Lecture 1

Inf1A: Introduction to Finite State Machines

1.1 Models of Computation

Most of us we think of *computers* as laptops, PCs and perhaps Cray supercomputers. In fact there are many more small computing devices in operation all around us: Microwave clocks, iPods, Cash Registers are all examples of machines which are doing computational processing. Each of these devices can be thought of as a *reactive system* - that is because each of them work by *reacting* to signals or inputs from the external world.

Throughout *Informatics 1* and the later years of your course (CS, AI, ...) you will learn about many specific types of "reactive systems" which have applications in the real world. You will learn different *programming languages*, which are used to construct software to perform specific tasks. You will also learn about special application areas and the type of challenges posed in constructing systems for those application areas.

Also throughout *Informatics 1* and later years, you will learn the importance of being able to *model the behaviour* of natural computational systems (in application areas such as human speech), as well as artificial systems.

In studying such a wide range of systems, it is very useful to have some simple models of computation which abstract away from the untidy details of programming or of "real life". This allows us to focus on the *computational power* of a particular model of computation - what it is possible to compute in the model of computation. It also allows us to focus on the *representational power* of a particular model of computation.

Therefore in this section of Informatics 1, we are going to focus on a class of very simple *abstract reactive systems*. We will consider *Finite State Machines* (FSMs), which are also called *Finite State Automata* in some textbooks. This particular *model of computation* is usually the first abstract model of computation to be studied in most Computer Science and Artificial Intelligence undergraduate courses, because it is so simple. In fact, the particular machines which we will construct as examples throughout this class will seem to be much "easier" than the programs that you have been implementing in the *Functional Programming* strand of *Informatics 1A*. However, the reason we are going to consider this simple model of computation (in which many of the typical examples are also very simple) is because we want to study Finite State Machines in detail. In order to be able to make general statements about our model of computation, it makes sense to start by considering a restrictive model.

In fact, even though the Finite State Machine model is very simple, it is still the basis for a lot of design activity across a range of application areas in hardware and software engineering. It is also the basis for a lot of work in pattern matching (such as the pattern matching which is performed by Google in searching documents for keywords) and for research into computer applications for speech and language processing.

1.2 The main idea

The main observation to make about reactive systems is that the response to a particular stimulus (a *signal*, or a piece of *input*) is not the same on every occasion. For example, in the case of a parking ticket machine, it will not print a ticket when you press the button unless you have already inserted some money. Thus the response to the print button depends on the previous *history* of the use of the system.

The Finite State Machine model restricts the number of different responses to a particular stimulus to be finite and to be fixed by the description of the machine. This is the big difference between the Finite State Machine model and other models of computation. Basically, in the Finite State Machine model, we can only construct machines which have a finite number of different states. The only effect that a stimulus can have is to (possibly[1]) generate a response from the machine and change state to some new state.

In the design of a Finite State Machine for a particular problem, it is usually the case that the input to the FSM will consist of an entire *sequence* of stimuli, rather than a single stimulus. We will usually use the terminology of an *alphabet* Σ, whose symbols correspond to particular stimuli that may be sent to the machine. Then our inputs will consist of a sequence of symbols from this alphabet, where the sequence is usually referred to as a *string* over the alphabet Σ.

The fact that we only are allowed to have a fixed number of states limits what we can compute (or model) with an FSM. When we read a new element of our input string and move to a new state, we expect the new state to reflect *what we want (or need) to remember* about the past inputs. Therefore, because we only have a fixed number of states, we are restricted to remembering a fixed finite amount about what has gone before. We will see later in Lecture 6 there are many simple counting tasks that do not fit within this model.

1.3 Details

In this section we describe the details of the finite-state machine definition. We first talk about how to describe finite-state machines, and then show how this description relates to the *behaviour* of the machine. We then go on to discuss some further technicalities of the definition.

We describe a FSM by providing the following descriptions:

1. A description of the stimuli that the machine will take into account. This is defined by giving a finite *input alphabet* consisting of a set of different *symbols*. The input

[1]Only the *transducer* type of Finite State Machine generates outputs. Most, though not all, of the machines we consider in this course will be acceptors, which do not generate responses.

alphabet is usually called Σ (pronounced sigma).

As an example, suppose the system (or computational process) which we are modelling has two pushbuttons, one labelled a, and the other b. Suppose that inputs to the system are given by pressing one or other of the buttons in turn. In this case we would probably define $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ to be our alphabet for our FSM (for modelling this system). We can also can think of the input to our FSM as a single stream of letters that can either be a or b (that is, a string over the alphabet Σ).

2. A description of the *responses* that the machine can generate. This is defined by giving a finite *output alphabet*. Most of the FSMs that we will give as examples will not have any output alphabet (they will not give any outputs). However, sometimes an output alphabet will be important, especially when we are using FSMs as a design tool for large systems (Lecture 5).

   The output alphabet is usually called Λ (pronounced lambda). For example, if the output alphabet for an FSM was $\Lambda = \{\mathsf{p}, \mathsf{q}\}$ we could imagine that the system has two lights on it (one labelled p, the other labelled q) and that system can turn them on and off independently, in response to given inputs, and depending on the current state of the system. Alternatively, we can think of the machine as having a single output stream that consists of a sequence of ps and qs.

