Lecture 3

Inf1A: Non-deterministic Finite State Machines

3.1 Introduction

In this lecture we deal with *non-deterministic* Finite State Machines, a class of machines which are a generalization of *deterministic* Finite State Machines As in Lecture 2, we will mostly be concerned with Finite State Machines which are *acceptors* (ie, have no output alphabet). Therefore we will also be concerned with the type of languages that can be accepted by some Finite State Machine.

It can be shown that when we consider *acceptors*, the class of non-deterministic FSMs is the same as the class of deterministic FSMs. This result, which is known as the *Conversion Theorem*, is difficult to prove, so we will only hint at the reasons it is true (full proof next year!). The result means that when we are considering acceptors, we can always use non-deterministic FSMs rather than deterministic FSMs if we want to (because there is always some deterministic FSM which will accept the same language as our non-deterministic machine).

3.2 Formal Definitions

The main difference between non-deterministic Finite State Machines and the machines we have encountered so far in our lectures is that in an non-deterministic FSM, there may be input strings which have more than one *trace*.

The concept of non-determinism gives us more flexibility when we are modelling reactive systems: we can use it to help us describe systems whose behaviour we cannot completely predict. There is a price to pay for this flexibility - we will no longer have the nice property that for every input string there is exactly one computation path (trace) that can be predicted from the description of the FSM. This means (for example) that the implementation of a non-deterministic FSM in a programming language will not be a straightforward task.

However, our main interest in non-deterministic FSMs is in the study of formal languages, for which they are very valuable. Here is the definition of an (*acceptor-type*) non-deterministic FSM:

Definition 3.1 (Non-deterministic FSM)
A Non-deterministic Finite State Machine *(N-FSM) of the* acceptor *type is a machine M defined as a 5-tuple*

$$M = (Q, \Sigma, s_0, F, \Delta)$$

consisting of

1. *a finite set Q of* states
2. *a finite alphabet* Σ
3. *a distinguished* start state $s_0 \in Q$
4. *a set $F \subseteq Q$ of* accepting states, *and*
5. *a* description Δ *of all the possible* transitions *of the FSM.*

   The description Δ is no longer required to be a function *from $Q \times \Sigma$ to Q, as it was for D-FSMs. It is a more general description, that associates some* subset *of Q with every state from Q, for every "symbol" from $\Sigma \cup \{\epsilon\}$. Formally Δ is a function of the form $\Delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is the* power set *of Q.*

The symbol Δ is pronounced "del-ta", just like δ - but the symbol Δ is Capital "del-ta". It may be the case that some of you have not seen the power set notation $\mathcal{P}(Q)$ before. The notation $\mathcal{P}(Q)$ denotes the "set of all subsets of Q". So every *element* of the power set $\mathcal{P}(Q)$ is itself a set. For example, consider the FSM from Figure 2.4, which has the states $1, 2$ and 3. In this case the state set Q is $\{1, 2, 3\}$. Therefore the power set $\mathcal{P}(Q)$ is $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. The symbol $\emptyset$ denotes the empty set.

Another way of viewing Δ is as a subset of $Q \times (\Sigma \cup \{\epsilon\}) \times Q$, that is, a ternary relation, where a tuple (q, a, q') is in Δ if a transition from q to q' is possible on input a (if and only if $q' \in \Delta(q, a)$). Therefore, Δ is sometimes referred to as the *transition relation* of the N-FSM.

Notice that the set of all N-FSMs includes, as a subset, the set of all D-FSMs (because every deterministic FSM automatically satisfies the definition of an N-FSM). This means that the following definition holds for both D-FSMs and N-FSMs. Recall from Lecture 1 that we say that the string x is accepted by an FSM if there is *some* trace for x that ends in a state from F. We need to extend the definition of a trace for N-FSMs (because we allow ϵ-transitions as well as transitions that consume a symbol of the input alphabet).

