Notes on Finite State Machines

5th edition

David Knight

July 25, 2013

Foreword

These notes provide an introduction to the theory of finite state automata, and regular expressions, and show how the two are related.

The notes cover deterministic and non-deterministic automata, and show how a non-deterministic automaton can be converted to an equivalent deterministic automaton.

The notes also show how to realise a deterministic finite automaton as a program in any imperative programming language, such as Java, C, or assembly-code. In addition, the notes show how to use the object-oriented design pattern known as "state".

The chapter on regular expressions shows how an expression can be transformed into a non-deterministic finite automaton, and thus realised as a program.

The development of these notes has been strongly influenced by the book: *Introduction to Automata Theory, Languages, and Computation*, by John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. If you find you would like further information on the material here, their book is highly recommended.

I thank my son, Alistair Knight, for preparing the diagrams in these notes. I also thank SangAh Lee for finding and reporting numerous typographic errors.

Undoubtedly errors remain in the text and diagrams. If you spot an error, I would appreciate your assistance in correcting it.

David Knight
25-Jul-2007

Contents

Chapter 0

Introduction

Most of your programming experience will have been acquired by writing conventional data-processing style programs that read data, process it, and generate output. Programs that simply read and write files are among the oldest uses of computers, and are known as *batch* programs — the program runs, computes and, when it is finished, the result is available.

If the program interacts with a user, and asks for data as it is needed for the computation, it is called an *interactive* program. Interactive programs were the next stage in the evolution of computers and software. You have probably written many interactive programs during your coursework so far.

However, many programs in the real world need to be able to process data as it arrives at the system. These programs are called *reactive* or *event-driven* because they need to be able to react to the data as soon as it arrives.

A familiar example of a reactive program is any program with a Graphical User Interface (GUI). Such a program does not wait for a specific button to be pressed, but is instead ready to respond to *any* button that is pressed.

Another application of reactive programs is embedded systems. These are systems that control electrical appliances such as DVD players, air conditioners, toasters, mobile phones, burglar alarms, car fuel injection systems, and many others. More than 99% of all microprocessors made in the world are hiding inside an embedded system.

Reactive programs have three main characteristics:

- *They are driven by the availability of data.* When a particular piece of data becomes available for processing, we say that an event occurs (hence event-driven programs). From a software viewpoint, the order in which events can occur is unpredictable — e.g. a DVD player cannot predict which button a user will press next. This means that the number of execution paths in a reactive program is *much* larger than in an interactive program. This fact alone makes writing correct code much more difficult (we have all experienced GUI programs that crash for no obvious reason).

- *They are concurrent.* Reactive systems are usually carrying out multiple activities at the same time. For example, a single web browser process can have multiple windows opened where a different download is in progress in each window. While doing this, the browser still responds to user mouse-clicks and key-strokes. From the viewpoint of the web browser, the arrival of data for each web-page or of a mouse-click is simply an event that can occur at any time.

- *The user is always in control.* While the program is processing data, it *cannot* choose to ignore the user, rather it *must* respond to the user's events (mouse-clicks, or key-strokes) in a timely fashion.

0.1 What are event-driven systems?

As noted above, reactive or event-driven systems can cover a broad spectrum of applications. The most common are:

Graphical User Interfaces (GUIs) It is *very* difficult to write correct code for GUIs without using event-driven techniques. In this course, we will show you how to construct *bug-free* GUIs.

Embedded Systems If you are an engineer, you will almost certainly be involved in designing or implementing an embedded control system for a product. The reason is simple: a microcontroller is the cheapest and most flexible way to implement the control section of almost every electronic product. Those of you who are studying the courses "Software Engineering and Project" or "Embedded Computer Systems" will find that knowledge of event-driven systems will significantly simplify the software for your projects.

Communication Protocols Modern computer systems are extensively networked, to allow them to exchange data. All computer communication depends on a protocol for the exchange of data. Communication protocols need to be able to respond to events, such as: arrival of a message, arrival of an acknowledgement, timeout after sending a message, and so on. All these events can occur in any order. A protocol can *only* be implemented reliably using event-driven programming techniques.

A "protocol stack" is a layered set of intercommunicating protocols that together permit computer-to-computer communication. Each layer in the stack is a reactive program. A protocol stack is thus, by its very nature, *highly* reactive, and *highly* concurrent.

Despite the variety of applications for event-driven systems, there are some well understood techniques for designing and implementing these systems. They can significantly reduce the time, effort and the likelihood of bugs when writing your code. They also will make your program *much* easier to change in the future.

By making clear design and implementation decisions at the beginning, you will retain intellectual control over the complexity inherent in all reactive systems.

0.2 A little history

The ideas behind finite state machines have been around for a long time now. In the 1930s, Alan Turing did research on computable functions, and invented the "Turing machine", a simple processor that has been proved to have the same capabilities as any current-day computer.

In the 1940s, McCulloch and Pitts modelled the behaviour of nerve networks, using ideas similar to those presented in these notes.

In the mid 1950s, papers by Mealy and Moore, from the Engineering community, had a big influence on the design of switching systems in telephone exchanges. Both described finite state machines as we now know them. We will look at the nature of their work a little later on.

At the end of the 1950s, Rabin and Scott published a paper that introduced non-deteministic finite state machines, and provided fruitful insights for further work. They received the ACM Turing award for this work.

0.3 An introductory example - the tap-light

Let us consider a simple but familiar example: an electric table lamp that is controlled by simply "tapping" it: when a user taps the lamp, it turns on; tap again, and it turns back off; tap again, and it turns back on; and so on. We can represent this behaviour in a *state-diagram*, as shown in fig. 1.

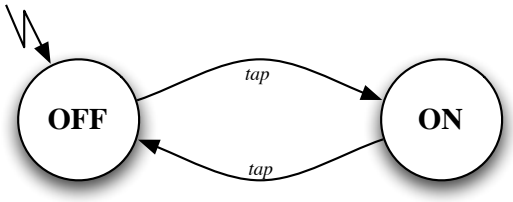


Figure 1: The tap-on/tap-off table light

In the diagram, the two circles, named off and on, represent the state of the light. The state of the light can be changed by the occurrence of an event, named *tap*. We show a transition from one state to another by a directed arc (A line with an arrow on one end) from one state to another, labelled with the name of an event. There is a special transition, shown as a "lightning-strike", that indicates the starting state of the diagram. (In this example, the starting state is named off.) A diagram such as this is called a *finite state machine*(FSM).

This state-diagram very neatly and succinctly captures the behaviour of the light: Initially, the light is off, and the FSM is in state off. Upon receipt of a tap event, the system changes state to on, indicating that the light is now on. The FSM is now in state on. Upon receipt of a tap event, the system changes state to off, indicating that the light is now off. The FSM is now in state off.

We can show the current state by putting a *dot* inside it. For example, immediately after starting this FSM, the diagram will appear as shown in fig. 2.

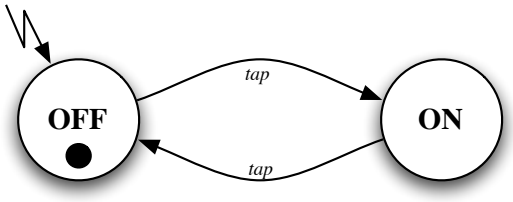


Figure 2: The tap-light in the *OFF* state

All this behaviour can be understood just by putting your finger on the current state (the one with the dot) and, upon receipt of an event, following the appropriate transition to the next state. By reading the diagram, it is easy to see what will happen in every situation.

Consider our previous diagram, with the tap-light in the *OFF* state. If we receive a tap event, the diagram changes to the *ON* state, as shown in fig. 3.

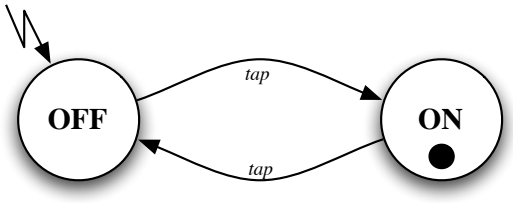


Figure 3: The tap-light in the *ON* state

0.3.1 Some nomenclature

A finite state machine has one or more *states*, represented on a state-diagram as circles. As a result of an external *event*, the FSM can make a *transition* from one state to another (possibly the same) state. A transition is represented on the diagram as a directed arc, labelled with the name of the event that causes the transition to be taken. The FSM has an *initial state*, represented on the diagram by a lightning-strike. The *current state* is represented by a large dot inside the state.

0.4 Another example - text recognition

We consider here another example, that arises in text-processing programs, such as a compiler. Suppose we wish to recognise the words "for", and "float", in a stream of text. We could construct a finite state machine, where the sequence of events is the sequence of characters in the input stream. The resulting FSM is shown in figure 4.

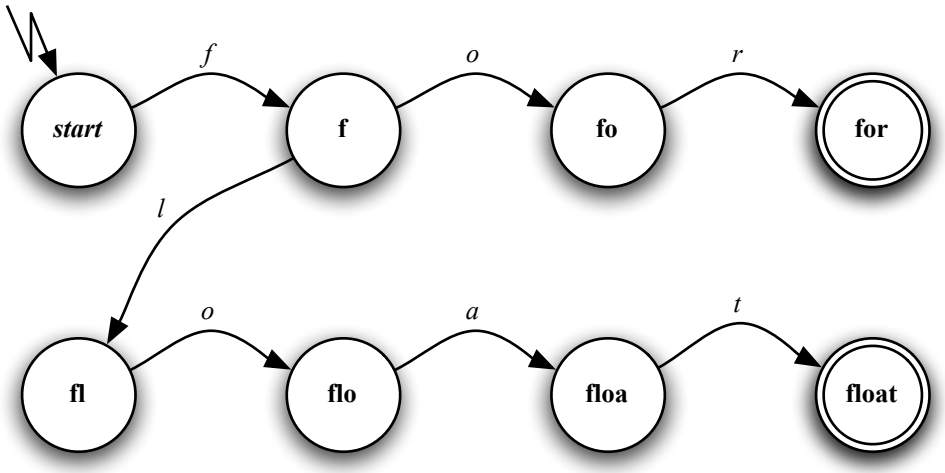


Figure 4: The text recogniser

In the FSM there are states named after the partially-recognised words: f, fo, fl, flo, and $floa$. There are also two *accepting* states named for and $float$, shown on the diagram as double-circled states. It is obvious that for each state, there are many possible input characters that will *not* be recognised. For example in state fo, the machine will not recognise the character x, because there is no transition labelled with the event x, from state fo. If an unrecognised event occurs, the machine "dies", and ceases to process symbols — it no longer has a *current state*. Effectively, the *dot* has been lost.

Diagrams of this kind often result from describing the behaviour of a *regular expression*, a topic we will deal with later.

0.5 Central concepts

There are a number of basic concepts that are central to the study of finite automata. These concepts are *alphabet* (a set of symbols), *string* (a list of symbols from an alphabet), and *language* (a set of strings from the same alphabet).

0.5.1 Alphabets

An *alphabet* is a finite, non-empty set of symbols. It is conventional to use the Greek letter Σ (sigma), to represent an alphabet. Some examples of common alphabets are:

1. $\Sigma = \{0, 1\}$, the set of binary digits.
2. $\Sigma = \{A, B, \cdots, Z\}$, the set of Roman letters.
3. $\Sigma = \{N, E, S, W\}$, the set of compass-points.

0.5.2 Strings

A *string* is a finite sequence of symbols drawn from an alphabet. A string is also sometimes called a *word*. Some examples of strings are:

1. 100101 is a string from the binary alphabet $\Sigma = \{0, 1\}$.
2. $THEORY$ is a string from the Roman alphabet $\Sigma = \{A, B, \cdots, Z\}$.
3. SE is a string from the compass-points alphabet $\Sigma = \{N, E, S, W\}$.

Empty string

The *empty string* is a string with no symbols in it, usually denoted by the Greek letter ϵ (epsilon). Clearly, the empty string is a string that can be chosen from any alphabet.

Length of a string

It is handy to classify strings by their *length*, the number of symbols in the string. The string $THEORY$, for example, has a length of 6. The usual notation for the length of a string s is $|s|$. Thus $|THEORY| = 6$, $|1001| = 4$, and $|\epsilon| = 0$.

Powers of an alphabet

We are often interested in the set of all strings of a certain length, say k, drawn from an alphabet Σ. This can be constructed by taking the *Cartesian product*, of Σ with itself k times: $\Sigma \times \Sigma \times \cdots \Sigma$. We can represent this symbolically, using exponential notation, as Σ^k.

Clearly $\Sigma^0 = \{\epsilon\}$, for any alphabet Σ, because ϵ is the only string whose length is zero.

For the alphabet $\Sigma = \{\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}\}$, we find:

$$\Sigma^1 = \{\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}\}$$

$$\Sigma^2 = \{\mathrm{NN}, \mathrm{NE}, \mathrm{NS}, \mathrm{NW}, \mathrm{EN}, \mathrm{EE}, \mathrm{ES}, \mathrm{EW}, \mathrm{SN}, \mathrm{SE}, \mathrm{SS}, \mathrm{SW}, \mathrm{WN}, \mathrm{WE}, \mathrm{WS}, \mathrm{WW}\}$$

$$\Sigma^3 = \{\mathrm{NNN}, \mathrm{NNE}, \mathrm{NNS}, \cdots, \mathrm{WWS}, \mathrm{WWW}\}$$

Σ^3, has 64 members, since it contains $4 \times 4 \times 4$ members.

The set of *all* strings that can be drawn from an alphabet is conventionally denoted, using the so-called *Kleene star*, by Σ^*, and of course has an infinite number of members. For the alphabet $\Sigma = \{0, 1\}$

$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, \cdots\}$

Clearly, $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$

Sometimes we do not want to include the empty string in the set. The set of *non-empty* strings is denoted by Σ^+. This is often referred to as the *Kleene plus*, by analogy with the Kleene star.

Clearly, $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$

And $\Sigma^* = \{\epsilon\} \cup \Sigma^+$

Concatenating strings

Let s be the string composed of the m symbols $s_1 s_1 s_2 \cdots s_m$, and t be the string composed of the n symbols $t_1 t_1 t_2 \cdots t_n$. The *concatenation* of the strings s and t, denoted by st, is the string of length $m + n$, composed of the symbols $s_1 s_1 s_2 \cdots s_m t_1 t_1 t_2 \cdots t_n$.

It is clear that the string ϵ can be concatenated with any other string s and that: $\epsilon s = s\epsilon = s$. ϵ thus behaves as the *identity value*, for concatenation.

0.5.3 Languages

A set of strings, all of which have been chosen from Σ^* of an alphabet Σ, is called a *language*. If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is said to be a *language over* Σ.

A language over Σ does *not* need to include strings with *all* the symbols of Σ. The implication of this is that when we know that L is a language over Σ, then L is also a language over any alphabet that is a *superset* of Σ.

The use of the word "language" here is entirely consistent with everyday usage. For example the language "English" can be considered to be a set of strings drawn from the alphabet of Roman letters.

The programming language *Java*, or indeed any other programming language, is another example. The set of syntactically-correct programs is the set of strings that can be formed from the alphabet of the language (the ASCII characters).

Using the alphabets we defined earlier, we can specify some languages that might be of interest to us:

1. The language consisting of valid binary byte-values (a string of 8 0's or 1's): $\{00000000, 00000001, \cdots, 11111111\}$ This is just Σ^8.
2. The set of even-parity binary numbers (having an even number of 1's), whose first digit is a 1: $\{11, 101, 110, 1001, 1010, 1100, 1111, \cdots, \}$
3. The set of valid compass directions: $\{\mathrm{N}, \mathrm{S}, \mathrm{E}, \mathrm{W}, \mathrm{NE}, \mathrm{NW}, \mathrm{SE}, \mathrm{SW}, \mathrm{NNE}, \mathrm{ENE}, \cdots\}$
4. Σ^* is a language over an alphabet Σ.
5. $\{\epsilon\}$, the language consisting only of the empty string, is a language over any alphabet. This language has just one string: ϵ.
6. $\emptyset$, the language with *no* strings, is a language over any alphabet. Note that $\emptyset \neq \{\epsilon\}$, because $\{\epsilon\}$ contains *one* string.

Notice also that an alphabet Σ is always of a finite size, but a language over that alphabet can either be of finite or of infinite size.

Chapter 1

Deterministic finite automata

In this chapter we look at *deterministic finite-state automata* (DFAs). we begin with an example, to get the "feel" of the transition-diagram representation of DFAs. We then examine a more formal definition of a DFA, formalise the definition of a transition diagram, and introduce transition tables.

1.1 Introduction to transition diagrams

Our example is a DFA, which we will name $zeroOne$, that recognises and accepts every string of 0s and 1s that ends with 01. Any other string of binary digits is to be rejected.

We can specify the language, L, of $zeroOne$ as:

$$L = \{s \mid s \text{ is any string of 1s and 0s, ending with } 01\}$$

or equivalently:

$$L = \{s01 \mid s \text{ is any string of 1s and 0s}\}$$

Examples of strings that are in the language, L, are: 01, 00000001, 0101, 1101101. These strings must be "accepted" by the $zeroOne$ DFA.

Examples of strings that are *not* in L are: 000, 10, 110, and ϵ. These strings must be "rejected" by the DFA.

A state in a DFA "remembers" things about the symbols that the DFA has previously seen. Considering our problem, it is clear that we are prepared to accept a lot of bits, (possibly an infinite number) but eventually we must receive a 0, and then a 1. We therefore need to be able to remember "we have received a 0", and also "we have received a 0 followed by a 1". This suggests that $zeroOne$ will have at least two states. In addition, we need a state to remember "nothing useful has happened yet". So it looks like there will be three states.

Let us name the states like this: q_0 remembers "Nothing useful has been received"; q_1 remembers "A 0 has been received"; and q_2 remembers "A 01 has been received".

We show the states as circles, with the name inside the circle. Clearly, q_0 is the starting state, which we show with a "lightning-strike" symbol. The accepting state is q_2, which we show with a double circle. Our diagram presently appears as shown in fig. 1.1.

We can now consider the transitions. If we are in state q_1 (we have previously received a 0), and we then receive a 1, we have correctly found a string that ends in 01, so we should enter state q_2. Thus there should be a transition from q_1 to q_2, labelled with 1.

If we are in state q_1, and we receive a 0, then we must still wait for a 1. Thus there should be a transition from q_1 to q_1, labelled with 0.

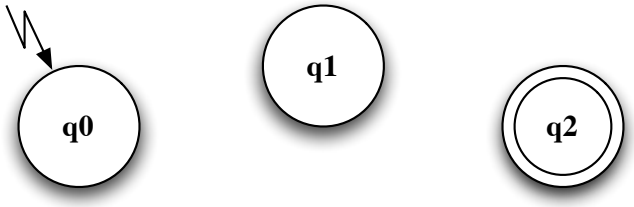


Figure 1.1: the $zeroOne$ DFA states

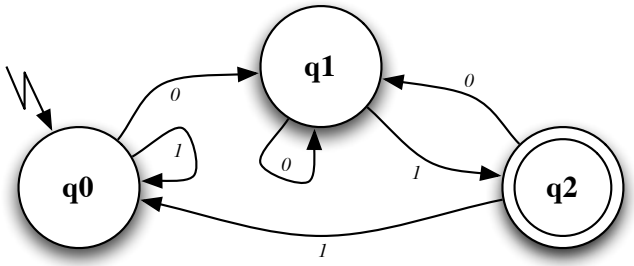


Figure 1.2: the $zeroOne$ DFA transition diagram

If we are in the starting state, q_0, and we receive a 0, this is good news, we should now enter state q_1. Thus there should be a transition from q_0 to q_1, labelled with 0.

If we are in the starting state, and receive a 1, this is no use to us, so we just remain where we are. Thus there should be a transition from q_0 to q_0, labelled with 1.

