

CS1Ah Lecture Note 15

Case Study: Cruise Control

The aim of the case study lectures is to provide slightly larger examples of the results, techniques, and ideas presented in the other lectures. This note has three main aims:

1. To illustrate the use of finite state machines in a “real world” context. This illustrates one area in which finite state machines are used extensively – there are many others;
2. To illustrate the main activities in the production of a software product in the context of creating a finite state machine. The example is very small but it does illustrate the main stages in creating a product;
3. To illustrate a simple way to translate from a deterministic finite state automaton to a Java program that “implements” the machine.

Embedded Controllers

Over the past ten to fifteen year a major change has taken place in the way we interact with a wide range of electromechanical devices. That change has been brought about by placing a computer program (usually called an embedded controller) between the human user and the controls of the machine. The purpose of that program is to allow humans to interact with the machine in terms of their needs rather than in terms of the basic controls of the machine. A good example of this is a television. Consider the first task you undertake after purchasing a new television:

Now: to set up the TV to operate properly we just press the sequence of buttons to initiate the setup function in the controller, it searches for strong signals, allocates different channels to different UHF frequencies and returns control to the user once all the strong signals have been allocated to channels.

10 or 15 years ago: the user had to select each channel in turn and then manually search the frequency spectrum to find different TV stations.

The striking difference is that the operation of today’s TV is in terms of an action any user can understand without needing to understand the internal operation of a TV. Ten or fifteen years ago the user was required to translate their desire to watch TV into a number of (rather arcane) actions that were explicable only in terms of the internal

functioning of a TV. The success of the strategy of inserting a program between the user and the machine is so successful that all but the very simplest machines have some kind of embedded controller.

This approach also has some disadvantages: poor programming can make the machine harder to use, the complexity of the program can add to unreliability, users have little understanding or fine control over the machine outside that permitted by the controller, controllers are often used to patch up poor basic design in the machine under control.

Cruise Control

On long car journeys drivers find it very tiring to keep up continuous pressure on the accelerator pedal. To avoid the need to do this many cars now have a system called *cruise control*. This is a controller that sits between the driver and the controls of the car (here the throttle that determines how fast the car travels). The cruise control system allows the driver to set a particular speed and then the controller maintains that speed until the driver changes the speed, uses the brake, or switches the system off.

These systems are usually controlled by a number of push-buttons on the dashboard or steering wheel of the car. The system usually has different modes of operation depending on the history of how the car has been controlled. The modes of operation (and subsidiary states arising in a particular mode) are usually modelled by a finite state machine and the meaning of the actions on the push buttons are given by saying what transitions take place between the states of the FSM when the buttons are pressed.

Specification

It is common practice in the world of embedded controller to use a very low level specification which is close to the implementation of the system. This practice arises out of the evolutionary approach to the design of such controllers. The next controller is very often just like the previous one with more bells and whistles.

Ideally the specification for a system should describe what the system should do – not how it does it. Here we list the main properties we would expect of the system. A real world system would have one or two extra features but would be quite close to this.

1. The driver should always be able to turn the system off.
2. The driver should be able to request the system to maintain the current speed.
3. The system should not operate after braking
4. The system should allow the driver to travel faster than the set speed by using the accelerator.

Design

Taking account of current practice in the industry (and hence user expectations of how the system will work) we can give a preliminary design that we believe matches the specification. The design consists of three components: the inputs and outputs to the system, the modes (or states) of the system, the effects of the inputs on the system in each state. Here we should expect to say explicitly what the effect of an input is for every state.

The inputs are:

- on: on/off button
- set: set the cruise speed to the current speed
- brake: the brake has been pressed
- accP: the accelerator has been pressed
- accR: the accelerator has been released
- resume: resume travelling at the set speed
- correct: indicates the car is travelling at the correct speed.
- slow: indicates the car is going slower than the set speed
- fast: indicates the car is going faster than the set speed

The outputs are:

- store: store the current speed as the cruise speed
- inc: increase the throttle
- dec: decrease the throttle

The main states (or modes) of the controller are:

Off: The system is not operational.