3. We describe the machine as a diagram consisting of circles that stand for distinct states in the machine. When we consider FSMs that are *acceptors*, states with *double circles* will have a special interpretation .

   When we come to write down a *formal description* of FSMs, each of the distinct states will have a distinct name. We usually use Q to denote the set of all states of the FSM.

4. There is often a fixed *start state* which is thought of as being the initial state of the Finite State Machine (before any input has been read). This state may have different names, depending on the particular system being modelled.

   Diagrammatically, the *state state* is indicated by an arrow which points to the start state, but which is not connected to any other state.

5. The computational behaviour of the machine is described in terms of *transitions*, which are arrows leading from one state to another.

   When we consider FSMs which are *transducers* (which produce outputs), then each transition is labelled by some pair of symbols a/b where a is drawn from Σ and b is drawn from Λ. Suppose we have such an arrow *from* state q to state q'. The idea is that if the machine is in state q and it receives the input a, then the machine generates output b and changes its state to state q'.

   When we consider FSMs which are *acceptors*, then each transition is labelled only with a symbol from Σ, since there are no outputs.

   On some occasions (see Lecture 3 onwards) we will want to be able to take a transition when we have no input. In these cases we can replace the letters on the transition by the symbol ϵ (pronounced *ep-sil-on*) indicating that no symbol is being read from the input. Sometimes we omit ϵs in diagrams since we can always clearly see where we would have used an ϵ.

Here is a very simple example of a parking ticket machine. The input alphabet is $\Sigma = \{\mathtt{m}, \mathtt{t}, \mathtt{r}\}$ standing for: inserting `m`oney, requesting a `t`icket, and requesting a `r`efund. The output alphabet is $\Lambda = \{\mathtt{p}, \mathtt{d}\}$ standing for: `p`rint ticket, and `d`eliver refund. The transitions and start state of the machine are shown here.

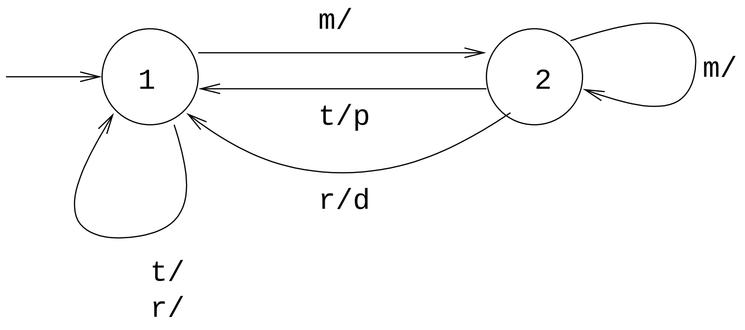

Figure 1.1: Parking meter FSM

The idea here is that the machine responds differently to a ticket request depending on whether any money has been deposited by the user of the system. If the user presses the button without inserting money then no ticket is printed. If money has been inserted then a ticket will be printed and the state changes back to the start state. So, the states capture the situation where no money has been inserted since the last ticket was printed and the situation where money has been inserted since the last ticket was printed.

Notice that the machine in Figure 1.1 is a *transducer*, because it produces outputs (not on all transitions, but on some of them).

1.3.1 Behaviour

In this section we investigate the type of behaviour a particular machine can exhibit. We do this by defining a *trace* of a finite-state machine. A *trace* for a Finite State Machine M is defined to be:

1. A finite sequence of alternating states and transition labels starting and ending with a state: $s_0, i_1/o_1, s_1, \ldots s_{n-1}, i_n/o_n, s_n$.
2. The first state, s_0, is the *start state* of M.
3. For every triple $s_{j-1}, i_j/o_j, s_j$ of the form (state, pair of input/output symbols, state) appearing in the sequence, there must be a transition in M from state s_{j-1} to s_j labelled i_j/o_j.

For example, one trace of the parking meter machine is: $1, \mathtt{m/}, 2, \mathtt{m/}, 2, \mathtt{m/}, 2, \mathtt{r/d}, 1$. This is the behaviour where the user deposits three coins and then decides to ask for a refund. The following is *not* a trace: $1, \mathtt{m/}, 1, \mathtt{m/}, 2, \mathtt{m/}, 2, \mathtt{r/d}, 1$ because the very first transition is not in the parking meter machine description (if it was and we kept the interpretation of the states then this would mean the machine could forget about money that had been deposited after the last ticket was printed). If we take a trace of a machine and drop the states from the trace then we get a possible sequence of stimulus/response pairs that could be generated by the machine. The overall behaviour of the machine is the collection of all traces that the machine could exhibit (notice that even for the parking machine this is an infinite set).

1.3.2 Transducers and Acceptors

When we are interested in the theoretical study of computation, we need to be quite precise about the type of behaviour we are interested in studying for FSMs. In the standard literature on Finite State Machines there are two main types of Finite State Machines, *transducers* and *acceptors*.