To be thorough, we must make sure that we know what happens in every state, otherwise $zeroOne$ will "die" for some input strings. If we are in state q_2, and we receive a 0 , we now have a string that ends in 010. This can become an acceptable string, provided we next receive a 1, so we see that we are now in state q_1. Thus there should be a transition from q_2 to q_1, labelled with 0.

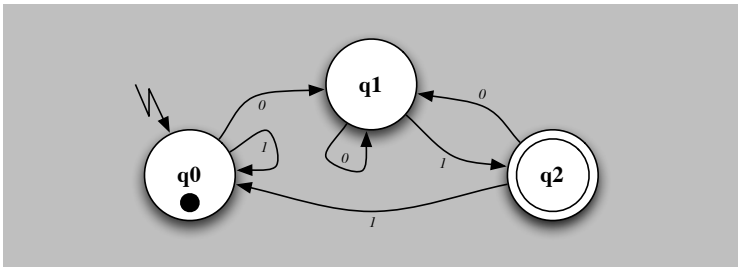
If we are in state q_2, and we receive a 1, we now have a string that ends in 011. This can only be accepted if we receive a 0 followed by a 1, so we must return to state q_0. Thus there should be a transition from q_2 to q_0, labelled with 1.

The complete transition diagram that corresponds to the $zeroOne$ DFA is shown in figure 1.2.

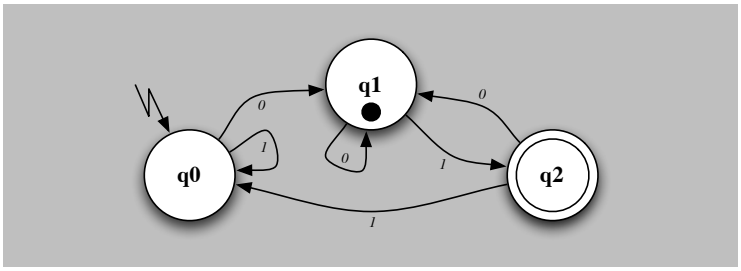
1.1.1 Example — the zeroOne DFA processing a string

We show here how the zeroOne DFA processes the string 011001.

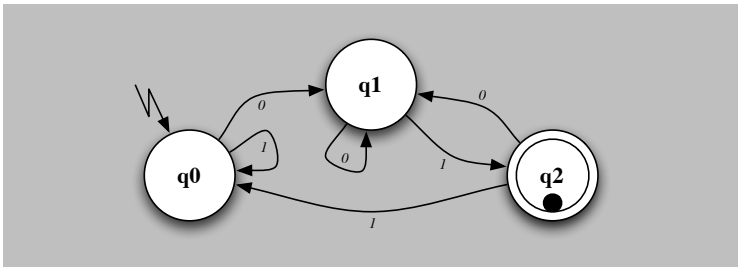
The DFA begins in its initial state, q_0.



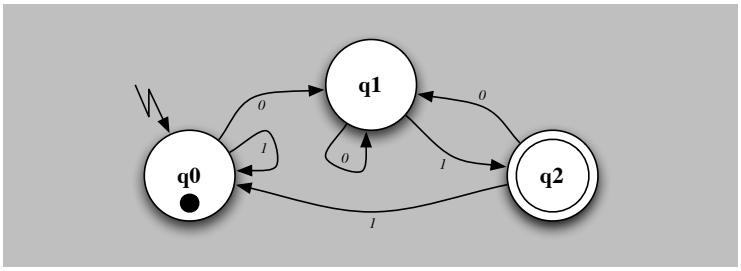
After processing the symbol 0, The DFA changes to state q_1.



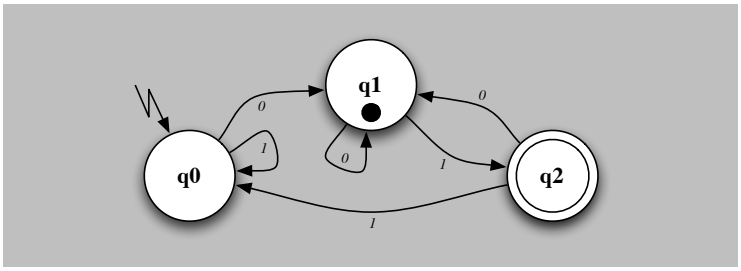
After processing the symbol 1, the DFA advances to state q_2. This is an accepting state, since the DFA has received a string that ends in 01.



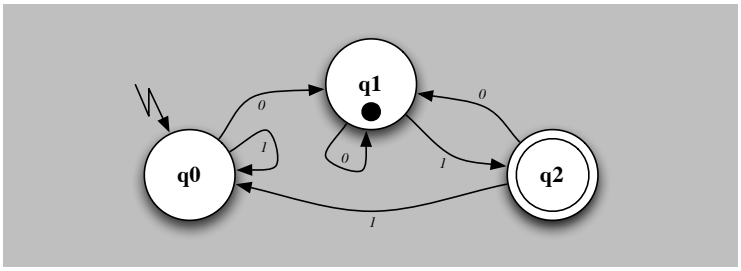
After processing the symbol 1, the DFA returns to state q_0.



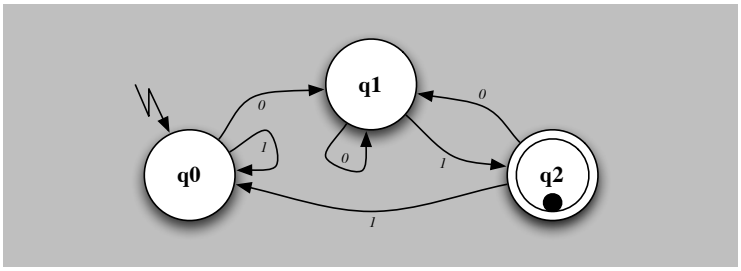
After processing the symbol 0, the DFA changes to state q_1.



After processing the symbol 0, the DFA remains in state q_1.



After processing the symbol 1, the DFA advances to state q_2. This is an accepting state, since the DFA has received a string that ends in 01.



1.2 Formal definition of a deterministic finite automaton

Specifying a DFA by means of a transition diagram is very convenient, and very easy for humans to interpret. However, two other notations are commonly used for describing DFAs:

- A *five-tuple*, a formal mathematical notation that defines a DFA as consisiting of a set of states, a set of input symbols, an initial state, a transition function, and a set of final states.
- A *transition table*, a table showing the values of the transition function, δ, for every combination of state and input symbol.

Let us now look at these alternative representations.

1.2.1 Five-tuple definition of a DFA

A *deterministic finite automaton* consists of:

1. A finite set of *states*, usually denoted by Q
2. A finite set of *input symbols*, usually denoted by Σ.
3. A *transition function* $\delta : Q \times \Sigma \rightarrow Q$, that takes a state and an input symbol, and returns a new state. The transition function is usually denoted by δ.
4. An *initial-state* $q_0 \in Q$. q_0 is one of the states in Q.
5. A set of *final* or *accepting* states $F \subset$Q, usually denoted by F. Clearly, F is a subset of Q.

A deterministic finite automaton ("DFA" for short), named M can thus be represented by a 5-tuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where Q is the set of states, Σ is the set of input symbols, δ is the transition function, q_0 is the initial state, and F the set of final states.

If we examine the $zeroOne$ example given earlier, we can write down the definition of the transition function, δ, simply by looking at each of the transitions. For example: From the transition between state q_1 and q_2, we conclude $\delta(q_1, 1) = q_2$. From the transition between state q_2 and q_0, we conclude $\delta(q_2, 1) = q_0$. Proceeding in a similar way we find: $\delta(q_1, 0) = q_1$, $\delta(q_0, 0) = q_1$, $\delta(q_0, 1) = q_0$, and $\delta(q_2, 0) = q_1$

1.2.2 Transition-diagram definition of a DFA

A *transition diagram* for a DFA $M = (Q, \Sigma, \delta, q_0, F)$, is a graph defined according to these rules:

- For every state in Q, there is a node in the graph.
- For each state, $q \in Q$, and each input symbol, $s \in \Sigma$: if $r = \delta(q, s)$ exists, then there is an arc from the node representing state q to the node that represents state r, labelled with the symbol s. To avoid clutter on the diagrams, when there are several symbols $s_1, s_2, \cdots, s_k$ for which $r = \delta(q, s_i)$, we can draw one arc and label it with a list of the symbols s_i.
- There is an arrow (drawn as a "lightning-strike") pointing into the node that represents state q_0. This arrow does not originate from any node. Its purpose is to show the initial state of the system.
- Nodes that correspond to accepting states are drawn with double circles. Nodes for all other states have a single circle.

1.2.3 Transition-table definition of a DFA

A *transition table* is simply a complete tabulation of the values of the transition function, for all states $q \in Q$, and all symbols $s \in \Sigma$. Conventionally, the rows correspond to the states, and the columns to the input symbols. The entry corresponding to the row q and the columns s gives the value of the new state, $\delta(q, s)$. The initial state is marked with an arrow, $\rightarrow$, and accepting states are marked with a star, $*$.

The previous $zeroOne$ DFA can be represented in tabular form like this:

$zeroOne$

δ	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
$*q_2$	q_1	q_0

1.3 How a DFA processes a string

To understand how a DFA decides whether to "accept" a string of symbols, we need to see how it processes a string. The set of all strings that a DFA accepts is called its *language*.

Suppose $s_1 s_2 s_3 \cdots s_n$ is a sequence of input symbols drawn from Σ, (the set of input symbols that the DFA can process). We start with the DFA in its initial state, q_0, and invoke the transition function δ to process the first input symbol: $\delta(q_0, s_1) = q^{'}$. The transition function returns a new state, $q^{'}$, that the DFA enters after processing the symbol s_1. The next symbol is processed similarly: $\delta(q^{'}, s_2) = q^{''}$, yielding a new state $q^{''}$. Continuing in this manner until all input symbols have been processed, we will (step-by-step) enter states $q^{'''}, q^{''''}, \cdots, q^{(n)}$. At each step, $\delta(q^{(i-1)}, s_i) = q^{(i)}$.

If $q^{(n)}$, the state reached after processing all the input symbols, is a member of the set of final states (i.e. $q^{(n)} \in F$) then the string of input symbols $s_1 s_2 s_3 \cdots s_n$ is "accepted", otherwise it is "rejected".

1.4 Extending the transition function to strings

The transition function allows us to specify how a DFA in a state, q, will move to state, r, upon receipt of a single symbol, s. We show this by writing $r = \delta(q, s)$.

We are interested in knowing what state a DFA will reach, starting at q_0, if it is presented with a *string* of symbols $w = s_1 s_2 \cdots s_n$.

We can specify an *string transition function* $\hat{\delta}$, that shows how a DFA that starts in any state, q, and receives a sequence of symbols, w, ends up in state r: $r = \hat{\delta}(q, w)$, like this:

$$\hat{\delta} : Q \times \Sigma^* \rightarrow Q$$

We can define the behaviour of $\hat{\delta}$ recursively, like this:

Base case If there is no input, the DFA stays in the current state, q. Formally:

$$\hat{\delta}(q, \epsilon) = q$$

Recurrence case If we are given a string of symbols $s_1 s_2 s_3 \cdots s_n$, we "chop" the first symbol s_1 from the string, then use δ to compute the state that would be reached, starting at state q:

$$r = \delta(q, s_1)$$

We then use the $\hat{\delta}$ function to process the remainder of the string, starting at state r: $\hat{\delta}(r, s_2 s_3 \cdots s_n)$. Combining the two pieces, we get:

$$\hat{\delta}(q, s_1 s_2 s_3 \cdots s_n) = \hat{\delta}(\delta(q, s_1), s_2 s_3 \cdots s_n)$$

We can see that a recursive evaluation according to these rules *must* terminate in a finite number of steps, because each recursive invocation of $\hat{\delta}$ occurs with a shorter string of symbols. Eventually, we *must* reach the base case: $\hat{\delta}(q, \epsilon)$.

1.5 The language of a DFA

It is now a simple matter to define the language of a DFA. The language is the set of all input strings that take the DFA from its initial state to an accepting state. Thus, for the DFA M, defined by:

$$M = (Q, \Sigma, \delta, q_0, F)$$

The language $L(M)$ of this DFA is defined by:

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

Chapter 2

Non-deterministic automata

The automata we have studied so far are called Deterministic Finite-state Automata (DFA) because when the DFA receive an input it is able to precisely determine its next state. *Deterministic behaviour is a highly desirable characteristic*, because it guarantees that we can implement a DFA as a program on a computer.

A Non-deterministic Finite state Automaton (NFA), has the property that it can be in more than one state at any one time. This (apparently weird) property allows us to express many automata more compactly than as a DFA.

Non-determinism appears to be an undesirable property (if the NFA cannot determine which state to enter next, what is it to do?) but it is not. It turns out that every NFA can be converted to an equivalent (though sometimes much larger) DFA, so NFAs can *always* be implemented, though with a bit more effort.

The attraction of the NFA is that we can more-easily *specify* the desired behaviour. Then, using a mechanical procedure, we can convert the NFA to a DFA, and thus implement it. This two-step approach neatly simplifies the overall task.

2.1 An informal view

In most respects, an NFA is the same as a DFA: It has a finite number of states; it accepts a finite set of input symbols; it begins in an initial state; and it has a set of accepting states. There is also a transition function, δ. Recall that in a DFA, $\delta(q, s)$ takes the current state, q, the current input symbol, s, and returns the next *state* to enter.

In an NFA, $\delta(q, s)$ takes a current state, q, the current input symbol s, and returns a *state-set* (a set of states) that the NFA enters. The set can contain zero, one, or more states.

Consider the $zeroOne$ recogniser described earlier. We can express this as an NFA as shown in fig. 2.1.

There are several things to notice about this diagram:

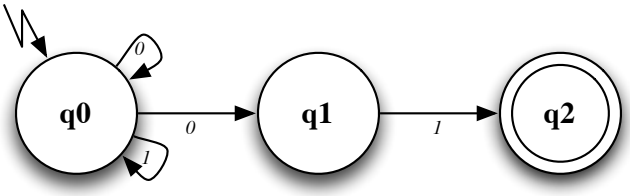


Figure 2.1: The zeroOne recogniser as an NFA

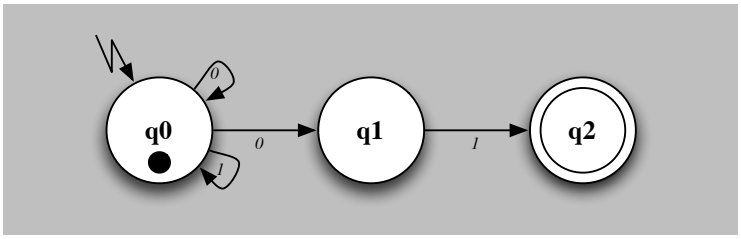
- The diagram has fewer transitions than the earlier DFA, so it appears simpler.
- There are *two* transitions from state q_0, labelled 0. Thus when a 0 is received, the NFA enters *both* state q_0 *and* state q_1. We will see how to think about this situation shortly.
- There is no transition corresponding to 0 from state q_1, and no transitions at all from state q_2. If these situations occur, the thread of the NFA's existence that corresponds to these states simply "dies". Other threads may continue to exist.

The central idea of this NFA is to try to "guess" when the final 01 has begun. Whenever it is in state q_0, and it sees a zero, it guesses that the 0 is the beginning of the final 01, so it enters state q_1. (If it guessed correctly, a subsequent 1 will cause it to enter state q_2.) However, just in case it makes a bad guess, and the 0 is *not* the beginning of a 01 sequence, the NFA "hedges its bets" and also remains in state q_0.

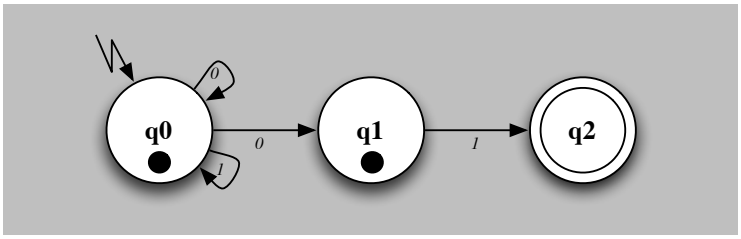
2.1.1 Example — the zeroOne NFA processing a string

To see how this works, let us process the same string 011001 that was handled by the previous DFA.

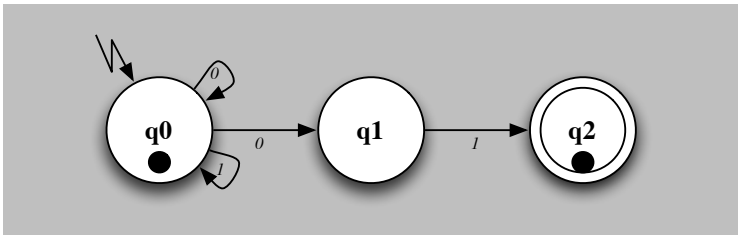
Initially, the NFA is in state q_0.



After processing the input symbol 0, the NFA enters states q_0 and q_1. There are thus *two* dots showing the current state.

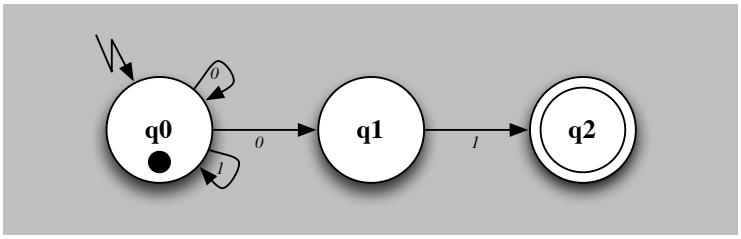


To process the input symbol 1, we must handle two cases. The dot in q_0 results in the next state being q_0. The dot in q_1 results in the next state being q_2. There are two current states q_0 and q_2, and the diagram now looks like this:

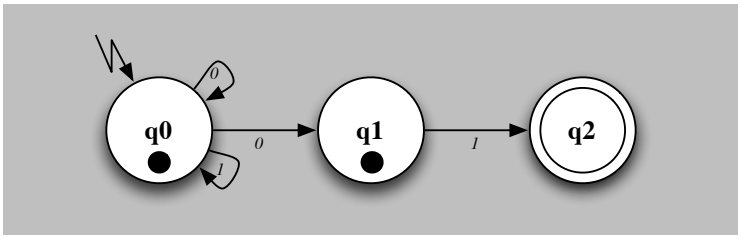


Since q_2 is an accepting state, the NFA has recognised a string that ends in 01.

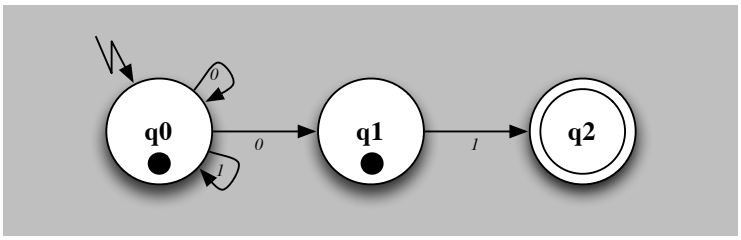
To process the input symbol 1, again we must handle two cases. The dot in q_0 results in the next state being q_0. q_2 has no outgoing transitions, so there is no next state. There is only one current state, and the diagram is:



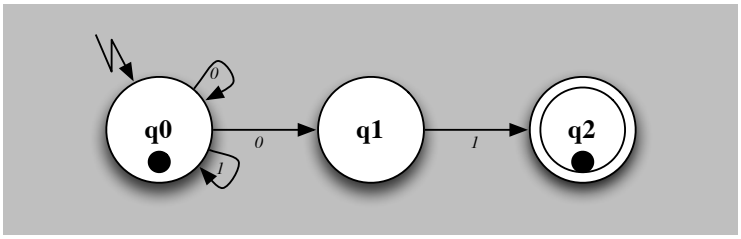
When we process the input symbol 0, the NFA makes two transitions to q_0 and q_1. There are again two current states, and the diagram is:



When we process the input symbol 0, there are two cases to handle: There is no outgoing transition from state q_1, so it is unable to handle the 0. State q_0 has two outgoing transitions labelled with 0, so the NFA enters states q_0 and q_1. The overall effect is for the NFA diagram to not change:



To process the input symbol 1, we again must handle two cases. The dot in q_0 results in the next state being q_0. The dot in q_1 results in the next state being q_2. There are two current states q_0 and q_2, and the diagram now looks like this:



Since q_2 is an accepting state, the NFA has recognised a string that ends in 01.