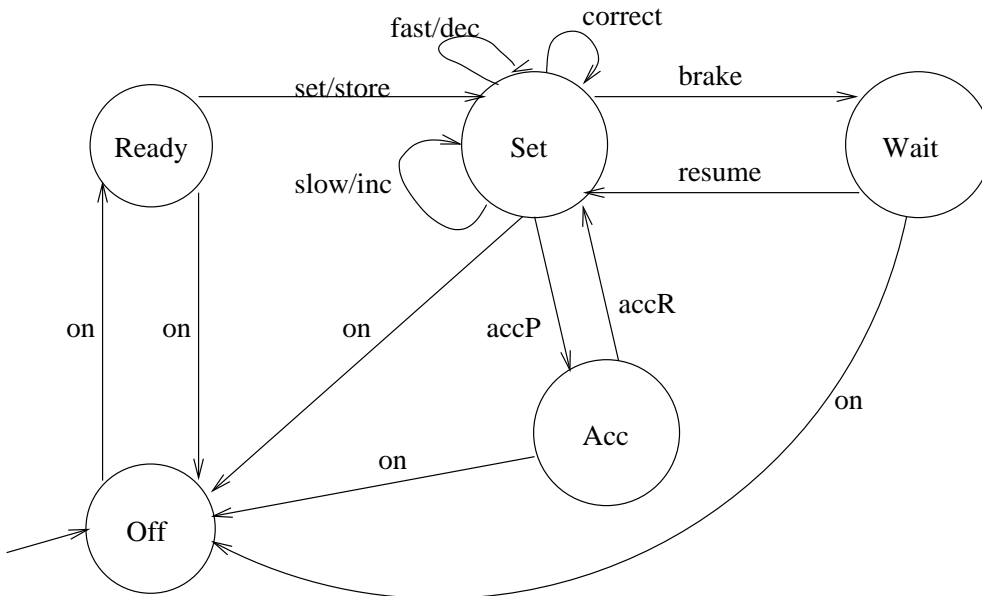
Ready: The system is switched on but so far no speed has been set to cruise at.

Set: A cruise speed has been set and the system is maintaining it.

Wait: The system has a set cruise speed but at some time the driver used the brake and caused the system to wait until the resume button is pressed to bring the car back into cruise control.

Acc: The system has a set cruise speed but the accelerator has been pressed to override cruise control until the accelerator is released.

We describe the FSM diagrammatically, the labels on the arcs represent inputs. If the label has the form *in/out* then output *out* is generated on the transition for input *in*. In this diagram, where we omit transitions from a particular state for some input we intend it should be a loop back to the same state with no output.



Testing

In testing the system we might first want to explore the behaviour of the system to see if we can discover anomalous behaviour. To do this we might imagine using the FSM to derive sequences of actions. In testing we usually use a *coverage criterion* to limit how much testing we do. Because the system is very small we might want to use an “all paths” criterion where the set of test inputs covers all the possible paths through the finite state machine (for very large machines we can’t do this).

We would also want to check that all the properties in the specification are satisfied by the design. Here we can see that they are satisfied by the design. For more assurance we might want to express the properties in the specification in some formal logic that would allow mathematical verification of the properties in principle.

We might also consider the transitions from each state in turn and check that their effect is correct. In our example consider the effect of a `set` operation in the *Wait* state. Here we might require that we also do a `store` output to reset the cruise speed. We might also argue that instead of a loop the arc labelled `set/store` might go to the *Set* state rather than looping on *Wait*.

Implementation

A finite automaton is a fairly abstract, idealised, notion of a computing device. It is very useful to support design activities for a wide class of systems, as exemplified by the Cruise Control example. To go from the abstract design expressed as a finite automaton to a piece of hardware or a combined hardware/software system requires us take many detailed decisions on how to relate the abstract definition to the hardware and program making up the *implementation* of the system. At a minimum this will involve:

- Working out how the notion of the input in the idealised definition relates to signals or computer inputs in the implementation. In the Cruise Control example

we just consider a sequence of symbols as the input. In an implementation each input to the controller will be a different signal from the sensors (pushbuttons, movement sensors, position sensors etc) picking up information from the driver and the car.

- Working out how to represent the state of the Finite Automaton. In hardware this will be done using flip-flops or memory. In software we will use some program variables to represent the state as stored values. In any case, the number of possible states in the implementation is likely to be quite a bit larger than the number of states in the abstract description of the Finite Automaton.
- Working out how the idealised notion of the output of a Finite Automaton relates to the signals or computer outputs in the implementation. Again the output is unlikely to be a single stream of symbols, instead it is likely to be several signals that are connected to the actuators that influence the equipment under control.