Definition 3.2: (Trace)
Let $M = (Q, \Sigma, s_0, F, \Delta)$ be an N-FSM. Let $x \in \Sigma^*$ be an input string to the machine. We say that

$$s_0, i_1, s_1, i_2, \ldots, s_{n-1}, i_n, s_n$$

is a trace *for the string x if for every i_j, $1 \leq j \leq n$ we have $i_j \in \Sigma \cup \{\epsilon\}$, if x is equal to the* concatenation[1] *of $i_1, \ldots, i_n$, and if for every $1 \leq j \leq n$, $s_j \in \Delta(s_{i-1}, i_j)$.*

Definition 3.3 (x accepted by an N-FSM)
Let $M = (Q, \Sigma, s_0, F, \Delta)$ be any N-FSM. Let $x \in \Sigma^*$. Then the string x is accepted by *M if and only if there is* some *trace for x in M such that the final state s_n of that trace is an element of the set F of accept states.*

[1] We will explain concatenation in the next lecture note. It essentially means "merging" of $i_1, \ldots, i_n$.

Definition 3.4 (Language accepted by an N-FSM)
Let $M = (Q, \Sigma, s_0, F, \Delta)$ be any N-FSM. Then the language $L(M)$ accepted by the N-FSM M is

$$L(M) \;=\; \{x \in \Sigma^* \mid M \textit{ accepts } x\}.$$

Definition 3.4: (Regular languages)
Let L be any language over the alphabet Σ (that is, L is some subset of Σ^*). Then L is a regular language *if there is* some *FSM M such that $L(M) = L$.*

In Definition 3.4 we have been unclear about whether we consider N-FSMs or the more restricted class of D-FSMs in making our definition of regular languages. We will assume we are are working with the general class of N-FSMs (though in fact it does not matter).

3.3 Some examples

In general, Non-deterministic Finite State Machines have the advantage that it can often be easier to specify a language (or in the *transducer* variant of N-FSMs, to model a complex system). We now give a few examples where we see this in action.

Before we start, notice that there is more than one "type of non-determinism" that an N-FSM can possess. The first type happens when we have some state q and some symbol $a \in \Sigma$ such that the set $\Delta(q, a)$ contains *more than one element*. That models the case when we have at least two arrows leaving q labelled with a. That means that whenever we follow a partial computation to the state q and the next input symbol is a, there are at least two possible "next states".

The second type of non-determinism happens when we have some state q such that the set $\Delta(q, \epsilon)$ is non-empty. That models the case when we have at least one arrow labelled ϵ leaving the state q. Then whenever a partial computation arrives at q, it make a transition to another state without reading *any* input symbol.

3.3.1 Example

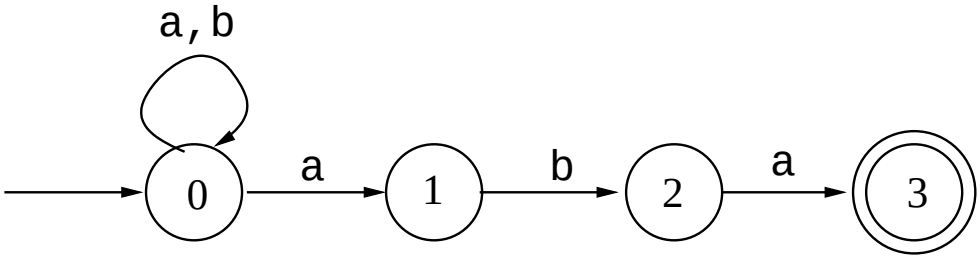

Figure 3.1: N-FSM to accept strings ending in aba

Figure 3.1 shows an N-FSM M to accept strings over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ which end in aba. The machine M has the formal description

$$M = \big(\{0, 1, 2, 3\}, \{\mathtt{a}, \mathtt{b}\}, 0, \{3\}, \Delta\big),$$

where the transition relation Δ is given by the following table:

Δ	a	b
0	$\{0,1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$

When we see the empty set symbol $\emptyset$ in the table, it tells us that for the given state and symbol, there is no possible transition to any state. Therefore there can be no trace through the given state reading the given symbol (though, since this is an N-FSM, there may be alternative paths/traces through the FSM).

If we look at the Finite State Machine in Figure 3.1, we can see that there are no transitions labelled ϵ (and therefore the table above does not need a column labelled ϵ. So for this example, the machine M only has one type of non-determinism.

