9 A framework for design

This chapter describes a basic programming framework that lets us build, test and analyze any interactive device. Once we have a framework for programming finite state machines, we can do all sorts of things efficiently. The design framework presented here is very simple, trimmed as it is for the purposes of the book. If you like programming, please be inspired to do better with your own design projects. If you don't like programming, it is still worth reading through this and the next few chapters, but be reassured that non-programming designers could use tools that hide the programming details. However, the insights a designer can achieve through this sort of programming are very important—so read the this and the next few chapters to see what can be achieved.

9.1 Future-proof frameworks

An important question to ask at the start is, "How extensible and future-proof is this approach?" A very cynical answer is that the framework holds together for the devices we want to cover in this book, and not much else, let alone the future. A more useful answer is that the framework is about *ideas*, which it illustrates perfectly clearly, and if you want to really use it, it will need developing and extending for whatever your particular application is. Most likely, you would build something like it into a design tool. Of course, if you are developing real production devices, you may not want to use JavaScript anyway—but the JavaScript ideas we describe will convert easily to any programming language.

A more realistic answer is that the framework is based on finite state machines. These are core technology for any pushbutton style user interface, though you may wish to make minor modifications to the framework to handle details of your specific applications.

▷ The framework is not restricted to "pushbuttons" as such, although they are practically ubiquitous ways of interacting with devices; for example, we'll have a look at a cordless drill—which has knobs and sliders—in section 9.8 (p. 316). We will discuss some variations on the framework, with ideas for programming in more professional ways, in section 9.7 (p. 312).


namco
HONDA

In the future, user interfaces will no doubt move beyond push buttons. Devices may be controlled by speech or even by direct thought, through some sort of brain implant. The framework will work just as well for such user interfaces, although we've only developed it here with button pressing in mind. In fact, a really important idea lies behind this: speech (or any other panacea) does not escape *any* of the interaction programming issues our framework makes clear and easy to explore.

9.2 A state machine framework

We first introduce our programming framework using a simple toy example—the framework is more important than the example, for the time being.

The first, and simplest, device we'll consider is a lightbulb, with three states: off, dim, and fully on. This can be drawn as a transition diagram with three circles, one for each of the states, and with arrows among them showing how one could change the state. As it happens, this lightbulb allows any state to go to any other state directly—but this is rarely the case with more complex devices. Figure 9.1 (facing page) is a state transition diagram showing how the lightbulb works.

The user interface is simple—buttons as well as states are called Off, On, Dim. For this device, regardless of what its called, a button *always* changes from any state to a state with that name. This makes the user interface very simple and consistent. There are only two disadvantages: if there are lots of states, there need to be lots of buttons, and—for the purposes of this book—there is a small risk that we may confuse the state names and the button names (in general, they need not be the same).

Most device descriptions are much bigger, but the lightbulb is simple and familiar enough to show it in its entirety.

There are many ways to program. For such a simple device, it would be tempting to go straight into programming: there are only three states and three buttons, and this is easily programmed. Indeed, we know that each button sets the state with the same name, so we may as well program as follows:

```
var state;
function action(button)
{ state = button;
}
```

However, if we wrote program code like that, we would have several problems. We would find it hard to modify the design (what happens when we change the name of a button, say to make a device for a foreign market?), and we would find analyzing the design hard (the code above is so terse that it isn't obvious what it does). We need to avoid both problems: we want a flexible framework that allows easy analysis to help the designer.

First, we will represent the device by a set of arrays. We have an array of button names, an array of state names, and a table (a two dimensional array) specifying how each button changes each state. The advantage of this approach is that it makes a working device, and it permits any analysis.

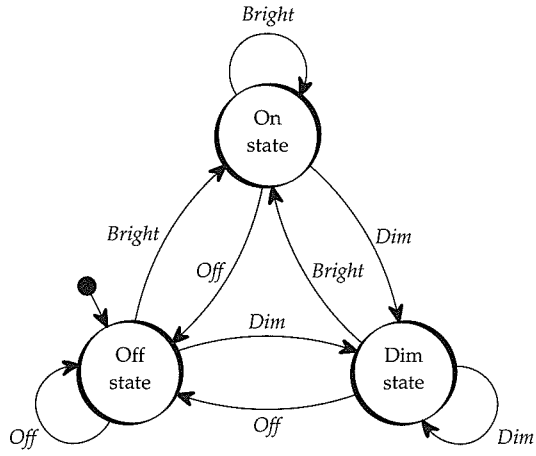


Figure 9.1: The transition diagram of a simple three-state lightbulb, which can be in any of three states: on, off, or dim. The initial state is Off, as shown by the default arrow. All actions are shown, even where they do not change states, so there are always three arrows out of all states.

Since every device has its own arrays, and we want to have a general approach for any device, it helps to group all of the device details together. JavaScript (and most other languages) allow data structures to be combined into single objects, and that is what we will do here.

The approach shown below is a pretty full description of how a device—in this case, our lightbulb—works.

```
var device = {
  notes:  "simple lightbulb, with dim mode",
  modelType:  "lightbulb",
  buttons:  ["Off", "On", "Dim"],
  stateNames:  ["dim", "off", "on"],
  fsm:  [[1, 2, 0], [1, 2, 0], [1, 2, 0]], // see Section 9.2.2 (p. 277)
  startState:  1,
  state:  0,
  manual:  ["dim", "dark", "bright"],
  action:  ["press", "pressed", "pressing"],
  errors:  "never",
  graphics:  "bulbpicture.gif"
};
```

Notice that we have allowed for quite a few more features: we've specified the initial state (the lightbulb will start at the state off); we've given the device a name and picture; and we've chosen simple words and phrases to describe what the device does. We can now concentrate on getting the simulation and analysis

to work, confident that we can easily change the device specification later if we want to.

9.2.1 How the specification works

The lightbulb definition creates a JavaScript object and assigns it to the variable `device`. The various fields of the object can be accessed by writing them after dots; for example, for this object the field `device.modelType` is the string `"lightbulb"`.

Some of the fields are arrays. For example `device.action` is an array with three strings: `["press", "pressed", "pressing"]`. The elements of arrays are accessed by using numbers written between square brackets. Given these values, `device.action[1]` will be the string `"pressed"`. Like Java and C, JavaScript counts array elements from 0, so `device.action[0]` is the first array element and `device.action[1]` the second.

Some devices won't have buttons to press; they might have pedals, levers, or knobs to turn. So the `device.action` field says how the user is supposed to do things: for this simple lightbulb, the basic action is to *press* a button. Since we might want to use the right word with different tenses, we've provided for present, past, and future forms of the word. The field `action` can be used to generate English explanations of how to use the device: we may want to generate text like, "If you are in the dark and want the lightbulb to be on, try **press**ing the On button," or, "To put the lightbulb off, **press** the Off button," or, "You **pressed** the On button, but the lightbulb is still on." You can see that the various English fields in the device specification will give us great scope for generating helpful text messages for users and indeed for our own analyses, making it easier for the designer to understand too.

The framework is simple but very flexible. If you want to change the names of the buttons, you could change the line that starts `"buttons:"` and give them different names. The advantage is that the complete definition of the lightbulb is kept in one place. It is quite separate from how we get the user interface to work and quite separate from how we make the device look.

An obvious limitation with the scheme is that I've only got one graphics image to cover all three states of the device. It might have been better to make `graphics` an array with, in this case, three images. With only three states and hence only three images, that seems to make sense, but in general we might have thousands of states and we would probably not want to end up having to organize thousands of pictures too. Instead, pictures would be better generated by a concise bit of program (which is how almost all graphical user interfaces work: all the images are drawn by program), and each state's specific drawing would be laid on top of a basic background image.

The lightbulb has three states, and as can be seen reading the `stateNames` field, they are called dim, off, and on; because of the way JavaScript array subscripts work, they are numbered 0, 1, and 2, respectively. The actual numbers are arbitrary (though they have to be used consistently throughout the definition). Note that I've chosen the numbers for buttons and states so that their names do not correspond, as in general there will be no clear-cut correspondence even though for

this simple device there is one we could have exploited. But we're trying to build a general framework for any device, not just for special cases.

9.2.2 State transition tables

The key part of the device's definition is the bit `fsm`. The numbers here define the finite state machine (FSM) that implements the device, the three-state lightbulb in this case. The definition of `fsm` in JavaScript, `[[1, 2, 0], [1, 2, 0], [1, 2, 0]]`, is equivalent to a simple matrix of numbers:

```
1  2  0
1  2  0
1  2  0
```

These particular numbers are not very exciting, but in general the `fsm` matrix tells us how to go from one state when a button is pressed to the next state. Every row of this matrix of numbers is the same only because the lightbulb is so simple, and its buttons always do the same things; in general, though, each row would be different.

The matrix structure is easier to understand when it is drawn out more explicitly as a table:

	Go to this state when this button is pressed		
When in this state	Off	On	Dim
0: dim	1: off	2: on	0: dim
1: off	1: off	2: on	0: dim
2: on	1: off	2: on	0: dim

From this table we can read off what each button does in each state. This lightbulb isn't very interesting: the buttons always do the same things, whatever the states.

The same information can be presented in many other forms, and often one or another form will be much easier to read for a particular device. State transition tables (STTs) are a popular representation.

The full state transition table for the lightbulb is particularly simple:

Action	Current state	Next state
Press Off	on off dim	off
Press On	on off dim	on
Press Dim	on off dim	dim

Each row in the table specifies an action, a current state, and the next state the action would get to from the current state. Other features may be added in further

columns, such as how the device responds to the actions, or the status of its indicator lights. (We would also need corresponding entries in the `device` specification.)

State transition tables can usually be made much shorter and clearer by simplifying special cases:

- If an action takes all states to the same next state, only one row in the table for the action is required (cutting it down from as many rows as there are states).
- If an action does not change the state, the row for that state is not required (with the proviso that if an action does nothing in any state, it needs one row to say so).
- If an action has the same next state as the row above, it need not repeat it.

Here is a state transition table for a simple microwave oven, illustrating use of all these rules:

Action	Current state	Next state
Press [Clock]	*any*	Clock
Press [Quick defrost]	*any*	Quick defrost
Press [Time]	Clock	Timer 1
	Quick defrost	
	Timer 1	Timer 2
	Timer 2	Timer 1
	Power 1	Timer 2
	Power 2	Timer 1
Press [Clear]	*any*	Clock
Press [Power]	Timer 1	Power 1
	Timer 2	Power 2

This table was drawn automatically in JavaScript from the specification given in section 9.4.6 (p. 286)—it was typeset using LaTeX, though HTML would have done as well. Unfortunately, and ironically because of the simplifying rules, the JavaScript framework code to generate this table is longer than it's worth writing out. The full code to do it is on the book's web site, mitpress.mit.edu/presson.

Our simple framework is written in JavaScript, because it has lots of advantages for a book, but a proper development framework would be much more sophisticated. It would allow designers to write state transition tables (and other forms of device specification) directly—just as easily as if they were editing a table in a word processor. Behind the scenes, a proper design tool constructs the finite state machine data, which is then used to support all the features of the framework. Ideally, the finite state machine would be reconstructed instantly when even the smallest change was made to the table, and every feature the designer was using—simulations, user manuals, analysis, whatever—would update itself automatically.

To recapitulate, from the framework model we can generate representations of the device—such as STTs and, later, user manuals, analysis, diagrams and help—but we can also (with a little more effort) use those representations to either edit

or generate what we started with. This creates a powerful two-way relationship, and it is *always* worth thinking about what opportunities there are to do this.

The general idea is called equal opportunity: everything has an equal opportunity to contribute to the process. When equal opportunity is used in a design process, there is no need to see one product—such as a user manual—as the output of the design (and therefore likely to come last) but as part of the design. If a technical author makes changes to the manual, those changes get carried back in the appropriate way to the original design, and the device would then behave differently—and no doubt have a better manual for it too.

9.3 Prototyping and concurrent design

Now that we have come this far, it's a simple matter to write a program that can run the device from the table. Since we are using JavaScript, we'll first build the device simulation as an interactive web page.

▷ Practical programming continues in section 9.4 (next page), but first, we want to philosophize about frameworks and design processes.

In conventional design, we distinguish between low-fidelity prototypes and high-fidelity prototypes. Typically, a low-fidelity prototype will be made out of paper drawings and will be used to help users and designers understand the major states of a system; high-fidelity prototypes are usually devices that look more realistic, typically being drawn using decent computer graphics.

The problem with this view of design is that in progressing from low to high-fidelity, you practically have to start again. You make paper models, throw them away, make more realistic models, throw them away, then start making a prototype on a computer—then you make the *real* device. You have to do several times as much work, and each stage loses whatever it learns: paper models for a low-fidelity prototype don't have any program code in them anyway; later prototypes are often written in Flash, Tcl/Tk or some other easy-to-use and flexible programming environment, but the final product has to be rebuilt as a standalone or embedded device using some completely different technology, such as programming in C or a burned in firmware on a chip.

Often the final implementation is such a technical process—it must deal with things like wireless communications protocols, battery management, screen updates including cool animations—that the programmers closely involved with the earlier design stages are now not involved at all.

Why be so inefficient? Why not have a design framework that covers a wide range of development needs? This chapter will get a design to a stage where it runs like an interactive web page; if you print off the web pages, you will have a low-fidelity paper prototype. If you get a user to interact with the web site (perhaps after putting in some nice images) you have a higher-fidelity prototype where you can start getting feedback from users. You'll also be able to generate user manuals and get lots of interesting design analysis to help you pinpoint design issues that need attention—or are opportunities for innovation. Crucially, as you discover

ways of improving the design—say, when a user spots something—you can revise the design and redo everything very efficiently.