2.2 Formal definitions of an NFA

2.2.1 Five-tuple definition of an NFA

A *non-deterministic finite automaton* (NFA) consist of:

1. A finite set of *states*, usually denoted by Q
2. A finite set of *input symbols*, usually denoted by Σ.
3. A *transition function* $Q \times \Sigma \rightarrow \{Q\}$, that takes a state and an input symbol, and returns a *set* of new states. The transition function is usually denoted by δ.
4. A set of *start-states* $Q_0 \subseteq Q$. Q_0 is a set containing one or more of the the states in Q.
5. A set of *final* or *accepting* states $F \subseteq$Q, usually denoted by F. Clearly, F is a subset of Q.

Notice that the there are two minor differences between a DFA and an NFA. The first difference is that the transition function for a DFA returns a *single* state, whereas for an NFA it returns a *set* of states.

The second difference follows from the first, and is that a DFA has a *single* start state, whereas an NFA has a *set* of start states.

A non-determinsitic finite automaton named N can be represented by a 5-tuple:

$$N = (Q, \Sigma, \delta, Q_0, F)$$

where Q is the set of states, Σ is the set of input symbols, δ is the transition function, Q_0 is the set of initial states, and F the set of final states.

2.2.2 Transition table definition of an NFA

Not surprisingly, we can represent the transition function for our example in tabular form, like this:

	$zeroOne$	
δ	**0**	**1**
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

The only difference between this table and the one for a DFA is that the *next-state* entries in the table are a *set* of states, instead of being a single state. For example, the entry for state q_0 given the symbol 0 is the set $\{q_0, q_1\}$. When there is no next state, we show the next state as an empty set, $\emptyset$.

2.3 How an NFA processes a string

2.3.1 The extended transition function, $Delta$

An NFA can simultaneously be in multiple states but, by convention, the transition function, δ, is specified in terms of a *single* state and *single* symbol, and returns a *set* of states. We will find it convenient to define a new function Δ that accepts as parameters a *set* of states, and a symbol, and returns a *set* of states.

If an NFA is in the set of states P, and receives an input symbol s, then the next state is determined by evaluating $delta$ for each state in that set (which will return a *set* of states), and then computing the union of those sets.

Formally:

$$Delta(P, s) = \bigcup_{p \in P} \delta(p, s)$$

2.3.2 Extending the transition function to strings

To understand how an NFA decides whether to "accept" a string of symbols, we need to see how it processes a string. The set of all strings that an NFA accepts is called its "language".

Suppose $s_1 s_2 s_3 \cdots s_n$ is a sequence of input symbols drawn from Σ, (the set of input symbols that this NFA can process).

We start with the NFA in its initial state-set, $R_0 = Q_0$, and use the extended transition function Δ to process the first input symbol to get the next the next state-set:

$$R_1 = \Delta(R_0, s_1)$$

R_1 is the set of new states that the NFA enters after receiving the input symbol s_1. We now take this state-set, and process the next symbol s_2, to find the next state-set, R_2:

$$R_2 = \Delta(R_1, s_2)$$

Continuing in this manner until all input symbols have been processed, we succesively enter sets of states $R_3, R_4, \ldots, R_n$. At each step,

$$R_i = \Delta(R_{i-1}, s_i)$$

If one of the states in R_n is an accepting state (i.e. if $R_n \cap F \neq \emptyset$), then the string of input symbols $s_1 s_2 s_3 \cdots s_n$ is "accepted". If not, it is "rejected".

2.4 The string transition function for an NFA

The extended transition function, Δ allows us to specify how an NFA in state-set Q, will move to state-set R upon receipt of a symbol s. We show this by writing $R = \Delta(Q, s)$.

We are interested in knowing what state-set an NFA will reach, starting at Q_0, if it is presented with a *string* of symbols $w = s_1 s_2 s_3 \cdots s_n$.

We can specify $\hat{\Delta}$, the string transition function, that takes a current *state-set* P, and a string of input symbols $w = s_1 s_2 s_3 \cdots s_n$, and generates the state-set R that the NFA will reach after processing those symbols: $R = \hat{\Delta}(P, w)$, like this:

$$\hat{\Delta} : \{Q\} \times \Sigma^* \rightarrow \{Q\}$$

(Note that $\{Q\}$ is a *set of sets of states* — the set of all subsets that can be created by selecting states from Q.)

We can define $\hat{\Delta}$ recursively, like this:

Base case If there is no input, the NFA stays in the current state-set, P. Thus the rule is:

$$\hat{\Delta}(P, \epsilon) = P$$

Recurrence case If we are given a string of symbols $s_1 s_2 s_3 \cdots s_n$, we "chop" the first symbol s_1 from the string, and use Δ to compute the state-set, T, that would be reached, starting at each state p in P:

$$T = \Delta(P, s_1)$$

We then use the $\hat{\Delta}$ function to process the remainder of the string, starting at T: $\hat{\Delta}(T, s_2 s_3 \cdots s_n)$. Combining the pieces, we get:

$$\hat{\Delta}(P, s_1 s_2 s_3 \cdots s_n) = \hat{\Delta}(\Delta(P, s_1)), s_2 s_3 \cdots s_n)$$

We can see that a recursive evaluation according to these rules *must* terminate in a finite number of steps, because each recursive invocation of $\hat{\delta}$ occurs with a shorter string of symbols. Eventually, we will reach the base case: $\hat{\Delta}(R, \epsilon)$.

2.5 The language of an NFA

We can define the language of an NFA in a very similar way to a DFA. The language is the set of all input strings that take the NFA from its initial state to an accepting state. Thus, for the NFA N, defined by:

$$N = (Q, \Sigma, \delta, Q_0, F)$$

The language $L(N)$ of this NFA is defined by:

$$L(N) = \{w \in \Sigma^* \mid \hat{\Delta}(Q_0, w) \cap F \neq \emptyset\}$$

Note that we must define the acceptance test slightly differently from a DFA. In a DFA, the $\hat{\delta}$ function returns a single state, and we check to see if that state is in F. For an NFA, $\hat{\Delta}$ returns a *set* of states. If one or more of those states is in F, then the NFA is in an accepting state. We express this condition by computing the intersection between the result of $\hat{\Delta}$ and F, $\hat{\Delta}(\cdots) \cap F$. If the intersection is *non-null*, we know that the NFA has reached an accepting state.

2.6 Equivalence of NFAs and DFAs

An NFA is often easier to construct than a DFA for a given language, which is sufficient justification for studying NFAs. You may be surprised to know that *every* NFA can be converted into an equivalent DFA, so it is always possible to implement an NFA.

In the worst case, an NFA with n states will require a DFA with $2^n - 1$ states. Fortunately, this case arises only rarely in practice, and a typical DFA has about the same number of states as the original NFA.

We can construct a DFA from an NFA by first creating all the non-empty *subsets* of the states of the NFA (there will be $2^n - 1$ such subsets, for an NFA with n states). Let us call each subset a *d-state*. We then build a DFA with a separate state for each d-state.

The goal is to take an NFA $N = (Q_N, \Sigma, \delta_N, Q_{0N}, F_N)$, and generate a corresponding DFA $D = (Q_D, \Sigma, \delta_D, q_{0D}, F_D)$ such that $L(N) = L(D)$.

We notice that the two automata have the same input alphabet, Σ, and the initial state of D (q_{0D}) is the d-state containing all the initial states of N (Q_{0N}).

The remaining parts of the automaton D can be constructed like this:

- Q_D is the set of all non-empty subsets of Q_N. We refer to each of these as a *d-state*. Since Q_N has n states, there will be 2^n subsets. After eliminating the empty subset, Q_D will have $2^n - 1$ d-states. Fortunately, in practical cases, many of these d-states will be unreachable from the initial d-state, and so can be thrown away. This can substantially reduce the complexity of the final DFA.

- The final-set of D ($= F_D$), is the set of d-states in Q_D, such that each d-state contains at least one accepting state of N. We can specify this as:

$$F_D = \{d \in Q_D \mid d \cap F_N \neq \emptyset\}$$

- For each d-state d in Q_D, and for each symbol t in Σ, we can define $\delta_D(d, t)$ by looking at each of the corresponding NFA states $s \in d$, and then seeing how N would handle each of those states for an input symbol t. Then we compute the union of those states, to get the actual d-state. Formally:

$$\forall d \in Q_D, \delta_D(d, t) = \Delta_N(d, t)$$

2.6.1 Converting our example NFA to a DFA

Taking our example of the $zeroOne$ NFA, we have: $Q_N = \{q_0, q_1, q_2\}$. Now Q_D is the set of all non-empty subsets of Q_N, thus the d-states in Q_D will be: $\{q_0\}$, $\{q_1\}$, $\{q_2\}$, $\{q_0, q_1\}$, $\{q_0, q_2\}$, $\{q_1, q_2\}$, and $\{q_0, q_1, q_2\}$. We can write:

$$Q_D = \{\{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$$

(Since N has 3 states, there are $2^3 - 1 = 7$ non-empty subsets, as we have just seen.)

The final states of the DFA are $\{q_2\}$, $\{q_0, q_2\}$, $\{q_1, q_2\}$, and $\{q_0, q_1, q_2\}$, since these d-states all include the final state q_2 of the original NFA.

When we construct the transition function, we find:

	$zeroOne$	
δ_D	**0**	**1**
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

To show how this table was derived, let us generate the entries for the d-state $\{q_0, q_2\}$. Consider first the case when the next input symbol is a 0. Within the d-state $\{q_0, q_2\}$ there are two states to consider, when we evaluate the Δ_N transition function:

$$\Delta_N(\{q_0, q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$$

We thus see that $\delta_D(\{q_0, q_2\}, 0) = \{q_0, q_1\}$, so $\{q_0, q_1\}$ goes into the 0 column.

Now consider the case when the input symbol is a 1. Once again, within the d-state there are two states:

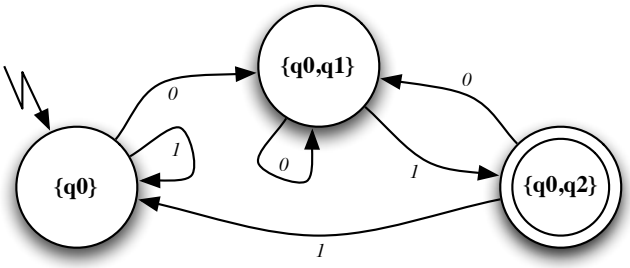
$$\Delta_N(\{q_0, q_2\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_2, 1) = \{q_0\} \cup \emptyset = \{q_0\}$$

We thus see that $\delta_D(\{q_0, q_2\}, 1) = \{q_0\}$. Which now goes into the 1 column.

The remaining entries in the table are computed in a similar way.

If we examine the table carefully, starting at the initial d-state $\{q_0\}$, we find we can reach only the d-states $\{q_0, q_1\}$, $\{q_0\}$, and $\{q_0, q_2\}$. All the other d-states are unreachable, and can safely be ignored. Notice that the DFA now has only three states, exactly the same as the original NFA! This is *much* better than the (potential) worst case of 7 states.

If we draw the DFA transition diagram that corresponds to this table, we get this diagram:



With the exception of the labels on the states, which are unimportant, because they are just names, this diagram is *identical* to the DFA we derived in chapter 1. This is, of course, exactly what we expected!

Chapter 3

Epsilon-NFAs

There is an extension that can be made to NFAs, that improves expressiveness, but does not add any fundamentally new capabilities. The idea of the extension is to allow an NFA to make a transition from one state to another *without* the need for an input symbol. We can think of this as *a transition caused by the empty string*, ϵ, which is why it is called an *ϵ-transition*.

An ϵ-NFA has the same capabilities as a standard NFA, because the class of languages that can be processed remains the same.

3.1 An informal view

We begin with an example: a signed integer is a string of characters. The first character can be an optional $+$ or $-$ sign, which is then followed by a sequence of decimal digits. Some examples of valid signed integers are: 12, -5, $+163$, and 9. Some illegal values are: $34A$, $-$, $-368-$, $3+$, and $3 + 4$.

Fig. 3.1 shows an ϵ-NFA, named $intRecog$, that can recognise a valid signed integer.

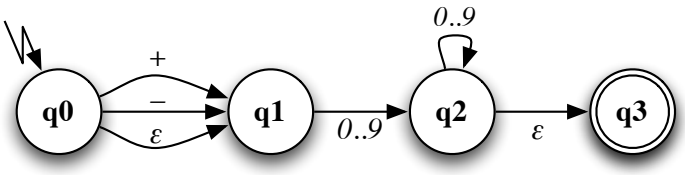


Figure 3.1: The intRecog ϵ-NFA

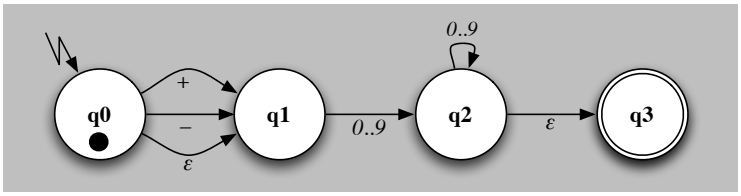
The diagram is very similar to the NFAs we studied previously, except it has two transitions labelled with ϵ. If the machine is in state q_0, it is permitted to make an ϵ-transition to state q_1, without requiring an input symbol. Similarly, if the machine is in state q_2, it can transition to state q_3 without requiring an input symbol.

The input alphabet for this NFA is $\Sigma = \{+, -, 0, 1, 2 \ldots 9\}$. (Notice that the alphabet does *not* include ϵ.) The set of states is $Q = \{q_0, q_1, q_2, q_3\}$. The initial state-set is $Q_0 = \{q_0\}$, and the set of accepting states is $F = \{q_3\}$

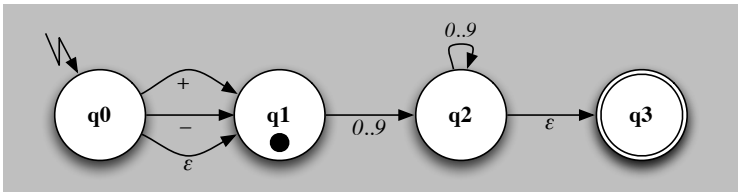
3.1.1 Example - the intRecog ϵ-NFA processing a string

Suppose we wish to recognise the string "+14".

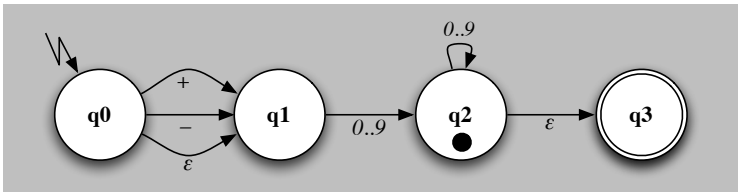
After initialisation (by the "lightnin strike"), the machine enters the state-set $\{q_0\}$. We can represent this with a dot on the diagram, like this:



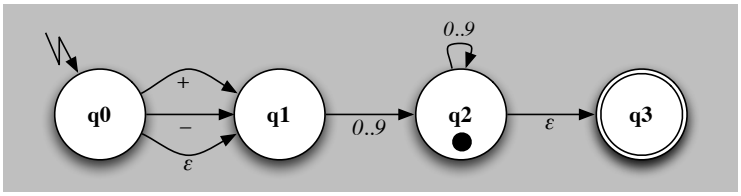
After processing the symbol $+$, the machine changes to the state-set $\{q_1\}$, and the diagram looks like this:



After processing the symbol 1, the machine changes to state-set $\{q_2\}$, and the diagram changes to this:



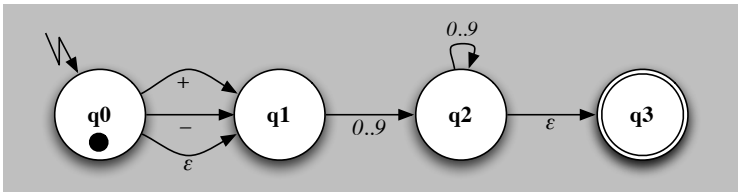
After processing the symbol 4, the machine remains in state-set $\{q_2\}$, and the machine looks like this:



If we ask "was this string recognised?", it seems that the answer is "no", becasue state q_2 is not an accepting state. However, there is an ϵ-transition from q_2 to q_3, so the machine *could* just transition to state q_3. Therefore we see that the machine *has* (correctly) recognised $+14$ as a valid string.

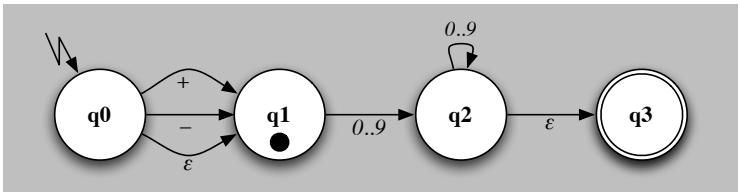
Now consider how it processes the string "65".

Once again, after initialisation, the machine begins in the state-set $\{q_0\}$, like this:

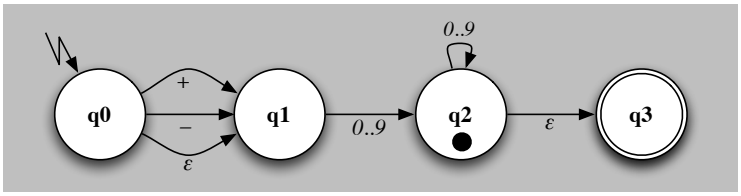


The machine now receives the symbol 6. Since there is no transition to handle this symbol from state q_0, a normal NFA would "die" at this point.

However, we have an ϵ-transition that we can take from state q_0 to q_1. After doing so, we arrive in state q_1.



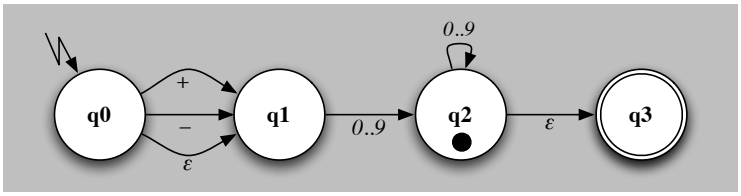
Now we are able to process the symbol 6, which takes us to state q_2, like this:



Notice that after taking an ϵ-transition, the rules require that the machine *must* process the symbol

6 by taking a normal transition. (If there had been a chain of epsilon transitions, the machine could take as many of them as it wanted. However, at the end it *must* process the 6.)

After processing the symbol 5, the machine remains in state-set $\{q_2\}$, and the machine looks like this:



Once again, because there is an ϵ-transition from q_2 to q_3, the machine has correctly recognised 65 as a valid string.