Once we have sorted this out there remains the issue of how we tell whether a proposed implementation really is an implementation. In the case of a Finite Automaton we expect that the implementation should accept the same input sequences as the Finite Automaton and for a given input the implementation should generate the same output as the Finite Automaton. This still leaves the question of how to relate the single input sequence in the abstract definition to a number of signals in the implementation — in the implementation we need to think about events happening simultaneously and this is not considered in the abstract model.

In general, the issue of how to relate an implementation to a more abstract design is an important issue in Computer Science and Software Engineering both in the development of theoretical work that tries adequately to capture this complex relation and practically in deciding whether some piece of hardware or hardware/software combination is faithful to an abstract design.

Finite Automaton to Java

The attached program `CruiseControl.java` is intended to be an implementation of the original (*i.e.* uncorrected) finite automaton for the Cruise Control system. In making this implementation we have taken the simplest decisions we can in order that the correspondence between the abstract design and the implementation is as straightforward as possible.

- The input is a sequence of strings read from the keyboard in response to a prompt from the program.
- The state of the automaton is represented by a single variable called `state` that stores a string that represents the current state of the automaton.
- The output of the implementation is a sequence of strings displayed on the screen of the computer.

Having made these decisions, we are ready to construct the program for the finite automaton.

Repeatedly Making Transitions

At the top level an implementation of a Finite Automaton should repeatedly get the next input and make the relevant transition to the the next state depending on the current state and the input until a final state is reached. This is a good example of where a while loop can be used. At the top level the structure of the implementation of the Cruise Control automaton is:

```
// Initialise Values

// Repeat Forever
while(true) {
    // Display Current State
    // Read input from keyboard
    // Make a transition
    // Display any output
}
```

Because there is no final state in the Cruise Control system the condition in the while is true so the program loops forever.

Transitions

In implementing the transitions we need to work out which state we are in and then change the value of the state depending. The obvious way to do this is to code the states and the input symbols as small integers and then use switch statements. This requires us to convert from strings to integers on input and from integers to strings on output.

Is it Correct?

Once we have our program we might reasonably ask whether it really implements the design. There are several approaches to establishing that the implementation is acceptable. Here we just consider testing and inspection.

Testing: This is an approach to finding flaws in the implementation. In testing one defines a collection of test inputs that are chosen to attempt to ensure that the program will work in all contexts it is expected to work in. In our case we might choose the test set to be all non-looping sequences of inputs in the graph of the Finite Automaton. This would exercise all the transitions in the design and might give some confidence they had been implemented.

Inspection: The text of the Java program implementing the Finite Automaton bears a close relationship to the transition graph of the Automaton. One way of checking the implementation is by inspecting the code to see it corresponds correctly to the transitions in the design.

Toward a Full Implementation

The Java program attached to the end of this note is still very far from a complete program for the Cruise Control problem. Here is a list of some of the work that needs to be done:

- Some of the transitions require the system to move data around. For example the set transition causes the current speed to be stored. In a full implementation this would require the use of additional variables to store the speed.
- Production code for such a system would include a much greater level of error checking and recovery. For example to see if the input is correct, to see the state variable is only set to one of the state names.
- Production code would probably have logging and monitoring code to monitor the use of the system, to detect errors or to allow the reconstruction of a sequence of events after an accident has taken place.

The Working Environment

The real working environment for systems like Cruise Control is very hostile to programmable devices. The level of Electromagnetic Interference in a car is very high. The effect on computers is to corrupt data and to cause random jumps in the program. This suggests the code might be constructed in a style that allows easy recovery from such incidents (short sequences of code, checking code has had its desired effect, automatically restarting if the system becomes inactive. This kind of coding adds a significant overhead to the simple approach taken here.

Evolution/Maintenance

After some months of operation the following accident report arrives at the car company¹:

The incident occurred when the driver was on a highway on a rainy night. The traffic was slow, travelling at about 40 mph. The driver engaged cruise control and set it to 40 mph. Later the rain cleared and the traffic got faster so the driver used the accelerator to increase the speed to 60 mph and travelled in this mode for some miles (the controller still in set mode but overridden by the accelerator).