The framework is surprisingly simple. Obviously what we are sketching is more pedagogical than real, but the approach—the ideas and motivation behind the approach—has many benefits. It helps makes design concurrent: everything can be done more-or-less at once, and no information need be lost between successive phases of the design process.

When we use a design framework to design *concurrently*, the following advantages become apparent:

- Complex information is shared automatically among different parts of the design process and with different people (programmers, authors, users) engaged in the design process.
- Design ideas and requirements formulated in the framework can be debugged and quality-controlled *once*, yet used for many device designs.
- Work need not get lost. The same design representation pervades everything and does not need to be redone or "repurposed" for other aspects of the design.
- Many problems for any part of the design can be identified immediately. For example, technical authors can start to write user manuals immediately; problems they face (say in explaining difficult concepts or in using draft manuals in training sessions) are then known from the start, rather than when it is too late.
- Rather than waiting for later stages of a design process to confront problems, it is possible to test how the "later" stages work almost immediately. The entire design life cycle can start to be debugged from the earliest moment, and feedback from draft final stages is available to improve the earliest conceptual work.
- It is possible to improve the design framework itself. Insights to improve the framework that help particular design projects are programmed into the framework and then are freely available to other projects.

These are grand claims for something so simple, but this is a different philosophical approach to design: you start with a simple framework and embellish it, by extending the framework in different directions as need arises. Since the extensions are automated, if you make any changes, the products are regenerated. The alternative is to have a complex design environment, where each product—low-fidelity, high-fidelity, whatever—is done from scratch, and has to be done again if there are any changes. Put another way, since every interactive device is a state machine, we build a framework to run and analyze state machines.

9.4 Prototyping as a web device

There are three main ways to convert a device into an interactive web site, which we'll now explore in turn.

9.4.1 Using HTML links within a page

The simplest and most direct way to convert a device specification into an "interactive experience" is to translate it into a *single* HTML page.

First, check that your JavaScript device specification works—for if it has any errors in it (like a missing comma) nothing will work. It's good practice to start off with something simple, make sure it works, then get more sophisticated—but only once you that know the foundations are sound.

```
var device = ...; // insert your device specification here.
document.write("<h1>Summary of "+device.modelType+"</h1><br>");
document.write("This device has "+device.stateNames.length+" states and ");
document.write(device.buttons.length+" buttons.");
```

▷ As described in section 4.5 (p. 105), where we defined the function `plural`, this JavaScript would generate better English if we wrote `... "This device has "+plural(device.stateNames.length, "state")+" and "+plural(device.buttons.length, "button")+"." ...`

You can either use + to join strings together (+ will also add numbers), or you can just as easily use several calls to `document.write` on the strings separately. It's a matter of personal style.

```
for( var s = 0; s < device.fsm.length; s++ )
{ document.write("<hr><a name="+s+">In state <b>"+device.stateNames[s]
    +"</b> you can press:</a><ul>");
  for( var b = 0; b < device.buttons.length; b++ )
    document.write("<li><a href=#"+device.fsm[s][b]+">"
      +device.buttons[b]+"</a></li>");
  document.write("</ul>");
}
```

This will generate a hugely insightful web page! It'll look like this:

In state **dim** you can press:
- Off
- On
- Dim

In state **off** you can press:
- Off
- On
- Dim

...

If you click on one of the buttons, shown in your browser as underlined hot text, the web page will scroll up and down to get the current state at the top (depending on your browser: you may need to make the window smaller so you can see it scrolling—otherwise, your browser won't seem to do anything if the target state is already visible in the window without scrolling). In a sense, you've not so much got an interactive simulation of an interactive device as a rather banal interactive (hypertext) user manual for it.

It's easy to do much better. Here's one idea: give the user some hints about what pressing a button will do:

```
for( var s = 0; s < device.fsm.length; s++ )
{ document.write("<hr><a name=new"+s+">In state <b>"
    +device.stateNames[s]+"</b> you can press:</a><ul>");
  for( var b = 0; b < device.buttons.length; b++ )
    if( device.fsm[s][b] != s )
    {  document.write("<li><a href=#new"+device.fsm[s][b]+">"
          +device.buttons[b]+"</a>");
       document.write(" - goes to state <b>"
           +device.stateNames[device.fsm[s][b]]+"</b>");
       document.write("</li>");
    }
  document.write("</ul>");
}
```

In this example, I changed the HTML link names from simple numbers (in the first example) to `new` followed by the state number (in the second example)—to ensure that the examples can safely work together without their link names clashing. I've also removed all transitions that do nothing; this makes the manual a lot shorter for many devices.

9.4.2 Multiple HTML pages

Rather than use a single page and rely on scrolling, we can convert a device into a multipage web site with lots of HTML files. The easiest way to do this is to have each HTML file correspond to a single state, HTML links among the pages then correspond to transitions. With this approach, it is very obvious that clicking a button changes state, since the browser loads a new page.

Typically, each page would say which state it was, and every page would have named hot text representing the device's buttons, much as before. Each page can have a name like `device1.html`, `device2.html` and so on, corresponding to the HTML description for what state 1, state 2, and so on do.

If you represent a device like this as lots of linked web pages, try using a web-authoring tool, like Dreamweaver, which can run checks on web sites to check that all the links work—for exactly the same reasons an interactive device should be checked.

▷ Dreamweaver has built-in tools to find pages you never link to and to find links that don't work. These are issues we will deal with later, in section 9.5 (p. 288), when we will write JavaScript to do these and other checks.

Representing a device as lots of web pages requires some way of writing lots of files automatically, which, unfortunately, is beyond JavaScript's capability when it is running in a browser.

Using one file, many URLs

Instead, we can "cheat" to achieve the same effect. Rather than naming lots of files `device1.html`, `device2.html`, and so on, we use the URLs `device.html?1`,

device.html?2, and so on. This is practically the same thing, except that everything can now be done with a *single* file, device.html, with the "search" part of the URL (what comes after the question mark character) selecting which state the device is in.

In the approach we used above, a for loop ran over all possible values of the state number. Now each page only has to deal with one state, but it gets the state number not from a for loop but from the search part of the URL, as follows:

```
var device = ...; // pick a device
var state = location.search.substr(1); // get the search string

if( state == "" || state < 0 ||  state >= device.fsm.length )
  state = device.startState; // use initial state if not given explicitly

document.write("<h1>"+device.modelType
   +" is "+device.stateNames[state]+"</h1>");

document.write("<ul>");
for( var b = 0; b < device.buttons.length; b++ )
  document.write("<li><a href=device.html?"
    +device.fsm[state][b]+">"+device.buttons[b]+"</a></li>");
document.write("</ul>");
```

Using URLs for states

An alternative to these simple (but systematic) methods of using the state number to directly select an HTML page is to use stateNames to apply URLs to states. Each state has a name that is the file you want the browser to show when the device is in that state (of course, if you want the names to be sensible, you could add an array url to the framework to serve the same purpose and keep stateNames with its original purpose).

Associating URLs with states reminds us that a state can be sophisticated, just as a URL could be a movie, an interactive form, or even a whole other web site, with *any* features within it. In other words, states in the framework are really clusters, in the sense of statecharts.

▷ For more on statecharts and clusters, see section 7.1 (p. 201).

9.4.3 Using forms

The third way of converting a device to an interactive web page can be done easily in JavaScript and is by far the best way for our purposes. It uses an HTML form to set up some buttons and a simple display, and the form text is updated with the state name (and anything else you want to say). This is an extremely easy and reliable way to get dynamic HTML working. Forms also give us buttons that look like buttons, so we can do better than using underlined hot text.

Loading up a web page in your browser with the code we will develop in this section will get a picture something like figure 9.2 (next page).

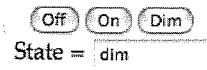


Figure 9.2: What the lightbulb device simulation looks like after you press the Dim button. The three buttons of the device are shown along the top row, and underneath is the text field that displays the current state name, all generated by the framework code.

First you need a basic form (written in standard HTML) for the buttons and a text field for the state name:

```
<form>
  <input type='button' value='Off' onMouseup='press(0)'>
  <input type='button' value='On'  onMouseup='press(1)'>
  <input type='button' value='Dim' onMouseup='press(2)'>
  <br>
  State = <input type='text' name='display' readonly><p>
</form>
```

Rather than calling specific button functions like `off` in the HTML code for the form, we've used a more general approach by having a single function `press` that can do *any* button; it just needs to be told which one. Thus `press(0)` means do whatever pressing Off should do, `press(1)` means do whatever pressing On should do, and so on.

The HTML form above can only handle a three-button device, and it is restricted to fixed button names (Off, On, Dim) at that. It is better to use JavaScript to generate a form for *any* device specification. We only need a JavaScript `for` loop to do it and to use `document.write` to generate the correct HTML. Here's how to do everything automatically:

```
function makeForm(d)
{ document.write("<form>");
  // generate one line for each button
  for( var s = 0; s < d.buttons.length; s++ )
  { document.write("<input type='button' ")
    document.write("value='"+d.buttons[s]+"' ")
    document.write("onMouseup='press("+s+")'>")
  }
  // plus some more HTML to get a working display ...
  document.write("<br>State = <input type='text' name='display' readonly>")
  // then finish the form
  document.write("</form>")
}
```

Now, if we write `<script>makeForm(device)</script>` anywhere, this will generate the HTML form code automatically. If the device is the lightbulb we defined earlier in this chapter, then this code will automatically generate the example form given above.

9.4.4 Look and feel

The last few sections developed very different appearances for the device, yet they all worked the same way. First, the device was presented as a single web page, then as many separate web pages of text, and finally we generated a form. The design and layout of the form was built into the code of `makeForm()`—the earlier two "look-and-feel" approaches were so banal we didn't even give the JavaScript that generated them a special name.

Every device we work with will now end up looking much like this. If we wrote a different `makeForm()` we could generate a form with a quite different layout, color scheme, or whatever. Then every device we worked with would look different, but consistently. So in particular, we can change how they look without changing how they work—a process sometimes called skinning.

Crucially, the look-and-feel of the interactive device is separated from its interaction rules. We can change one without affecting the other. In a professional design environment, we might have different people working on each aspect separately. If changes are made to the look and feel, say, changing the color scheme, the other people on the design project do not need to know; conversely, if the device specification is changed, the look and feel is automatically revised to work on the new device.

In an ideal world, we would invent a language (no doubt in XML, so that other programs could help us store and manipulate it) to describe the look and feel of a device, just as we invented a language (actually a JavaScript data structure) for specifying the state machine part of the device. Here we've simply written JavaScript code to specify the look and feel. Although JavaScript is very easy to fiddle with, it isn't a good language for a design tool to control.

Our `makeForm()` might be totally flexible, but our design framework has no insight into how it works. For example, if we accidentally made the background and the text colors the same, nothing in the form would be readable—and there is no general way that we can program in checks like this that would work with every device and every look and feel we used.

▷ Section 11.7.1 (p. 402) presents a case for adding look and feel information in the framework (rather than into program code).

In a way, then, the possibility of improving the approach to look and feel, and the advantages of doing so, is a small parable about the philosophy of the design framework. The design framework as we develop it in this book only works with interaction, not look and feel.

9.4.5 Initial checking

Creating the interactive form is only one part of initializing a device, and it is useful to have a single function to do all the necessary work:

```
function initialize(d)
{ d.state = d.startState; // start off the device in the initial state
  makeForm(d);
  ... other details to be added ...
}
```

It's convenient to have a function `displayState` to display the state name by writing the name of the state to the form field:

```
function displayState(d)
{ document.forms[0]["display"].value = d.stateNames[d.state];
}
```

The initialization function will call `displayState(d)` to make sure that the initial state is correctly displayed. A more exciting simulation of a device would show different images for each state, rather than just writing strings of text—the state description—to a field, as we've done here.

Now all that is needed is the next state function, which we've called `press`. Notice how `press` uses the `fsm` table to work out the next state from the current state and the number of the button the user has pressed. It then calls `displayState` to show the user the new state name.

```
function press(buttonNumber)
{ device.state = device.fsm[device.state][buttonNumber];
  displayState(device);
}
```

It takes a little bit more tinkering to get it all perfect, but when all the details are sorted out, we can run the lightbulb simulation as a self-contained web page.

9.4.6 Specifying a different device

Once we have a framework, changing or revising the device specification is easy. The *same* Javascript code will work for *any* device: we can easily change the definition of `device` to change its behavior. Here is the device specification for a microwave oven:

```
var device2 = {
  notes: "A basic microwave oven, based on a Hyundai model",
  modelType: "Microwave oven",
  buttons: ["[Clock]","[Quick defrost]","[Time]","[Clear]","[Power]"],
  stateNames: ["Clock","Quick defrost","Timer 1",
               "Timer 2","Power 1","Power 2"],
  fsm: [
        [0,1,2,0,0],
        [0,1,2,0,1],
        [0,1,3,0,4],
        [0,1,2,0,5],
        [0,1,3,0,4],
        [0,1,2,0,5]
       ],
```

```
  startState: 0,
  action: ["touch","touched","touching"],
  manual: ["has the clock running","doing quick defrost",
           "using Timer 1","using Timer 2",
           "on Power level 1","on Power level 2"
          ],
  errors: "never",
  graphics: ""
};
```

▷ We shall use this definition of a microwave oven as a frequent example. The oven was discussed at length in Jonathan Sharp's PhD thesis (see the further reading at the end of this chapter, p. 323); we use his *exact* definitions to emphasize the generality of the approach.