3.2 Formal definitions of an ϵ-NFA

3.2.1 5-tuple definition of an ϵ-NFA

An ϵ-NFA, E, can be defined by the tuple:

$$E = (Q, \Sigma, \delta, Q_0, F)$$

The meanings of Q, Σ, Q_0, and F are the same as for an NFA, but the definition of δ, is altered slightly: it accepts a state in Q, and *either* an input symbol s ($s \in \Sigma$) *or* ϵ, and returns a set of states. We can express this formally:

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \{Q\}$$

where $\{Q\}$ is a set of sets of states, i.e. the *set of all subsets* of Q.
For example, for the intRecog ϵ-NFA described earlier, we have:

$$Q = \{q_0, q_1, q_2, q_3\}$$

$$\Sigma = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$Q_0 = \{q_0\}$$

$$F = \{q_3\}$$

3.2.2 Transition-table definition of an ϵ-NFA

The transition table for an ϵ-NFA is almost identical to that for an NFA, except there is an extra column to specify the effect of the ϵ-transitions.

For example, the transition-table for the $intRecog$ machine described earlier is:

$intRecog$

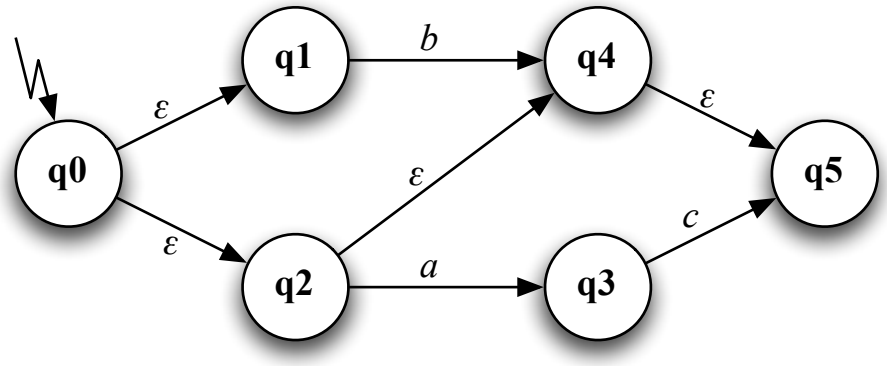
δ	ϵ	$+, -$	$0, 1, \ldots, 9$
$\to q_0$	$\{q_1\}$	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$
q_2	$\{q_3\}$	$\emptyset$	$\{q_2\}$
$*q_3$	$\emptyset$	$\emptyset$	$\emptyset$

3.3 The epsilon-closure

To define the language of an ϵ-NFA, we need to understand a new concept, called the *epsilon-closure*, that is a property of each state. The epsilon-closure of a state q is a set of states. The set includes the state q itself, and all states that can be reached from q, by following *only* ϵ-transitions.

3.3.1 Example of epsilon-closure

Consider the ϵ-NFA shown here:



By looking at the diagram and working backwards, we immediately see:

- $eclose(q_5) = \{q_5\}$
- $eclose(q_4) = \{q_4, q_5\}$, because there is an ϵ-transition from q_4 to q_5
- $eclose(q_3) = \{q_3\}$, because there are no ϵ-transitions leading from state q_3
- $eclose(q_2) = \{q_2, q_4, q_5\}$, because states q_4 and q_5 are reachable from q_2 by taking ϵ-transitions
- $eclose(q_1) = \{q_1\}$, since there are no ϵ-transitions leading from q_1
- $eclose(q_0) = \{q_0, q_1, q_2, q_4, q_5\}$, because we can reach all of these states from q_0, by taking one or more ϵ-transitions

3.3.2 Formal specification of epsilon-closure

We can formally define the epsilon-closure, $eclose(q)$, of a state q like this:

Base case State q is in $eclose(q)$. Formally: $q \in eclose(q)$.

Recurrence case For all states p that are in $eclose(q)$, if state r is reachable from p by an ϵ-transition, then r is in $eclose(q)$. Formally:

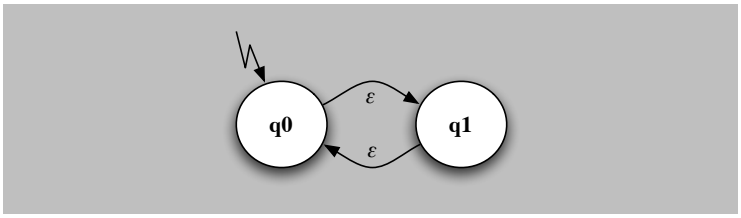
$$\forall p \in eclose(q), \delta(p, \epsilon) \neq \emptyset \text{ and } \delta(p, \epsilon) \subseteq eclose(q)$$

3.3.3 Special cases of epsilon-closure

There are a couple of special cases to worry about.

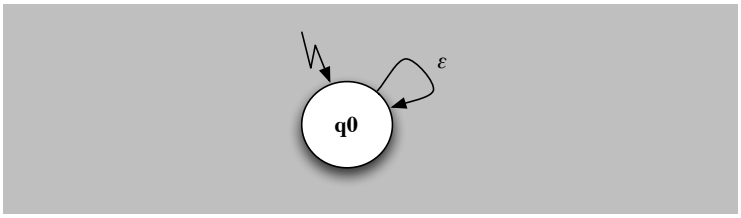
Consider this diagram that has two states connected by ϵ-transitions:

We see that $eclose(q_0) = \{q_0, q_1\}$.



Here is the limiting case of the previous problem:

Once again, there is no problem: $eclose(q_0) = \{q_0\}$.



Thus we see that loops of ϵ-transitions among one or several states cause no problem with the definition of $eclose$.

3.3.4 The extended epsilon-closure

The epilson closure we have defined, $eclose$ takes as parameter a *single* state, and returns a *set of states*.

We will find it convenient to define another function, $ECLOSE$ (note the capital letters!), that takes as parameter a *set of states* and returns a *set of states*. The $ECLOSE$ of a set of states is simply the union of the $eclose$ of each of the individual states, thus the definition is:

$$ECLOSE(Q) = \bigcup_{q \in Q} eclose(q)$$

3.4 The string transition function of an ϵ-NFA

We are interested in knowing what state-set an ϵ-NFA will reach, starting at state-set Q_0, if it is presented with a *string* of symbols $w = s_1 s_2 s_3 \cdots s_n$.

The *string transition function* $\hat{\Delta}$, shows how an ϵ-NFA that starts in any *state-set* P, and receives a sequence of symbols, w, ends up in state-set R.

We can specify the $\hat{\Delta}$ function like this:

$$\hat{\Delta} : \{Q\} \times \Sigma^* \rightarrow \{Q\}$$

(Remember that $\{Q\}$ is the set of all subsets that can be created by selecting states from Q.)

We can define the behaviour of $\hat{\Delta}$ recursively like this:

Base case $\hat{\Delta}(P, \epsilon) = P$ which says that if there is no input, the ϵ-NFA remains in the current state-set P.

Recurrence case We proceed by imagining that we have chopped one symbol, s_1, from the input string, w, and then taking one step forward in the ϵ-NFA machine with that symbol. Finally, starting at the state-set thus reached, we use the extended transition function to process the remainder of the input string, $s_2 s_3 \ldots s_n$. More formally:

1. let $V = ECLOSE(P)$ here V is the set of all states from which the ϵ-NFA could possibly make a transition. It includes the current state-set (P), plus all the states that the machine could reach by taking one or more ϵ-transitions.
2. let $T = \Delta(V, s_1)$ T is the set of all states that will be reached by taking one transition with the symbols_1, starting from the state-set V. (It includes all states *directly* reachable from the current state-set, P, by a transition s_1, *and* all the states *indirectly* reachable by a sequence of epsilon transitions, followed by a transition s_1.) Thus, T is the *next* state-set.

3. We can process the remainder of the input string, (i.e. $s_2, s_3, \ldots, s_n$) by recursively invoking the extended transition function, starting from state-set T: $\hat{\Delta}(T, s_2 s_3 \ldots s_n)$.

By combining the above three steps, we see:

$$\hat{\Delta}(P, s_1 s_2 s_3 \ldots s_n) = \hat{\Delta}(\Delta(ECLOSE(P), s_1), s_2 s_3 \ldots s_n)$$

A recursive evaluation according to these rules *must* terminate in a finite number of steps. Each invocation of $\hat{\Delta}$ occurs with a shorter string of symbols, so we must (eventually) reach the base case: $\hat{\Delta}(P, \epsilon)$, and the recursion will stop.

3.5 The language of an ϵ-NFA

let $E = (Q, \Sigma, \delta, Q_0, F)$ be an ϵ-NFA. We can define $L(E)$, the language of E in a very similar way to that for an NFA. The language is the set of all input strings that take E from its initial state-set Q_0 to a state-set in which *at least one* of the states is an accepting state.

We can find this in three steps:

1. The string transition function, $\hat{\Delta}$ allows us to find the set of states T that E will occupy after processing an input string w:

$$T = \hat{\Delta}(Q_0, w)$$

2. However, there may still be some ϵ-transitionsfrom those states to one (or more) final states. We take account of this by computing, R, the states that are reachable from T by taking only ϵ-transitions:

$$R = ECLOSE(T)$$

3. The string w is accepted by E if at least one of the states in R is in F.

We can compress these three steps into one compact formal statement, like this:

$$L(E) = \{w \in \Sigma^* \mid ECLOSE(\hat{\Delta}(Q_0, w)) \cap F \neq \emptyset\}$$

3.6 Eliminating ϵ-transitions

Given any ϵ-NFA, E, we can find a non-deterministic finite-state automaton (NFA), N, that accepts the same language as E. We do this by eliminating the ϵ-transitions.

Let $E = (Q_E, \Sigma, \delta_E, Q_{0E}, F_E)$ be an ϵ-NFA. We will create an equivalent NFA, $N = (Q_N, \Sigma, \delta_N, q_{0N}, F_N)$, defined like this:

1. N has the same states as E, thus $Q_N = Q_E$. We will see later that sometimes this results in N having unreachable states (states that can never be reached from the initial states). Unreachable states can safely be deleted, thus the number of states in N maybe fewer than in E.

2. E starts in the state-set Q_{0E}, so N will start in the same set of states. Thus:

$$Q_{0N} = Q_{0E}$$

3. The set of final state for E is F_E. It is possible that there are are one or more ϵ-transitions *leading* to from states in Q_E to states in F_E. If so, the states from which those ϵ-transitions come must be regarded as being in the set F_N. Formally:

$$F_D = \{q \in Q_E \mid eclose(q) \in F_E\}$$

4. To compute the transition function, δ_N, we note that for each state q in Q_E, the corresponding transition in N will be expanded to take account of any ϵ-transitionsthat exist from state p. Formally, for any state q in Q_E, we can write:

$$\forall s \in \Sigma : \delta_N(q, s) = \Delta_E(eclose(q), s)$$

This procedure must be applied to *all* the states, q, in Q_E to generate the complete $delta_N$ transition function.

3.7 Example1: Convert integer-recogniser ϵ-NFA to NFA

Earlier we showed an ϵ-NFA that recognised integers. Here is the transition function for that automaton (shown as a table):

$intRecog$

δ	ϵ	$+, -$	$0, 1, \ldots, 9$
$\rightarrow q_0$	$\{q_1\}$	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$
q_2	$\{q_3\}$	$\emptyset$	$\{q_2\}$
$*q_3$	$\emptyset$	$\emptyset$	$\emptyset$

We now convert this ϵ-NFA to a NFA using the method just described.

3.7.1 Compute ϵ-closure

We begin by computing the ϵ-closure, $eclose$, of each state, since we will need this information several times:

state	ϵ-closure
q_0	$\{q_0, q_1\}$
q_1	$\{q_1\}$
q_2	$\{q_2, q_3\}$
q_3	$\{q_3\}$

3.7.2 Compute the starting state-set

The starting state-set of the NFA will be the same as the starting state-set of the ϵ-NFA:

$$\begin{aligned} Q_{0N} &= Q_{0E} \\ &= \{q_0\} \end{aligned}$$

3.7.3 Compute the transition function

We now derive the transition function for *each* state and *each* input symbol.

Let us consider state q_0 first, with the input sybol $+$. From the rules given earlier: $\delta_N(q, s) = \Delta_E(eclose(q), s)$ so:

$$\begin{aligned}\delta_N(q_0, +) &= \Delta_E(eclose(q_0, +)) \\ &= \Delta_E(\{q_0, q_1\}, +) \\ &= \delta_E(q_0, +) \cup \delta_E(q_1, +) \\ &= \{q_1\} \cup \emptyset \\ &= \{q_1\}\end{aligned}$$

Thus $\delta_N(q_0, +) = \{q_1\}$. It is easy to see that the case for $-$ is the same: $\delta_N(q_0, -) = \{q_1\}$.

Now consider the digits $(0 \ldots 9)$ case, we find:

Similar reasoning applies to the other digits. We see $\delta_N(q_0, 0 \ldots 9) = \{q_2\}$.

$$\begin{aligned}\delta_N(q_0, 0) &= \Delta_E(eclose(q_0, 0)) \\ &= \Delta_E(\{q_0, q_1\}, 0) \\ &= \delta_E(q_0, 0) \cup \delta_E(q_1, 0) \\ &= \emptyset \cup \{q_2\} \\ &= \{q_2\}\end{aligned}$$

Thus the first row of the NFA transition table looks like this:

$intRecog$

δ_N	$+, -$	$0, 1, \ldots, 9$
$\rightarrow q_0$	$\{q_1\}$	$\{q_2\}$

We now repeat this process for state q_1. State-set $\{q_1\}$ is simple to handle, because $eclose\{q_1\}) = \{q_1\}$. If we process a $+/-$ symbol from this state, we die (ie next-state set= $\{\}$), and if we process $0 \ldots 9$, the next-state set is $\{q_2\}$.

Thus the next row of the table shows:

q_1	$\emptyset$	$\{q_2\}$

Similar reasoning applies to state $\{q_2\}$, yielding this row:

q_2	$\emptyset$	$\{q_2\}$

Finally, we process q_3, yielding this row:

q_3	$\emptyset$	$\emptyset$

3.7.4 Compute the final-state set

To determine which states are final states, we must check to see if the ϵ-closureof any state of the NFA contains a final state of the original ϵ-NFA. For each state q in Q_E we evaluate the predicate: $eclose(q) \cap F_E \neq \emptyset$, which will have the value either *false* or *true*.

Working through this for q_0, we find:

$$eclose(q_0) \cap \{q_3\} \neq \emptyset$$
$$\{q_0, q_1\} \cap \{q_3\} \neq \emptyset$$
$$\emptyset \neq \emptyset$$
$$\text{false}$$

Therefore q_0 is not a final state.

Similar reasoning applies to q_1.

For state q_2 we find:

$$eclose(q_2) \cap \{q_3\} \neq \emptyset$$
$$\{q_2, q_3\} \cap \{q_3\} \neq \emptyset$$
$$\{q_3\} \neq \emptyset$$
$$\text{true}$$

Therefore q_2 *is* a final state.

Similar reasoning tells us that q_3 is also a final state. The final-state set is therefore: $\{q_2, q_3\}$.

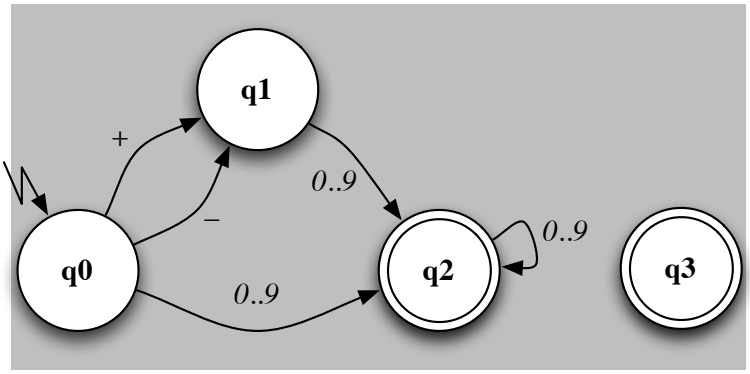
3.7.5 The complete NFA table

Thus the transition-table for the completed NFA is:

$intRecog(NFA)$

δ_N	$+, -$	$0, 1, \ldots, 9$
$\rightarrow q_0$	$\{q_1\}$	$\{q_2\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\{q_2\}$
$*q_3$	$\emptyset$	$\emptyset$

This table corresponds to this NFA diagram:

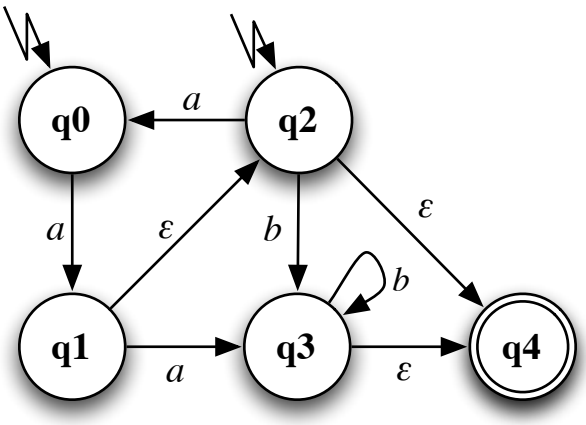


As you can see, it is only slightly more complex than the original ϵ-NFA. After a moment's examination you can also see that:

- It is obviously a correct implementation of the original ϵ-NFA;
- State q_3 is unreachable from any state, and could be deleted; and
- Although we aimed to produce an NFA, this diagram is in fact a *DFA*. (This doesn't always happen — we just got lucky this time.)

3.8 Example2: Convert an ϵ-NFA to an NFA

Here is the diagram of an ϵ-NFA with a lot more ϵ-transitions.



The transition table representation of this ϵ-NFA is:

$example2$

δ_E	a	b	ϵ
$\rightarrow q_0$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\{q_3\}$	$\emptyset$	$\{q_2, q_4\}$
$\rightarrow q_2$	$\{q_0\}$	$\{q_3\}$	$\{q_4\}$
q_3	$\emptyset$	$\{q_3\}$	$\emptyset$
$*q_4$	$\emptyset$	$\emptyset$	$\emptyset$

We will now convert this ϵ-NFA to a NFA.

3.8.1 Compute ϵ-closures

We begin by computing the ϵ-closure, $eclose$, of each state, since we will need this information several times:

state	ϵ-closure
q_0	$\{q_0\}$
q_1	$\{q_1, q_2, q_4\}$
q_2	$\{q_2, q_4\}$
q_3	$\{q_3, q_4\}$
q_4	$\{q_4\}$

3.8.2 Compute the starting state-set

The start-states of the NFA will be the epsilon-closure of the start-states of the ϵ-NFA:

so the NFA will begin in the state-set $\{q_0, q_2\}$.

$$\begin{aligned} Q_{0N} &= ECLOSE(Q_{0E}) \\ &= ECLOSE(\{q_0, q_2\}) \\ &= eclose(q_0) \cup eclose(q_2) \\ &= \{q_0\} \cup \{q_2\} \\ &= \{q_0, q_2\} \end{aligned}$$

3.8.3 Compute the transition function

We now derive the transition function for *each* state and *each* input symbol.

