine at step k.
- G: an output function — depending on which approach you like, the output is determined by the state

alone $o = G(s_k)$ for the Moore machine, or the state and inputs $o = G(s_k, i)$ for the Mealy machine.

If you want to use a state machine to perform a sequence of actions, and need to wait between actions, each action in the sequence with a delay has to be converted to a separate state. State machines are reactive; a long-running task like `sleep()` or `download_webpage()` needs to be converted to an action to initiate the task, which is an output, and the completion of that task is an input to the state machine.

Our silly horoscope example would become something like this:

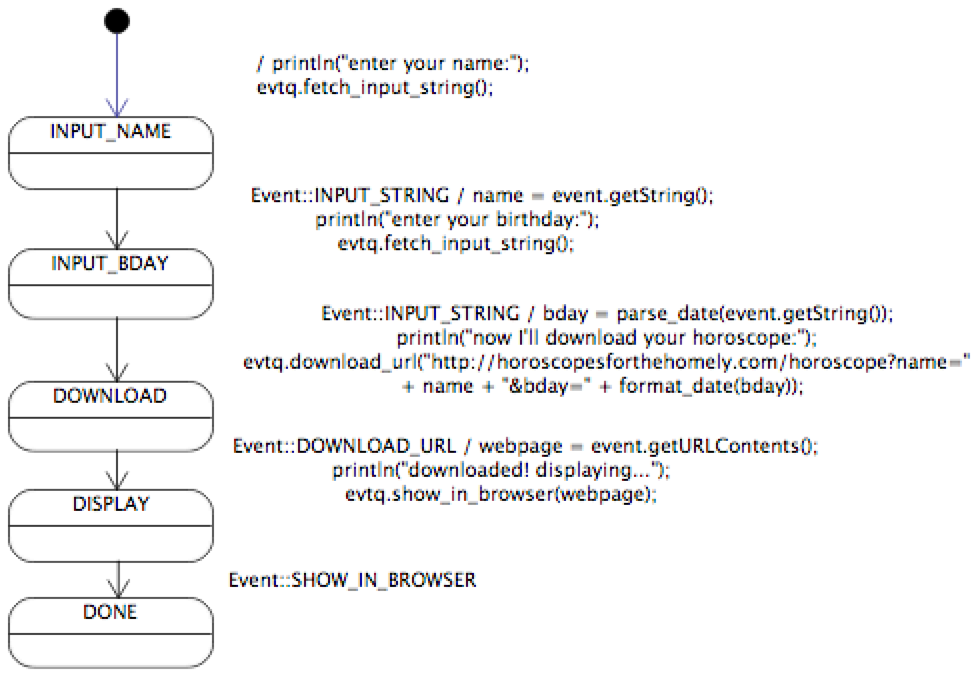
```
enum State {
  STATE_START,
  STATE_INPUT_NAME,
  STATE_INPUT_BDAY,
  STATE_DOWNLOAD,
  STATE_DISPLAY,
  STATE_DONE
} state = STATE_START;

EventQueue evtq = ...;

while (true)
{
  enum State next_state = state;
  switch (state)
  {
    case STATE_START:
      println("enter your name:");
      next_state = STATE_INPUT_NAME;
      evtq.fetch_input_string();
      break;
    case STATE_INPUT_NAME:
      Event event = evtq.check(Event::INPUT_STRING);
      if (event != null)
      {
        name = event.getString();
        println("enter your birthday:");
        evtq.fetch_input_string();
        next_state = STATE_INPUT_BDAY;
      }
      break;
    case STATE_INPUT_BDAY:
      Event event = evtq.check(Event::INPUT_STRING);
      if (event != null)
      {
        bday = parse_date(event.getString());
        println("now I'll download your horoscope:");
        evtq.download_url("http://horoscopesforthehomely.com/horoscope?name="
                + name + "&bday=" + format_date(bday));
        next_state = STATE_DOWNLOAD;
      }
      break;
    case STATE_DOWNLOAD:
      Event event = evtq.check(Event::DOWNLOAD_URL);
      if (event != null)
      {
        webpage = event.getURLContents();
        println("downloaded! displaying...");
        evtq.show_in_browser(webpage);
        next_state = STATE_DISPLAY;
      }
      break;
    case STATE_DISPLAY:
      Event event = evtq.check(Event::SHOW_IN_BROWSER);
      if (event != null)
      {
        next_state = STATE_DONE;
      }
      break;
  }
  state = next_state;
}
```

There! Isn't that easy? We turned 9 brief, straightforward lines of code into 62 lines using an event queue. And this is why people don't like state machines. Actually, the main issue isn't so much designing the state machine, but rather implementing it. Even in 2015, we don't really have a good, widely-accepted set of tools for working with state machines in languages like C or Java. (Later in this article, I'll talk about the ones I'm aware of.)

If we just wanted to *design* the state machine, not implement one, it would be a little bit simpler; a state diagram would look something like this:


/ println("enter your name:");
evtq.fetch_input_string();
INPUT_NAME
Event::INPUT_STRING / name = event.getString();
println("enter your birthday:");
evtq.fetch_input_string();
INPUT_BDAY
Event::INPUT_STRING / bday = parse_date(event.getString());
println("now I'll download your horoscope:");
evtq.download_url("http://horoscopesforthehomely.com/horoscope?name="
+ name + "&bday=" + format_date(bday));
DOWNLOAD
Event::DOWNLOAD_URL / webpage = event.getURLContents();
println("downloaded! displaying...");
evtq.show_in_browser(webpage);
DISPLAY
Event::SHOW_IN_BROWSER
DONE

The black dot at the top is the UML symbol for the initial transition, which occurs each time the state machine starts executing. The remaining states have transitions demarcated by `cond / action` where `cond` is the condition causing the transition, and `action` is the action that occurs on the transition. Because there are actions (or outputs) that occur on transitions, this is a Mealy machine; if the outputs were only on the states, it would be a Moore machine. State diagrams have been around since the 1950's, and haven't changed much, although there have been some additions in the 1980's that we'll talk about later.