An FSM which is an *acceptor* can be thought of as defining a subset of the set of all sequences of the input alphabet Σ. For the class of FSMs which are acceptors, the output alphabet Λ is always empty, and therefore the *transitions* of the FSM are labelled with only one symbol (the input symbol which is read by that transition). The definition of a *trace* as the same as before except that the trace now looks like $s_0, i_1, s_1, \ldots s_{n-1}, i_n, s_n$.

When a FSM is an acceptor, we have an extra definition to make - the extension is to mark some states as being *accepting states*. Formally, this is done by defining the set F of accepting states (usually just by listing which states are accepting). In diagrams, accepting states are represented by double circles.

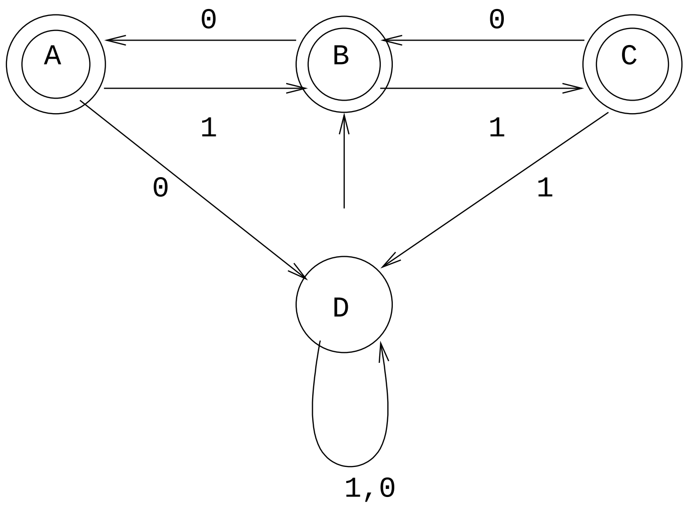

Figure 1.2: An acceptor

The FSM in figure 1.2 accepts sequences of `0`s and `1`s in which the number of `0`s never exceeds the number of `1`s by more than one and *vice versa*. In this case the sequence `0101101` is accepted because the trace corresponding to it has an accepting state (C) as its final state. The trace corresponding to this sequence is:

$$B, \mathtt{0}, A, \mathtt{1}, B, \mathtt{0}, A, \mathtt{1}, B, \mathtt{1}, C, \mathtt{0}, B, \mathtt{1}, C$$

By contrast, the sequence `0101001` is not accepted by the FSM because the trace corresponding to it is:

$$B, \mathtt{0}, A, \mathtt{1}, B, \mathtt{0}, A, \mathtt{1}, B, \mathtt{0}, A, \mathtt{0}, D, \mathtt{1}, D$$

and D is not an accepting state. *Failure to accept* a sequence can arise in two ways:

- We are able to construct a trace corresponding to the sequence but the final state in the trace is not an accepting trace.
- We cannot construct a trace for the given sequence. For example, if we omit state D in the above acceptor then it would be impossible to construct a trace for the sequence `0101001`.

The second case above only happens when the set of *transitions* is incomplete. That is, case two only happens if there is some (state, symbol) pair (q, i) for $q \in Q, i \in \Sigma$ such that no transition leaving q is labelled by i. We will discuss this further in Lecture 2.

For a given FSM M we say that the *language accepted by M* is the set of strings over the alphabet Σ that are accepted by M.

1.3.3 Deterministic Machines

In this discussion we just consider acceptors, but most of what we say here carries over to transducers (in some circumstances things can be a bit more delicate).

Nothing we have said so far implies that any two transitions originating at the same state *must be* labelled with different inputs. Neither have we implied that if there is a state q with an ϵ-transition leaving that state, then it is the case that that transition *must be* followed (without reading any input symbol). What this means is that when we have "competing transitions" (transitions labelled with the same inputs, or with ϵ-transitions) leaving the same state, there may be more than one trace corresponding to a particular sequence of inputs.

Finite State Machines of the kind described above are the most general types of Finite State Machines (ignoring the fact that we consider acceptors rather than transducers).

For any FSM M with an input alphabet Σ, we say that an input sequence from Σ^* is accepted if *at least one trace* for the sequences ends in an accepting state. In some circumstances we would like to know that there is *at most one* trace for any input sequence. If this is so then the FSM is called *deterministic*. Machines which are not deterministic are called *non-deterministic*.

In later years of your studies you will see there is a method for converting any non-deterministic acceptor to a deterministic acceptor. However, this may result in an exponential increase in the number of states.[2]

1.4 Summary

In this lecture note we have:

- Considered reactive systems.
- Given a fairly informal definition of Finite State Machines.
- Studied how the description of an FSM defines the *behaviour* of the FSM.
- Seen how FSMs can be considered to be acceptors or transducers.
- Considered the distinction between deterministic and non-deterministic FSMs.

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

Mary Cryan, October 2004

[2]This means if the non-deterministic machine has n states, the equivalent deterministic FSM may have as many as 2^n states. This is not always the case but it can happen.