Let us consider state q_0 first, with the input sybol a. From the rules given earlier: $\delta_N(q, s) = \Delta_E(eclose(q), s)$

Thus $\delta_N(q_0, a) = \{q_1\}$.

$$\begin{aligned}\delta_N(q_0, a) &= \Delta_E(eclose(q_0, a)) \\ &= \Delta_E(\{q_0\}, a) \\ &= \delta_E(q_0, a) \\ &= \{q_1\}\end{aligned}$$

Now consider the symbol b:

$$\begin{aligned}\delta_N(q_0, b) &= \Delta_E(eclose(q_0, b) \\ &= \Delta_E(\{q_0\}, b) \\ &= \delta_E(q_0, b) \\ &= \emptyset\end{aligned}$$

Thus the first row of the NFA transition table looks like this:

$example2$

δ_N	a	b
q_0	$\{q_1\}$	$\emptyset$

We repeat this process for state q_1, with input a:

$$\begin{aligned}\delta_N(q_1, a) &= \Delta_E(eclose(q_1, a) \\ &= \Delta_E(\{q_1, q_2, q_4\}, a) \\ &= \delta_E(q_1, a) \cup \delta_e(q_2, a) \cup \delta_E(q_4, a) \\ &= \{q_3\} \cup \{q_0\} \cup \emptyset \\ &= \{q_0, q_3\}\end{aligned}$$

And for input b:

$$\begin{aligned}\delta_N(q_1, b) &= \Delta_E(eclose(q_1, b) \\ &= \Delta_E(\{q_1, q_2, q_4\}, b) \\ &= \delta_E(q_1, b) \cup \delta_e(q_2, b) \cup \delta_E(q_4, b) \\ &= \emptyset \cup \emptyset \cup \emptyset \\ &= \emptyset\end{aligned}$$

Thus the next row shows:

q_1	$\{q_0, q_3\}$	$\emptyset$

Processing q_2 with a, we find:

$$\begin{aligned}\delta_N(q_2, a) &= \Delta_E(eclose(q_2, a) \\ &= \Delta_E(\{q_2, q_4\}, a) \\ &= \delta_E(q_2, a) \cup \delta_E(q_4, a) \\ &= \{q_0\} \cup \emptyset \\ &= \{q_0\}\end{aligned}$$

and with b:

$$\begin{aligned}\delta_N(q_2, b) &= \Delta_E(eclose(q_2, b) \\ &= \Delta_E(\{q_2, q_4\}, b) \\ &= \delta_E(q_2, b) \cup \delta_E(q_4, b) \\ &= \{q_3\} \cup \emptyset \\ &= \{q_3\}\end{aligned}$$

So the next row of the table is:

q_2	$\{q_0\}$	$\{q_3\}$

For q_3 with a we get:

$$\begin{aligned}\delta_N(q_3, a) &= \Delta_E(eclose(q_3, a) \\ &= \Delta_E(\{q_3, q_4\}, a) \\ &= \delta_E(q_3, a) \cup \delta_e(q_4, a) \\ &= \emptyset \cup \emptyset \\ &= \emptyset\end{aligned}$$

And with b:

$$\begin{aligned}\delta_N(q_3, b) &= \Delta_E(eclose(q_3, b) \\ &= \Delta_E(\{q_3, q_4\}, b) \\ &= \delta_E(q_3, b) \cup \delta_e(q_4, b) \\ &= \{q_3\} \cup \emptyset \\ &= \{q_3\}\end{aligned}$$

So the fourth row of the table is:

q_3	$\emptyset$	$\{q_3\}$

And finally for q_4 with a:

$$\begin{aligned}\delta_N(q_4, a) &= \Delta_E(eclose(q_4, a) \\ &= \Delta_E(\{q_4\}, a) \\ &= \delta_E(q_4, a) \\ &= \emptyset\end{aligned}$$

And with b:

$$\begin{aligned}\delta_N(q_4, b) &= \Delta_E(eclose(q_4, b) \\ &= \Delta_E(\{q_4\}, b) \\ &= \delta_E(q_4, b) \\ &= \emptyset\end{aligned}$$

So the last row of the table is:

$$q_4 \parallel \emptyset \mid \emptyset$$

3.8.4 Compute the final-state set

We now determine if any of the states, q, are final-states, by evaluating the predicate: $eclose(q) \cap F_E \neq \emptyset$

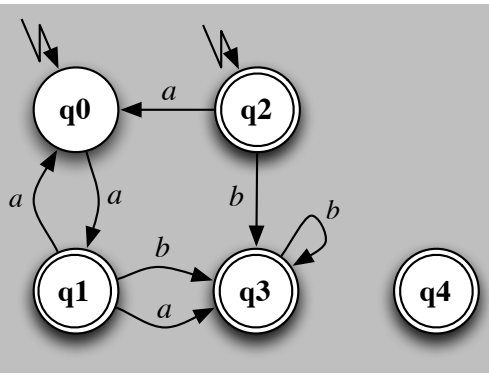
$$\begin{aligned}eclose(q_0) \cap \{q_4\} \neq \emptyset &\Rightarrow \text{false} \\ eclose(q_1) \cap \{q_4\} \neq \emptyset &\Rightarrow \text{true} \\ eclose(q_2) \cap \{q_4\} \neq \emptyset &\Rightarrow \text{true} \\ eclose(q_3) \cap \{q_4\} \neq \emptyset &\Rightarrow \text{true} \\ eclose(q_4) \cap \{q_4\} \neq \emptyset &\Rightarrow \text{true}\end{aligned}$$

3.8.5 The complete transition table

Here is the complete table and corresponding diagram:

$example2(NFA)$

δ_N	a	b
$\rightarrow q_0$	$\{q_1\}$	$\emptyset$
$*q_1$	$\{q_1\}$	$\{q_3\}$
$\rightarrow q_2$	$\{q_1\}$	$\{q_3\}$
$*q_3$	$\emptyset$	$\{q_3\}$
$*q_4$	$\emptyset$	$\emptyset$



3.9 Lazy/Greedy evaluation

The procedures we have so far described for eliminating ϵ-transitions can be described as *lazy evaluation* — the machine does not evaluate an ϵ-transitionuntil a real symbol arrives and needs to be processed.

There is an alternative view: whenever we have an opportunity to execute an ϵ-transition, the machine could immediately take it. This approach is known as *greedy evaluation*.

The difference between the two approaches is quite small — it depends on when the ϵ-closureoperation is performed.

With *lazy* evaluation, we have seen that we perform the ϵ-closure operation *before* performing the $\Delta(q, s)$ step to process a symbol s. Lazy evaluation requires us to do a final ϵ-closureoperation before deciding whether the machine has reached an accepting state.

With *greedy* evaluation, the order of operations is reversed: after performing the $\Delta(q, s)$ step to process a symbol s, the machine executes an ϵ-closureoperation, to "rush ahead". Greedy evaluation requires us to perform an ϵ-closure operation at the *beginning*, to "rush ahead" after the machine has been initalised.

3.10 Converting the intRecog ϵ-NFAto a NFA by greedy evaluation

3.10.1 Compute ϵ-closures

We begin by computing the ϵ-closure, $eclose$, of each state, since we will need this information several times:

state	ϵ-closure
q_0	$\{q_0, q_1\}$
q_1	$\{q_1\}$
q_2	$\{q_2, q_3\}$
q_3	$\{q_3\}$

3.10.2 Compute the starting state-set

The start-states of the NFA will be the ϵ-closureof the start-states of the ϵ-NFA:

$$\begin{aligned} Q_{0N} &= ECLOSE(Q_{0E}) \\ &= ECLOSE(\{q_0\}) \\ &= eclose(q_0) \\ &= \{q_0, q_1\} \end{aligned}$$

3.10.3 Compute the transition function

We now derive the transition function for *each* state and *each* input symbol.

Let us consider state q_0 first, with the input sybol $+$. With greedy eavluation, we the order of evaluation is the reverse of lazy evaluation: $\delta_N(q,s) = ECLOSE(\delta_E(q,s))$ so:

$$\begin{aligned}\delta_N(q_0,+) &= ECLOSE(\delta_E(q_0,+)) \\ &= ECLOSE(\{q_1\}) \\ &= eclose(q_1) \\ &= \{q_1\}\end{aligned}$$

Thus $\delta_N(q_0,+) = \{q_1\}$. It is easy to see that the case for $-$ is the same: $\delta_N(q_0,-) = \{q_1\}$.

Now consider the digits $(0\ldots9)$ case, we find:

$$\begin{aligned}\delta_N(q_0,0) &= ECLOSE(\delta_E(q_0,0)) \\ &= ECLOSE(\emptyset) \\ &= \emptyset\end{aligned}$$

Similar reasoning applies to the other digits. We see $\delta_N(q_0, 0\ldots9) = \{q_2\}$.

Thus the first row of the NFA transition table looks like this:

	$intRecog$	
δ_N	$+,-$	$0,1,\ldots,9$
q_0	$\{q_1\}$	$\emptyset$

We now repeat this process for state q_1, with $+/-$ symbols:

$$\begin{aligned}\delta_N(q_1,+) &= ECLOSE(\delta_E(q_1,+)) \\ &= ECLOSE(\emptyset) \\ &= \emptyset\end{aligned}$$

And for $0\ldots9$:

$$\begin{aligned}\delta_N(q_1,0) &= ECLOSE(\delta_E(q_1,0)) \\ &= ECLOSE(\{q_2\}) \\ &= eclose(q_2) \\ &= \{q_2,q_3\}\end{aligned}$$

Thus the next row of the table shows:

q_1	$\emptyset$	$\{q_2,q_3\}$

Repeating this for state q_2, with $+/-$ symbols:

$$\begin{aligned}\delta_N(q_2,+) &= ECLOSE(\delta_E(q_2,+)) \\ &= ECLOSE(\emptyset) \\ &= \emptyset\end{aligned}$$

And for $0 \ldots 9$:

$$\begin{aligned} \delta_N(q_1, 0) &= ECLOSE(\delta_E(q_1, 0)) \\ &= ECLOSE(\{q_2\}) \\ &= eclose(q_2) \\ &= \{q_2, q_3\} \end{aligned}$$

Thus the next row of the table is:

q_2	$\emptyset$	$\{q_2, q_3\}$

Finally, we process q_3, yielding this row:

q_3	$\emptyset$	$\emptyset$

3.10.4 Compute the final-state set

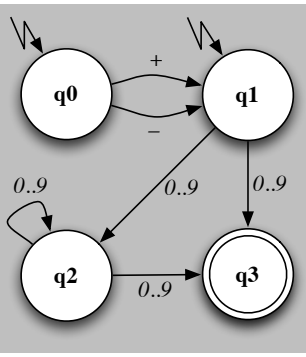
With greedy evaluation, the final state-set of the NFA is exactly the same as that of the ϵ-NFA. Thus $f_N = f_E = \{q_3\}$.

3.10.5 The complete NFA table

Combining the pieces, the transition-table for the intRecog NFA, and the corresponding diagram is:

$intRecog(NFA)greedy$

δ_N	$+, -$	$0, 1, \ldots, 9$
$\rightarrow q_0$	$\{q_1\}$	$\emptyset$
$\rightarrow q_1$	$\emptyset$	$\{q_2, q_3\}$
q_2	$\emptyset$	$\{q_2, q_3\}$
$*q_3$	$\emptyset$	$\emptyset$



We leave the task of converting the $example2$ ϵ-NFA to an NFA by greedy evaluation as an exercise.

Chapter 4

Regular expressions

Up until now we have been examining finite state automata in a variety of forms. Now we will examine a notation, called *Regular expressions* that can express the same languages as finite state automata, but in a more compact and user-friendly way.

A regular expression describes strings of characters in a compact way. Before we define formally how regular expressions work, let us see some examples:

sequence A regular expression ab stands for the character a followed by the character b. This idea can be carried out to arbitrary length, so (for example) $abcd$ is the character a followed by b followed by c followed by d.

alternation The regular expression $a \mid b$ stands for either the character a or the character b (but not both). The string $ab \mid cd$ stands for the string of characters ab or the string cd. Notice that sequence takes precedence over alternation. There can be multiple alternation operators, like this: $a \mid b \mid c$, which means a or b or c.

repetition The regular expression a^* stands for any one of the strings ϵ,a,aa, aaa,$aaaa$,... The expression ab^* stands for the any of the strings a,ab, abb,$abbb$,... Notice that repetition takes precedence over sequence.

grouping Parentheses can be used to group parts of regular expressions. For example the expression $(a \mid b)(c \mid d)$ represents the strings ac, ad, bc, and bd.

4.1 The operators of regular expressions

Regular expressions denote languages. For example, the regular expression: $01^* \mid 10^*$ denotes the language consisting of all strings that start with a single 0, and are followed by zero-or-more 1s, or start with a single 1, followed by zero-or-more 0s.

Let us now consider these operations more formally.

1. The concatenation of two languages L and M, denoted by LM is the set of strings that can be formed by taking any string in L, and concatenating it with any string in M. We usually denote concatenation by just putting the two languages in sequence, as we have previously shown.

   For example, if L is the language {1,01,10,101}, and M is the language {ϵ,0}, then LM is the language {1,01,10,101,010,100,1010}. The first four strings of LM are simply those of L concatenated with ϵ. The remaining three strings come from L concatenated with 0. Note that the string 10 is generated twice by this process, but appears only once in LM (because it is a *set*, not a *list*).

2. The union of two languages L and M, denoted by $L \mid M$, is the set of strings that is in L or M or both. For example if $L = \{01, 11, 100\}$ and $M = \{1, 101\}$, then $L \mid M = \{1, 01, 11, 100, 101\}$.

3. The closure (also called the *Kleene star*) of a language L, denoted by L^* represents the set of all strings that can be formed by taking any number of strings from L, with repetitions permitted, and concatenating them.

   For example, if $L = \{0, 1\}$, then L^* is the set of all strings of zero-or-more ones and zeros.

   if $L = \{0, 10\}$, then L^* is the set of all strings of ones and zeros where a one is always followed by a zero. e.g. 0,10,1010,10010, but not 0110. Formally:

$$L^* = \bigcup_{i \geq 0} L^i = L^0 \cup L^1 \cup L^2 \cdots$$

   where $L^0 = \{\epsilon\}$, $L^1 = \{L\}$, and $L^i = LL^{i-1}$ (i copies of L concatenated together).

Closure is tricky, so here are two examples to clarify the concept:

Example 1 If $L = \{0, 10\}$, then $L^0 = \{\epsilon\}$, since the set of all strings of length zero consists only of the string ϵ). $L^1 = L = \{0, 10\}$. $L^2 = LL = \{00, 010, 100, 1010\}$. $L^3 = LLL = \{000, 0010, 0100, 01010, 1000, 10010, 10100, 101010\}$, continuing in this fashion, we can compute $L^4, L^5 \cdots$. We compute L^* by taking the union of all these sets:

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cdots$$

$$L^* = \epsilon \cup \{0, 10\} \cup \{00, 010, 100, 1010\} \cup \{000, 0010, 0100, 01010, 1000, 10010, 10100, 101010\} \cup \cdots$$

$$L^* = \{\epsilon, 0, 10, 00, 010, 100, 1010, 000, 0010, 0100, 01010, 1000, 10010, 10100, 101010, \cdots\}$$

Example 2 Now consider a different language L which is the set of all strings of zero-or-more 0s. Clearly L is an infinite language, unlike the previous example which was finite. Despite this, it is not hard to derive L^*: $L^0 = \epsilon$, $L^1 = L$, $L^2 = L$, $L^3 = L$, and so on. So $L^* = L$.

4.2 Precedence of operators

Like all algebras, the regular expression operators have an assumed order of "precedence", which means that operators are associated with their operands in a particular order. We are familiar with precedence from high-school algebra: The expression $ab + c$ groups the product before the sum, so the expression means $(a \times b) + c$. Similarly, when we encounter two operators that are the same, we group from the left, so $a - b - c$ means $(a - b) - c$, and *not* $a - (b - c)$. For regular expressions the order of evaluation is:

repetition. The star ($*$)operator has highest precedence. It applies to the smallest string of symbols to its left, that is a well-formed expression.

sequence. The concatenation operator has the next level of precedence. After grouping all the stars to their operands, we group all concatenation operators to their operands. All expressions that are adjacent (without an operator in between) are concatenated. Since concatenations is associative, it does not matter what order we group the operands, though it is conventional to group from the left. Thus abc is grouped as $(ab)c$.

alternation. The alternation operator (|) has the lowest precedence. The remaining operators are grouped with their operands. Alternation is also associative, so the order of grouping does not matter, but again the convention is to group from the left.

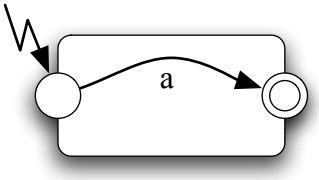
For example, applying the rules to the expression $01^* \mid 10 \mid 1^*0$, we group the star first, giving $0(1^*) \mid 10 \mid (1^*)0$. Then we group the sequence operators, giving $(0(1^*)) \mid (10) \mid ((1^*)0)$. Finally we group the alternation operators, to get: $(((0(1^*))) \mid (10)) \mid ((1^*)0)$.

4.3 Converting finite automata to regular expressions

Regular expressions and finite automata define the same class of languages, so it is possible to transform a DFA into a regular expression, and a regular expression into a DFA. We will ignore the transformation from DFA to regular expression, since this is not often used.

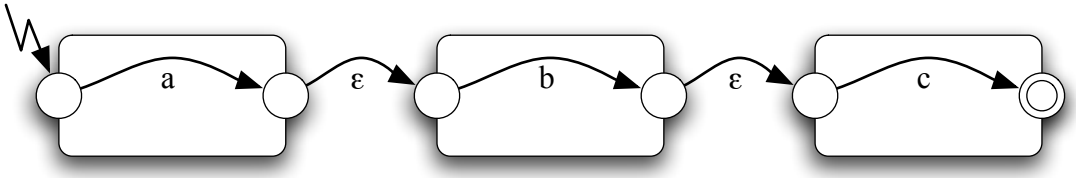
We can transform a regular expression to an ϵ-NFA, in a sequence of trivial steps, as we will now see. First we will need to construct some "building blocks" for constructing the NFA.

The simplest regular expression, a, can be converted to a two-state NFA, with a start-state and a final state, by writing the character a on the transistion, like this:



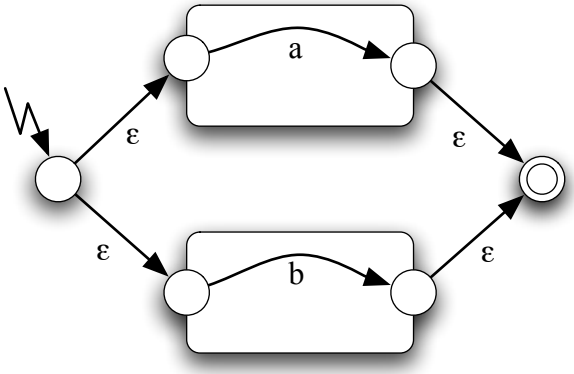
It is clear that this NFA accepts a string consisting only of the character a.

A sequence of simple regular expressions abc, can be handled by converting each character in the expression to a two-state NFA, and then joining the NFAs togetherwith ϵ transitions, like this:



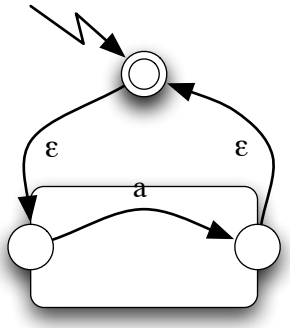
It is immediately obvious that this NFA accepts a string consisting only of the sequence of characters abc.

A regular expression of the form $a \mid b$, can be handled by building two simple NFAs, to handle the a and b, and then running them in parallel. Again, we use ϵ transistions to glue the parts together, like this:

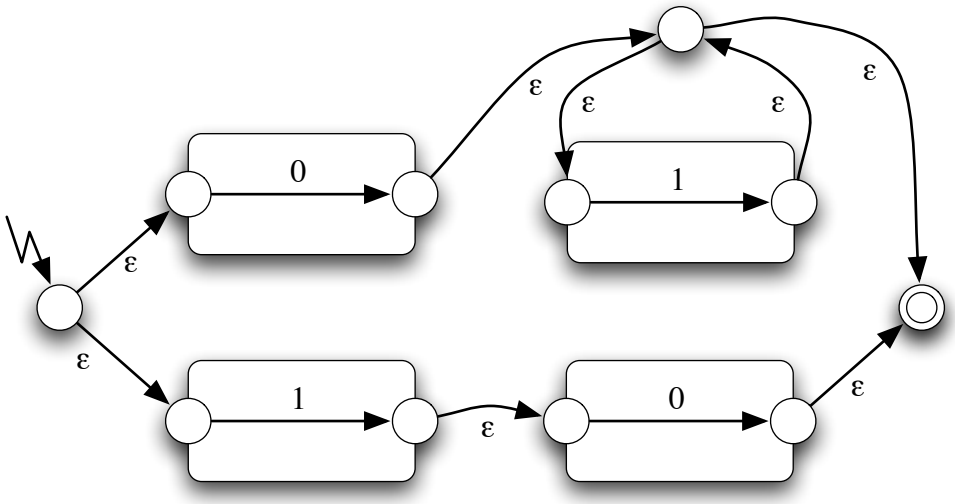


Once again, it is immediately clear that this NFA will accept a string consisting of either a or b.