If we just do `device = device2` the framework will obligingly change from simulating a lightbulb to simulating this microwave oven.

9.4.7 Using an image map

Our definition of the lightbulb included a picture file: instead of boring buttons, an image and image map could be used to make the interaction simulation far more realistic. If you want to use an image map so the user can click on regions within the large picture that correspond to buttons or knobs, then the device specification should include coordinates for its hot regions.

▷ An example of using an image map is given in section 9.5.1 (p. 291), though the image there is the device's state transition diagram, not its buttons and front panel.

9.4.8 Handling errors

Although the lightbulb has no scope for errors, the device specification had some information about errors, namely, a string `device.errors`. This field could be set to `"beep"` if we wanted the device to beep when there is an error, to `"never"` if we wanted to ignore errors, or perhaps to something more complicated, say, to identify some states with an error. For this book, I decided not to make error handling a big issue for the framework, because the way errors are handled affects almost all of the framework and makes it much more complicated than we need to bother about here.

In fact, error handling is always tricky, and most interactive systems do it very badly. If we put decent error handling into the framework, we'd only have to do it once (perhaps leaving some details to the requirements of specific devices), and then every device developed with it would "inherit" the careful error handling.

For some devices we might want to warn the user that pressing a button does nothing. We could use code like this:

```
if( state == device.fsm[device.state][buttonNumber]
     && device.errors != "never" )
  alert(device.action[2]+" "+device.buttons[buttonNumber]+
        " doesn't do anything in this state!");
```

Notice that we used the pressing device action from the `device.action` array. The program will test whether `device.errors != "never"` again and again, so better programming might call for writing:

```
var reportErrors = device.errors != "never";
...
if( reportErrors && state == device.fsm[device.state][buttonNumber] )
...
```

Remember that most errors we can report to users (while they are interacting with the simulation) could have been detected before running the device. For instance, if we think that buttons having no action is an error, then we can write program to find all such cases automatically and then modify the design to eliminate them before a user ever interacts with it.

9.4.9 Logging a commentary on user actions

The device specification includes a simple manual entry for each state as well as a choice of words for what users do when they press buttons. Here's one way to use the manual part of the specification to provide a running commentary on the user working the device:

```
function press(buttonNumber)
{ ...
  document.forms["info"].comment.value = "You "+device.action[1]+
          " "+device.buttons[buttonNumber]+
          ", and "+(same? "nothing happened.":
                          "it is "+device.manual[device.state]+" now.");
  ...
}
```

We could easily extend the framework to have a button [PROBLEM] that would let the user write a description of any problem they have found, and we could automatically annotate the user's comments with the state where it was found (and perhaps also the last few states so a designer looking at the report would know how the user got there and experienced the problem).

9.5 Basic checking and analysis

Any device may be specified with accidental errors the designer hasn't noticed. One job for a framework is to perform checks that basic design errors have not been made—this is an important part of design that is often forgotten in ordinary programming, where the designer just writes code to do things and then has no easy way to check what has been done.

If every program is written from scratch, the effort to do detailed checking will rarely seem worthwhile: most of the useful checks are not particularly easy to program (and it is always going to be easier to hope that the design is correct than do hard work). Instead, in a framework, we only have to write the checking program code *once*, and it is then *always* available for every device we want to use. That is a very useful gain and a good reason to use a framework.

The first and simplest thing to check is that we have a valid finite state machine with the right number of buttons. The properties to check are as follows:

- The basic fields are defined in the device specification. (In a strongly typed programming language like Java this step is not be required.)
- The number of buttons and the number of states conform to the size of the various `fsm`, `indicators`, `manual`, `action` fields. (This is a check that both Java and JavaScript cannot do without some explicit programming.)
- The `startState` and all entries in the `fsm` are valid state numbers. (Again, this is a check that Java and JavaScript cannot do without programming.)
- Report on whether any states have no actions in them. This is generally an error, except for special devices.
- Report on whether any states have only one action in them. This is an example property of a device that may or may not be a design error; it may be deliberate in some states for some devices, but generally it is worth highlighting for a designer to take note.

  ▷ Pointless states can be identified visually from transition diagrams; they raise a potential design issue discussed in section 6.3.3 (p. 184).

In JavaScript almost anything goes, so we have to check explicitly that strings are strings, numbers are numbers, and so on. It is easiest to define some handy functions to do this and to report errors as necessary:

```
function isString(s, name)
{ if( typeof s != "string" )
    alert(name + " should be defined as a string");
}

function isWholeNumber(n, name, lo, hi)
{ if( typeof n != "number" || Math.floor(n) != n || n < lo || n > hi )
    alert(name + " (which is "+n+
                  ") should be a number between "+lo+" and "+hi);
}
```

```
function isStringArray(a, name, size)
{ if( a == undefined )
    alert(name+" field of device not defined");
  if( a.length != size )
    alert(name + " should be a string array, size "+size);
  for( var i = 0; i < size; i++ )
    isString(a[i], name + "["+i+"]");
}
```

The curious bit of code in `isWholeNumber`, namely, `Math.floor(n) != n`, checks that `n` is a whole number in the appropriate range, `lo` $\leq$ `n` $\leq$ `hi`. The standard JavaScript `Math.floor` function removes any decimal part of a number, so 2.3 becomes 2; of course, 2.3 is not equal to 2, so the function determines that 2.3 (or any number with a fractional part) is not a whole number. We need to check whether the type of `n` is a number too since `Math.floor` would fail (returning `NaN`—not a number) if it wasn't.

In Java, or another statically typed language, we could check this much more easily by declaring things to be `int`—then the Java compiler would do the checks with no further effort on our part, but JavaScript does not check whether a number is an integer or a floating point number.

Next we want to report on any pointless states. This is just a simple matter of examining each state and counting the number of transitions from that state to other states. If a state has only one exit, then it possibly serves no purpose, unless that state does something useful for the user as a side effect. In any case, the designer should know the list of pointless states in order to review that they do indeed have some use; otherwise, they should be deleted.

```
function reportPointlessStates(d)
{ for( var i = 0; i < d.fsm.length; i++ )
  { var count = 0;
    for( var j = 0; j < d.fsm[i].length; j++ )
      if( d.fsm[i][j] != i )
        count++;
    if( count == 1 )
      alert("State "+d.stateNames[i]+" may be pointless.");
  }
}
```

In a perfect world (that is, one where the framework is developed to be more sophisticated and "industrial strength") designers could record their decisions: each state could have a flag to say that it is alright for it to be "pointless," and the check code would not report it again. Indeed, as soon as you write any program code like this, many other checks and features will occur to you. For example, why not check that the count of transitions from a state is at least 1, because surely 0 would be an error too? Checking that a single state has no exit is one problem, which we could report with a simple change to `reportPointlessStates`, but what if a set of several states have no common exit, but each state within the set does? That's harder to check.

▷ We will make more powerful checks like this later, in section 9.6 (p. 297).

With these functions, the checking is now straightforward:

```
function checkDevice(d)
{ // check strings
  isString(d.modelType, "modelType");
  isString(d.notes, "notes");
  isString(d.errors, "errors");
  isString(d.graphics, "graphics");
  // how many buttons?
  var b = d.buttons.length;
  // how many states?
  var s = d.stateNames.length;
  isWholeNumber(d.startState, "startState", 0, s-1);
  // check right number of things of right type
  isStringArray(d.buttons, "buttons", b);
  isStringArray(d.stateNames, "stateNames", s);
  isStringArray(d.action, "action", 3);
  isStringArray(d.manual, "manual", s);
  // check fsm
  if( d.fsm.length != s )
    alert("fsm does not have "+s+" rows");
  for( var i = 0; i < s; i++ )
  { if( d.fsm[i].length != b )
      alert("fsm row "+i+" does not have "+b+" columns");
    for( var j = 0; j < b; j++ )
      isWholeNumber(d.fsm[i][j], "fsm["+i+"]["+j+"]", 0, s-1);
  }
  reportPointlessStates(d);
}
```

Of course the `checkDevice` function can be applied to *any* device. This generality is one of the huge advantages of using a framework: once you notice that one device has potential problems you can identify (or even fix) automatically, modifying your framework will avoid those problems in every other device too. The same applies to all the analytical results. Good ideas can automatically be made available to every device we want to try out.

▷ Chapter 8, "Graphs," presents many ideas for measuring the effectiveness of interactive devices, in particular whether the device *structure* has any dead-ends and other traps for the user—issues that are not checked with the basic code here.

Finally, we must not forget to actually check the device, by calling the function `checkDevice(device)` on the device we intend to run.

9.5.1 Drawing transition diagrams

The easiest way to draw a transition diagram is to exploit an existing tool to do it for us. There are several graph-drawing standards, and we just need use one to generate a basic description of the finite state machine, and then run a program to

do the work for us. When drawing is automated, if we change our device specification, the diagrams can be automatically updated—with speed and accuracy.

To draw diagrams, we will use Dot, an open source standard for drawing graphs. Dot specifications can be read into many programs, which then do the details of drawing, creating web pictures, PostScript, or whatever graphic formats we want.

If state *a* has a transition to state *b*, we write this in Dot by saying `a -> b`, and if we want to say that this transition is caused by some action, such as pressing a button (On), then we tell Dot the transition has a named label by writing `a -> b [label="On"]`.

Here is a basic description of the microwave oven (generated automatically from the last example in our framework, on p. 286) in Dot:

```
digraph "Microwave oven" { /* basic Dot description */
  0->0 [label="[Clock]"];          0->1 [label="[Quick defrost]"];
  0->2 [label="[Time]"];           0->0 [label="[Clear]"];
  0->0 [label="[Power]"];          1->0 [label="[Clock]"];
  1->1 [label="[Quick defrost]"];  1->2 [label="[Time]"];
  1->0 [label="[Clear]"];          1->1 [label="[Power]"];
  2->0 [label="[Clock]"];          2->1 [label="[Quick defrost]"];
  2->3 [label="[Time]"];           2->0 [label="[Clear]"];
  2->4 [label="[Power]"];          3->0 [label="[Clock]"];
  3->1 [label="[Quick defrost]"];  3->2 [label="[Time]"];
  3->0 [label="[Clear]"];          3->5 [label="[Power]"];
  4->0 [label="[Clock]"];          4->1 [label="[Quick defrost]"];
  4->3 [label="[Time]"];           4->0 [label="[Clear]"];
  4->4 [label="[Power]"];          5->0 [label="[Clock]"];
  5->1 [label="[Quick defrost]"];  5->2 [label="[Time]"];
  5->0 [label="[Clear]"];          5->5 [label="[Power]"];
}
```

The JavaScript to generate this would be easy—just a couple of `for` loops to run around the definition of `device.fsm`. However, if you run this through Dot, it is immediately obvious that there are lots of ways to improve it; we'll make the following improvements in our JavaScript:

- The states need names, and we could identify the start state specially.
- If a state has a transition to itself (that is, the action does nothing)—for instance like `0->0` and `4->4` above—we needn't show the arrow for the transition. This simplification will remove a lot of clutter from the drawing.
- If the same action transitions between two states in both directions—for instance like the pair `2->3 [label="[Time]"]` and `3->2 [label="[Time]"]` above—then we can draw a single arrow but with arrowheads on both ends, rather than two separate arrows.
- If several actions do the same transition, we draw them as a single arrow but with a label made out of all the actions. Thus `1->1 [label="[Quick defrost]"]` and `1->1 [label="[Power]"]` do the same thing and should be

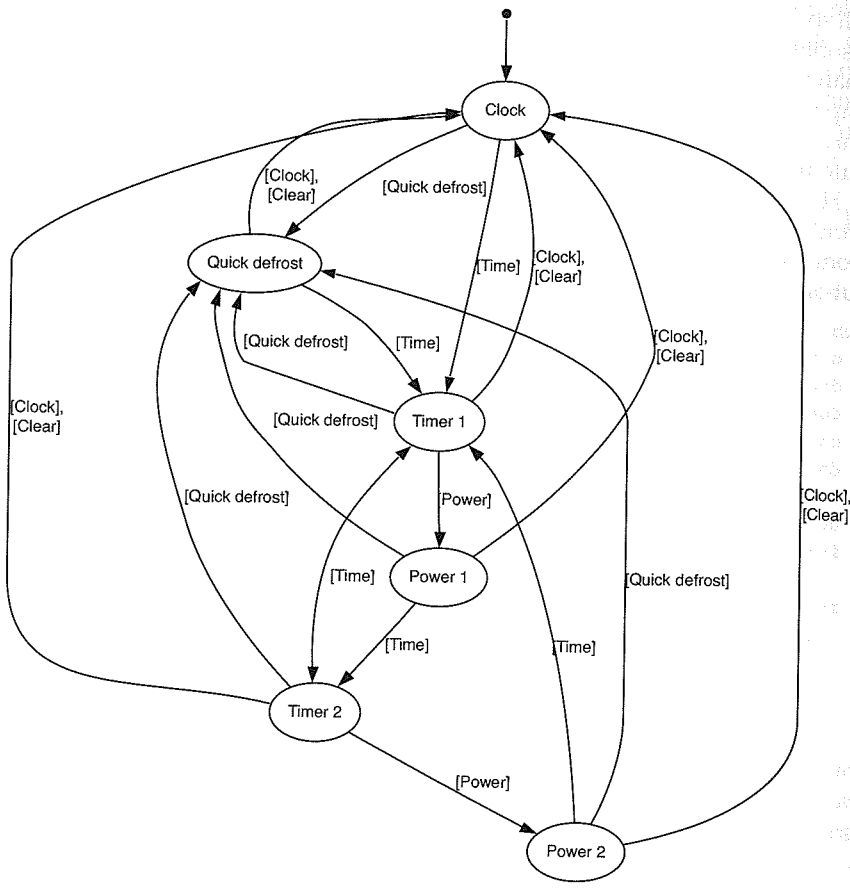
combined to make a single transition `1->1 [label="[Quick defrost], [Power]"]`.