Now examine the machine, and notice that any string of `a`s and `b`s is accepted by the machine if it passes through the sequence of states $1, 2$ and 3 at the very end of its trace. This happens if and only if the string ends in `aba`. Therefore the language accepted by the N-FSM of Figure 3.1 is

$$L(M) = \{x\texttt{aba} \mid x \in \{\texttt{a},\texttt{b}\}^*\}.$$

Notice that because the machine is non-deterministic, there may be many traces for a given string $x \in \{\texttt{a},\texttt{b}\}^*$. For example, it is an interesting task to count the number of traces for `ababa`.

3.3.2 Example

Next consider the language accepted by the deterministic FSM in Figure 1.2 of Lecture 1. This FSM accepts a certain language over the alphabet $\Sigma = \{\texttt{0},\texttt{1}\}$. We have already discussed the language accepted by the D-FSM of Figure 1.2 a little. We note that it is certainly the case that for every string x accepted by that D-FSM, the number of `0`s and of `1`s in x differ by at most 1. However, a stronger property must hold if x is to be accepted. If you re-examine Figure 1.2 and experiment with a few extra strings from $\Sigma = \{\texttt{0},\texttt{1}\}$, you will eventually work out that the language accepted is the set of all strings x over $\Sigma = \{\texttt{0},\texttt{1}\}$ such that x consists of a sequence of `01`s and `10`s, possibly with a `0` or `1` added at the end.

It takes some work to decipher which language is accepted by Figure 1.2, even when we have a diagram of the FSM. In Figure 3.2 we give an N-FSM which accepts exactly the same language, and we observe that it is much easier to decipher the language accepted by this particular machine. This is because we can represent the `01` and `10` choices by separate branches in the FSM, and use ϵ-transitions to move to either of them. Notice that our new machine has both types of non-determinism, as is clear from the Δ-table below Figure 3.2. However, the only times we have a choice of "next state" happens when we are reading an ϵ, so in some sense we are only really dealing with ϵ-non-determinism for this example.

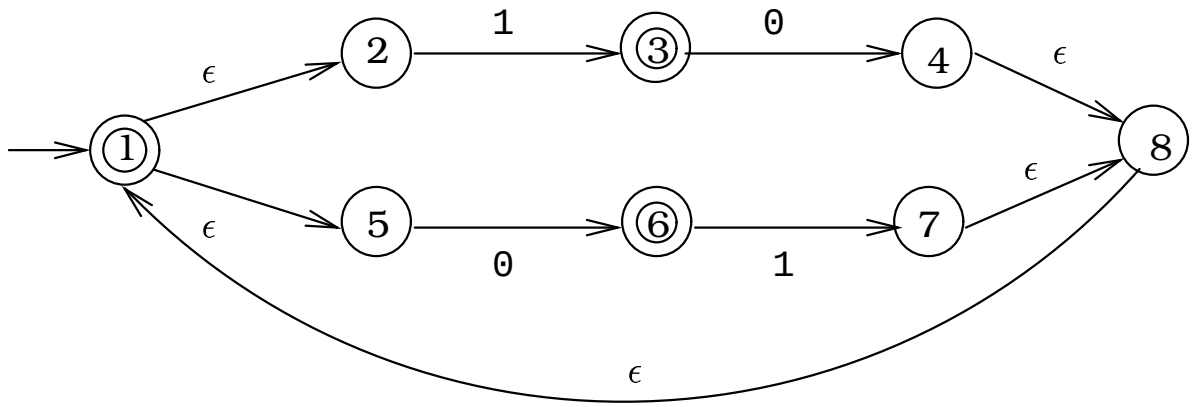

Figure 3.2: An N-FSM for the same language as Figure 1.2.