Depending on the level of formality, there are a few options you have in drawing state diagrams:

- **whether to show default transitions that lead from a state S back to itself** (i.e. no state change) — if you were designing a logic circuit from discrete gates to implement a state machine, you'd want to make sure you have an unambiguous definition for each possible combination of inputs. In most modern systems, however, the default behavior of maintaining state when no indicated transaction matches, is an implied behavior, and it's easy to implement in software, and probably is easy to specify in hardware description languages like Verilog or VHDL. (Digital design is one of my weaknesses, so if you're reading this and you know how these languages work with state machine design, send me a message — I'd love to hear more details.)

- **whether to report actions/outputs in detail** — in the systems I work on, state diagrams are usually used more as a high-level system design tool rather than a complete description, so I would probably not put all that information in each of the transition actions, replacing it instead with a short note in English or pseudocode (e.g. "fetch web page"), with the details left up to the programmer.

This kind of looks like a flowchart, the other clunky unsexy design tool from the 1950's that you were probably forced to learn in school, and decided to shun forever. Flowcharts are really closely related to state diagrams, the only real difference being that in a flowchart, the assumption is that there is a sequential, imperative program executing, and the position in the flowchart is usually assumed to correspond to the program counter in most computer architectures. The program counter is an implicit representation of the behavioral state the program is in — whereas with a state machine, you maintain and update the state explicitly, and there doesn't need to be a program counter (state machines can run on simple digital logic circuits).

When behavioral flow is very strongly linear, like in this example, there's not too much benefit in using a state diagram. The expressive brevity of most computer languages is so good and natural-sounding — at least to most programmers — that we really like to cling to the idea of writing a program as an imperative sequence of tasks. State machines seem like an odd and clumsy approach to use, and we don't really have software tools to express the equivalent state machine as succinctly as a short imperative program.

The real power of state machines is subtle, and FSMs really shine in a few situations when you don't have a pure sequence of events, but rather some odd behavior with branches and loops and things that don't translate easily to a simple linear sequence of blocking function calls:

- FSM state diagrams can greatly reduce the chances of human error:
    - They are a way of describing and documenting desired behavior. In most cases I can review a FSM diagram much more easily than code or a text description in English.
    - They force you to understand your system behavior more clearly.
    - Using a state machine implementation, rather than hand-coded spaghetti logic, can ensure the structure is implemented correctly.

- FSMs are a formal mechanism for describing computing behaviors, and are widely used in the theoretical design of parsers and regular-expression engines. But if you felt queasy about state machines when you started learning about them, try sinking your teeth into a computer science text on parsing; you'll see homework problems with useless little theoretical regular expressions, containing lots of *a*'s and *b*'s, converted into useless little state machines — aka DFA = deterministic finite automaton — and your taste for them will probably be turned off for good. That isn't a good way of appreciating the state machine unless you like theoretical gunk. (As a side note: there are some good papers, a few of them by Russ Cox and another by Eli Bendersky, that get into the relationship between FSMs and regexps. The widely-used parser generator tools Lex and Yacc apparently use state machines, although in the case of Yacc they are augmented by a stack, forming something called a push-down automaton.)

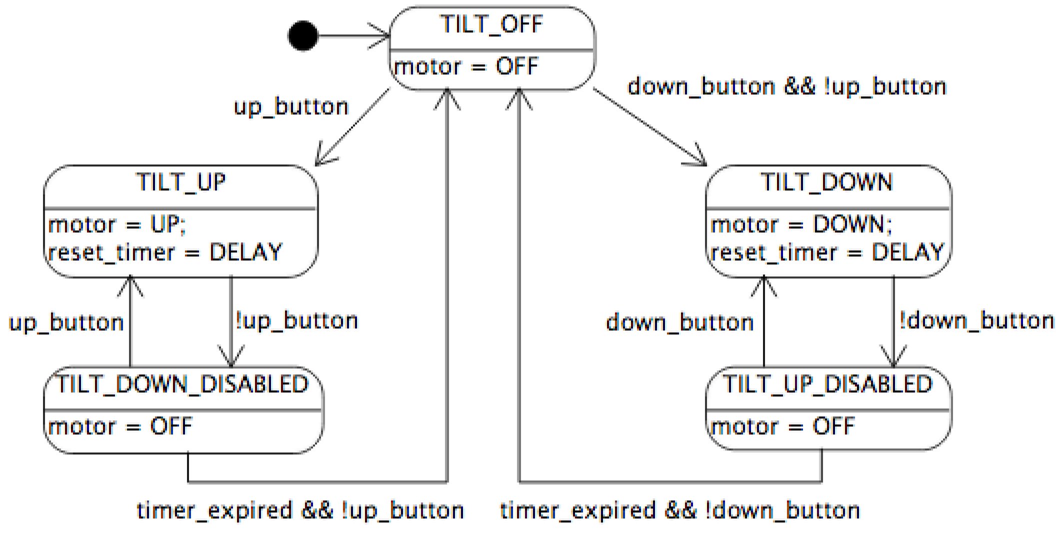
Of these points, the first is far more important for most of us. Let me just repeat one of the reasons to use a state diagram:

—

They force you to understand your system behavior more clearly.

—

The silly horoscope example doesn’t really illustrate this well, so let’s go back to the motorized couch example and draw a state diagram:

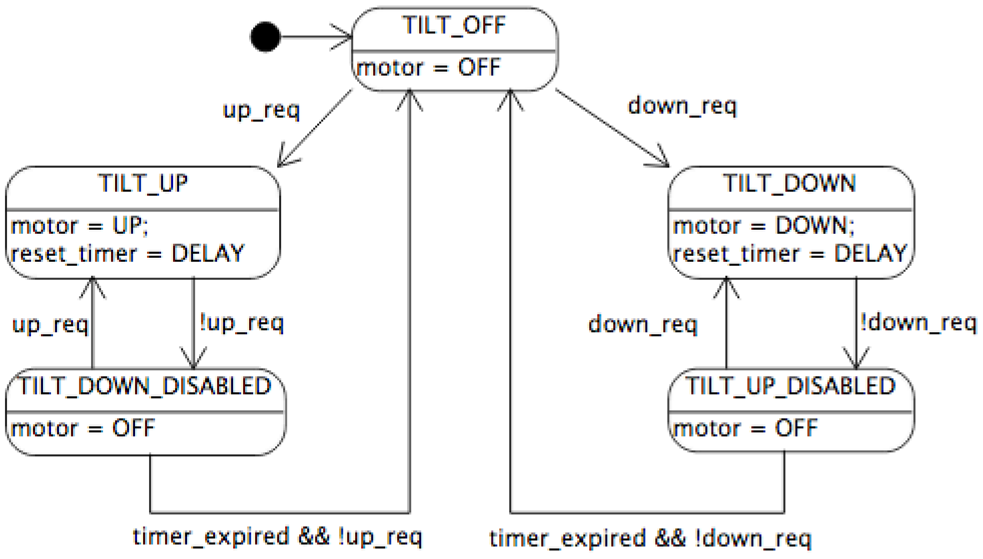


Note a few things here:

- The two states `TILT_UP` and `TILT_DOWN` map very cleanly to the physical system behavior (one state for tilting the motor up, another for tilting the motor down)
- Splitting the concept of “motor off” into three separate “off” states might not be something you’d think about until you sit down and write out a diagram, but think about it:
    - in `TILT_DOWN_DISABLED` we don’t allow the motor to tilt down for a while, because the motor has recently been turning in the up direction
    - in `TILT_UP_DISABLED` we don’t allow the motor to tilt up for a while, because the motor has recently been turning in the down direction
    - in `TILT_OFF` there are no restrictions, since there has been enough time elapsed since the motor was last energized.
- In `TILT_OFF` we are forced to handle the case where both buttons are pressed; here we let the up button take priority, and let the person using the couch enter the `TILT_UP` case regardless of whether the down button is pressed, whereas `TILT_DOWN` can only be entered from `TILT_OFF` if the up button is not pressed.
- Some aspects of our system design become simpler in this design:
    - we only need one timer (not two)
    - in the `TILT_xxx_DISABLED` states, we only need to consider the button press for the direction the motor was recently traveling, since the other button is disabled.

Technically, our button contention logic is a little inconsistent here: if you press both buttons in `TILT_OFF`, up takes priority, but if you press both buttons in `TILT_DOWN` or `TILT_UP_DISABLED`, that up-taking-priority-over-down behavior disappears. This may or may not be a more intuitive behavior to customers using the couch, but the important point is that specifying the behavior in a diagram forces you to consider it.

One technique I use a lot in state diagrams is to make the conditions more abstract, and define them more completely outside the diagram. In this case, I might choose to represent two mutually-exclusive conditions, `up_req` and `down_req` (up request and down request) in the diagram:



Then I can define `up_req` and `down_req` and `timer_expired` separately in terms of the two input signals, giving priority to the up button in all cases:

```
up_req         = up_button
down_req       = !up_button && down_button
```

This diagram's behavior is slightly different: priority is consistent in all cases, the diagram is cleaner, and the details of prioritization are given in text accompanying the diagram.

Hierarchy and Statecharts

One of the problems with the classical Mealy or Moore state machines occurs when you get into real systems, and there is structure in the state machine. This happens in almost every real-world case, except for the few trivial behaviors that can be described in something simple like the case of our motorized couch.

Here's some examples:

Maybe your software has component A, which acts as a state machine, and another component B, which acts as a state machine, and the two are intertwined; sometimes in one state of component A, we start over with component B. A classical system where A has 6 states and B has 5 states would potentially yield 30 separate states, and a diagram starts to be unwieldy.

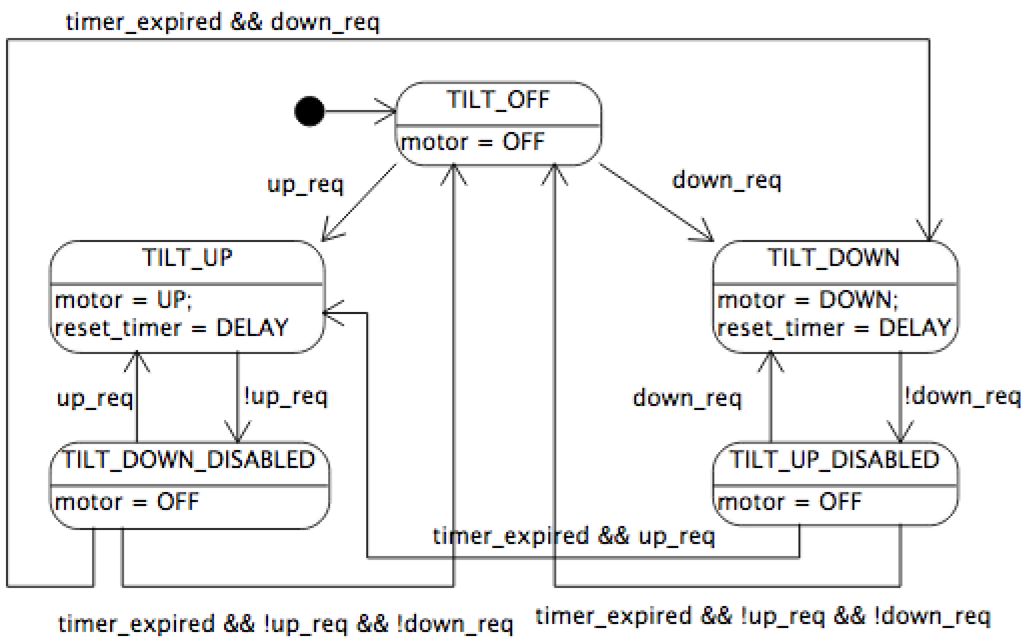
Or maybe your state machine has history: we go from any of states Q1, Q2, Q3, and Q4 into state Q5 when a button is pressed, but when the button is pressed again, we go back to the state we came. The classical approach of state machines is to be memoryless: everything is captured by the state itself, and to get the desired effect, we'd have to split state Q5 up into four separate states that duplicate the same behavior but transition into different states.