Finally, to handle the regular expression a^*, we simply add a pair of ϵ transitions to the NFA for a. One of the transitions allows us to completely bypass the NFA (thus allowing "zero-times"), and the other transition allows an infinite number of repetitions (thus allowing "-or-more times"). The diagram therefore looks like this:

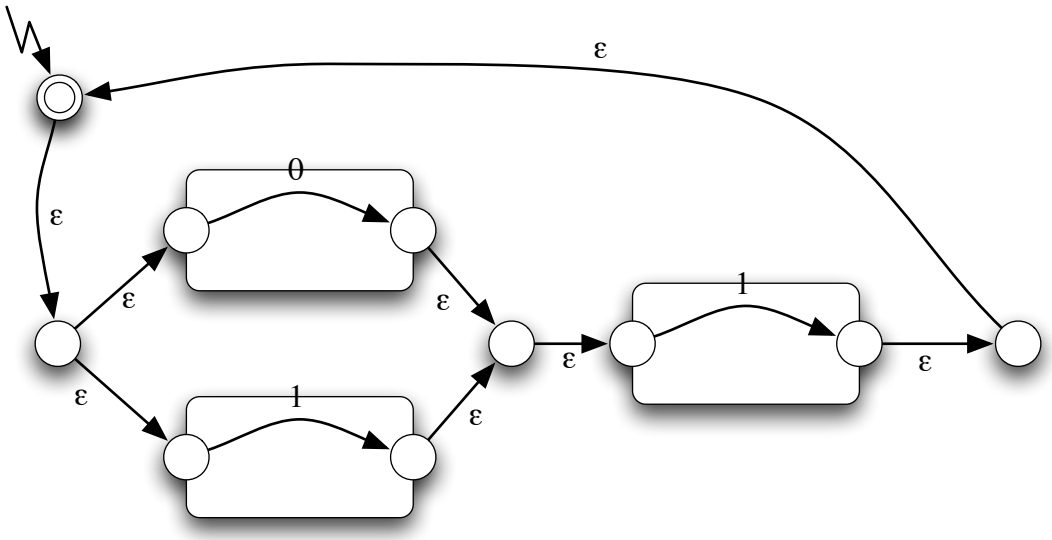


We can now compose these diagrams to handle any regular expression. For example, conside the regular expression $01^* \mid 10$. If we insert parentheses, to make the meaning clear, we get $((0(1^*)) \mid (10)$ which can immediately be converted to this diagram:



It is now a simple matter of using the techniques we already know to reduce this ϵ-NFA to a deterministic finite-state automaton.

Similarly, the expression $((0 \mid 1)1)^*$ can be converted to this diagram:



4.3.1 What is the use of this transformation?

Regular expressions are a very compact way of *expressing* a language. DFAs are a very efficient way of *implementing* a language recogniser. The transformation we have just described trivially converts a regular expression into an ϵ-NFA, and we already know how to convert any NFA into a DFA. Thus we now have a highly efficient way of turning any regular expression into an executable program.

There are numerous tools available, that do this job. One example is *Jlex*, which you may have used during a compiler-construction course.

4.4 Regular expressions in Unix

The regular expressions described earlier have enough expressive power to describe any regular language. However some expressions can become a bit clumsy. Unix extends the set of operators that are available, to make the task of expressing a language even easier. The additional operators available in Unix are:

any The character . (dot) means "any character".

list The sequence of characters $[abcde]$ means $a \mid b \mid c \mid d \mid e$. This saves about half the typing, since we don't need to type the $\mid$.

We can also write $[b - e]$, to represent the consecutive sequence of characters beginning at b and ending at e. For example $[a - z]$ means any lower-case letter. [A-Za-z0-9] means any letter or digit. This is a *big* saving in typing. If we want to include a $-$ sign in the list, we just put it first or last in the list. We could describe a signed (one-digit) integer with $[-+][0 - 9]$.

There are few predefined patterns: $[:digit:]$ means "any digit", $[:alpha:]$ means any alphabetic character, and $[:alnum:]$ means "any letter or digit". To use these patterns, they *must* appear in a list: $[[:digit:]]$ matches a single digit.

optional The character ? placed after an expression means "zero-or-one-of". For example the pattern $(+|-)?[[:digit:]][[:digit:]]*$, describes an integer number with an optional preceding sign. We could also express this as: $[+-]?[[:digit:]][[:digit:]]*$.

repetition. There is an additional repetition operator, $+$, that means "one-or-more-of". The regular expression $R+$ has the same meaning as $RR*$. We can now express our previous definition of signed integer as $[+-]?[[:digit:]]+$.

multiple The operator {n} (where n is a positive integer) placed after a regular expression means "n-copies-of". For example the expression $(ab)\{3\}$ has the same meaning as $ababab$.

nomagic Clearly, some of the characters we have special meanings (such as $*$ [? .). We can tell Unix to treat these characters *without* their magic interpretation by preceding them with a backslash character (\). For example, the regular expression $\backslash * a \backslash ?$ will only match the string of three characters $*a?$.

Here are some example Unix patterns:

Integer A signed integer can be specified by : $[+-][0-9]+$, or equivalently $[+-][[:digit:]]+$.

Identifier A typical programming language identifier can be specified by: $[a-zA-Z][a-zA-Z0-9\_]*$, or this: $[[:alpha:]][[:alnum:]\_]*$.

Chapter 5

Implementing DFAs

In this chapter, we show how FSMs can be implemented in a programming language, such as Java, and in assembly language, such as DLX. We look at Mealy and Moore FSMs, a combined Mealy-Moore FSM, and finally at the design-pattern known as *state*. Before we do so, however there are a few preliminaries:

5.0.1 Events and transitions

A finite state machine is driven by *events*. An event causes the machine to make a *transition* from one state to another (possibly the same) state. When we write a program, we will signal that an event has occurred by calling a method (with the same name as the event) within the FSM. For example the method:

```
public void tick()
```

would be called to indicate a $tick$ event had occurred. And the method:

```
public void keyPressed(char c)
```

would be called to indicate a $keyPressed$ event had occurred, and that the key that was pressed had character-code c.

5.0.2 Remembering the state

We need a way to label the states of our FSM. Since the number of states is usually small, we will use an integer variable, and rather than just use numbers for the states, we will define named constants that represent the states, using Java's `private static final int ...` We will also need a variable to remember the *current* state. For example:

```
private static final int DEAD_ST= 0;
private static final int ALIVE_ST= 1;

private int state;
```

5.0.3 Actions

To be interesting, we want our FSM to perform some kind of *action* in response to its input events. We will show an action by calling a method. Most commonly, an action causes effects *outside* the FSM, perhaps by emitting an event to another FSM. But actions can also be internal to a FSM. The only requirement is that an action does not cause a change of state.

5.1 Example: An apartment light

To make the following sections more concrete, imagine we are solving a practical problem that has arisen in an apartment block.

You are the landlord for a three-storey block of apartments, where access to the upper floors is reached by stairs. At night, a light is needed to enable the tenants to safely climb the stairs. Obviously, you can put a switch at each floor, to allow the tenants to switch the light on as they approach the stairs, and switch it off after they reach their own floor.

Unfortunately, there is a problem with this scheme: if two people approach the stairs, the first one will turn the lights on and start climbing. The second, seeing the lights are already on, will simply climb. When the first person reaches her floor, she switches the lights off, plunging the second person, who is halfway up, into darkness. Experience shows that tenants are aware of this problem, and so *don't* switch the light off when they reach their floor — they simply leave it on "for the next person" — thus wasting electricity.

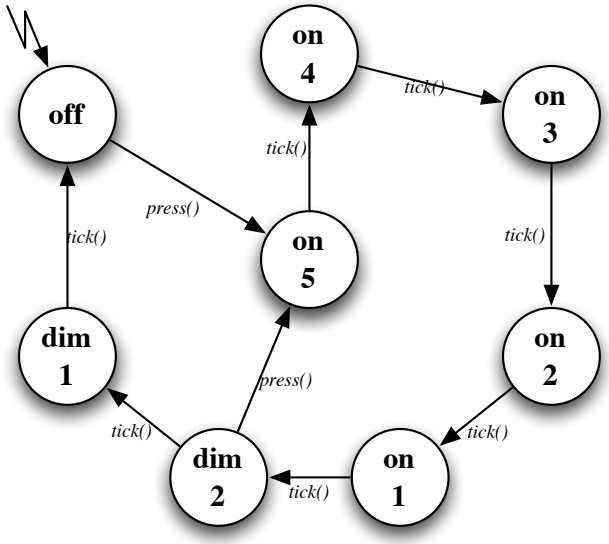
To solve this problem, you have decided to install a computer-controlled timer, that will work like this (assume the light is initially off):

- When a person presses a button at one of the floors, the light will turn on and a five-minute timer will be started.
- After five minutes the lamp brightness will reduce to 70%, to warn anyone on the stairs that it is about to switch off.
- If a person now presses a button the light will return to full brightness and the five-minute timer will be restarted.
- If no press is detected for two minutes, the light will turn off.

5.2 An FSM model of this problem

To specify this model as an FSM, we realise that it has at least three distinct states: off, dim, and on. To handle the timer we must make some assumptions about the frequency of the ticks. Let us assume that there is one tick per minute. When we enter the *on* state, we have 5 minutes to wait, before entering the *dim* state. After one tick, we have 4 minutes to wait, after another tick we have 3 minutes to wait, and so on. We need to be able to remember that we have received the ticks, so we will need additional states.

Here is a diagram showing the FSM for this problem:

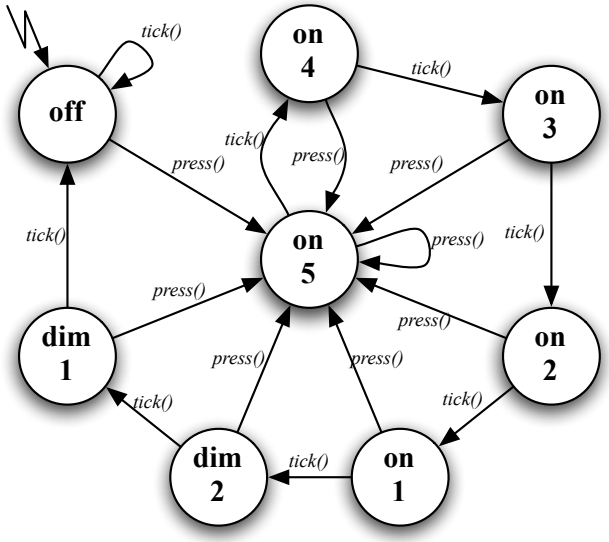


Note that the starting state is off, as indicated by the lightning-strike. If a $press()$ event occurs, the machine makes a transition to the $on5$ state. When a $tick()$ event occurs, the machine changes to the $on4$ state. A $press()$ event will return it to the $on5$ state, but a $tick()$ event will cause the machine to enter the $on3$ state. An examination of the diagram shows that after five $tick()$ events, the machine enters the $dim2$ state, and after a further two $tick()$ events, it enters the off state.

It is a simple matter to determine whether the design is complete, by checking that *every* event is handled in *every* state. When we do this, we discover that the informal specification above is incomplete: it fails to specify what happens if the user presses a button when the light is already on in states $on5$, $on4$, $on3$, $on2$, $on1$, and $dim1$. It is clear that the $press()$ event should result in a transition to the $on5$ state, to restart the timer.

The specification also fails to say what happens to a $tick()$ event in state off. Clearly, we should simply ignore this event, and remain in the off state.

Thus the complete state diagram is:



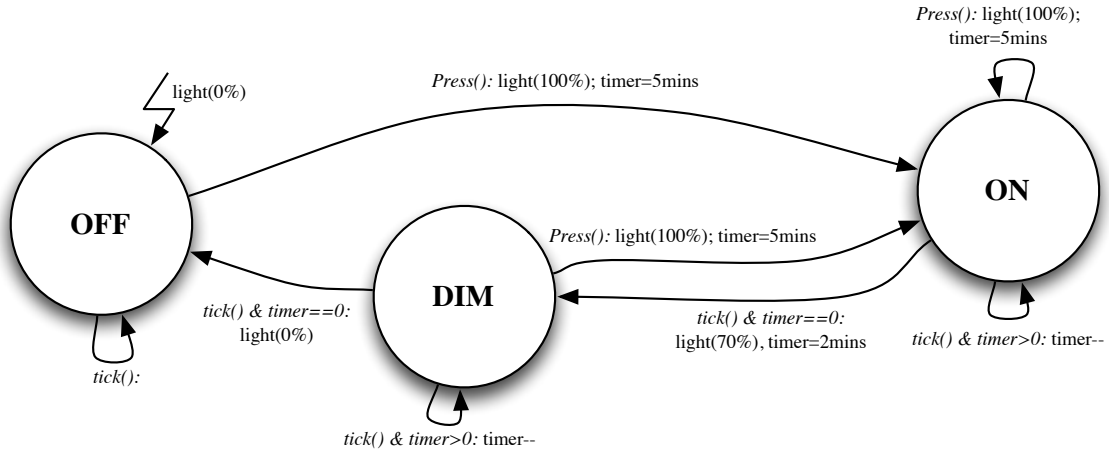
Despite the fact that the problem appeared trivial, we have ended up with a *large* diagram. It could have been far worse! Consider what the diagram would have looked like if ticks had occurred once every *second*, instead of once per minute: there would have been more than 400 states!

Fortunately, there is a solution to this problem: *extended* finite state machines. An extended finite state machine has states, like an FSM, but it also has private *variables*. We can fold most of the states associated with the timer into a single variable that we decrement, as time passes. This *greatly* reduces the complexity of the diagrams, as we shall see. As programmers, we find that almost every FSM that we build is an extended FSM. For this reason, we will continue to use the name FSM, despite the fact that we are really talking about an extended FSM.

5.3 Mealy machine

In 1955, G.H. Mealy published an important paper *A method for synthesizing sequential circuits*, that introduced the notion of finite state machines where the actions of the machine were generated from the *transitions* of the machine. We can think of the actions being generated as the machine *exits* its current state, so we call them *exit actions*.

If we review our apartment-light problem, it seems clear that there are three states: off, dim, and on, with a variable to act as a timer. Remembering that every event must be handled in every state, we find that the behaviour of the apartment-light device can be represented as a three-state Mealy-style FSM, like this:



There are two events driving this machine: `press()`, arising from the user pressing a button, and `tick()`, arising from a clock. There is one external action by the machine: `light(brightness)`, where the intensity of the light is set to a new value. The FSM also alters the value of an internal timer variable.

5.3.1 Implementing the example

In our earlier considerations of the timing for this light controller, we assumed that ticks occurred at one-minute intervals. This means that our timing has a rather big variability: if we press the button *just after* a tick, the timing will be accurate; if we press *just before* a tick, the times will be short by one tick; on average, the time will be short by half a tick.

To reduce the variability, we can make the ticks come more frequently, say once per second. That way, the average error is only half a second — easily small enough to be ignored. A five-minute delay will therefore require 300 ticks.

There will be three methods in the program:

- `initialise()`, that will force the FSM to its initial state (light off). We can call intialise at *any* time to force the machine back to its initial state;
- `press()`, to respond to button-press events; and

- `tick()`, to inform us of the passing of a unit of time.

```
public class ApartmentLightMealy
{
    //Tuning parameters
    //(Time in 1-second units)
    private static final int ON_TIME= 300;
    private static final int DIM_TIME= 120;

    //Declare the names for the states
    private static final int OFF_ST= 0;
    private static final int DIM_ST= 1;
    private static final int ON_ST= 2;

    //Here is the state variable
    private int state;
    private Light light;
    private int timer;

    public ApartmentLightMealy(Light light)
    {
        this.light= light;
        initialise();
    }

    public void initialise()
    {
        light.set(0);
        state= OFF_ST;
    }

    public void press()
    {
        switch( state ){
        case OFF_ST:
            light.set(100);
            timer= ON_TIME;
            state= ON_ST;
            return;

        case DIM_ST:
            light.set(100);
            timer= ON_TIME;
            state= ON_ST;
            return;

        case ON_ST:
            light.set(100);
            timer= ON_TIME;
            state= ON_ST;
            return;
        }
    }

    public void tick()
    {
        switch( state ){
        case OFF_ST:
            //Ignore it
            return;

        case DIM_ST:
            if( timer>0 ){
                timer--;
                state= DIM_ST;
                return;
            }

            //timer expired
            light.set(0);
            state= OFF_ST;
            return;

        case ON_ST:
            if( timer>0 ){
                timer--;
                state= ON_ST;
                return;
            }

            //timer expired
            light.set(70);
            timer= DIM_TIME;
            state= DIM_ST;
            return;
        }
    }
}
```

Figure 5.1: Apartment-light example implemented as a Mealy FSM

There are *exit-actions* associated with the transition from each state, that change the value of the local variable called timer, and alter the state of the external light.

The Java code for our example, implemented as a MealyFSM, is shown in figure 5.1.

We can understand the operation of the code by examining its behaviour when the FSM is in the state Dim.

When a press event occurs, the `press()` method is called. In the Dim state, the press method executes the exit-action, by setting the light to 100%, and resetting the timer to the `ON_TIME` value, and then sets the state to On.

When a tick occurs, the `tick()` method is called. Again, the method contains a switch statement with a case for each state. In the Dim state, the method tests if the value of the timer is non-zero (indicating that the timer has not yet expired). If so, the timer is decremented, and the state is set to Dim. If the timer *is* zero, the method sets the light to 0%, then sets the state to Off.

The behaviour in the other two states is similar.

The finished program is quite short, and its structure follows directly from the Mealy FSM diagram. Translation from the diagram to the Java is "mechanical" — if the diagram is correct, the Java code will automatically be correct also. This is a very attractive property.

5.3.2 The general case

The general procedure for constructing a Mealy FSM is straightforward. The following points refer to the numbered sections in the example program shown in figure 5.2.

```
public class MealyGeneral
{
    //(1)Declare the names for the states
    private static final int STATENAME1= 0;
    private static final int STATENAME2= 1;
    ...more state declarations

    //(2)Here is the state variable
    private int state;

    //(3)FSM global variables go here

    //(4)Constructor
    public MealyGeneral(...params...)
    {
        initialise();
    }

    //(5)Initialisation routine
    public void initialise()
    {
        ...other initialisations
        state= STATENAME;
    }

    //(6)Event handler for event1
    public void event1(...params...)
    {
        switch( state ){
        case STATENAME1:
            (7)...exit actions
            state= STATENAME;
            return;

        case STATENAME2:
            ...exit actions
            state= STATENAME;
            return;

        (8)...more cases for other states
        }
    }

    //Event handler for event2
    public void event2(...params...)
    {
        switch( state ){
        case STATENAME1:
            ...exit actions
            state= STATENAME2;
            return;

        ...more cases for other states
        }
    }

    (9)...More handlers for other events
}
```

Figure 5.2: General form of Mealy Finite State Machine

1. Define constants, named after the states of the FSM, and assign a unique integer value to each state.

2. Create an integer variable named `state`, to hold the current state of the FSM.

3. Allocate any additional variables needed within the FSM, such as timers or counters.

4. The constructor can perform any once-only initialisations, such as making connections to other objects in the program. Its last action should be to call the `initialise()` procedure.