All sorts of ideas will occur to you on further ways to improve the diagram. Dot is a sophisticated language and can take lots of hints about what you want. You can change the color and shape of the states, the arrowhead styles, and so on. What is important for us is that Dot draws a good enough diagram with no effort on our part. If we modify the finite state machine definition, the diagram will be updated automatically. Designers need easy and efficient tools, and this is one of them.

Here's the JavaScript code to achieve the improvements mentioned above. I won't describe the workings of the code in detail; if you copy it out (or copy it from mitpress.mit.edu/presson), it will work and generate HTML, which you then cut-and-paste to a Dot program to draw the graph.

```
function drawDot(d) // generate Dot code for any device d
{ document.write("digraph \""+d.modelType+"\" {\n");
  document.write("size=\"4,4\";\n");
  document.write("node [shape=ellipse,fontname=Helvetica,fontsize=10];\n")
  document.write("edge [fontname=Helvetica,fontsize=10];\n");
  document.write("start->"+d.startState+";\n");
  document.write("start [label=\"\",style=filled,height=.1,");
  document.write("       shape=circle,color=black];\n");
  for( var s = 0; s < d.fsm.length; s++ ) // state names
    document.write(s + " [label=\"" + d.stateNames[s] + "\"];\n"); // *
  for( var s = 0; s < d.fsm.length; s++ ) // single arrows
    for( var t = 0; t < d.fsm.length; t++ )
      if( t != s ) // ignore self arrows
      { var u = true;
        for( var b = 0; b < d.buttons.length; b++ )
        if( d.fsm[s][b] == t && d.fsm[t][b] != s ) // single arrows only
        { document.write(u? s + "->" + t + " [label=\"": ",\\n");
          u = false;
          document.write(d.buttons[b]);
        }
        if( !u ) document.write("\"];\n");
      }
  for( var s = 0; s < d.fsm.length; s++ ) // double arrows
    for( var t = s+1; t < d.fsm.length; t++ )
    { var u = true;
      for( var b = 0; b < d.buttons.length; b++ )
      if( d.fsm[s][b] == t && d.fsm[t][b] == s )
      { document.write(u? s + "->" + t + " [dir=both,label=\"": ",\\n");
        u = false;
        document.write(d.buttons[b]);
      }
      if( !u ) document.write("\"];\n");
    }
  document.write("}");
}
```

After the definition of this function, call `drawDot(device)` in the JavaScript. Here's the diagram it draws for the microwave oven—with no further touching up:



If you are keen, you can tell Dot to associate a URL with each state and then you can click on the transition diagram (in a web browser) and make the device work by going to the state you've clicked on. To do so, add the string

```
"URL=\"javascript:displayState(state = "+s+")\""
```

into the line marked * above—the URL runs the JavaScript to simulate the device.

This transition diagram is a detailed technical diagram that might be of interest to the designer but is too detailed to be of much use to a user. Instead, we can generate code for Dot (or whichever drawing program we are using) to make things more helpful for users.

The following Dot code draws a four-state transition diagram but uses photographs of the device taken when it was in each of the states, using Dot's parameter `shapefile` to use a picture file rather than a geometric shape. Here the device

is a Worcester Bosch Highflow-400 central heating system, and the transitions occur when the user presses the Select button. A diagram like figure 9.3 (next page) might be useful in a user manual.

Below, we've only shown the Dot code, not the JavaScript that could generate it. You would write some JavaScript that generated the Dot code for only part of the system (that is, a subgraph), rather than the whole system, for instance, based on lists of states that are needed to make each diagram to illustrate the user manual.

```
"Off" -> "Twice" -> "Once" -> "On" -> "Off";

"Off"   [shapefile="Off.jpg" label=""];
"Twice" [shapefile="Twice.jpg" label=""];
"Once"  [shapefile="Once.jpg" label=""];
"On"    [shapefile="On.jpg" label=""];
```

The easiest way to extend the framework is to add a line to the `device` specification like `stateNames: ["dim", "off", "on"]`, but giving the file names (or URLs) of images: `stateImages: ["dim.jpg", "off.jpg", "on.jpg"]`, for example.

Instead of photographs or pictures, you could generate text into a template representing the device's screen: you could have a function `generateScreen()` that puts together the appropriate string representing the screen for any state. The possibilities are endless!

It is worth emphasizing again (and again) that once you have set up your framework, you can draw, analyze, and animate interactive devices automatically with no further effort. And, the diagrams, analyses, or animations get modified automatically (and correctly) with no further effort when you revise the device specification.

▷ Instead of Dot, the JavaScript code can be modified to generate XML or specifications in other languages just as easily from our framework. See 7.7 (p. 219) for more on SCXML, the XML for statecharts.

9.5.2 Adding more features

You can keep on adding features to the framework indefinitely. However, the whole point of our approach is that the "way it works" features are clear and well defined, and any cosmetic ideas—like pictures—can be added in independently. Whatever fancy ideas we have to make the simulation more realistic, we should keep at its heart the nice, elegant finite state machine simulator we can understand, draw, and think about easily.

Very often we will want a state to encapsulate or do several things. Originally we had a single function, `press`, to do everything, which has the huge advantage of ensuring consistency, but now we want to be able to handle some states differently. We can either modify `press` so that it does different things in different states or add a field (like `stateNames`) to list functions for each state.

We may want it to draw a picture, or we might want it to handle something quite complex for the device like initiating a phone call.

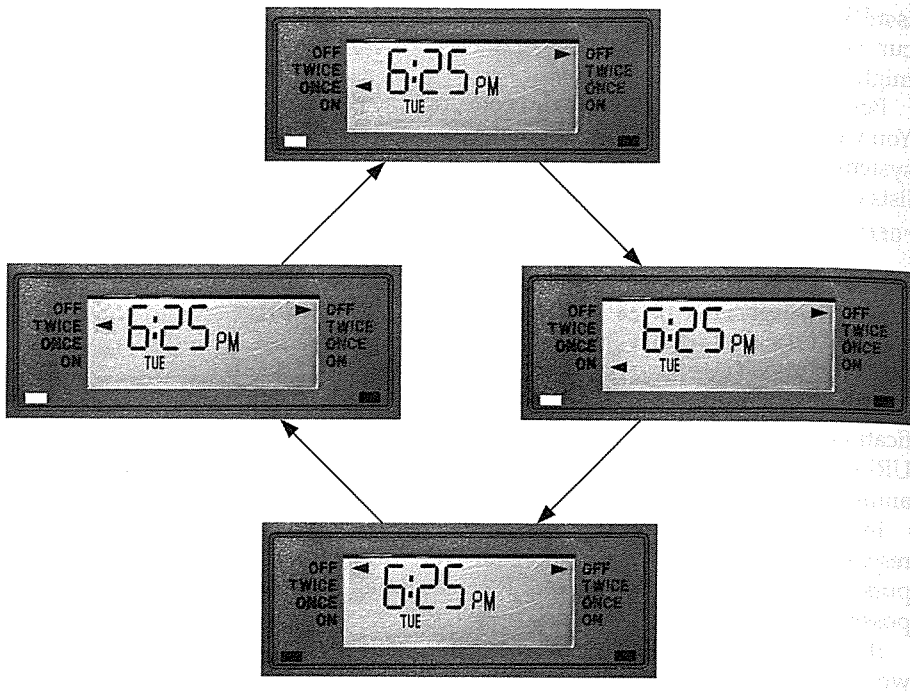


Figure 9.3: A state transition diagram, drawn automatically but using photographs of the device in each of the relevant states. This sort of diagram can be used directly in user manuals. The device here is a domestic central heating boiler, and the states are hot water off, on twice a day, on once a day, or on continuously. Note that the photographs show both the state indicator and the heater-on light.

One approach to handle state encapsulation is to arrange each state to call its own JavaScript function. The function can then do what it likes *in* that state. In many cases, simple modifications to press will seem easiest:

```
function press(buttonNumber)
{ state = device.fsm[device.state][buttonNumber];
  displayState(state);
  switch( state )
  { case 1:  dostate1(); break; // do something special in state 1
    case 5:  dostate5(); break; // do something special in state 5
    default: // in this design, no other states do anything special
             // but the default lets other states work consistently
             // if we added code here for them
             ...
  }
}
```

Before long, this approach of overriding the definition of each state will get too complex, especially if you are tempted to write all the code directly into `press` itself. Much better is to improve the way each state is defined centrally in the `device` definition. One way to do it is to have a (potentially) different `press` function for every state:

```
var device = {
 notes: "Simple lightbulb, with dim mode",
 ...
 stateNames: ["dim", "off", "on"],
 stateFunctions: [stateLit, stateOff, stateLit],
 ...
};
```

and `press` does it's basic work then dispatches to the appropriate state function, `stateLit`, `stateOff`, or whatever:

```
function press(buttonNumber)
{ device.state = device.fsm[device.state][buttonNumber];
  displayState(device);
  device.stateFunctions[device.state](); // do special action
}
```

It's important that everything can be done from the *same* core representation of devices. The JavaScript framework we've designed will work well even if we change, perhaps radically, the definition of the device we are simulating or analyzing. The interaction, as done here in Javascript, the drawings and all of the analysis and manual writing is done directly from the same definition. We can easily change the design (by editing the `device` specification) and then rerun the simulation or the analyses. If the ideas sketched out in this chapter were used in a real design process, it would permit concurrent engineering—doing lots of parts of the design at the same time. Some designers could work on getting the interaction right, some could work on the analysis, some could work on implementation (like writing better code than we did just now), and so on.

9.6 Finding shortest paths

Designers want to know whether users can do what they want to, and if they can, how hard it is. Let's use the framework to answer these design and usability questions.

The data structure `device.fsm` describes the complete graph of the user interface design. Here is the data for the microwave oven again:

	Buttons →				
States	0	1	2	0	0
↓	0	1	2	0	1
	0	1	3	0	4
	0	1	2	0	5
	0	1	3	0	4
	0	1	2	0	5

This table, which is stored in `device.fsm`, tells us the next state to go to. Given a button press (column) in any state (row), the entry tells us the next state to be in. For example, the last line (which is state number 5) tells us that button 0 (which is called Clock and is the first column) would take the device to state 0—this is the number in the first column of that row.

To work out how many button presses it takes to get from any state to any other state, we need a new table that has rows *and* columns for states; each entry will tell us how many presses it takes to get from one state to the other.

By reading the state-button table above, we can work out the beginning of this new table, which we will call `matrix`. If a button pressed in state i causes a transition to state j, then we want to have a 1 in entry `matrix[i][j]` to indicate that we know we can get from state i to state j in one step. Also, we know as a special case we can get from state i to state i in no steps at all; so every entry `matrix[i][i]` should be zero. Any other entry in the matrix we'll set to infinity because we think (so far) that there is no way to get between these states.

Note that `matrix` has less information in it than `device.fsm`, since we have lost track of buttons: the new matrix only says what next states are possible from a given state, whereas `device.fsm` *also* says which button should be pressed to do it.

All the code that follows is written inside a function, `shortestPaths(d)`, and then `shortestPaths(device)` will work out the shortest paths for that device. Throughout the following code, `d` will be the device we're working on.

Inside the function, we first need a few lines to create a blank matrix of the right size:

```
var fsm = d.fsm;     // convenient abbreviation
var n = fsm.length; // number of states

var matrix = new Array(n);
for( var i = 0; i < n; i++ ) // create the 2 dimensional table
  matrix[i] = new Array(n);
```

For a finite state machine of n states, if any shortest route takes more than n steps it must be impossible: $n-1$ steps is always sufficient to visit every state in the machine (if it's possible, that is), so we can define "infinity" for our purposes to be any number n or larger. The value $n+1$ will do nicely.

```
var b = fsm[0].length; // number of buttons
var infinity = n+1;    // no route can take this many steps!

for( var i = 0; i < n; i++ ) // build one-step matrix
{ for( var j = 0; j < n; j++ )
    matrix[i][j] = infinity;
  for( var j = 0; j < b; j++ )
    matrix[i][fsm[i][j]] = 1;
  matrix[i][i] = 0; // zero steps to get from state i to itself
}
```

The tricky line in the inner `for` loop, `matrix[i][fsm[i][j]] = 1`, says, "we're in state `i`, and we know button `j` gets us to state `fsm[i][j]`, and therefore the entry in the matrix for state `i` to state `fsm[i][j]` should be 1."