These kinds of behaviors are not hard to understand when described informally, but when they are translated to a formal finite state machine, the resulting state diagram is complex and loses its visual simplicity and clarity. The mathematics professor David Harel published a paper in 1987, *Statecharts: A Visual Formalism for Complex Systems*, in which he described a number of ways of capturing the complexity in a more simple visual manner through modifications of the traditional state diagram:

- addition of hierarchy
- parallelism (showing two state machines working simultaneously)
- history

Harel’s paper is well-written and worth a read: he introduces these techniques in the context of his digital wristwatch. Remember those things, with their handful of input buttons repurposed for many different purposes? (press the MODE button briefly and it changes from a clock to a stopwatch to a timer; hold down the MODE button and it lets you set the time) Described as a state machine, even with Harel’s statecharts, their behavior is surprisingly complex.

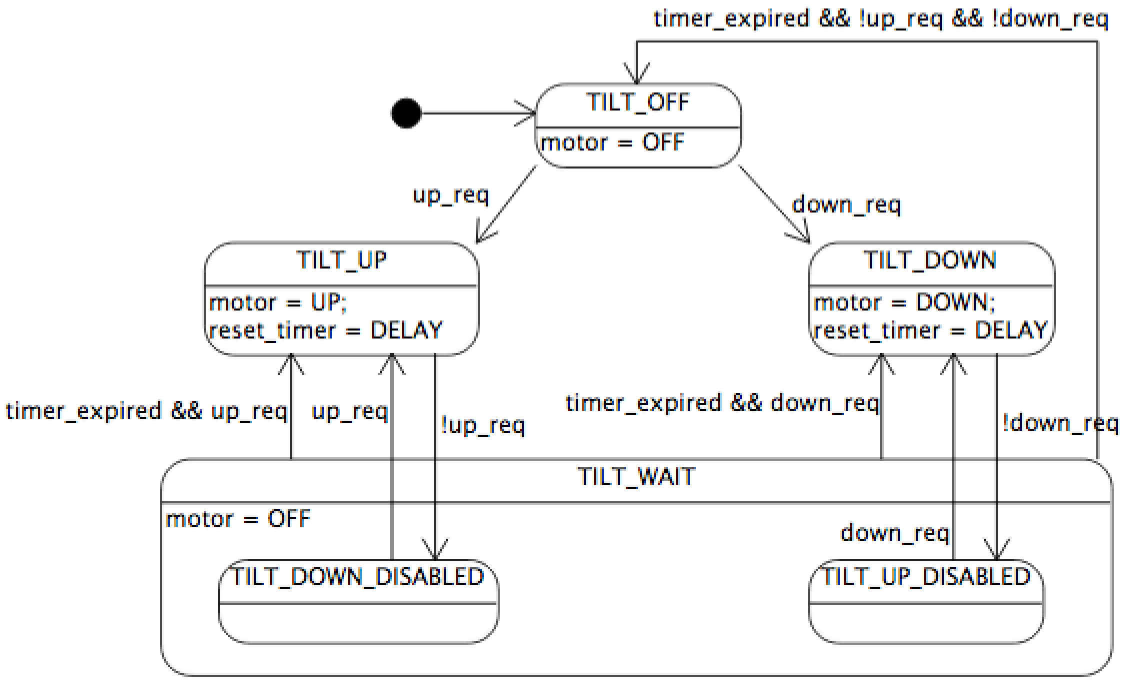
Let’s make another minor change to our state diagram of the motorized couch:



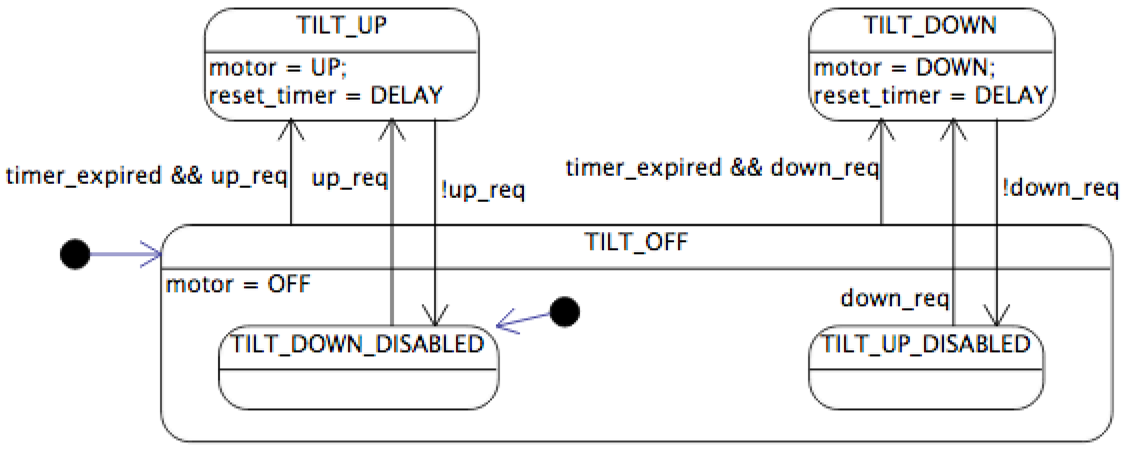
Here, when the timer has expired, we allow the motorized couch to go directly from the `TILT_UP_DISABLED` state to the `TILT_UP` state, and directly from the `TILT_DOWN_DISABLED` state to the `TILT_DOWN` state, skipping the `TILT_OFF` state. This is a fairly trivial change from the end user’s perspective, but these kinds of little changes may have significance for the system constraints. If you are executing state machine updates at a moderate rate, say at 1 millisecond intervals, an extra state introduces a 1 millisecond delay, and although a person may never notice, that delay may mean the difference between a constraint met or violated.

I frequently work on motor controllers, and details like these are often important. For example, a motor controller with a bootstrap gate drive on each half-bridge requires a short period of time with the lower transistor on, so that bootstrap capacitors can be charged up, before the upper transistor is allowed to turn on. In that case, a delay is intentional and important. In other cases, like fault-handling of an overcurrent condition, additional delays are detrimental.

In any case, the state diagram shown above is starting to get a little messy when we add in the important details. We can, however, redraw it as a statechart:



This makes things a little simpler; we can refactor out common output and transition behavior from the `TILT_UP_DISABLED` and `TILT_DOWN_DISABLED` states, by making them substates of a common `TILT_WAIT` state in which the motor is off. In fact, drawing it this way makes it easier to see that, as long as we start with the timer expired, we can merge the `TILT_WAIT` and `TILT_OFF` states:



The other black dot in the `TILT_OFF` state dictates which substate we start in, whenever we enter `TILT_OFF` without explicitly going to a particular substate. (After all, we have to be unambiguous.)