The transition relation Δ is given by the following table:

Δ	0	1	ϵ
1	$\emptyset$	$\emptyset$	$\{2,5\}$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\{4\}$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$	$\{8\}$
5	$\{6\}$	$\emptyset$	$\emptyset$
6	$\emptyset$	$\{7\}$	$\emptyset$
7	$\emptyset$	$\emptyset$	$\{8\}$
8	$\emptyset$	$\emptyset$	$\{1\}$

Some of the strings of the language accepted by M are

$$L(M) = \{\epsilon, \mathtt{0}, \mathtt{1}, \mathtt{01}, \mathtt{10}, \mathtt{010}, \mathtt{101}, \mathtt{1010}, \ldots\}$$

and so on. It is a worthwhile exercise to consider some longer sequences and check that they are accepted by Figure 3.2 if and only if they were accepted by the original machine in Figure 1.2 (since part of our goal was to check both machines accept the same language).

3.4 Building N-FSMs from the bottom up

We are now familiar with the definition of non-deterministic Finite State Machines and have some idea of the way they work. We have seen a couple of examples of these machines, and discussed how it is very important to test for correctness of the FSM.

We will now focus on the main advantage of N-FSMs (as opposed to D-FSMs), which is that they allow us to build large FSMs in a piecewise fashion, by fitting smaller FSMs together in an appropriate fashion. We will see that the ϵ-transitions allowed in an N-FSM will be very helpful for combining small machines into larger machines which perform more complex computations. Throughout this section we will assume that we are dealing with acceptors, to make life simpler. It will also make our life easier to assume that the FSMs that we are dealing with have just *one* accept state. Therefore we make the following observation:

Observation: Any FSM that has multiple accepting states can be converted to one with a single accepting state by connecting the existing accepting states to a new accepting state using ϵ-transitions and relabelling all the existing accepting states as normal states.

We now define some operations on FSMs and argue (but do not formally prove) that they are correctly defined. These operations will allow us to build up expressions that can be helpful in defining large FSMs. In the following diagrams we use ϵ transitions to join up machines. Whenever a smaller FSM is being used as a building block in a larger construction, we will represent the smaller FSM by an ellipse (with a label), and we will indicate the state state and the accept state of the small machine by the left-hand circle and the right-hand circle inside the ellipse respectively.

Sequence: To construct the machine that performs the computation of the N-FSM M_1 followed by the computation of the N-FSM M_2, we construct a new machine M with a new start state and a new accepting state. Then we

1. add an ϵ-transition from the new start state to the start state of M_1;
2. add an ϵ-transition from the accept state of M_1 to the start state of M_2;
3. add an ϵ-transition from the accept state of M_2 to the new accept state.
4. change the status of the original accept states of M_1 and M_2 so that neither of these are accept states any more.

Then the new machine M will accept any string which consists of a string belonging to $L(M_1)$ followed by a string belonging to $L(M_2)$. Formally, we say that $L(M) = \{xy \mid x \in L(M_1), y \in L(M_2)\}$. A diagram showing this construction is given in Figure 3.3. Any accepting trace of M_1M_2 will consist of an accepting trace of M_1 followed by an accepting trace of M_2.

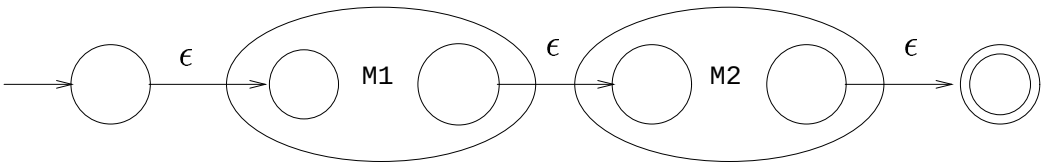

Figure 3.3: M_1M_2 — M_1 followed by M_2

Choice: We use the choice construction when we want to construct an N-FSM M which will choose to *either* perform the computation of the N-FSM M_1 *or* perform the computation of the N-FSM M_2, but not both. We construct M by adding a new start state and a new accept state, by adding an ϵ-transition from the new start state to the start state of M_1 and also to the start state of M_2, and by adding an ϵ-transition from the accept states of M_1 and of M_2 to the new accept state. Then we change the status of the accept states of M_1 and M_2 so they are no longer accepting.

A diagram showing this construction is given in Figure 3.4.