Here is what the cost matrix looks like after running that initialization code:

	To state ...					
From state ...	0	1	2	3	4	5
0: Clock	0	1	1	∞	∞	∞
1: Quick defrost	1	0	1	∞	∞	∞
2: Timer 1	1	1	0	1	1	∞
3: Timer 2	1	1	1	0	∞	1
4: Power 1	1	1	∞	1	0	∞
5: Power 2	1	1	1	∞	∞	0

I printed this table directly from the actual values in `matrix`, by using a couple of `for` loops:

```
for( var i = 0; i < n; i++ ) // display matrix
{ for( var j = 0; j < n; j++ )
    if( matrix[i][j] == infinity )
      document.write("&infin; ");
    else
      document.write(matrix[i][j]+" ");
  document.write("<br>");
}
```

It would not be hard to use an HTML `<table>` to lay the data out nicely.

The symbol ∞ in the table (`&infin;` in HTML) means for the time being we don't know a state transition is possible—that is, it might take an infinite number of steps—to get from one state to another. More precisely, the matrix entries mean: it can take 0 steps to get from one state to another (if they are the same state); it can take 1 step to get from one state to another (if there is a single button press that does it directly); or it can take more, which we've represented as ∞.

In short, for any pair of states, say i and j, the matrix tells us whether we can get from one to the other (in zero or more steps). We now use a standard algorithm, the Floyd-Warshall algorithm, that considers every state k and determines whether we can get from i to j via the state k more efficiently. If we can go $i \to k \to j$ more efficiently than going directly from $i \to j$, we record it in the matrix. Thus we expect those ∞ values to reduce down to smaller values. The direct cost is `matrix[i][j]` and the indirect cost, via k, is `matrix[i][k]+matrix[k][j]`. Whenever the indirect cost is better, we can improve the recorded cost. Here's how to do it:

```
for( var k = 0; k < n; k++ )
  for( var i = 0; i < n; i++ )
    for( var j = 0; j < n; j++ )
    // replace cost of ij with best of ij or ikj routes
    { var viak = matrix[i][k] + matrix[k][j];
      if( viak < matrix[i][j] )
        matrix[i][j] = viak;
    }
```

At the end of running these three nested for-loops, the program has tried every way of getting from anywhere to anywhere. The inner two loops find the best way of getting from state i to j via any intermediate state k, but the outer loop ensures that we have tried all intermediate routes via 1 to $k-1$ first. Thus when the outer loop is finished, we know the best ways of getting between any two states via any state $1 \ldots k$, which covers all possible cases. This is sufficient to know all of the best routes through the finite state machine.

Here is the result of running this code on our cost matrix:

	To state ...					
	0	1	1	2	2	3
	1	0	1	2	2	3
From	1	1	0	1	1	2
state	1	1	1	0	2	1
	1	1	2	1	0	2
	1	1	1	2	2	0

We can write the same information as a mathematical matrix (just put it between round brackets), which is highly suggestive of things you might want to do if you know matrix algebra; however, to go there is beyond the scope and immediate interest of this book.

▷ For more on matrices, see box 11.1, "Matrices and Markov models" (p. 382).

Note that there are no ∞ (infinity) symbols in this shortest path table; this means that (for this device) it is possible to get from any state to any other in a finite number of steps—thus this device is strongly connected. In particular, each entry in the matrix gives the least number of steps it will take between two states (specifically, `matrix[i][j]` is the cheapest cost of getting from state i to state j by any means).

It goes without saying that our knowing the least number of steps to do anything does not stop a user from getting lost or perhaps deliberately taking a longer route. But we can be certain a user cannot do better than these figures.

In technical terms, the costs are a lower bound on the user interface complexity, as measured by counting button presses. That is one reason why the costs are so important; we have a conservative baseline to understand the user's performance—or lack of it—regardless of how good or sophisticated the user is they cannot do better than these figures. If the costs turned out to be bad (or, more likely, some of the costs were bad) there is nothing a user can do and nothing training can do; we either have to live with a difficult-to-use feature (some things we may *want* to be difficult), or we have to fix the design.

If we multiply the counts in the cost matrix by 0.1 second, we get an estimate of the minimum *time* a fast user would take—and of course we could do some experiments and get a more accurate figure than our guess of 0.1 second; then the timings become a much more useful design measure.

9.6.1 Using shortest paths

Once we have worked out the shortest paths matrix, we know the most efficient cost of getting from anywhere to anywhere. For instance, to get from Clock (state 0)

to Power 2 (state 5) takes 3 button presses—that's the number 3 in the top right, at the end of the first line in the matrix above. Here's the best route between these two states:

1. Press [Time] to get from Clock to Timer 1
2. Press [Time] to get from Timer 1 to Timer 2
3. Press [Power] to get from Timer 2 to Power 2

In fact, a designer might be interested in *all* the hardest operations for a device; for this device there are two, both taking at least 3 steps (the user will takes longer if they make a mistake; the analysis shows they cannot take *less* than 3 steps). The other worst case is getting from Quick defrost to Power 2. If the worst cases take ∞ steps, then very likely there is something wrong with the design: some things the device appears designed for are impossible to do.

Some devices—like fire extinguishers—may be designed to be used only once, and then the designer will expect an infinite cost for getting back from the state "extinguisher used" to "extinguisher not used." But even then, it would help the designer to have these states and problems pointed out automatically by the framework. For instance, why not make the fire extinguisher refillable?

To find the best route, the correct sequence of button presses, not just the total cost to the user of following the best route, requires a few extensions to the program.

We need to keep another table `via[i][j]` to record the first step along the best route $i \to j$. Every time we update the cost of the route, we've found a better step. Here are the details:

- In the original initialization of `matrix` we add an initial value to `via`:

```
for( var j = 0; j < b; j++ )
{ matrix[i][fsm[i][j]] = 1;
  via[i][fsm[i][j]] = j; // to get from i to fsm[i][j], press button j
}
```

- In the easy case of going from a state to itself, the first thing to do is to go there.

```
matrix[i][i] = 0;
via[i][i] = i;
```

- In the code that updates `matrix`, if it is better to get from i to j via k, we replace the first step of $i \to j$ with the first step of $i \to k \to j$, which is already in `via[i][k]`:

```
var viak = matrix[i][k] + matrix[k][j];
if( viak < matrix[i][j] )
{ matrix[i][j] = viak;
  via[i][j] = via[i][k];
}
```

- Finally, after finding all the shortest paths, to get the best route between any two states $i \rightarrow j$, the following code will print out the route:

```
var start = i;
var limit = 0;
while( start != j )
{ var nextState = device.fsm[start][via[start][j]];
  document.write("Press "+device.buttons[via[start][j]]
      +" to get to "+device.stateNames[nextState]+"<br>");
  start = nextState;
  if( limit++ > n ) { document.write("?"); break; }
}
```

  This code uses the variable `limit` to ensure that we don't try to solve an impossible task—in case it is called when the route has the impossible length ∞. Recall that no route in a finite state machine need take longer than the number of states; otherwise, it is going around in circles (it must visit some state twice) and therefore can't be a shortest route to anywhere.

The code in the last step above prints out the sequence of user actions a user needs to do to get a device from state i to state j. If we put that code inside a couple of loops to try all values of i and j we can find out how often each button is used. We could then design the device so that the most popular button (or buttons) are large and in the middle. Or we might discover that the popularity of buttons is surprising (some buttons may never be used on shortest paths, for instance), and then we'd want to look closer at the device design. Here's one way to do it:

```
var bcount = new Array(b); // an array with one element per button
for( var i = 0; i < b; i++ )
  bcount[i] = 0;
for( var i = 0; i < n; i++ )
 for( var j = 0; j < n; j++ )
  { var limit = 0, start = i;
    while( start != j )
    { var nextState = device.fsm[start][via[start][j]];
      bcount[via[start][j]]++; // count how often buttons are used
      start = nextState;
      if( limit++ > n ) break;
    }
  }
for( var i = 0; i < b; i++ )
  document.write(device.buttons[i]+" rated "+bcount[i]+"<br>");
```

9.6.2 Characteristic path length

Now that we have worked out shortest paths, the characteristic path length is just a matter of finding their average. Here's how, printing the results in HTML:

```
var sum = 0, worst = 0;
for( var i = 0; i < n; i++ )
  for( var j = 0; j < n; j++ )
  { sum = sum+matrix[i][j];
    if( matrix[i][j] > worst )
      worst = matrix[i][j];
  }
document.write("Characteristic path length "+(sum/(n*n))+"<br>");
document.write("Worst case path length "+worst);
```

For our microwave oven example, the characteristic path length is 1.22, and the worst case is 3.

▷ The characteristic path length was introduced in section 8.9 (p. 264).

Since the microwave oven happens to have as many buttons as states, the characteristic path length could have been as low as 1, with a worst case of 1 too. Perhaps the designer should take a closer look to see whether the extra effort for the user is justified by greater clarity or other factors.

9.6.3 Eccentricity

The eccentricity of a state measures the cost of the *worst possible* thing you could try in that state, that is, getting to the most distant state you could get to from that state.

In graph theory terms, the eccentricity of a vertex is the maximum distance it has to any other vertex. The diameter and radius of a graph are then defined as the largest and smallest, respectively, of the vertex eccentricities. Here's how to find out all the information:

```
var radius = infinity, diameter = 0;
for( var i = 0; i < n; i++ )
{ var eccentricity = 0;
  for( var j = 0; j < n; j++ )
    if( matrix[i][j] > eccentricity )
      eccentricity = matrix[i][j];
  document.write("<b>"+d.stateNames[i]
      +"</b> eccentricity is "+eccentricity+"<br>");
  if( eccentricity > diameter )
    diameter = eccentricity;
  if( eccentricity < radius )
    radius = eccentricity;
}
document.write("Diameter (worst eccentricity) = "+diameter+"<br>");
document.write("  Radius (least eccentricity) = "+radius+"<br>");
```

The designer will be especially interested in any states that have a very high or a very low eccentricity; the extreme values might indicate an oversight, or, of course, they may be intentional—sometimes a designer *wants* to make certain things hard to do.

For the microwave oven, it turns out that the diameter is 3, and the most eccentric states are Clock and Quick defrost, and in both cases the worst thing to try to do is to change to the state Power 2.

Clock is a state used for setting the clock, so you'd expect to have to get out of clock setting, start power setting, then set Power 2, so a shortest path to Power 2 of 3 doesn't sound unreasonable. If a user is doing a Quick defrost, they are not very likely to want to ramp up the power to its highest—it's easy to burn stuff on the outside but leave the middle frozen. So we have a no reason to worry about the two largest eccentricities.

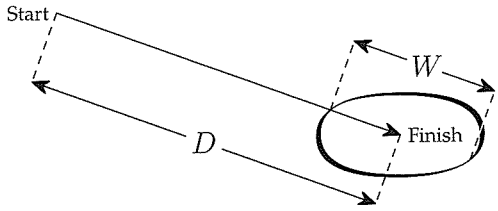
9.6.4 Alternative and more realistic costs

We've described shortest paths as a way of finding the most efficient route, or path, from any state to any other. By "efficient" we meant using the least number of button presses, or user actions of some sort if they aren't button presses. It's time to find out whether we can be more accurate—and whether being more accurate helps us design better systems.

If buttons are close to each other, a human can operate them with one hand. If the buttons are several meters apart, then they will be much harder to use; in fact, if they need pressing simultaneously, two users might be needed or a user might have to put their feet on a bar and touch two distant buttons to guarantee their limbs are kept out the way of the dangerous machinery.

To be more realistic, we should include more information in the design framework about the device, such as button and indicator sizes and positions. Secondly, we should get some data from real people about how they perform with different representations of devices.

Instead of measuring how long users take, which requires an experiment, it is possible to estimate how long they take from theory. For each action on a push-button device, the user has to do three things: decide which button to press, move their fingers to get to the button, and then press the button. The Hick-Hyman law says roughly how long a user will take to decide which button to press: the time is $a + b\log(N+1)$, where N is the number of choices the user has to decide between—here, it will be the number of buttons. The Fitts law says how quickly the user can move to hit the button, depending on the distance to move and how large the target button is: the time for a movement is $c + d\log(D/W+1)$, where D is the distance to move and W the width of the button measured in the direction the finger is moving. If the device has tiny buttons, then you must also take account of the size of the user's fingers.



Box 9.1 Facilities layout Most problems have appeared under different but more-or-less equivalent guises before. Improving the design of interactive devices by looking at shortest paths and user data is "the same" as the facilities layout problem. You have some facilities, like workstations in a factory, and you want to minimize the cost of people carrying stuff around the factory.

How people can move stuff around the factory is a graph. You collect data on where people go and how often, you work out the shortest paths, and you then move the facilities around to reduce the overall costs of moving things. This problem is very well studied; it is estimated that about 10% of a nation's industrial effort goes not into just moving stuff around, but into rearranging facilities or building new ones to move stuff around!

It's the same problem as deciding where to put interesting states in a state machine, or deciding where to put interesting pages in a web site, if you know shortest paths and have some usage data. It turns out that finding a solution to the problem is NP-hard, which is just a fancy way of saying nobody knows a good way of solving the problem.

It's nice to know that re-designing an optimal web site, or re-designing an optimal device, based on user data is NP-hard, for this is effectively saying it's a very hard job, however you do it. On the other hand, since the problem is so hard, if you do not use a systematic way of re-design—using one of the standard algorithms for solving the problem—then it is very unlikely to be much of an improvement.