And now we have a diagram that unambiguously captures the behavior we want, and is fairly simple to understand. We can use this as an specification for system behavior when we are implementing it, so that our software engineers can work from the statechart, rather than having to read a few paragraphs in English and try to interpret them correctly.

These characteristics — unambiguous specification, ease of understanding, and strong correlation between system simplicity/complexity and visual simplicity/complexity — are what really makes drawing statecharts worthwhile. They're not just homework assignments. Most real systems won't have more than a dozen states at any level of hierarchy in the statechart, and if they do, you should raise a concern and ask why; I've mentioned the evils of complexity in several articles, and will get on my soapbox again to say that you should try to keep your system as simple as possible, as you will save time and resources in the long run.

Determining Desirable System Behavior: An Exercise in

Managing Complexity

It’s worth dwelling on this point for a little bit. Here is Harel’s description of why he got involved in the topic of state machines. In December 1982, he was a professor at the Weizmann Institute of Science in Israel, and had begun consulting work for Israel Aircraft Industries on an avionics project called Lavi:

> Here is how the problem showed up in the Lavi. The avionics team had many amazingly talented experts. There were radar experts, flight control experts, electronic warfare experts, hardware experts, communication experts, software experts, etc. When the radar people were asked to talk about radar, they would provide the exact algorithm the radar used in order to measure the distance to the target. The flight control people would talk about the synchronization between the controls in the cockpit and the flaps on the wings. The communications people would talk about the formatting of information traveling through the MuxBus communication line that runs lengthwise along the aircraft. And on and on. Each group had their own idiosyncratic way of thinking about the system, their own way of talking, their own diagrams, and their own emphases.
>
> Then I would ask them what seemed like very simple specific questions, such as: “What happens when this button on the stick is pressed?” In way of responding, they would take out a two-volume document, written in structured natural language, each volume containing something like 900 or 1000 pages. In answer to the question above, they would open volume B on page 389, at clause 19.11.6.10, where it says that if you press this button, such and such a thing occurs. At which point (having learned a few of the system’s buzzwords during this day-a-week consulting period) I would say: “Yes, but is that true even if there is an infra-red missile locked on a ground target?” To which they would respond: “Oh no, in volume A, on page 895, clause 6.12.3.7, it says that in such a case this other thing happens.” This to-and-fro Q&A session often continued for a while, and by question number 5 or 6 they were often not sure of the answer and would call the customer for a response (in this case some part of the Israeli Air Force team working with the IAI on the aircraft’s desired specification). By the time we got to question number 8 or 9 even those people often did not have an answer! And, by the way, one of Jonah Lavi’s motivations for getting an outside consultant was the bothersome fact that some of the IAI’s subcontractors refused to work from these enormous documents, claiming that they couldn’t understand them and were in any case not certain that they were consistent or complete.
>
> In my naïve eyes, this looked like a bizarre situation, because it was obvious that someone, eventually, would make a decision about what happens when you press a certain button under a certain set of circumstances. However, that person might very well turn out to be a lowlevel programmer whose task it was to write some code for some procedure, and who inadvertently was making decisions that influenced crucial behavior on a much higher level. Coming, as I did, from a clean-slate background in terms of avionics (which is a polite way of saying that I knew nothing about the subject matter…), this was shocking. It seemed extraordinary that this talented and professional team did have answers to questions such as “What algorithm is used by the radar to measure the distance to a target?”, but in many cases did not have the answers to questions that seemed more basic, such as “What happens when you press this button on the stick under all possible circumstances?”.
>
> In retrospect, the two only real advantages I had over the avionics people were these: (i) having had no prior expertise or knowledge about this kind of system, which enabled me to approach it with a completely blank state of mind and think of it any which way; and (ii) having come from a slightly more mathematically rigorous background, making it somewhat more difficult for them to convince me that a two-volume, 2000 page document, written in structured natural language, was a complete, comprehensive and consistent specification of the system’s behavior.

And later on:

> They would repeatedly say things like, “When the aircraft is in air-ground mode and you press this button, it goes into air-air mode, but only if there is no radar locked on a ground target at the time”. Of course, for anyone coming from computer science this is very familiar: what we really have here is a finite-state automaton, with its state transition table or state transition diagram. Still, it was pretty easy to see that just having one big state machine describing what is going on would be fruitless, and not only because of the number of states, which, of course, grows exponentially in the size of the system. Even more important seemed to be the pragmatic point of view, whereby a formalism in which you simply list all possible states and specify all the transitions between them is unstructured and non-intuitive; it has no means for modularity, hiding of information, clustering, and separation of concerns, and was not going to work for the kind of complex behavior in the avionics system. And if you tried to draw it visually you’d get spaghetti of the worst kind. It became obvious pretty quickly that it could be beneficial to come up with some kind of structured and hierarchical extension of the conventional state machine formalism.
>
> …
>
> For some mysterious reason, the basic graphics of statecharts seemed from the very start to vibe well with the avionics engineers at the IAI. They were very fast in grasping the hierarchy and the orthogonality, the high- and low-level transitions and default entries, and so forth.
>
> Interestingly, the same seemed to apply to people from outside our small group. I recall an anecdote from somewhere in late 1983, in which in the midst of one of the sessions at the IAI the blackboard contained a rather complicated statechart that specified the intricate behavior of some portion of the Lavi avionics system. I don’t quite remember now, but it was considerably more complicated than the statecharts in Figs. 7–9. There was a knock on the door and in came one of the air force pilots from the headquarters of the project. He was a member of the “customer” requirements team, so he knew all about the intended aircraft (and eventually he would probably be able to fly one pretty easily too…), was smart and intelligent, but he had never seen a state machine or a state diagram before, not to mention a statechart. He stared for a moment at this picture on the blackboard, with its complicated mess of blobs, blobs inside other blobs, colored arrows splitting and merging, etc., and asked “What’s that?” One of the members of the team said “Oh, that’s the behavior of the so-and-so part of the system, and, by the way, these rounded rectangles are states, and the arrows are transitions between states”. And that was all that was said. The pilot stood there studying the blackboard for a minute or two, and then said, “I think you have a mistake down here, this arrow should go over here and not over there”; and he was right.
>
> For me, this little event was significant, as it really seemed to indicate that perhaps what was happening was “right”, that maybe this was a good and useful way of doing things. If an outsider could come in, just like that, and be able to grasp something that was pretty complicated but without being exposed to the technical details of the language or the approach, then maybe we are on the right track. Very encouraging.