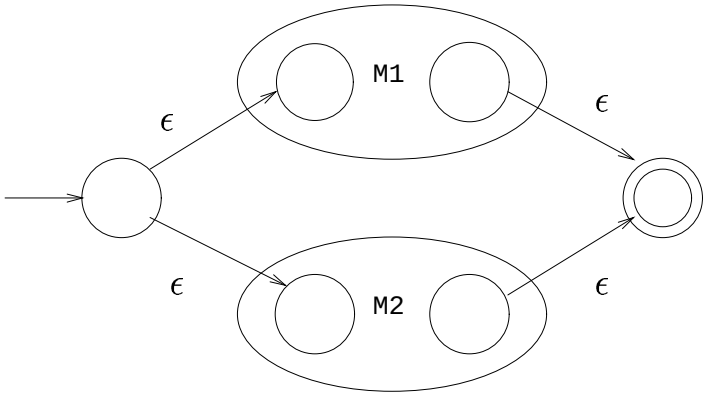

Figure 3.4: $M_1 \mid M_2$ — M_1 or M_2

Repeat: We use the repeat construction when we want to construct an N-FSM M^* which can perform the computation of a given N-FSM M as many times as necessary, including 0 times. The machine M^* is constructed by adding a new start state and accept state. Then we add an ϵ-transition from the new start state to the start state of M, from the original accept state of M to the new accept state, and also add a direct ϵ-transition from our new start state to our new accept state (this is to make sure that we can perform the computation 0 times and end in an accept state). We also add an ϵ-transition from the accept state of M to the start state of M, to allow iterations of the FSM M.

A diagram showing this construction is given in Figure 3.5.

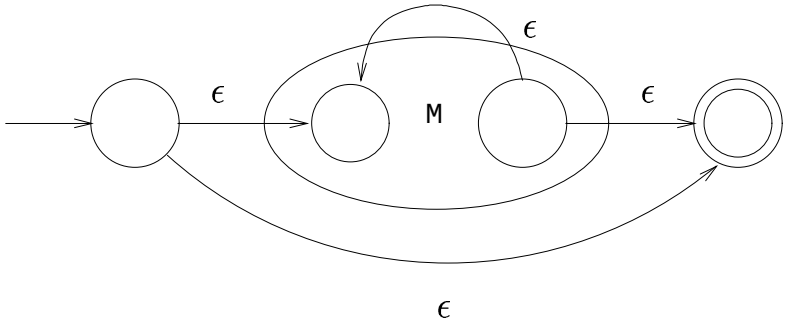

Figure 3.5: M^* — repeat M zero or more times

Intersection: Computing the intersection of the strings accepted by two machines, M_1 and M_2, requires more work than the previous examples. It is necessary to construct a new machine M that simulates *both* of the machines, at each step inspecting to see if a joint move is possible. It is possible to give a construction that works properly for N-FSMs but for this particular operation we will only describe the construction for FSMs without ϵ-transitions (just to make the construction slightly simpler). The construction of the machine M that recognises the intersection is achieved as follows:

- We define the set of states of the new machine to be *pairs of states*, one state from M_1, and one state from M_2. So if M_1 has p states and M_2 has q states the machine for $M_1 \cap M_2$ has pq states.

- The start state of the new machine is the pair of the start states of M_1 and M_2.
- There is only one accepting state of the new machine. It is the state which is the pair of the accepting states of M_1 and M_2.
- There is a transition labelled a in the new machine between state (p, q) and (r, s) if and only if there is a transition labelled a between p and r in M_1 *and* a transition labelled a between q and s in the M_2.

It is not very difficult to reason that a string x will be accepted by the intersection machine if and only if it is accepted by both M_1 and M_2.

