Lecture 6

Inf1A: Transducers for Modelling Systems

6.1 Introduction

In this lecture we will return to the study of *transducer-style* Finite State Machines. We will specifically consider transducer-style FSMs for modelling computational systems. We will see that important software engineering issues such as *design*, *testing* and *verification* are also important in the context of Finite State Machines.

Many computational systems that arise in practice are *persistent* systems, in the sense that we do not think of them as computing one limited computational function; but rather as performing an arbitrarily-long series of small actions in response to a arbitrarily-long sequence of inputs (or stimuli). Finite State Machines can be very effective for modelling such systems.

We will base this lecture on a single case study of a particular persistent system, the Cruise-Control system commonly found in modern cars. We will consider design, testing, verification and implementation issues for this system. In fact, although this lecture note is quite different to the lecture notes for the other Finite State Machine lectures, we will see that we have been dealing with these issues right throughout the course.

Cruise Control Systems

On long car journeys some drivers find it tiring to keep up continuous pressure on the accelerator pedal. To avoid the need to do this many cars now have an *embedded system* called *cruise control*. This is a controller that directly links the driver with the throttle that determines how fast the car travels. The cruise control system allows the driver to specify the particular speed that he/she wants to travel at, and then the controller maintains that speed until the driver changes the speed, uses the brake, or switches the system off.

Cruise Control systems are more commonly found in American cars than in British cars (and are probably more useful on long US highways), but they have also become reasonably common in Britain over the last few years. A Cruise Control system is usually controlled by a number of push-buttons on the dashboard or on the steering wheel of the car. The system usually has different *modes of operation* depending on the different buttons which have been pushed so far. In this lecture we will see that these modes of operations (and any subsidiary states) can be modelled by the *states* of a Finite State Machine. We will see that changes in the mode of operation can be modelled by transitions of a *transducer-style* Finite State Machine.

6.2 Specification of the System

In modelling a computational system, we have to make choices about the level of *specification* we will give for that system. By *specification*, we mean a precise description of how the system adapts as a response to certain inputs. For very complex software engineering systems, we often design the system as a series of increasingly detailed designs - and in this case, the top-level specification will be a very *high-level* specification (not giving too much detail). However, when we are modelling a small *embedded system* (such as a Cruise control system) with limited functionality, it can be possible to give a low-level specification that is not too complex.

The specification for a system is different to a program or an implementation of the system, because a specification should describe *what* the system should do - not *how* it does it. You will see that in later years that when *formal specification languages* are used to specify computational systems, these languages tend to be based on some form of *logic*.

We will now specify (in English) some very basic properties that we expect of our Cruise Control system.

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

A real world system would have one or two extra features but would be quite close to this. In practice, a specification will often be more precise than what we have written for the Cruise Control system.

6.3 Design

In software engineering, the second step in constructing a system is to come up with a design that adheres to the specification.

In our design for the Cruise Control system, we will have three decisions to make: we need to decide what are the inputs and outputs to the system, what are the modes (or states) of the system, and what are the effects of the inputs on the system in each state. The third of these aspects will determine the behaviour of our system. We will need to specify the desired effect of an input, for every possible state the system can be in.

Our inputs will be taken from (i) the driver and (ii) the vehicle. They 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 for the system are:

Off: The Cruise Control 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 behaviour of the system by defining the "next state" that will result from a given input to a given state. The purpose of the *design phase* is to refine the specification to describe a behaviour for the system, which will conform to the specification. In other words, we describe *how* the system can be constructed (without implementing it - that is the final phase). The purpose of carrying out the design phase is to work out how the system will be realised, without worrying about implementation-dependent issues. Our design should be in a form to make it easily to implement. Therefore when we model a system as an FSM, the FSM should be a deterministic (usually transducer-style) FSM.