Harel’s account of statecharts is a compelling reason to use them for system specification. In an ideal world, the right tools not only allow for unambiguous specification of system behavior, but actually allow system designers to see complexity and reduce it, leading to a simpler and easier-to-maintain system.

Michael Breen’s account of the development of the Philips CDR870 compact disc recorder brings another perspective:

> Specification of the user-visible behaviour therefore began on a provisional basis, a final decision being deferred until progress on this would allow a more reliable assessment of feasibility. Because of the urgency of the schedule, it was agreed to proceed more-or-less straight to the construction of a formal model of the external behaviour, in co-operation with the person responsible for deciding on the details of the user interface: as a finite state machine model, the specification would be precise and unambiguous, thus helping to fulfill our quality

> requirement; it would also be essentially executable, thus helping to reduce implementation effort.
>
> The specification began well. For example, it was easy to model the behaviour of the CDR's tray as a finite state machine which responded to the pressing of the open/close button by changing from the "closed" state to the "opening" state. However, for this example, there is an exception: if the system happens to be recording when the open/close button is pressed then the convention is to treat this as a mistake and ignore it (forcing the user to press the stop button first). And, as more functions were incorporated into the model, the number of unusual scenarios to be considered increased substantially, with many of them being easy to miss. Not unusually for a system with a user interface, the complexity of the CDR was therefore quite disproportionate to its small size. Thus, it was possible to consider the behaviour of individual components, like the tray, and satisfy oneself that they were correct, only later to discover cases in which the overall system response was incorrect.
>
> The main problem was the lack of any systematic way to find these unusual scenarios, or corner cases. Initially, we believed that diligent effort would reduce the number of such cases that had been overlooked to the very low level necessary. As specification progressed, however, there was diminishing confidence on the part of the author in particular, as principal modeler and "scribe", that this point could be arrived at quickly enough. Further, it would be difficult to know that we had reached that level of quality — or when we were likely to reach it — with the degree of certainty one would wish in committing to a project with a hard deadline and minimal margin for scheduling error.
>
> Some other problems might in part be ascribed to the particular notation used for the model, which was Statecharts [10]. It was known from the outset that the person deciding the details of the user interface would have difficulty reviewing the statecharts. However, we understood there would also be difficulties with other notations based on the concept of communicating finite state machines. Given the pressing circumstances, it was agreed that he would sign off on a specification in this form trusting that it accurately modeled the behaviour discussed in the meetings. Nevertheless, this was obviously not ideal for either party and negated some of the advantages of having a formal specification.
>
> Since states may be organized hierarchically in Statecharts, the notation also permits a relatively high degree of structuring. However, this was not an advantage during the specification process as it was difficult to make good structuring decisions while the behaviour was still being elicited. For example, a decision to nest certain other states within the "tray closed" state might initially seem elegant but later prove awkward when trying to incorporate other elements into the model. Weighing such structuring decisions involved unwanted effort that would be better deferred to a separate design activity; changing any such decision additionally carried the risk of introducing errors into the specification by failing to preserve some correctly-modeled behaviour.

Breen's group decided to use a series of constraint tables during requirements development, showing which actions were possible and which were not.

> Completed, the original CDR specification uses 16 state variables, the values of which allow for almost 10^9 system states; however, taking the constraints into account, fewer than 400,000 of these states are possible in the model. The system behaviour is defined in response to 50 different events using a total of 365 transition rules.
>
> The method proved completely successful in eliciting the unusual scenarios that needed to be considered: Every possible combination of event and system state had to be covered by some rule and filling in all of the cells in the tables ensured that no dependency or detail of the response was overlooked. Whenever a new variable was added to the model, a new column of empty cells was created in existing rules, serving as a reminder of the potential dependencies or changes that remained to be considered as a result of the addition.

…

> A specification should be organized as a reference document rather than an introductory narrative about the system: though it is more difficult to browse, it is better in the long run as the information is in a form that allows easy reference throughout the project [11]. In the case of the CDR specification, one could confirm the response for any case simply by turning to the pages for the appropriate event and finding the applicable rule. As a comprehensive and detailed oracle, it reduced the danger of spurious problem reports from integration testers and it forestalled potentially time-consuming arguments on the details of the system behaviour (a particular problem for user interfaces, where everyone has an opinion): not only did it unambiguously state the required response, its explicit form helped to confirm this was not an unforeseen accident of implication; in effect, it declared, “Yes, this specific case was considered and this is what was decided.”
>
> The organization of the specification also allowed regression tests to be prepared with a minimum of additional documentation: to ensure broad coverage, tests were associated with all of the rules; each test referenced the rule it was principally designed to exercise through its unique identifier in the specification.

Hmm. Both accounts agree that a formal specification is very important, but Harel’s motivation for statecharts was rejection of a thick specification document in favor of simpler graphical charts, whereas Breen’s account noted the rejection of statecharts in favor of a more linear and enumerative specification document.

Which is right?

The graphical vs. textual argument, found in both documentation and programming, is something of a holy war and often a matter of personal preference; in the end, I would say whatever gets the job done more reliably is the right choice. Although I’m inclined to lean towards Harel on this one, in part because I feel a picture is worth a thousand words, and in part because I have this hunch that Philips isn’t the best organization for coming up with simple and clear system behavior. (Breen’s writing style, being more exhaustive than illustrative, is one symptom; another is the frustration my grandmother and I had with one of their LCD picture frames.)

Probably if each aspect of your system behavior is simple enough to fit on a one-page statechart, you should use the statechart approach. On the other hand, if they don’t fit on one page, and can’t be simplified to fit on one page, I would run away screaming and find another project to work on. But that’s just me.