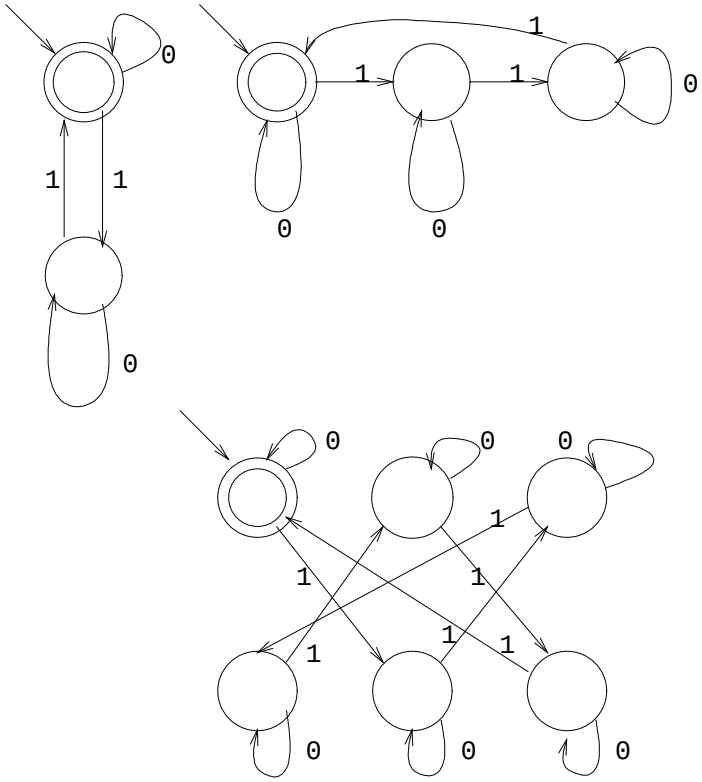

Figure 3.6: $M_1 \cap M_2$ — intersect M_1 and M_2

The example in Figure 3.6 illustrates the machine for the intersection of a particular two state FSM and a particular three state machine FSM. It's not so easy to give a diagram for the general case!

Interleaving: Now we consider the question of constructing a machine which performs the computations of two FSMs M_1 and M_2, but allows the computations to interleave in any order. This is a generalization of the **sequence** operation. Like **intersection**, the construction is a bit more complicated than our earlier three operations (it is also a bit less useful, this operation is not commonly used). We construct the FSM M from the two FSMs M_1 and M_2 as follows:

- Again, our set of states of M corresponds to *pairs of states*, one from M_1, and another from M_2.

- There is a transition labelled a in the new machine between state (p, q) and (r, s) if and only if there is a transition labelled a between p and r in M_1 and $q = s$, *or* a transition labelled a between q and s in the M_2, and $p = r$.

  The example in Figure 3.7 illustrates the machine for the interleaving of a particular two state FSM and a particular three state FSM.