The constants a, b, c, d are usually found from experiment, and depend on the users, their age, fitness, posture, and so on.* The total time to do anything is the sum of these two laws for each user action required. Whatever the exact values, we can immediately see that there are some generic things that can be done to speed things up. We could:

- Make buttons larger, so W is larger.
- Make buttons closer, so D is smaller.
- Reorganize buttons into a better layout, to reduce average distances.

  ▷ See box 9.2 (p. 310) for an example of this idea.

- We could reduce N, for instance by illuminating only active buttons, or we could use a dynamic approach to showing buttons, say using a touch screen that reconfigures itself. However, if the screen changes, although N is reduced, we would find that the "constants" increase, as the user has a harder job to decide which button to press—they cannot learn a fixed layout.

  ▷ Illuminating active buttons is discussed in section 11.3.1 (p. 383).

- On devices with touch-sensitive screens that show buttons, we can change the size of buttons depending on what state the device is in. For example, making

* Often these laws are given differently, using logarithms to base 2. Since the constants are experimentally determined it does not matter what logs are used.

the buttons that are used at the beginning of any sequence of user actions larger will help the user notice them (as if they are lit up).

- We could change the constants by "modifying" the user rather than the device: we could train the user to be faster, or we could select users who were good at our tasks.
- We don't always design by minimizing costs; we could try to increase costs instead. If the user is using a gambling machine, we may want the users to spend as long as possible losing money. Or, although we know that hitting the big red [EMERGENCY OFF] button switches a device off immediately, it may not be the best way of switching it off normally; we should make its cost higher, for instance putting it under a protective cover.

Programming Fitts Law times

The design framework could be extended to allow designers to edit the look and feel of a device in a drawing program. As the graphic designer moves and changes the sizes of buttons, they would get continuous feedback from the design framework analyzing its performance. Such ideas would take us beyond the scope of this book, but we'll explain how to get the framework to use Fitts Law (or any other formula) to estimate how long a user takes to do tasks. To do this, we need to extend the framework with details about where buttons are and what sizes they are. For simplicity, we'll assume all buttons are circles of radius `r` and they are on a flat surface.

If `buttonLocation` is an array of (x, y) coordinates, the following function called `Fitts`, converts the distances between buttons into the time, in seconds, Fitts Law says it will take a user. Fitts Law doesn't need to know the units of measurement, but the radius and (x, y) coordinates must be given in the same units. (Which just shows Fitts Law is an approximation: if the distances are huge, then Fitts Law will underestimate the times.)

The following code uses constants typical of typewriter keyboard speeds:

```
function Fitts(b, c) // time taken moving between buttons
{ var dx = device.buttonLocation[b].x-device.buttonLocation[c].x;
  var dy = device.buttonLocation[b].y-device.buttonLocation[c].y;
  var d = Math.sqrt(dx*dx+dy*dy); // distance to move
  if( d < r || b == c ) return 0.14; // a double tap
  return 0.16+0.07*Math.log(d/device.r+1); // index finger
}
```

The code shows Fitts Law numbers estimated for index-finger movement; use `0.18+0.09*Math.log(d/device.r+1)` for slower thumb movement.

If we are not sure what constants to use the function `Fitts` can be modified to return `Math.log(d/device.r+1)/Math.log(2)`, and it will then be returning the index of difficulty (IOD), which is a number representing how hard something is to do, without regard for the exact timing.* Then the algorithm will then return not

* The index of difficulty is properly measured in bits, and hence needs $\log_2$.

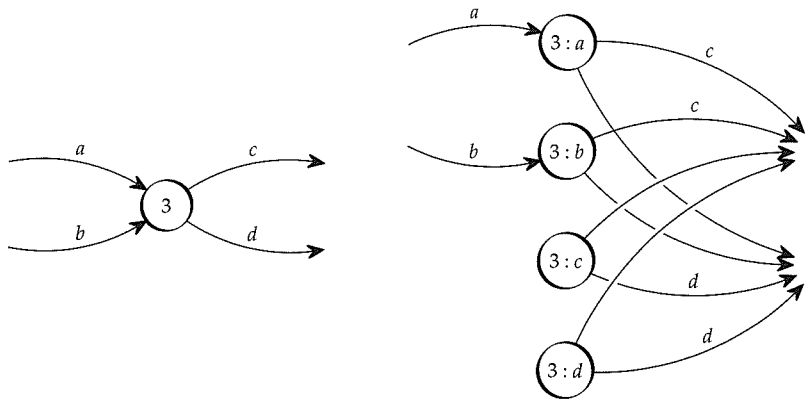


Figure 9.4: How a single-state (state 3, left) with two transitions to it (a and b) would be replaced with four states (right), each "remembering" the original state's different ways of getting to it. Now the transitions "know" how the user's finger has moved: for example, transition c from state $3 : a$ required the user to move their finger from button a to button c. In this example, two of the new states ($3 : b, 3 : c$) are unnecessary as there is no way to enter them. (For clarity, the righthand diagram does not show all transition labels, and states $3 : a$ and $3 : b$ will also have more transitions to them than are shown.)

best times, but best measurements of the index of difficulty, depending on button layout and the device interaction programming.

We can't reuse the shortest paths algorithm directly because the time it takes a user to get from one state to another depends on what the user was doing *before* they got to the first state. How long it takes to get from state to state depends on the distance the user's finger moves. Pressing button [A], say, got the device into state 2, then the user moved their finger from [A] to press [B] to get the device to the next state. The shortest path algorithm assumes each action (arc) has a fixed cost—but to use Fitts law we need to know which button the user pressed to get the device to the first state: for instance, the physical distance [A] to [B], if [A] got us to state 2, or the physical distance [C] to [B], if [C] got us to state 2. Somehow the states need to know which button the user pressed to get there.

Suppose the device has got B buttons and N states. We create a new "device" with B states for each of the original device's states. Give the new states names like $s : b$, so that in the new device the state $s : b_0$ represents entering the original device's state s by pressing button b_0—thus we have states that record how the user got to them. If on the old device pressing button b_1 went from state s to state t, then on the new device state $s : b_0$ will go to state $t : b_1$. This transition will take time according to the Fitts Law to move the finger from the location of b_0 to the location of b_1, and we now have enough details to work the timings out.

To program this in JavaScript, we give the state we've called $s : b$ the unique number $sB + b$. The code starts by creating a table of all weights for all the new state transitions, first initializing them to ∞:

```
var w = new Array(NB);
for( var i = 0; i < NB; i++ )
{ w[i] = new Array(NB);
  for( var j = 0; j < NB; j++ )
    w[i][j] = infinity; // default is no transition
}
```

Then we construct the new "device":

```
for( var i = 0; i < N; i++ )
  for( var b = 0; b < B; b++ )
  { var u = device.fsm[i][b]; // b takes us from i to u=fsm[i][b]
    for( var c = 0; c < B; c++ )
      // we've just pressed b, now we press c
      // pressing c takes us from u to fsm[u][c]
      w[u*B+b][device.fsm[u][c]*B+c] = Fitts(b, c);
  }
```

We can now use the familiar Floyd-Warshall algorithm to find fastest paths. This part of the code works exactly as before—though for simplicity here we are not recording the actual paths taken.

```
for( var k = 0; k < NB; k++ )
  for( var i = 0; i < NB; i++ )
    for( var j = 0; j < NB; j++ )
      if( w[i][k]+w[k][j] < w[i][j] )
        w[i][j] = w[i][k]+w[k][j];
```

▷ The Floyd-Warshall algorithm was introduced in section 9.6 (p. 299).

We've found fastest paths but paths in the *new* "device." We need to translate back to the original device: that is, all states $s : b$ need to be called s whatever button got there. We create a new table v for the results:

```
var v = new Array(N);
for( var i = 0; i < N; i++ )
{ v[i] = new Array(N);
  for( var j = 0; j < N; j++ )
    v[i][j] = infinity;
}
for( j = 0; j < NB; j++ )
  for( k = 0; k < NB; k++ )
    if( v[Math.floor(j/B)][Math.floor(k/B)] > w[j][k] )
      v[Math.floor(j/B)][Math.floor(k/B)] = w[j][k];
```

We now have a table with entries `v[i][j]` that gives the best time in seconds it takes a user to get from state `i` to state `j`. Note that the timing is only useful for pushbutton devices; common devices like cameras, with knobs and sliders, will require slightly different treatment than the "raw" Fitts Law. Devices like the

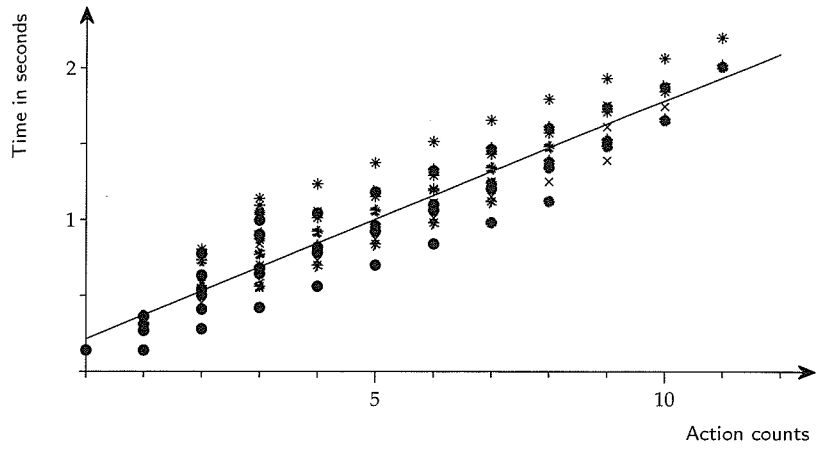


Figure 9.5: A plot of Fitts Law timings (vertically) against button press or action counts (horizontally) for all possible state changes on the PVR used in section 10.3. Button counts are closely proportional to user times.

Points close enough to overlap are rotated, like × ✱ ✱ ● ● ··· ● —the more points at the same coordinates, the darker the blob. The graph also shows a best-fit line $0.21+0.16x$.

radio—section 3.10 (p. 80)—require the user to spend a significant time interpreting the display, and Fitts Law does not consider that either. If your device takes many button presses to change states, Fitts Law will become inaccurate as it increasingly under-estimates times, ignoring the user checking the device feedback and thinking.

In figure 9.5 (this page), I've used the program code and shown Fitts Law timings plotted against button counts to do the same tasks. As can be seen, the timings and counts are closely related (at least for this device), so button press counts—which the basic shortest paths algorithm gives directly—will be sufficiently accurate for most interaction programming design issues (in fact, as accurate as you could expect from either experiments or from Fitts Law alone). However, if you wish to design button layouts, the plot also shows us that user timings can vary by a factor of two or more simply because of the time taken for finger movement; for button layout, then, Fitts Law can be very helpful.

Familiarity

Another useful cost we can explore is familiarity. There are many ways to represent familiarity, but suppose we keep a matrix inside the interactive device, initially with the one-step costs: 1 if a transition is possible, 0 if it is not possible. Now, every time the user does something, the corresponding transition "cost" is

Box 9.2 Button usage Finding the absolutely best layout of buttons is hard, especially all possible button positions are considered in all possible layouts (rectangles, circles, hexagons ...). It may be just as useful to find out which are the most used buttons. A model of the JVC HRD580EK PVR suggests that buttons are used very differently:

Button press or action	Relative use on shortest paths		Button press or action	Relative use on shortest paths	
[Record]	67.36%	(1876)	[Operate]	2.87%	(80)
[Stop/Eject]	15.44%	(430)	[Forward]	1.94%	(54)
[Pause]	8.58%	(239)	[Rewind]	1.94%	(54)
[Play]	5.03%	(140)	insert tape	1.87%	(52)

The [Record] seems to be used excessively: it has too many uses in this design—surely, there aren't that many ways of recording? Thus, some of its uses might be reconsidered and handed over to another, possibly new, button. Or, if we do not change the meaning of the button (which would change the figures in the table), it should be physically positioned in the centre of the buttons, to reduce hand or finger movement times.

Inserting a tape is used 1.87% of the time, which also seems high—surely there is only one thing inserting a tape should do? As it happens, on the JVC PVR, inserting a tape *also* switches the PVR on if it is off, so inserting a tape is faster than switching on, then inserting a tape. Again, analysis questions a user action possibly having too many uses.

▷ The JVC HRD580EK PVR is used in section 10.3 (p. 330). The path length data is illustrated in a bar chart in figure 11.4 (p. 381).

incremented. The matrix counts how often a user does something; we can assume counts indicate how familiar the user is with each action. Now maximum cost paths indicate paths that the user would tend to take themselves. We can find solutions that are not fastest but—we hope—are most familiar. Alternatively, we might want to train the user to be proficient in all ways of using a device; then we would want to recommend they learn and do things they are *not* familiar with.

If the user was a pilot, nurse, or fireman, then they might be required to be familiar with *every* operation of the device; the ideas here then allow the framework to tell them what they have missed or not yet done enough of. Perhaps their professional license requires them to do a certain sort of procedure every year or so; if so, we can check, and tell them what they have missed.

These ideas give advice depending on what users do. Instead, we could fill the cost matrix with how experts think the device should be used. In this case, the shortest paths will automatically give expert advice.

▷ More uses of costs to use in products (rather than just for the designer to analyze) are discussed in section 10.7 (p. 351), particularly 10.7.5 (p. 358), and 11.5 (p. 392).