5. Create a public procedure named `initialise()` or `reset()` that does whatever is required to get the FSM into the initial state. If there are any FSM global variables, this is the correct place to initialise them. The last action of the initialise procedure must set the initial value of the `state` variable. Note that the initialise procedure must contain *all* the code needed to correctly force the FSM back to the initial state, regardless of what its current state might be.

6. Create a procedure to handle each event to which the FSM will respond, with the name of the event it handles. There will therefore be as many event procedures as there are events. An event procedure can accept parameters, to specify more precisely how the event is to be handled. For example, an event named `keypress(char ch)` might have as a parameter the ASCII-code of the key that was pressed. Inside each event procedure there will be a switch statement, controlled by the `state` variable.

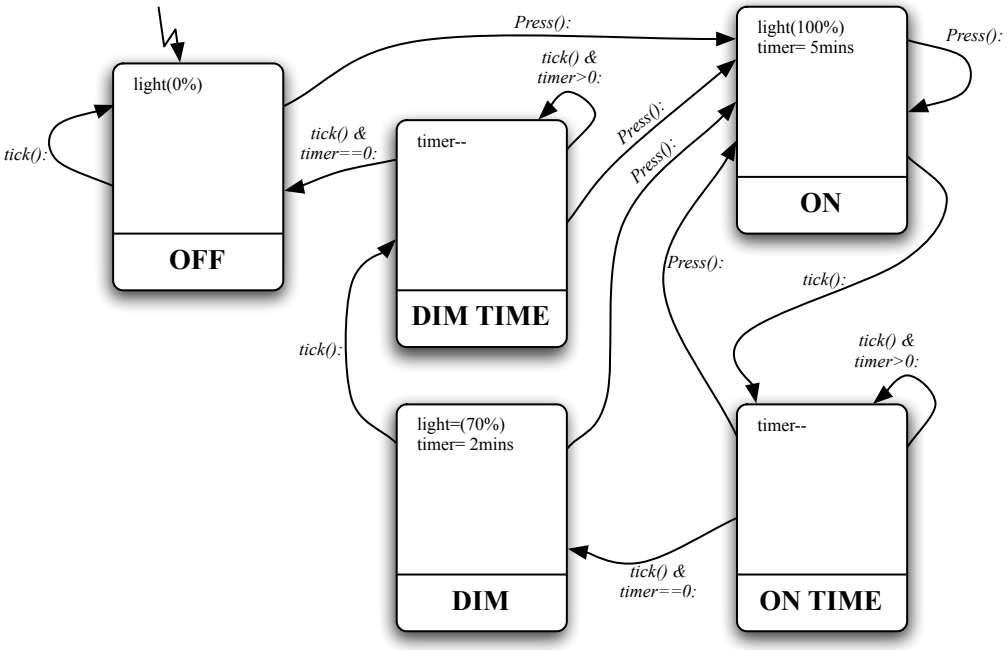
7. For each state, there is a case to handle the processing for that state. Each state can perform whatever *exit-actions* are appropriate. The *last* thing to do, after handling a case, is to assign a new value to the `state` variable, thus causing a transition to a new state. To make it clear that no further processing is required, write `return`, rather than `break`, at the end of the case.

8. Note that there must be an explicit case for *every* state. If there is a reason to believe that an event is "impossible", *do not* omit the case! Instead, write:
`throw new RuntimeException("Impossible!").` If the reasoning is correct, this exception will never occur. However, if there is a flaw, (perhaps a bug elsewhere in the program) the exception will report the problem, rather than just ignoring it.

5.4 Moore machine

in 1956, E.F. Moore published an influential paper *Gedanken experiments on sequential machines*, in which he described finite state machines whose actions were generated from the *new state* of the machine. We can think of these actions as being generated as we *enter* a new state.

We can build the apartment-light in the Moore style, but it turns out that we need a total of five states to implement the required behaviour. Here is the resulting Moore FSM:



Notice that the actions for each state are now written *inside* the state, not on the transitions.

As before, there are two events driving this machine: `press()` and `tick()`. There is one external action from it: `light(brightness)`, and of course the machine updates the internal timer variable.

```
public class ApartmentLightMoore
{
    //Tuning parameters
    //(Time in 1-second units)
    private static final int ON_TIME= 300;
    private static final int DIM_TIME= 120;

    //Declare the names for the states
    private static final int OFF_ST= 0;
    private static final int DIM_ST= 1;
    private static final int DIM_TIME_ST= 2;
    private static final int ON_ST= 3;
    private static final int ON_TIME_ST= 4;

    private int state;
    private Light light;
    private int timer;

    public ApartmentLightMoore(Light light)
    {
        this.light= light;
        initialise();
    }

    public void initialise()
    {
        next(OFF_ST);
    }

    private void next(int newState)
    {
        state= newState;
        switch( state ){
        case OFF_ST:
            light.set(0);
            return;

        case DIM_ST:
            light.set(70);
            timer= DIM_TIME;
            return;

        case DIM_TIME_ST:
            light.set(70);
            timer--;
            return;

        case ON_ST:
            light.set(100);
            timer= ON_TIME;
            return;

        case ON_TIME_ST:
            light.set(100);
            timer--;
            return;
        }
    }

    public void press()
    {
        switch( state ){
        case OFF_ST:
            next(ON_ST);
            return;

        case DIM_ST:
            next(ON_ST);
            return;

        case DIM_TIME_ST:
            next(ON_ST);
            return;

        case ON_ST:
            next(ON_ST);
            return;

        case ON_TIME_ST:
            next(ON_ST);
            return;
        }
    }

    public void tick()
    {
        switch( state ){
        case OFF_ST:
            return;

        case DIM_ST:
            next(DIM_TIME_ST);
            return;

        case DIM_TIME_ST:
            if( timer>0 ){
                next(DIM_TIME_ST);
                return;
            }

            next(OFF_ST);
            return;

        case ON_ST:
            next(ON_TIME_ST);
            return;

        case ON_TIME_ST:
            if( timer>0 ){
                next(ON_TIME_ST);
                return;
            }

            next(DIM_ST);
            return;
        }
    }
}
```

Figure 5.3: Apartment light implemented as a Moore FSM

5.4.1 Implementing the example

The event procedures are very simple, since they just decide which state to enter next, and new method, `next(int newState)`, handles the entry actions. The Java code is shown in figure 5.3.

5.4.2 The general case

The procedure for constructing a Moore FSM is straightforward. The following notes refer to the numbered sections in the example program shown in figure 5.4.

```
public class MooreGeneral
{
    //(1)Declare the names for the states
    private static final int STATENAME1= 0;
    private static final int STATENAME2= 1;
    ...other states

    //(2)Here is the state variable
    private int state;

    //(3)FSM global variables go here


    //(4)Constructor
    public MooreGeneral(...params...)
    {
        initialise();
    }

    //(5)Initialisation routine
    public void initialise()
    {
        ...initialisation of FSM globals
        next(STATENAME);
    }

    //(6)Handle actions on entry to a state
    private void next(int newState)
    {
        state= newState;
        //(7)Execute entry actions
        switch( state ){
        case STATENAME1:
            ...entry action
            return;

        case STATENAME2:
            ...entry actions
            return;

        ...more cases for other states
        }
    }

    //(8)Event handler for event1
    public void event1(...params...)
    {
        switch( state ){
        case STATENAME1:
            next(STATENAME);
            return;

        case STATENAME2:
            next(STATENAME);
            return;

        (9)...more cases for other states
        }
    }

    //Event handler for event2
    public void event2(...params...)
    {
        switch( state ){
        case STATENAME1:
            next(STATENAME);
            return;

        ...more cases
        }
    }

    (10)...More handlers for other events
}
```

Figure 5.4: General form of Moore Finite State Machine

1. Define constants, named after the states of the FSM, and assign a unique integer value to each state.

2. Create an integer variable named `state`, to hold the current state of the FSM.

3. Allocate any additional variables need within the FSM, such as timers or counters.

4. The constructor can perform any once-only initialisations, such as making connections to other objects in the program. Its last action should be to call the `initialise()` procedure.

5. Create a public procedure named `initialise()` that does whatever is required to initialise the global variables (if any), then calls `next(···)` to make a transition to the initial state. At minimum, the initialise procedure must set the initial state by calling the `next(···)` method. Note that a user should be able to call the initialise procedure at any time, to force the FSM back to its initial state.

6. Create a procedure `next(int newState)` that will be called whenever a transition to a new state is required. The first thing the procedure does is to assign the value of `newState` to `state`.

7. The `next` method must contain a switch-statement controlled by `state`. Each case in the switch-statement contains the code for the actions to be executed on entry into that state. To make it clear that no further processing is required after each case, write `return`, rather than `break`, at the end of the case.

8. For each event that the machine will respond to, create a procedure to handle that event. Inside every event procedure there will be a switch statement, controlled by the `state` variable.

9. For each state, there is a case, to handle the processing for that state. Typically, this part of the program will consist of conditional statements to decide which state to enter next. At the end of handling a case, the last statement must be a call to the `next(···)` method, to set the new state. To make it clear that no further processing is required, write `return`, immediately after the call to `next`.

   If there is a reason to believe that an event is "impossible", *do not* omit the case! Instead, write `throw new RuntimeException("Imposible!")`. If the reasoning is correct, this exception will never occur. However, if there is a flaw, (perhaps a bug elsewhere in the program) the exception will report the problem, rather than just ignoring it.

10. There will be as many event handlers as there are events.

5.5 Combined Mealy-Moore machine

As we have already seen, when solving the same problem, a Mealy machine has fewer states than the equivalent Moore machine. Yet the Mealy and Moore FSM models each have attractive properties:

- The Mealy machine provides very great expressive power, since a FSM with n states can have as many as n^2 transitions, and actions are associated with each transition. However, observing that the FSM is in a particular state, s, is *not* sufficient to enable us to determine the actions that were taken prior to entering that state. (They, of course, depended on the transition that led the machine to state s.)

- The Moore machine is conceptually simpler, because its actions are associated with each state. The transitions of the machine serve only to change the state of the machine, but have no direct infuence on the actions taken. Thus when the machine is in a particular state, s, the actions that were taken by the machine can readily be determined.

We can think of a Mealy machine as generating actions as it *exits* the current state, and a Moore machine as generating actions as it *enters* the next state.

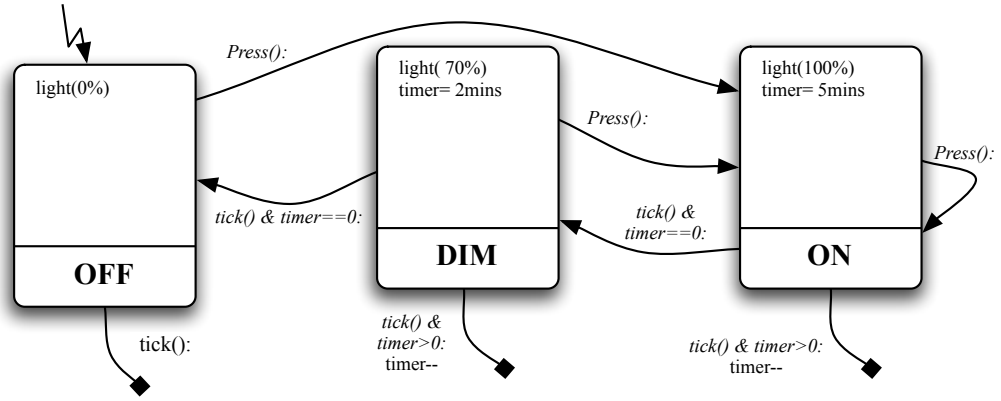
It is clear that we could build a FSM that behaved like both of these models at the same time: It would have *exit-actions*, like the Mealy machine, and *entry-actions*, like a Moore machine.

When an event occurs, we respond to it in the usual (Mealy) way, by executing some exit-code, determined by the event and the current state. As usual, the code is able to update variables in the FSM, and to perform actions to the outside world. The code ends with a call to `next(···`, to specify the next state, and (in the Moore way), execute *entry* action of that new state.

The combined Mealy-Moore machine offers one additional facility not available in the separate machines: partial-transitions. When an event occurs, we respond to it in the usual (Mealy) way, by executing some exit-code, determined by the event and the current state. Normally, the code would now call `next(··)`, and move to the next state. For a partial-transition, however, we simply execute `return`, which causes the FSM to remain in the current state *without executing the entry-actions*. Clearly a partial-transition is *not* the same as a transition to the current state.

A partial-transition can be surprisingly useful. It is represented on FSM diagrams as an arc that ends in "mid air".

If we specify our earlier apartment-light example using a combined Mealy-Moore machine, and exploit all the features described above, we get an even simpler-looking FSM, than our earlier Mealy solution. It has some actions on the arcs, and some in the states, and makes use of partial transitions:



5.5.1 Implementing the example

The Java code for our example, implemented as a Mealy-Moore FSM is shown in 5.5.

5.5.2 The general case

The procedure for constructing a Mealy-Moore FSM is a straightforward combination of the Mealy and Moore approaches. The following notes refer to the numbered sections in the example program shown in figure 5.6.

1. Define constants, named after the states of the FSM, and assign a unique integer value to each state.
2. Create an integer variable named `state`, to hold the current state of the FSM.
3. Allocate any additional variables needed within the FSM, such as timers or counters.
4. The constructor can perform any once-only initialisations, such as making connections to other objects in the program. Its last action should be to call the `initialise` procedure.

```
public class ApartmentLightMealyMoore
{
    //Tuning parameters
    //(Time in 1-second units)
    private static final int ON_TIME= 300;
    private static final int DIM_TIME= 120;

    //Declare the states
    private static final int OFF_ST= 0;
    private static final int DIM_ST= 1;
    private static final int ON_ST= 2;
    private int state;

    private Light light;

    //FSM Globals
    private int timer;

    public ApartmentLightMealyMoore(Light light)
    {
        this.light= light;
        initialise();
    }

    public void initialise()
    {
        next(OFF_ST);
    }

    private void next(int newState)
    {
        state= newState;
        switch( state ){
        case OFF_ST:
            light.set(0);
            return;

        case DIM_ST:
            timer= DIM_TIME;
            light.set(70);
            return;

        case ON_ST:
            timer= ON_TIME;
            light.set(100);
            return;
        }
    }

    public void press()
    {
        switch( state ){
        case OFF_ST:
            next(ON_ST);
            return;

        case DIM_ST:
            next(ON_ST);
            return;

        case ON_ST:
            next(ON_ST);
            return;
        }
    }

    public void tick()
    {
        switch( state ){
        case OFF_ST:
            //Ignore it
            return;

        case DIM_ST:
            if( timer>0 ){
                 timer--;
                return;
            }

            //Timer expired
            next(OFF_ST);
            return;

        case ON_ST:
            if( timer>0 ){
                timer--;
                return;
            }

            //Timer expired
            next(DIM_ST);
            return;
        }
    }
}
```

Figure 5.5: Apartment light as a Mealy-Moore Finite State Machine

```
public class MealyMooreGeneral
{
    //(1)Declare the names for the states
    private static final int STATENAME1= 0;
    private static final int STATENAME2= 1;
    ...other states

    //(2)Here is the state variable
    private int state;

    //(3)FSM global variables go here


    //(4)Constructor
    public MealyMooreGeneral(...params...)
    {
        initialise();
    }

    //(5)Initialisation routine
    public void initialise()
    {
        state= STATENAME;
        ...other initialisations
    }

    //(6)Handle actions on entry to a state
    private void next(int newState)
    {
        state= newState;
        //(7)
        switch( state ){
        case STATENAME1:
            ...entry actions
            return;

        case STATENAME2:
            ...entry actions
            return;

        ...other cases
        }
    }

    //(8)Event handler for event1
    public void event1(...params...)
    {
        //(9)
        switch( state ){
        case STATENAME1:
            ...exit actions
            next(STATENAME2);
            return;

        case STATENAME2:
            ...exit actions
            next(STATENAME1);
            return;

        ...more cases
        }
    }

    //(11)Event handler for event2
    public void event2(...params...)
    {
        switch( state ){
        case STATENAME1:
            ...exit actions
            next(STATENAME1);
            return;

        case STATENAME2:
            ...actions
            next(STATENAME2);
            return;

        ...more cases
        }
    }

    ...More handlers for other events
}
```

Figure 5.6: General form of Mealy-Moore Finite State Machine

5. Create a public procedure named `initialise()` that does whatever is required to get the FSM into the initial state. This is the right place to initialise the FSM global variables (if any). At minimum, the initialise procedure must call the `next` method to make a transition to the initial state. Note that a user should be able to call the `initialise` method at any time, to force the FSM back to its initial state.

6. Create a method named `next(int newState)` that will be called whenever a transition to a new state is required. The first thing the procedure does is to assign the value of `newState` to `state`.

7. The `next` procedure must contain a switch-statement controlled by `state`. Each case in the switch-statement contains the code for the actions to be executed on entry into that state. To make it clear that no further processing is required after each case, write `return`, rather than `break`, at the end of the case.

8. For each event that the machine will respond to, create a procedure to handle that event. Inside every event procedure there will be a switch statement, controlled by the `state` variable.

9. For each state, there is a case, to handle the processing for that state. Each state can take whatever *exit-actions* are appropriate. At the end of handling a case, there are two options: call the `next` method to make a transition to a new state; or do nothing, to indicate a partial transition. Either way, write `return`, rather than `break`, at the end of the case.

10. There will be as many event handlers as there are events.

```
machine ApartmentLight
    int timer
    constant int ON_TIME= 300;
    constant int DIM_TIME= 120;

//Initialisation mechanism
initial
    //Initialisation goes here
    next OFF

//Description of a state
state OFF:
    //Entry actions for this state
    light(0%);

    //Events processed in this state
    when press()
        //Exit actions go here...

        //Transition to next state
        next ON;

    when tick()
        //Partial transition just returns
        return;
    endwhen;
```

```
state DIM:
    light(70%);
    timer= DIM_TIME;

    when press()
        next ON;

    when tick()
        if timer>0 then
            timer--
            return;
        endif;
        next OFF;
    endwhen;

state ON:
    light(100%);
    timer= ON_TIME;

    when press()
        next ON;

    when tick()
        if timer>0 then
            timer--
            return;
        endif;
        next DIM;
    endwhen;
endmachine;
```

Figure 5.7: “Ideal” realisation of Mealy-Moore Finite State Machine

5.5.3 What we'd really like to say

The Mealy-Moore Java program we have written expresses our state-machines using the language constructs provided in Java. (And, by evolution, therefore, also in C, C++ and C#, as well as many other languages.) The finished program is "ok" — it does what we want, and it is reasonably easy to modify, but it is not "beautiful", and it could be a lot clearer. There is a lot of "junk" necessary to specify the machine: We must declare the state variable, build the case statements, write the $next$ procedure, and so forth. All of these steps require discipline, and care, or errors will be introduced

In an ideal world, the Java compiler would allow us to directly express the apartment-light program as shown in figure 5.7.

Notice here that the program is *very* short, and *very* readable. Each state is clearly marked, and the entry actions are visually associated with that state. The handling of each event in a state is specified by a `when(...)` clause, and it is quite clear exactly what happens for each event. The transition to the next state is handled by a `next` statement, (or a `return`, in the case of a partial transition).

Java, of course, does not yet allow us to say this(I live in hope!). The code shown previously is about the best that can be achieved within the limits of the language. On a performance note, it is not clear how well the Java compiler is able to "optimise-away" all the additional stuff that we are forced to add.