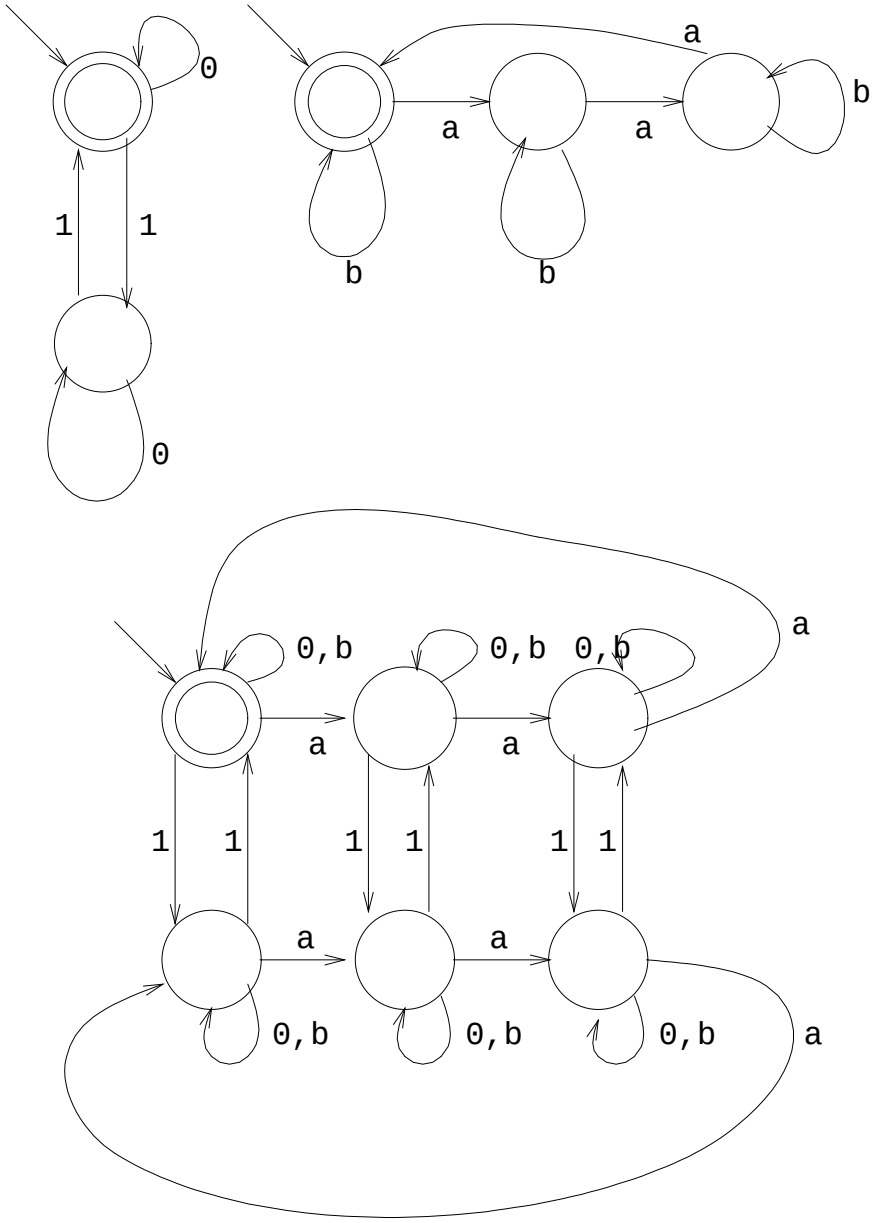

Figure 3.7: $M_1 \mid\mid M_2$ — interleave M_1 and M_2

Other Operations: The operations we have defined above certainly offer a good way of writing down complex FSMs while keeping control of the way they interact. It may be that to be really useful we would need other operations. One example is the idea of being able to interrupt the current activity to deal with some other activity and then return to the state the interrupt took place in when the interrupting activity is complete. Can you think of any others?

Now we have all these operations we should show how to use them. If we use x to stand for the simple two state machine that only accepts the letter x and so on for all the other letters then we can use the operations to combine these basic machines to give us structured descriptions of the FSM. From then on we can treat the detailed description as implementation detail and work with the structured definitions.

For example, the expression ((0 | 1)*11(0 | 1)*) represents the machine that accepts all sequences of 0s and 1s that contain two successive 1s. Using the definition of the operations we can build up the machine for this expression piece-by-piece. Figure 3.8 accepts any sequence of zeroes and ones. Figure 3.9 uses the machine of Figure 3.8 to build our final machine.