9.6.5 Task/action mapping

The shortest paths program finds paths between states. Unfortunately, the user is not interested in states as such; states are abstract things inside the device. The device can do what it likes, as it were, but the user is interested in achieving things in the world. More specifically, the user has goals, and they want to undertake tasks to achieve their goals.

Of course, this is a gross simplification; a user's goal might be to have fun and there may be no specific task associated with that experience. But for a great many cases, the user will have goals—such as getting a device they are using to do something important for them—and their task then amounts to doing the actions that get the device to actually be in the right state to have done whatever they wanted.

The general question is to find the best way of getting from the set of states the device may be in (they need not know specifically) to the set of states that achieve the goal the user wants to achieve (they may not know these specifically either). The user's goal may further require that certain intermediate states are visited. For example, a user might want to get a CD player from any state (however it was last left) to playing a track on a new CD. What should they do?

The standard term for doing this is called solving the task/action mapping problem. Given the task (or the goal that ends the task), how do we map that into the actions the user should do?

What the user should do is spelled out by the program; what the program should do and how it works is our present interest. Clearly, there are as many solutions to the programming problem as there are different sorts of user problems that we might want to handle. The best way of starting is to introduce indicators to the device framework, which indicate things the user is particularly interested in. Indicators, then, represent sets of states. For example, a device can be off in several ways—with a CD in it or no CD in it, say—but the user knows it is off. We can then have an indicator off, to represent the set of (in this case) two off states.

Users then specify their goals in terms of changes to indicator settings. The previous task we described is simply to go from (any state) indicating off to (any state, preferably the nearest) indicating play.

▷ Details of how to do this sort of thing are given in section 10.7 (p. 351), where we will examine how to solve problems for microwave oven tasks and video recorder tasks. Indicators are introduced in section 8.1.3 (p. 232); see also section 10.3.2 (p. 334).

Once we have some program code to solve user problems, it can be used in four quite different ways:

- For the user, as we already know, it can solve *particular* problems.
- For the designer, it can solve *every* user problem and then generate statistics, such as the average and worst-case costs for the user.
- For the technical author (or the designer), who writes user manuals and perhaps interactive help for users, it allows them to review all sensible solutions to problems so they can be written up accurately.

▷ User manuals can be generated with automatic help; see section 11.5 (p. 392).

- The fourth point of view is the most interesting: how hard is it for us to work out how to write a program to solve the user's task/action mapping problems? The harder it is for us to solve the problem, then certainly the harder it will be for users (who know less about the device than we do). We may find that our programs need hints; if so, so will the users.

  In some cases, we may find that the task/action mapping problem is very hard to solve (it might take exponential time to solve, or worse); then we really need to redesign the device—because if a problem is theoretically that hard, the user *must* find it hard too.

  ▷ The key importance of the designer getting insight into the difficulty (or ease) of the user solving problems is part of the computer science analogy, which we introduced in section 5.7 (p. 157).

9.7 Professional programming

Our style of defining a device is very simple, so there are inevitably more "professional" ways of doing it. This chapter is not about how to program. I want you to know, rather, what is possible and easy to do—and the automatic benefits of being simple and systematic. If we had used a more sophisticated way of programming in this book, we could certainly handle much larger device specifications more easily and more reliably, but then too much text would have been taken up with explaining sophisticated programming ideas; we would have lost the simplicity at the core of the approach.

Here are some suggestions for improving the framework, depending on what you want to do:

- With lots of states, keeping track of the differences becomes impractical, but to fill in most array fields in `device` you have to be aware which state is which. Instead, a single state field should be an array of objects. Thus each state says what its name is, and what its user manual text is. Instead of writing,

```
var device = {
 ...
 stateNames: ["dim", "off", "on"],
 fsm: [[1, 2, 0], [1, 2, 0], [1, 2, 0]],
 ...
 manual: ["dim", "dark", "bright"],
 ...
};
```

  you would write more like this (shown on the next page):

```
var device = {
 ...
 states:
 [{name: "dim", fsm: [1, 2, 0], manual: "dim"},
  {name: "off", fsm: [1, 2, 0], manual: "dark"},
  {name: "on",  fsm: [1, 2, 0], manual: "bright"}],
 ...
};
```

This way of coding brings everything in each state much close together and therefore helps you get the correspondences right. If you know JavaScript, you can use object constructors rather than repeatedly writing out `name`, `fsm`, and `manual`.

- The finite state machine field, `fsm`, is defined using state numbers, and it is sometimes too easy to mix up state numbers. Instead, every reference to a state should be to its name. If you want to do this, you will probably want to preprocess the device to build an internal specification that uses numbers (because they are much more efficient when the device is running).

  ▷ Section 9.5 (p. 288) gives other ideas for preprocessing the device specification to improve its quality.

- Strings are readable and make a device specification very clear, which is why we're using them a lot in this chapter. But, what happens if you mistype a string? The device would misbehave. If you are using a typed language like Java, there are many better approaches. You could have a state constructor, say,

  ```
  State sOff = new state("off", "dark");
  ```

  The advantage is that Java will only let you use `sOff` where you need a state, and you can only use states in those places. You could not get a button and a state confused. (Here you'd need to add actions to each state separately, say, `sOff.addAction(button, nextstate)`.)

- If you spend another half hour or so programming, you will be able to generate code to work on hardware (or Java, or whatever) from the JavaScript framework we're using, and you will then be able to get the real device to work straight from the framework.

- It is easy to add special-purpose features to the framework, but we won't dwell on them in this book. For example, the framework can be extended to be a pushdown automaton in which each user action stores the last state in a stack (in JavaScript, it would be an array, using the methods `push` and `pop`). The Undo button is then programmed to pop the stack and restore the last state.

  ▷ Section 10.8 (p. 362) suggests how to implement consistent user interfaces by using program code to define interaction features.

- Our framework has finite state machines as explicit data structures. It's tedious writing down finite state machines like this—every bit as tedious as drawing state transition diagrams. Just as statecharts improve the visualization of state machines, there are many programming language approaches (including SCXML, the XML statechart notation) that improve the clarity of finite state machine specifications.

  ▷ You can use JavaScript as a specification language to build the finite state machine directly; we show how in section 9.8 (p. 316). We discuss more general approaches to design tools in section 12.6 (p. 428). For other ideas extending the framework see section 9.5.2 (p. 295).

9.7.1 Going further: JSON

Our framework is written in JavaScript, which may give the impression that it is not as serious as, say, a framework written in Java or C++. In fact, our framework will work in many other languages *directly*, and therefore is as serious as any other approach. JavaScript has inspired the JavaScript object notation, JSON, which uses what is essentially JavaScript notation to define objects in a portable way that can be used in a very wide variety of languages, including ActionScript, C, C++, C#, Lisp, Haskell, Java, JavaScript (of course), Perl, PHP, Python, Ruby, and Tcl/Tk.

JSON is exactly our framework notation, except that JSON's field names have to be written in quote marks—although most JSON systems don't worry unless the field names need funny characters (and none of ours do). So when we wrote definitions of devices like `{fsm: [[0,1],[1,1]]...}`, all we need to do is change them to `{"fsm": [[0,1],[1,1]]...}` and they are then proper, strict JSON.

If you want to program using our framework in Java, get hold of the Java-JSON package, use it to read in the JSON notation, then you have objects in Java that you can use exactly as we have been doing in this book.

▷ You can get more details from www.json.org

9.7.2 Going further: Phidgets

Our framework works nicely in web browsers on any platform, which is an advantage of using JavaScript, but you might want to build *real* systems, not on-screen devices restricted to a web browser style of interaction.

Phidgets are a very nice way to get into programming hardware. Phidgets are so-called because they are the physical equivalent of on-screen widgets—buttons, text fields and so on.

Phidgets allow you to build user interfaces using LCD displays, LED indicators, buttons, knobs, sliders, touch sensors, RFID tags, motors and so on—as well as relays, so you can control real systems. The touch sensors can be placed behind paper you have printed with button designs or controller layouts, so you can get realistic prototype designs to work very easily.

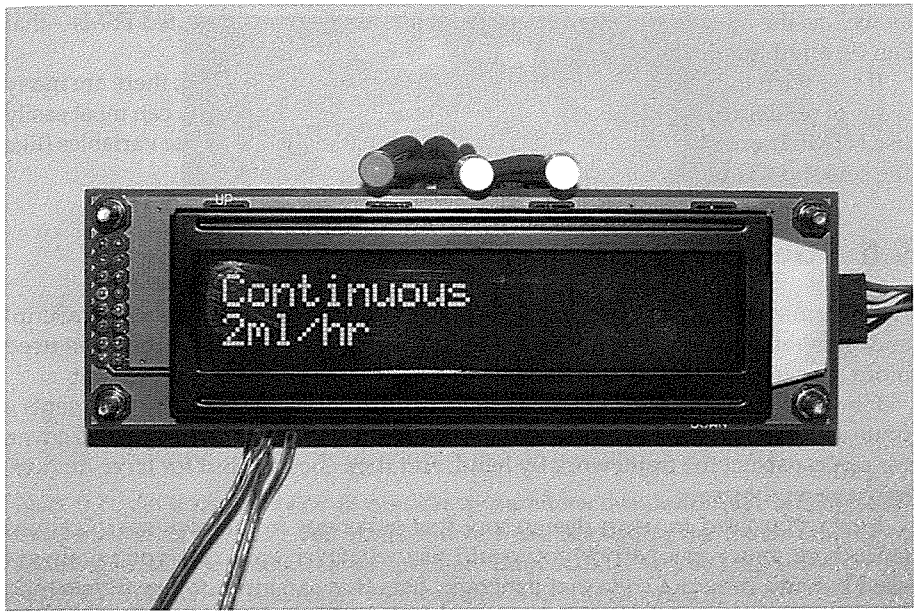


Figure 9.6: A Phidget, connected to a USB port on a PC, simulating a device display.

Phidgets use USB interfaces and are well supported. So, for example, you could use JSON to get our framework into ActionScript, C or Java, then use the Phidget libraries provided for these languages to run the hardware. Figure 9.6 (this page) shows a simple user interface built out of Phidgets, running a syringe pump user interface written using this book's framework.

▷ You can get more details from www.phidgets.com

9.7.3 Going further: web sites, XML and so on

In our framework, a state is "just" a name. We can add more attributes to states, such as manual sections, images, indicators, and so on indefinitely, but each extension makes the framework more complex. A neater and more general extension is to make states themselves HTML or XML pages (the state names could be URLs). Now a state can not only display different images and text, but a state can have internal programmed activities as well—anything an HTML page can do, even be another interaction framework. In fact, we have obtained a hierarchical FSM; where the states at one level are FSMs in their own right at the next lower level.

For example, one state might allow the user to enter numbers for, say, selecting a CD track or dialing a phone number, but at the higher level of the FSM, these are details that are abstracted away. Thus we get the advantages of a nice, clean FSM,

as well as the advantages of arbitrary programming—or even nested FSMs—that can do anything in any state.

If you wish to build hierarchical FSMs like this in Java/C++/C#, there are many ways of rendering sub-FSMs; if you continue using JavaScript, you can most easily use HTML frames to maintain and carry the simulated device's state variables (like the selected CD track number) from one sub-page to another.

9.8 Generating device specifications

A problem with our framework is that device designs look like so-much JavaScript and numbers. To designers, interactive devices just don't *look* like that! We need more conceptual ways of representing devices that are easier to think about.

How can we go from conceptual designs to a table of numbers that represents a finite state machine? For very simple devices, such as this book has space to cover, we can construct the framework by hand, and then check it. But for large devices, we have a problem.

Either you need to use a design tool that generates framework specifications (and there aren't any yet) or you could use whatever approach you've always used but add program code that generates data that can input to our framework. For example, you might build a conventional device in Java or C++, and write in it calls to generate framework data, perhaps in JSON. This way, you get the benefits of your usual approach *plus* all the analysis and benefits our framework provides. You might consider translating the code in this book so that the framework's benefits can be obtained within the same program; there's no need to use JavaScript.

Another way is to write a program to construct the finite state machine. Now, you can use the high-level features of your programming language to specify the *structure* of the interaction programming.

Figure 9.7 (facing page) shows an electric drill, which is a very simple FSM, but with an interesting structure. The drill has 8 user actions, twisting knobs, pressing buttons—not counting "indirect" actions such as removing the battery or letting it go flat. The finger trigger controls the drill motor. On the assumption that we are interested in examining how the drill behaves at T different speeds, we need to model T different trigger positions. The drill then has $216T$ possible states—the 216 comes from the combinations of knobs and buttons. However, some states are disallowed by the physical construction of the drill: for instance, when it is locked off, the trigger must be out and the drill cannot be running. More accurately, then, there are $144T + 72$ possible states. In our program below, we will take $T = 2$, that is, we will only distinguish between the drill running or not running, not the exact speed it is running at.

With $T = 2$, the drill model has 360 states. In one state, for instance, it can be running clockwise in second gear in hammer mode. There are 8 different actions—increasing the clutch, decreasing the selected gear, and so on—so the drill could have $360 \times 8 = 2,880$ transitions. In fact it only has 1,746 transitions. The "missing" transitions are impossible because of physical constraints in the design. For

Figure 9.7: An unusual device to consider as an interactive device, but a finite state machine nevertheless, with 360 states—more if we distinguish the motor running at various speeds. The device can be specified using a statechart or program—see figure 9.8 (p. 319) for the statechart and section 9.8 for the program.

example, you can only change direction from clockwise to anti-clockwise if the motor is not running.