Tools and Implementation Methods

I said earlier in this article that we don’t really have a good, widely-accepted set of tools for working with state machines. That’s not quite true. They’re just not as accessible to the general programming population as other tools like IDEs and compilers. Here are some of the tools I’m aware of:

State Machine Diagram Editors

I used ArgoUML for the diagrams I drew in this article. It’s free and open-source, and runs on Java, so it works on Windows and OS X, and should run on Linux. I’d give it a B- rating: ArgoUML is pretty easy to create statecharts (as well as many other UML diagrams), but the editor feels old and clunky, like an early 1990’s vector drawing tool. All the information you store in it is retained, so it’s not just a graphic presentation tool, but if you enter in the right things, someone should be able to create code based on your project. This property is called being “executable”, and it means there is enough machine-readable information in the design file to fully specify your state machine.

Astah is another UML editor with support for statecharts. It is published in several editions; the Community edition is free and has the most limited feature set, but they sell a Professional Edition for $299 at the time of this writing. I don't have much experience with this software to tell whether it's better or worse than ArgoUML. One of my coworkers likes it a lot, but he ran into some of the feature limitations of the free edition. Another coworker at Microchip, Justin Bauer, gave a good presentation at the MASTERs Conference in 2012 about using Astah for state machine design along with another tool, SinelaboreRT, to generate C code for use with Microchip's PIC microcontrollers.

QM (QP Modeler) — this is a graphical editor specifically aimed at the QP statechart library; see the next section for more details. I wasn't that impressed with QM.

Other programs which are graphical editors for visual purposes (but which don't retain a machine-readable description of a state machine) are yEd from yWorks, and Microsoft Visio. I used to like Visio before Microsoft bought it. The yEd editor is free, and although it's much more limiting than Visio, it does well as an editor for graphs (i.e. combinations of nodes + directed edges) with a nice auto-layout feature, so it's a decent match for drawing state machine diagrams.

Other software I will talk about in the next section is not limited to just diagram editing.

State Machine Implementation Tools

There are a few different techniques of implementing state machines in C/C++ or Java or other similar languages, and they boil down to one of the following:

- native language support
- hand-coded implementation
- tabular implementation
- unintentional state machines
- State pattern
- use of a library
- model-based code generation

I'll cover these one at a time.

- **Native language support** — in all the languages I know of, there isn't any built-in support for state machines, except for maybe Erlang. Pity.

- **Hand-coded implementation** — this is the technique I showed at the beginning; typically you have a big switch statement where you figure out what to do in each state, and figure out the next state. If you're lucky, you'll get something without errors.

- **Tabular implementation** — the tabular approach is to create a state transition table of triples (source state, destination state, input condition), something like this:

```
enum Input {
  IN_NULL = 0,
  IN_UP_BUTTON = 1,
  IN_DOWN_BUTTON_NOT_UP = 2,
  IN_NOT_UP_BUTTON = 3,
  IN_NOT_DOWN_BUTTON = 4,
  IN_TIMER_EXPIRE = 5,
};
enum State {
  STATE_OFF = 0,
  STATE_UP = 1,
  STATE_DOWN = 2,
  STATE_UP_DISABLE = 3,
  STATE_DOWN_DISABLE = 4,
};
typedef struct
{
  enum State source;
  enum State destination;
  enum Input input;
} StateTransition;

const StateTransition transitionTable[] = {
  {  STATE_OFF, STATE_UP,   IN_UP_BUTTON },
  {  STATE_OFF, STATE_DOWN,IN_DOWN_BUTTON_NOT_UP },
  {  STATE_OFF, STATE_OFF, IN_NULL },

  {  STATE_UP,   STATE_DOWN_DISABLE, IN_NOT_UP_BUTTON },
  {  STATE_UP,   STATE_UP, IN_NULL },

  {  STATE_DOWN,  STATE_UP_DISABLE, IN_NOT_DOWN_BUTTON },
  {  STATE_DOWN,  STATE_DOWN, IN_NULL },

  // more state transitions omitted
};
```

and then at the initialization of your application, you process the table, optionally storing the entries into hashmaps to make it an O(1) lookup. Here all the conditions have to be enumerated. It doesn't translate that cleanly for the input actions; you have to translate each possible combination of raw conditions (e.g. input button pressed, voltage below 3V, etc.) into one of several mutually exclusive values in the `Input` enumeration, so that at each step, exactly one of the `Input` values represents the inputs to the state machine. (I suppose you could replace the `Input` values with a series of function pointers that return `true` or `false` after evaluating some arbitrary series of conditions, and make the order of transitions in the table important so that the first matching condition dictates which state is next.) Then for each update of the state machine, use the table to determine the next state. It's not pretty, but at least it's not complicated, and it doesn't require anything special (= works in plain old C).

- **Unintentional state machines** — Don't make this mistake! We've all done it — you add some logic in your program, and some persistent system flag, and by accident, *poof*, there it is: buried with its tendrils wrapped around your application is a hidden state machine. Here's one I've done in the past: adding a square wave generator to a large signal-processing procedure:

```
int16_t sqwave = 1000;
uint16_t sqwave_counter = SQWAVE_PERIOD;

...

void main_system_step()
{
  ...

  if (sqwave_counter > 0)
  {
    --sqwave_counter;
  }
  else
  {
    sqwave = -sqwave;
```

```
      sqwave_counter = SQWAVE_PERIOD;
    }
    output_signal = plain_signal + sqwave;
  }
```

Here there are two states, positive and negative. Consider whether it would be better to pull out this logic to a separate state machine with those two states mentioned explicitly in the source code. And that’s not even a really bad one. I’ve seen software where there is some `flags` variable containing binary state flags, and scattered around the program are lines of code that read and write from the bits of this variable. When you see this sort of thing even start to rear its head, nip it in the bud and make sure it is designed and documented clearly as an explicit state machine. Don’t let the source code be the specification for state machine behavior!