Coming to the exit ramp the driver turned off and released the accelerator to coast up the ramp. At that point the cruise control aimed to stabilise the speed at the set level (40 mph). The driver was taken by surprise and lost control of the car which travelled through a stop sign without braking. Fortunately no accident occurred.

¹This example is courtesy of *Do you know what mode you're in? An analysis of mode error in everyday things*, Anthony Andre (Interface Analysis Associates, San Jose, CA) and Asaf Degani (San Jose State University, CA)

During the life of a product many accident reports are lodged with the manufacturer. These are usually analysed to see if there is some systematic source of accidents that can be eliminated if it presents a significant threat. The problem identified here is just such a case. One possible solution is to treat the accelerator like the brake, and require the driver to explicitly press the resume button in order to return to cruise control after accelerating. This would require the controller software to be amended and updated in vehicles, probably by replacing the controller chip.

Summary

- We have seen how finite state machines can be used to represent the modes of systems and are a useful descriptive tool (some hardware design tools that use FSMs can handle many million states).
- We have seen to translate from a deterministic finite state automaton to a Java program that “implements” the machine.

Murray Cole, 27th November 2002.

Implementation

```
// This implements the controller as it stood before the
// acceleration error was reported. Can you fix it?
import java.io.*;
public class CruiseControl {
    // Inputs
    public static final int on = 1;
    public static final int set = 2;
    public static final int brake = 3;
    public static final int accP = 4;
    public static final int accR = 5;
    public static final int resume = 6;
    public static final int correct = 7;
    public static final int slow = 8;
    public static final int fast = 9;
    // Outputs
    public static final int store = 10;
    public static final int inc = 11;
    public static final int dec = 12;
    // States
    public static final int OFF = 13;
    public static final int READY = 14;
    public static final int SET = 15;
    public static final int WAIT = 16;
    public static final int ACC = 17;

    public static void main(String[] args) throws IOException {
        // Initialise Keyboard Input
        BufferedReader keyboard =
            new BufferedReader(new InputStreamReader(System.in));

        int input = 0;
        int output = 0;
        int state = OFF;    // Initial State

        // Repeat Forever
        while(true) {
            // Display Current State
            switch(state) {
                case OFF: System.out.println("State: OFF"); break;
                case READY: System.out.println("State: READY"); break;
                case SET: System.out.println("State: SET"); break;
                case WAIT: System.out.println("State: WAIT"); break;
                case ACC: System.out.println("State: ACC"); break;
                default: System.out.println("Error: Undefined State");
                    System.exit(1);
            }
        }
    }
}
```

```
}
// Read input from keyboard
System.out.print("Input: "); String in = keyboard.readLine();
if (in.equals("on")) input = on;
else if (in.equals("set")) input = set;
else if (in.equals("brake")) input = brake;
else if (in.equals("accP")) input = accP;
else if (in.equals("accR")) input = accR;
else if (in.equals("resume")) input = resume;
else if (in.equals("correct")) input = correct;
else if (in.equals("slow")) input = slow;
else if (in.equals("fast")) input = fast;
else input = 0;

// The state machine is defined below - the system remains in the
// same state unless a new state is explicitly set.

if (input != 0) {
    // Clear Output
    output = 0;

    // State Machine Actions
    switch(state) {
        case OFF: // OFF State
            switch(input) {
                case on: state = READY; break;
                default: break;
            }
            break;

        case READY: // READY State
            switch(input) {
                case on: state = OFF; break;
                case set: output = store; state = SET; break;
                default: break;
            };
            break;

        case SET: // SET State
            switch(input) {
                case on: state = OFF; break;
                case brake: state = WAIT; break;
                case accP: state = ACC; break;
                case fast: output = dec; break;
                case slow: output = inc; break;
                case correct: break;
                default: break;
            }
        }
    }
}
```

```
    }
    break;

    case WAIT: // WAIT State
        switch(input) {
            case on: state = OFF; break;
            case resume: state = SET; break;
            default: break;
        }
        break;

    case ACC: // ACC State
        switch(input) {
            case on: state = OFF; break;
            case accR: state = SET; break;
            default: break;
        }
        break;
    }
}

// End of State Machine Definition

// Display any output
switch(output) {
    case store: System.out.println("Output: store"); break;
    case inc: System.out.println("Output: inc"); break;
    case dec: System.out.println("Output: dec"); break;
    default: break;
}
// Print separator
System.out.println("--");
}
}
```