For our Cruise Control system, we will give this definition of the system behaviour in diagrammatic form, as a Finite State Machine (see Figure 6.1). Any label of the form *in/out* in the FSM should be read as meaning that output *out* is generated on the transition for input *in*. Note that our machine is *deterministic*, but not *strictly deterministic*. In general we would require a *strictly deterministic* design, because we will want to design a system

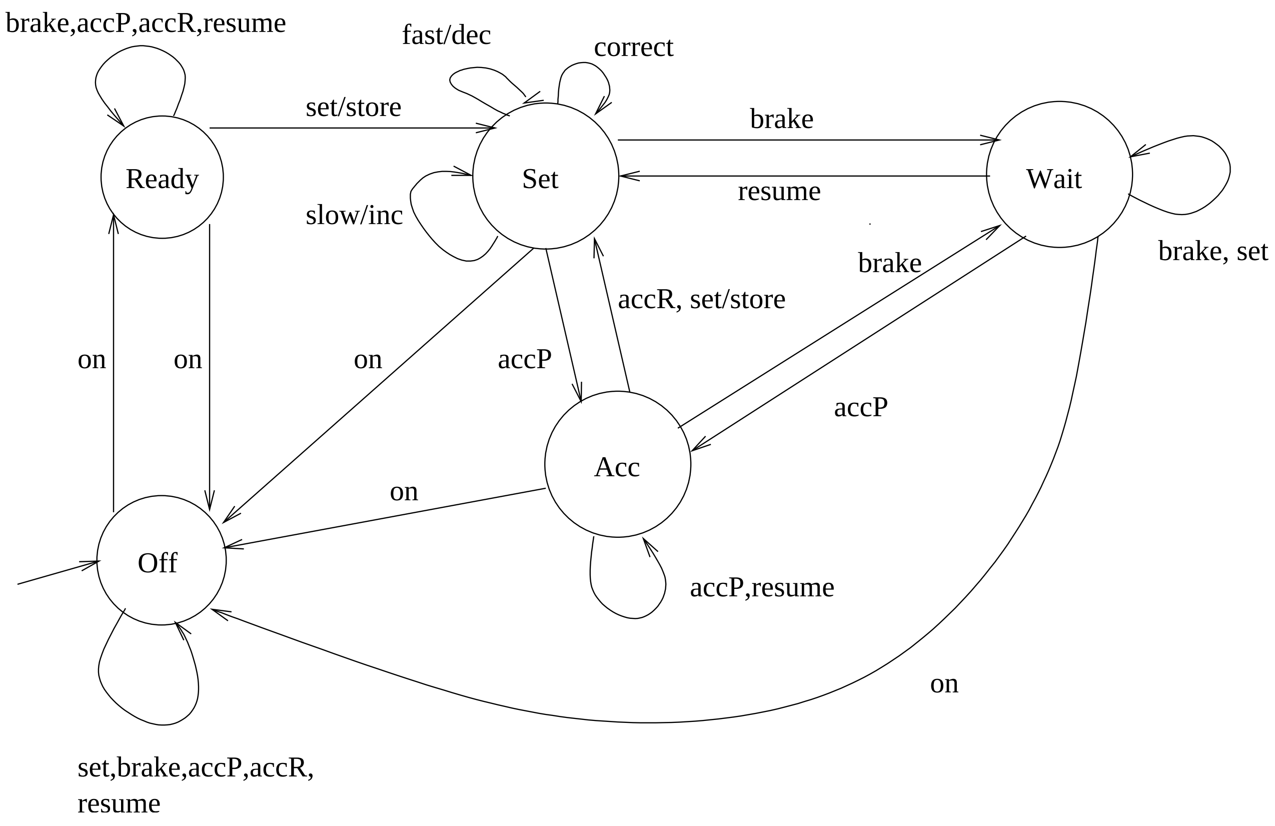

Figure 6.1: An FSM describing the behaviour of the Cruise Control system

whose behaviour is defined for all sequences of inputs. However, for the design of the cruise control FSM, we use some simple facts about cars (for example, the `accR` input can only directly follow a `accP` input) to allow us to omit some transitions.