- **State pattern** — See the Design Patterns book. The State pattern is an object-oriented approach. The idea is you create various class instances that encapsulate the behavior in each state, including determining which state is next. In Java, a state would naturally map to an interface like the following:

```
/*
 * Encapsulates other state variables in our system.
 * This is a plain value class, but it could be any
 * object type encapsulating the persistent information
 * in our system.
 */
class OtherState {
    public double x1;
    public int i;
}

interface State {
   public State step(OtherState otherState);
}
```

so that each state executes and determines which state is next. Simple states could be implemented as Java `enum` singletons:

```
enum MyState implements State
{
    FOO {
       MyState execute(OtherState otherState) {
           otherState.x1 += 3.0;
           otherState.i++;
           return BAR;
       }
    },
    BAR {
       MyState execute(OtherState otherState) {
           otherState.x1 -= 1.0;
           otherState.i--;
           return (i % 3 == 0) ? FOO : BAR;
       }
    },
    // more states omitted
}
```

In C++ it's a little more cumbersome to create this behavior, but you can imagine defining a State as a virtual base class, and implementing various subclasses for the individual states.

- **Use of a library** — the free C++ Boost libraries include two modules for implementing state machines:

    - the Boost::statechart module
    - the Boost::msm module

  Both use the power of Big Obscure Obfuscated Sophisticated Templates to create the correct logic to implement a state machine, with a minimum amount of runtime overhead, but they require you to understand both the intricacies of template programming in C++ and the quirky methods used in each module.

  As an alternative to the Boost libraries, Miro Samek's QP Framework libraries are suitable for plain C on embedded systems and are dual licensed (free GPLv2, or paid commercial license). I first found out about statecharts from reading Samek's *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*, and although it's been a while since I've looked at the QP library, at the time it seemed like a viable option for a project we were considering. He's written a tutorial article in Dr. Dobbs, and there are a few slideshow presentations floating out there which show examples of using QP in C (it uses preprocessor macros and function pointers). In recent years, Samek has written a new edition of his book, *Practical UML Statecharts: Event-Driven Programming*.

- **Model-based code generation** — here we turn to external tools used to draw statecharts and automatically generate application code from them directly. Again, I'm aware of a few possibilities, that vary widely depending on how much money you are willing to spend. I have little to no experience with any of them, however.

    - Miro Samek's company produces a free graphical tool called QM (QP Modeler) which can be used to draw statecharts and generate code using the QP framework. I've played around with it a little bit, and found its state-diagram drawing facilities to be rather clumsy.

    - SinelaboreRT — the term *sine labore* means "without work" in Latin; this program will take the UML from visual editors like Astah and ArgoUML, and generate C code from it. I've seen it demonstrated with 8-bit microcontrollers from Microchip, so the run-time overhead isn't very large. Sinelabore isn't free, but it's not that expensive ($119). Their website shows an example including a sample of generated code. I'd probably start with this software because of its price range and its proven targeting to small embedded systems.

    - Stateflow from The Mathworks is well regarded as an editor and generator of state machine code, although it's not cheap: TMW's website quotes $3,000 for a single license of Stateflow, and that's in addition to the base cost of MATLAB, and more toolboxes (e.g. Simulink Coder) for generating C/C++ code, so you're probably talking about $10-20K total to generate C++ code from a Stateflow-designed statechart. On the plus side, it appears to be less expensive than the IBM Rational tools.

    - Statemate — About the same time David Harel was developing his statechart technique, he and some colleagues started a company, called AdCad, which by 1986 had developed statechart software called Statemate. Shortly afterwards, AdCad morphed into I-Logix, which was acquired by Telelogic AB in 2006, which then got absorbed into IBM two years later. IBM's pricelist for Statemate lists a number of *à la carte* packages, but it looks like it would cost something like $40K-$50K for one license to use Statemate to design a state machine and generate C code. But I suppose if you're working on avionics applications and need to be confident your systems don't suffer from software bugs, you should be using the best software tools out there. Aircraft crashes are expensive failures.

Wrap-up and Further Reading

There's not really much else to say. State machines vary from the simplest of systems (managing an on/off actuator) to complex high-reliability systems where there might be dozens of subsystem state machines all interacting with each other in subtle ways.

In summary:

- Finite state machines are a way of describing the different discrete modes of a system, and how they transition from one state to the next based on the foreseeable input conditions.
- State machines are event-driven systems, and unlike blocking sequences of function calls, allow you to run multiple instances simultaneously.
- Architecting a state machine using a state diagram is a way to show the exact behavior of the state machine.
- Statecharts add some more powerful features to traditional state diagrams, including hierarchy and parallelism, to help organize and simplify state machines by introducing structure.
- State diagrams and statecharts are a way of visualizing the complexity of a system behavior. Use these tools to review and develop that system behavior, and keep an awareness of your system complexity. Aim for simplicity, and shun excess complexity.
- While small state machines can be hand-coded, more complex state machines are better left to more rigorous techniques like the State pattern, state machine libraries, or model-based code generation, to ensure they are implemented correctly.
- Beware of accidental state machines lurking in your application. Don't let the source code be the specification for state machine behavior!
- Even if they're not as glamorous as other areas of software engineering, state machines are a vital part of system design, and should never be overlooked.

Further reading

- "A Method for Synthesizing Sequential Circuits," George H. Mealy, Bell System Technical Journal 34: 5. September 1955.
- "Gedanken-experiments on Sequential Machines," Edward F. Moore, pp 129 – 153, Automata Studies, Annals of Mathematical Studies, no. 34, Princeton University Press, Princeton, N. J., 1956. The diagrams at the beginning of this article are from Moore's paper.
- "Finite Automata and Their Decision Problems", M.O. Rabin and D Scott, IBM Journal of Research and Development, Volume 3 Issue 2, April 1959.
- "Statecharts: A Visual Formalism for Complex Systems", David Harel, Science of Computer Programming, Volume 8 Issue 3, June 1, 1987.
- "Statecharts in the Making: A Personal Account", David Harel, Proceedings of the third ACM SIGPLAN conference on History of programming languages, Pages 5-1-5-43, 2007.
- "UML Tutorial: Finite State Machines" Robert C. Martin.
- "Understanding State Machines", The MathWorks.

Next up: **abstraction**.

Previous post by Jason Sachs:
Optimizing Optoisolators, and Other Stories of Making Do With Less
Next post by Jason Sachs:
Voltage Drops Are Falling on My Head: Operating Points, Linearization, Temperature Coefficients, and Thermal Runaway

---