The best way to define this device in the framework is to write a program that provides the details. It would be far too tedious and error-prone to write out everything by hand. Every physical constraint will appear in the program as an explicit test.

Interestingly, the user manual for the drill (a DeWalt DC925) only mentions one of the constraints—it says you must not change gear while the motor is running. Trying to specify this device as a FSM therefore highlights a possible design issue: should the gears be improved so that nothing need be said in the user manual? Would it be a better tool if the gears were modified? It would certainly be easier to use, and with a reduced risk to the user of wrecking the gearbox, which the manual currently warns about. So, even without doing any analysis with the framework, the *discipline* of being explicit about the interaction design of the device has raised a design issue.

What follows is one way to program it in JavaScript. First we need state numbers for every possible state the drill can be in. The drill allows the user to set the gear, the clutch, and so on, to various settings:

```
var clutchSteps = 24, directionSteps = 3,
    gearSteps = 3,    triggerSteps = 2;

// all values are numbered from zero
// e.g., the program uses 0,1,2 for the drill's gears 1,2,3
```

```
function conceptualState(trigger, clutch, gear, direction)
{ return trigger+triggerSteps*(clutch+clutchSteps*
                                (gear+gearSteps*direction));
}
```

Given some user settings, for example that the clutch is in position 3, we use `conceptualState` to generate the state number, which will be a number from 0 to 431. We allow for 432 states because, so far in the programming, we are not worrying about the physical constraints that disallow some user actions in certain states.

In our framework, each state needs a name and we can generate these automatically:

```
function stateName(t, c, g, d)
{ var n;
  n = t == 1? "running ": d == 1? "": "stopped ";
  n += d == 0? "reverse, ": d == 1? "locked ": "forwards, ";
  if( c == clutchSteps-1 ) n += "hammer mode";
  else if( c == clutchSteps-2 ) n += "drill mode";
  else n += "clutch set to "+(c+1);
  n += ", gears set to "+(g+1);
  return n;
}
```

This function converts the numbers that describe the state to a string. For example, `c` is the clutch position; the code treats the clutch as a number 0 to 23, modeling the position of the twist control on the drill, which has 24 positions (there are 22 different strengths of screwdriving, a drilling mode, and a hammer drilling mode). Calling `stateName(1, 4, 2, 2)` generates the string `"running forwards, clutch set to 5, gears set to 3"`—a good state to do some screwdriving. The function doesn't ever say `"stopped locked"` but just `"locked"` as it is obvious that when it's locked it's also stopped.

To test the code on *all* states, it can be put inside nested `for` loops to try it in all possible states. As we shall need nested `for` loops running over all the drill's states several times, we write a function to make them easier to manage:

```
function every(doThis)
{ for( var t = 0; t < triggerSteps; t++ )
    for( var c = 0; c < clutchSteps; c++ )
      for( var g = 0; g < gearSteps; g++ )
        for( var d = 0; d < directionSteps; d++ )
          doThis(t, c, g, d);
}
```

At this point, we can test the state names work by the following code, which defines a test function and calls it for all combinations of settings:

```
function test(t, c, g, d)
{ document.write(stateName(t, c, g, d)+"<br>");
}

every(test);
```

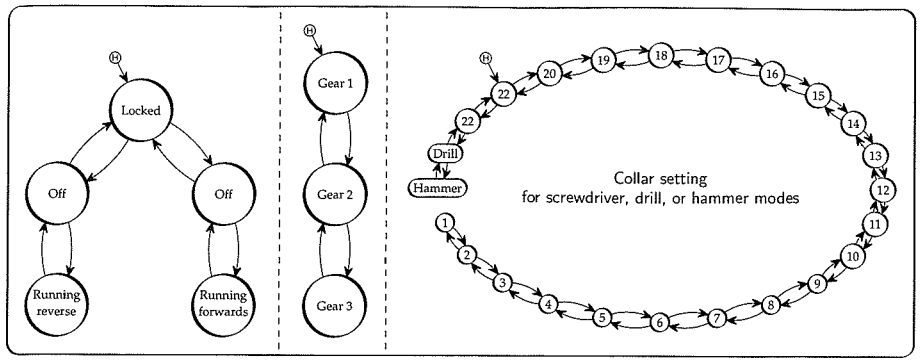


Figure 9.8: A statechart description of the DC925 drill shown in figure 9.7 (p. 317). Unusually, this device has no self-loops—if the user can physically do an action, the device changes state. In contrast, pushbuttons on most devices can be pressed even when they do nothing, which creates self-loops in their FSM model.

Amongst the mass of output this generates, it will print "`running locked`" a few times, which is a combination that ought to be impossible! You can't press the trigger in to get it to run when the direction slider is in the central, locked, position. We obviously still have more programming to do.

Next, for all possible user actions in any state, we need to work out what transitions are possible:

```
function action(t, c, g, d)
{ transition(t, c, g, d, "trigger in", t+1, c, g, d);
  transition(t, c, g, d, "trigger out", t-1, c, g, d);
  transition(t, c, g, d, "clutch increase", t, c+1, g, d);
  transition(t, c, g, d, "clutch decrease", t, c-1, g, d);
  transition(t, c, g, d, "gear up", t, c, g+1, d);
  transition(t, c, g, d, "gear down", t, c, g-1, d);
  transition(t, c, g, d, "direction +", t, c, g, d+1);
  transition(t, c, g, d, "direction -", t, c, g, d-1);
}
```

This function is saying, for all of the features on the drill, things like "if the current state is t, c, g, d then we could increase the clutch setting by 1, and if we did, it would be in state $t, c+1, g, d$." If we wrote `every(action)` this would generate calls to the function `transition` for *every possible* thing that could be done in *every possible* state. Unfortunately, not all the states and not all actions are actually possible. For example, if the drill is locked, the trigger cannot be pressed in to make it run; and if the drill is set to gear 2, we can't increase the gear to 3, because there are only three gears (gears are numbered 0, 1, 2, even though the drill itself calls them 1, 2, 3). We need to program a check on the drill's constraints:

```
function allow(t, c, g, d)
{ if( t < 0 || t >= triggerSteps ) return false;
  if( c < 0 || c >= clutchSteps ) return false;
  if( g < 0 || g >= gearSteps ) return false;
  if( d < 0 || d >= directionSteps ) return false;
  if( d == 1 && t != 0 ) return false;
  return true;
}
```

The important point is that this function and `transition`, discussed below, capture *all* the device's constraints in one place. The complex constraints are captured in a clear programmatic way. For example, the last test in the code above, `if( d == 1 && t != 0 ) return false` effectively says, "if the drill is locked, then the trigger must be out and the motor off."

Writing the code prompted me to think more about the drill's constraints: have I written accurate code for this book? It turns out that you can stall the drill if you try hammer drilling with the motor in reverse and simultaneously apply some pressure. This could happen if you are drilling, for instance, reinforced concrete and wish to free a drill bit that has got stuck—if you reverse the drill but leave it in hammer mode, the drill itself may get jammed. It must be an oversight that this case is not mentioned in the user manual. The extra code needed to express this constraint in the function `allow` is `if( d == 0 && t == 1 && c == clutchSteps-1 ) return false`, or in words, "if in reverse, and the trigger is pushed in, and the clutch is set to hammer, then disallow this state."

Our framework requires consecutive state numbers 0, 1, 2, 3 ... with no gaps, so if some states are not allowed we need a way of mapping conceptual state numbers to real state numbers, skipping the states that are not allowed. The easiest approach is to construct a map as a JavaScript array:

```
var map = new Array(triggerSteps*clutchSteps*gearSteps*directionSteps);

var n = 0;
function makeMap(t, c, g, d)
{ if( allow(t, c, g, d) )
    map[conceptualState(t, c, g, d)] = n++;
}

every(makeMap);
```

After running this code, `map[s]` gives us a number 0 to 359 of allowed state numbers, provided that `s` is an allowed state from the original 432 conceptual states; otherwise `map` is undefined.

Recall that the framework requires each state to be named. Here's how the map can be used to name the real states: if a conceptual state is allowed, map the state number to a real state number, then give it its state name. As before, we generate the state name from the combination of gears, clutch settings and so on, using the function `stateName` we've already defined. Notice we use `every` to conveniently run over all possible states.

```
drill.stateNames = new Array(n);
function nameEachState(t, c, g, d)
{ if( allow(t, c, g, d) )
    drill.stateNames[map[conceptualState(t, c, g, d)]] =
          stateName(t, c, g, d);
}

every(nameEachState);
```

To complete the framework FSM, we must make sure all the user's state transitions are allowed; that means both the state we are coming from `t0...` and the state we are going to `t1...` are allowed. For the drill, if both states are allowed, the transition between them is always allowed. More complex devices would need further programming to allow for more complex conditions on what transitions are allowed—for example, although the drill allows us to change gear when it is running, the manual warns this is a bad idea because the gears may grind, and this constraint could be handled by writing `... && (t0 == 0 || g0 == g1)`—meaning "trigger out (motor not running) or the gears aren't changed."

```
function transition(t0, c0, g0, d0, button, t1, c1, g1, d1)
{ if( allow(t0, c0, g0, d0) && allow(t1, c1, g1, d1) )
    drill.fsm[map[conceptualState(t0, c0, g0, d0)]][lookup(button)] =
              map[conceptualState(t1, c1, g1, d1)];
}
```

The details we haven't yet provided are for initializing the FSM and defining the function `button`, needed as a way of getting a button number from the button name.

```
drill.fsm = new Array(n);
for( var i = 0; i < n; i++ )
{ drill.fsm[i] = new Array(drill.buttons.length);
  // if you try an action, by default stay in the same state
  for( b = 0; b < drill.buttons.length; b++ )
    drill.fsm[i][b] = i;
}

function lookup(button)
{ for( var b = 0; b < drill.buttons.length; b++ )
    if( button == drill.buttons[b] )
      return b;
  alert("Button "+button+" isn't recognized!");
}
```

The `alert` in the function `lookup` will happen if we misspell a button name anywhere; it's a useful check.

Now we've finished the programming, calling `every(action)` will create the FSM we wanted. The FSM will have hundreds of rows like `[346, 195, 352, 340, 346, 344, 346, 346]`, *but we need never look at them.* We should use the framework to analyze the drill's properties rather than looking directly at the raw data.

Generating a FSM might take a long time—especially in a slow language like JavaScript—but it only needs to be done once. It doesn't matter how complicated the functions like `every` and `allow` are; write them clearly, without worrying how inefficient they seem. The point is to make the device's interaction structure clear.

▷ Physical constraints are closely related to *affordance*, a topic covered in section 12.3 (p. 415).

9.9 Conclusions

This chapter has introduced and explored the benefits of programming interaction frameworks: general purpose programs that can run, simulate, or analyze any device. The advantage of a framework is that it can check and measure all sorts of useful properties—and it becomes worth doing so because we can compare and contrast different designs very easily. Without a framework, each device is a completely different programming problem, and it probably won't seem worth going to the trouble of writing high-quality program code to evaluate it.

In particular, this chapter developed a framework in JavaScript that readily allows a device to be specified, simulated, and analyzed. Professional programmers will perhaps want to redesign the framework for their favorite languages, and there is much to be gained by doing so.

We could develop the simple framework here into a full-blown design tool. There are many possibilities ... it's astounding that the user interfaces of most devices are so routine—even a simple framework helps designers become more certain in their processes, and in turn become more confidently creative.

9.9.1 Some *Press On* principles

- Once there is a prototyping framework, changing or revising a device specification is easy and reliable → section 9.4 (p. 286).
- A framework can check for many basic design errors—something that cannot be done when a system is written in a standard programming language → section 9.5 (p. 288).
- Find all least cost paths through a device; any impossible paths should be justified carefully or fixed → section 9.6 (p. 301).
- By programming user problems, the designer can get useful statistics and measurements about a device's overall behavior → section 9.6 (p. 311).
- Technical authors should use automatic tools working with device specifications so they have reliable advice for users → section 9.6 (p. 311).
- The harder it is for a designer or a programmer to solve task/action mappings, the harder it will be for the user; find a way to make it easier → section 9.6 (p. 312).

9.9.2 Further reading

Before starting any project, do an internet search and see whether somebody has solved your problems for you, or defined standards that you can take advantage of. There are many programs and libraries on the web you can start from more easily than working on your own.

- Dot is an open source research project at AT&T. Full details of Dot can be found at www.research.att.com/sw/tools/graphviz. I used Pixelglow's award winning OSX implementation, GraphViz, to draw the graph earlier in this chapter.
- Kaye, J., and Castillo, D., *Flash MX for Interactive Simulation*, Thomson, 2003. The authors professionally develop medical devices and training systems, and their book is the best book for programming in Flash. It gives many examples of device simulators and has more material on statecharts. Their techniques for building interactive systems can be copied into other languages.
- MacKenzie, I. S., Motor Behavior Models for Human-Computer Interaction, in Carroll, J. M., ed., *HCI Models, Theories and Frameworks*, pp27–54, Morgan Kaufmann, 2003. Scott MacKenzie provides a very good discussion of Fitts Law and other models.
- Sharp, J., *Interaction Design for Electronic Products Using Virtual Simulations*, PhD thesis, Brunel University, 1997. This thesis gives the definition of the microwave oven we used.
- There are many programming tools and standards out there, from JavaHelp to XML. In particular, SVG is a standard for vector graphics, effectively an open source version of *Flash*.