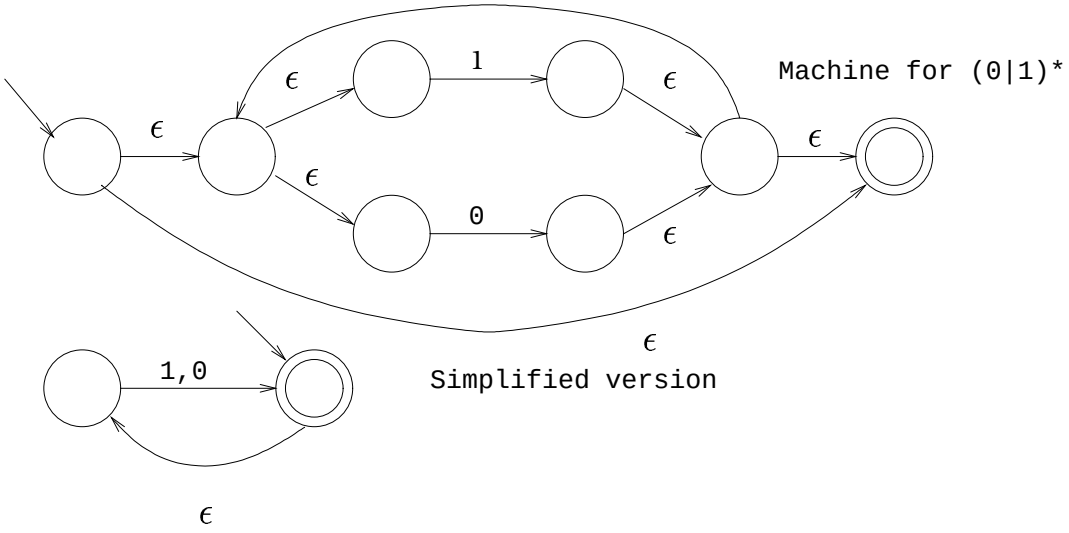

Figure 3.8: Machine for (0 | 1)*

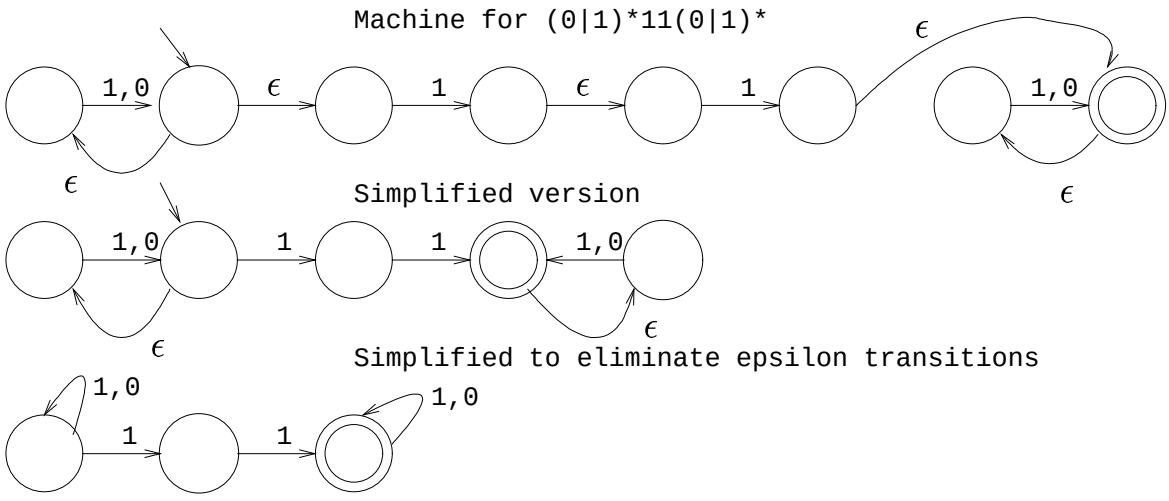

Figure 3.9: Machine for ((0 | 1)*11(0 | 1)*)

Notice that following this strict procedure for building FSMs means that we end up with a lot of extra ϵ-transitions and intermediate states that we do not really need. When we follow a strict of rules, we often end up with redundancy in the FSM.

3.5 D-FSMs versus N-FSMs

We have already mentioned that there is a *theorem* called the *Conversion theorem*, that proves that N-FSMs and D-FSMs recognise exactly the same class of languages, the class of *regular languages*. We are not going to prove this theorem in this course.

It is natural to ask why we bother to consider two different types of Finite State Machines, when they represent exactly the same class of languages? The reason is that the two frameworks have different advantages. The determinism of a D-FSM is important when implementing a program (or physical machine) for recognising a regular language. N-FSMs are not as well suited to language recognition because nondeterminism gives the machine the possibility of making choices. In general, when an N-FSM is directly used for language recognition, it is necessary to backtrack and explore all possible choices: a time consuming activity.

However, N-FSMs have the advantage that they are a lot easier to build, especially when we want to construct a FSM to accept a relatively complicated class of languages. So for the purposes of modelling languages (and often for modelling systems), they provide a more intuitive framework than D-FSMs.

3.6 Summary

In this lecture note we have

- Given a formal definition of a Non-deterministic FSM (N-FSM) and of the language accepted by an N-FSM.
- Given a couple of example N-FSMs which are simpler than corresponding D-FSMs to accept the same languages.
- Shown how N-FSMs can provide a framework for building Finite State Machines from the bottom up.

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