In general, when using FSMs for the design phase of the software engineering cycle, it is sometimes useful to assume that if a particular state has no transition for some input, this corresponds to having a loop (with no output) back to the same state on that input. Notice that this is entirely **different** to what we assume for acceptor-type FSMs (and even most transducer-type FSMs). You shouldn't ever assume this by default - only when you are told this assumption is being made.

6.4 Testing

Many errors in the software engineering life cycle creep in at the design phase, and can be discovered at this stage. Therefore testing our design of a system (in reference to a specification for the system) is an important step.

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 Cruise Control Finite State Machine of Figure 6.1 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).

The most important part of the testing phase is ensuring that the properties of the specification are satisfied by the design.

For the Cruise Control example, it is not too difficult to check that properties 1.-4. given in Section 6.2 are satisfied by the design of Figure 6.1. For example, clearly property 1. is satisfied, because there is a transition from each state of the FSM to the **Off** state, where this transition is taken when there is a second push of the `on` (power) button.

Another way of testing the design involves considering the transitions from each state in turn and checking that their effect is correct. In our example consider the effect of a `set` operation in the **Wait** state. In our current design, trying to re-`set` in the **Wait** state has no effect. However, it might be the case that we would prefer that the effect would be to do a `store` output to reset the cruise speed, and transition to the **Set** state. Then the design FSM would need to have an arrow labelled `set/store` from **Wait** to **Set**, instead of looping at **Wait**. Note that this choice (for the effect of pressing `set` while in the **Wait** state) is a *design issue*, since the desired behaviour was not given as part of the specification.

6.5 Implementation

A finite automaton is a fairly abstract, idealised, notion of a computing device. It is very useful for supporting design activities for a wide class of systems, as we have seen with the Cruise Control example. In fact a number of Software Engineering Tools to support design with Finite State Machines are available on the internet: a few examples are `Esterel`, `Lustre` and `Argos`.

To actually construct a computational system for use in the real world, it is necessary to move from the abstract to an actual implementation. This is the final step in the Software Engineering life cycle. The implementation step often requires us to make a lot of decisions about how we will realise our design in terms of hardware and software. Some issues that will become important are:

- It will be necessary to work out 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.
- We need to decide how to represent the states 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 that the number of states in the abstract description of the Finite Automaton.
- Finally we need to decide how the outputs of a Finite Automaton are realised as signals or computer outputs in the implementation.

Once these implementation issues have been dealt with, we need to ensure that our implementation is a faithful representation of the design. When the design is done as a

Finite State Machine, we expect that the implementation should accept the same input sequences as the FSM and that for any given input, the implementation should generate the same output as the FSM (subject to our decisions about how we realise inputs and outputs in software and hardware).

In practice, the implementation of a system will involve an extensive *testing* phase.

Testing: This is an approach to finding flaws in the implementation. The usual approach to testing is to define a collection of test inputs that are chosen to attempt to ensure that the program will work in all contexts. In the Cruise Control case we might choose the test set to be all *non-looping* sequences of inputs in the Finite State Machine. This would exercise all the transitions in the design and might give some confidence they had been implemented correctly.

For safety-critical systems, testing is not enough. Usually it is necessary to have a *verification phase*, where we formally prove (by some method of formal reasoning like Logic) that the system is *guaranteed* to behave correctly (and safely) for all possible sequences of inputs.

6.6 Summary

We will not cover the implementation of the Cruise Control system in this course. It would certainly be possible to implement the software for the Cruise Controller in an appropriate programming language, possibly in `Java` (which you will study in *Informatics 1B* next semester).

Remember that we have seen some implementations of Finite State Machines already during the *Computation* section of *Informatics 1A*. In Lecture note 2 we showed how to implement a FSM to recognise odd numbers in unary in Haskell. Also, in our lab for week 8, most of you will have implemented a number of Finite State Machines in Haskell. For all these implementations, you will have informally tested your implementation to make sure it gives the same results as the original FSM.

These notes have been adapted from two sets of earlier notes, the first due to Stuart Anderson, Murray Cole and Paul Jackson, and the second due to Martin Grohe and Don Sannella.

Mary Cryan, November 2004