5.5.4 The apartment-light example - in DLX

To give a feeling for how efficiently a Mealy-Moore FSM can actually be implemented on a modern computer, we present here a solution to the apartment-light problem written in assembly-code for the DLX machine. The purpose of showing the code is *not* that you should learn it by heart —that would be pointless. Rather, you should see that the cost of an FSM, when it is implemented tightly in assembly code is very small indeed. The DLX version of the program directly implements the "ideal" version shown earlier. The overheads associated with receiving an event, and determining which event-handler to execute is cheap (just four instructions), and the overhead of the entry-actions is also very low (three instructions).

The high-level language versions of our example program tend to look more complex than they actually are, but the assembly code shows the truth. An FSM *really is* the way to build the control logic of an event driven program.

Figure 5.8 shows the apartment-light example implemented in DLX (except for a few unimportant details that have been omitted).

Each event procedure contains a *switch* statement, implemented as a lookup table, that decides which body of code is to be executed. For our example, there are just three instructions and three pointers.

The code for the event-handler of state, st, responding to an event ev, is labelled st_ev. In assembly code, we can put code anywhere we like, so we are able to put the event-handling code for a state immediately after the entry-code for that state, thus keeping the state's logic neatly together in one place.

The entry-code for state, st, is simply labelled st. To get the effect of `next t`, we simple execute a jump to t.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                           ;
;   ApartmentLight in DLX                   ;
;                                           ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;Tuning parameters
;(1-second units)
ON_TIME .equ   300
DIM_TIME .equ  120

;Declare constants for the names of the
 Since we use the constants as
;an index into a word-sized table, they
;must be a multiple of 4.
OFF_ST  .equ    0
DIM_ST  .equ    4
ON_ST   .equ    8

state   .space  4
timer   .space  4

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       EVENT press()
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
press   lw      r1,state
        lw      r1,press1(r1)
        jr      r1

press1  .word   off_press
        .word   dim_press
        .word   on_press

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       EVENT tick()
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
tick    lw      r1,state
        lw      r1,tick1(r1)
        jr      r1

tick1   .word   off_tick
        .word   dim_tick
        .word   on_tick


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       Initialise
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
initialise
        j       off

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       STATE off
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
off     addi    r1,r0,OFF_ST    ;set state
        sw      state,r1

        addi    r1,r0,0         ;set light
        ;(more here)
        jr      r31

;-------------------------------------
;       WHEN press()
;-------------------------------------
off_press j     on              ;NEXT on

;-------------------------------------
;       WHEN tick()
;-------------------------------------
        jr      r31             ;Ignore it


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       STATE dim
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
dim     addi    r1,r0,DIM_ST    ;set state
        sw      state,r1
        addi    r1,r0,70        ;set light
        ;(more here)
        addi    r1,r0,DIM_TIME  ;init timer
        sw      timer,r1
        jr      r31

;-------------------------------------
;       WHEN press()
;-------------------------------------
dim_press j     on              ;NEXT on

;-------------------------------------
;       WHEN tick()
;-------------------------------------
dim_tick lw     r1,timer        ;timer>0?
        sgt     r2,r1,r0
        bf      r2,off          ;No, NEXT off
        subi    r1,r1,1         ;timer--
        sw      timer,r1
        jr      r31             ;done


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       STATE on
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
on      addi    r1,r0,ON_ST     ;set state
        sw      state,r1
        addi    r1,r0,100       ;set light
        ;(more here)
        addi    r1,r0,ON_TIME   ;init timer
        sw      timer,r1
        jr      r31

;-------------------------------------
;       WHEN press()
;-------------------------------------
on_press j      on              ;NEXT on

;-------------------------------------
;       WHEN tick()
;-------------------------------------
on_tick lw      r1,timer        ;timer>0?
        sgt     r2,r1,r0
        bf      r2,dim          ;No, NEXT dim
        subi    r1,r1,1         ;timer--
        sw      timer,r1
        jr      r31             ;done

;       end of machine
```

Figure 5.8: The Apartment light Mealy-Moore FSM, in DLX

5.6 The *state* pattern

There is a design-pattern called *state*, that the object-oriented programming community uses to implement Finite state machines. In the pattern, we write a separate (private) class to describe the behaviour of each state of the FSM, and then exploit the dynamic-dispatch mechanism of the object-oriented language to transfer control to the required code.

Exploiting dynamic dispatch eliminates the need for the `switch` statement that appears in the FSMs shown earlier. To handle the entry actions, we write the code in the constructor for each private class. While eliminating the `switch` is attractive, we now have a large amount of "clutter" in the program as a result of all the private classes. It is not clear which form of the program is really "better".

5.6.1 Implementing the example

The Java code for the apartment light using the *state* pattern Mealy-Moore FSM is shown in figure 5.9.

5.7 The dangers of do-it-yourself

To finish off this chapter, we will show the dangers of building "do-it-yourself" state-machines, instead of following the discipline we have described here. Many programmers are tempted to say "I can get this right without a FSM". Unfortunately, this statement is rarely true. Most often, the program that results will be hard to understand, hard to modify, and will *not* be correct. In this section, we will try to convince you of the advantage of FSMs.

5.7.1 A microwave oven

If you have studied the Computer Systems course recently, you may recognise this example!

You are required to write a program to control a simple microwave oven, according to this specification:

The oven has a keyboard, with number buttons for the digits $0 \cdots 9$, and three other buttons labelled $Start$, $Stop/Clear$ and $Door$. There is a numeric display, to show the cooking time remaining. Inside the oven is a *light* to make it easy to see the food, and show when the oven is on, a *turntable*, to rotate the food, a *magnetron* that generates the microwave energy to cook the food, and a *beeper* to indicate when cooking is complete.

When the oven is first started, it must display $--\,--$ (four dashes), to indicate that it is ready. To enter a cooking time, simply press the number keys. If you keep on pressing number keys, the display only shows the last four keys you pressed. Your program must suppress leading zeros from the display (show the number 12 as 12, not 0012). To clear the displayed value, press the $stop/clear$ button, and the display will return to $--\,--$.

To start cooking the food, press the $Start$ key. Your program must cause the turntable to rotate, the light to be turned on, and the magnetron to be activated. While the timer value is non-zero, your program should decrement the displayed value once each second.

When the timer reaches zero, your program must stop the turntable, turn off the light, turn off the magnetron, and then sound the beeper for three seconds.

```
public class StatePatternFsm
{
    //Tuning parameters
    //(Time in 1-second units)
    private static final int ON_TIME= 300;
    private static final int DIM_TIME= 120;

    private Light light;
    private State state;
    private int timer;

    public StatePatternFsm(Light light)
    {
        this.light= light;
        initialise();
    }

    public void initialise()
    {
        state= new OffState();
    }

    public void press()
    {
        //Forward it to the current state
        state.press();
    }

    public void tick()
    {
        //Forward it to the current state
        state.tick();
    }

    ////////////////////////
    //Prototype state-class
    ////////////////////////
    private abstract class State
    {
        public abstract void press();
        public abstract void tick();
    }

    ////////////////
    //OffState-class
    ////////////////
    private class OffState extends State
    {
        public OffState()
        {
            light.set(0);
        }

        public void press()
        {
            timer= ON_TIME;
            state= new OnState();
        }

        public void tick()
        {
            //Ignore it
        }
    }

    ////////////////
    //DimState-class
    ////////////////
    private class DimState extends State
    {
        public DimState()
        {
            light.set(70);
        }

        public void press()
        {
            timer= ON_TIME;
            state= new OnState();
        }

        public void tick()
        {
            timer--;
            if( timer>0 ){
                return;
            }

            state= new OffState();;
        }
    }

    ///////////////
    //OnState-class
    ///////////////
    private class OnState extends State
    {
        public OnState()
        {
            light.set(100);
        }

        public void press()
        {
            timer= ON_TIME;
        }

        public void tick()
        {
            timer--;
            if( timer>0 ){
                return;
            }

            timer= DIM_TIME;
            state= new DimState();;
        }
    }
}
```

Figure 5.9: Apartment ligh implemented using the *state* pattern

If, during cooking, you press the $Stop/Clear$ button, the turntable must stop, and the magnetron must be turned off. The light must remain on, and the timer must hold its present value. If you press the $Stop/Clear$ button a second time, the timer is cleared, the light is turned off, and the display returns to $--:--$.

If, while cooking is suspended, you press $Start$, cooking resumes at the current timer setting.

No matter what the oven is doing at the time, whenever you press the $Door$ button, your program must stop the cooking process (if it is currently cooking), unlock the door and turn on the light. When the door has been closed again, your the oven should display whatever it was doing just prior to the door being opened. Note that if cooking was interrupted when the door was opened, it does *not* automatically resume as soon as the door is closed. You must press the $Start$ button to resume cooking, or the $Stop/Clear$ button to terminate cooking.

Wherever the specification is incomplete, you should arrange that the oven behaves in a way that a user would find "reasonable".

If you had *not* attended these lectures, you might have created a Java program like the one shown in figure 5.10.

Before you read any further, read the program, and decide whether the program behaves according to the specification.

5.7.2 Discussion

The specification for the behaviour does not seem so complex — after all, it is *only* a microwave oven!

There are three events driving the machine: `press(char ch)` that handles button-pressses, `tick()`, that handles clock-ticks, and `doorClosed()`, that handles door closure.

Despite all this, the program behaviour is not obvious. There are three boolean variables: $cooking$, that is true when the oven is cooking, $doorOpen$, that is true when the door is open, and $beeping$, that is true when the oven is beeping. There are also two timers: $cookTimer$, that keeps track of cooking time, and $beepTimer$, that handles beeping. From the program point of view, the important aspect of a timer is whether it has expired (is zero) or has not-expired (is non-zero).

We thus have five boolean properties, each with two possible states ($false$ or $true$). Overall, there are 32 unique combinations for the values of these variables. Our "simple" oven potentially has 32 states!

In reality, we know some of the states are mutually exclusive: For example, $doorOpen$, $cooking$, and $beeping$ are all mutually exclusive — if the door is open, the oven *should not* be cooking, and should not be beeping. Also, if the $cookTimer$ is being used to time a cooking cycle, the $beepTimer$ is *not* being used. Clearly this reduces the actual number of states — but by how much?

```
public class BadOven
{
    private static final char DOOR= 'd';
    private static final char RUN= 'r';
    private static final char STOP= 's';

    private boolean cooking;
    private boolean doorOpen;
    private boolean beeping;

    private Timer cookTimer= new Timer();
    private Timer beepTimer= new Timer();
    private Door door= new Door();
    private Magnetron magnetron= new Magnetron();
    private Light light= new Light();
    private Beeper beeper= new Beeper();

    public void press(char key)
    {
        if( key==DOOR ){
            if( beeping ){
                beeper.off();
                beeping= false;
            }

            door.unlock();
            magnetron.off();
            light_on();
            cooking= false;
            doorOpen=true;
        }

        if( key==RUN ){
            if( !doorOpen && !cooking &&
              !cookTimer.isZero() ){
                cooking= true;
                magnetron.on();
                light.on();
            }
        }

        if( key==STOP ){
            if( cooking ){
                cooking= false;
                magnetron.off();
            }else{
                light.off();
                cookTimer.set(0);
            }
        }

        if( '0'<= key && key<='9' ){
            //Number key
            cookTimer.addDigit(key);
        }
    }


    public void tick()
    {
        if( cooking ){
            cookTimer.decrement();
            if( cookTimer.isZero() ){
                magnetron.off();
                cooking= false;

                beepTimer.set(30);
                beeper.on();
                beeping= true;
            }
        }

        if( beeping ){
            beepTimer.decrement();
            if( beepTimer.isZero() ){
                beeper.off();
                light.off();
            }
        }
    }

    public void doorClosed()
    {
        door.lock();
        doorOpen= false;
        if( cookTimer.isZero() ){
            light.off();
        }
    }
}
```

Figure 5.10: A bad way to build a microwave oven

5.7.3 The right way to build an oven controller

The right way to implement this controller is to design an FSM that gives the desired behaviour, and then build a program from the diagram. Figure 5.11 shows a FSM diagram that matches the specification above.

The diagram contains just four states! *Far* fewer than the 32 that our naive solution used. By "playing" with this diagram, we can determine *precisely* what will happen when the oven is operated. There are *no* ambiguous cases, and therefore *no* unspecified or unexpected behaviours.

The diagram can be turned directly into a Mealy-Moore style FSM in Java, in a few minutes, using the technique shown earlier.

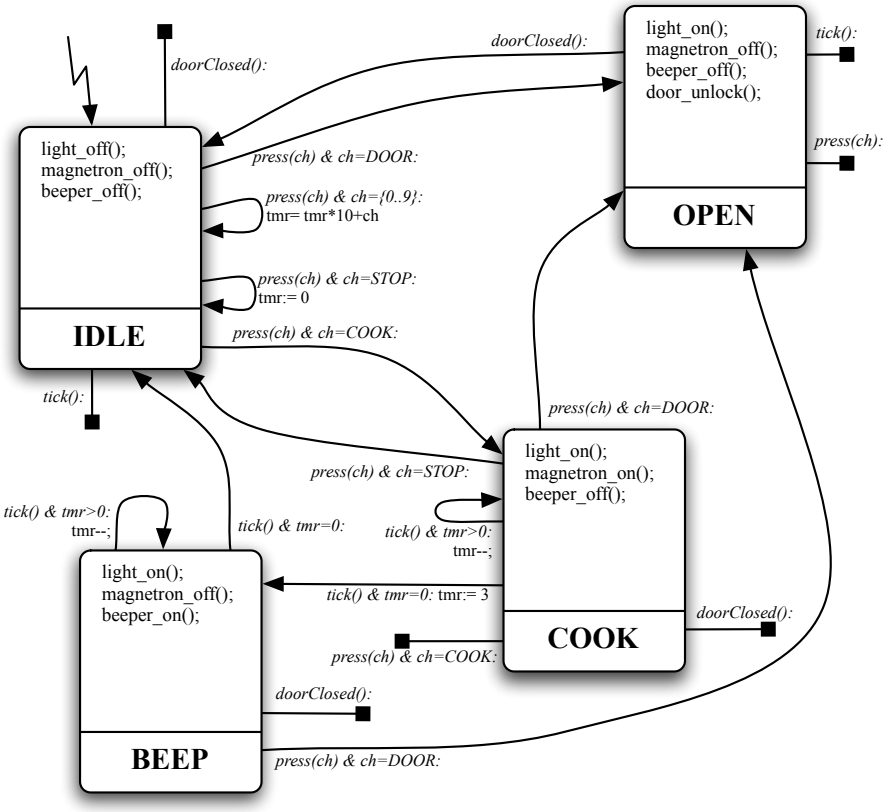


Figure 5.11: FSM for the microwave oven

5.7.4 The moral of the example

The important lesson to learn from this example is that the *moment* the control logic of a program starts to be complex (as soon as you have more than *two* states!), you should build a formal FSM. Any attempt to use "tricky boolean flags", or to exploit strange values of variables is likely to cause far more problems than it solves.

The *hard* part of the microwave oven controller is creating the state diagram. But effort spent at this stage pays off handsomely during debugging and when modifications are needed in future. Converting the diagram into a Java program is the *easy* part!

Our bad oven had 32 states (based on the 5 boolean properties), but only four of them really existed. Exactly what did the oven do in the other 28 states? Was it possible to accidentally enter

one of those “unused states”? Was it hazardous to the user? These questions are extremely difficult to answer!

Chapter 6

Petri nets

6.1 Introduction

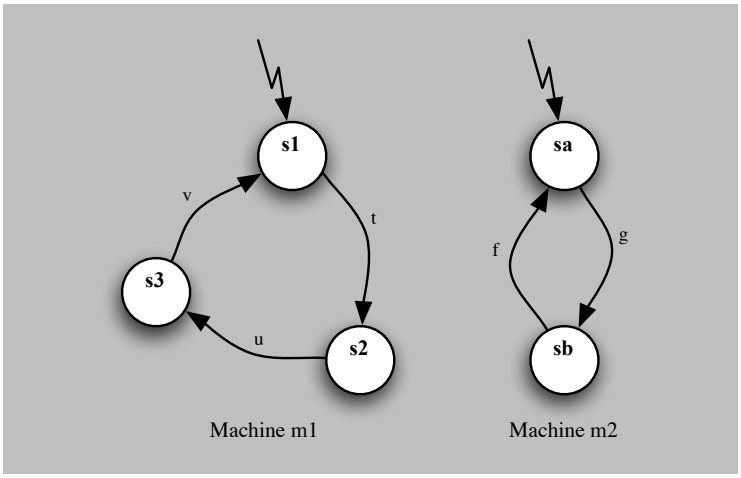
So far we have considered systems that consist of only a single FSA. In practice, systems do not have just *one* FSA, they are instead composed of multiple FSAs interacting with one another. We will now examine the problems that arise when we deal with multiple interacting FSAs.

The first papers on finite state machines were published in the mid 1950s. Not long after, in the early 1960s, a German researcher named Carl Petri published the results of his investigations of interacting finite state machines, and introduced a notation that is now known as a *Petri Net*.

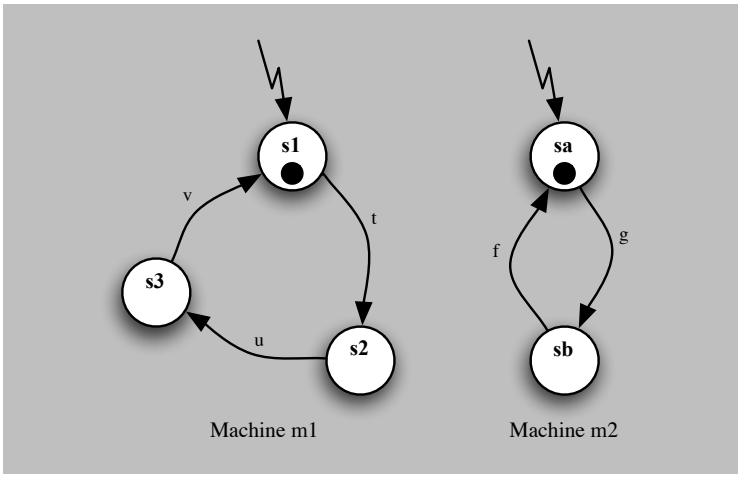
6.1.1 The origin of Petri-nets

We don't actually know how Petri got his idea, but the following story is probably (more-or-less) correct.

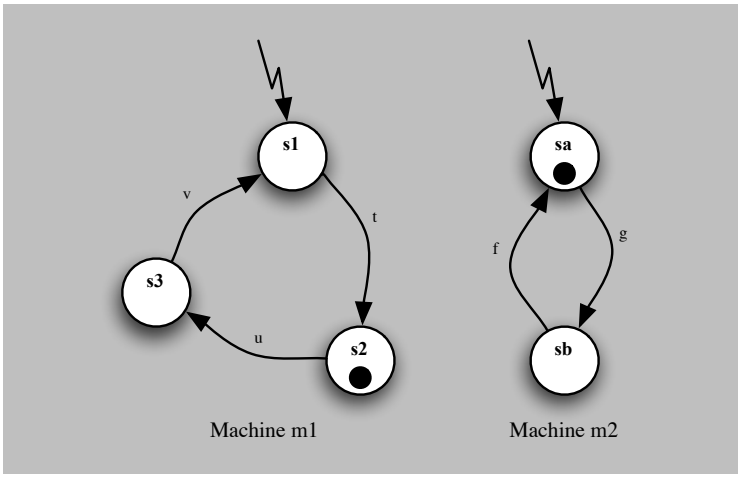
Consider two simple finite-state machines, like this:



We can show the state of an FSA by drawing a *black dot* in the current state. Immediately after machine m1 has executed its initial transition, it will be in state $s1$. Similarly machine $m2$ will be in state sa. We represent this on the diagram like this:

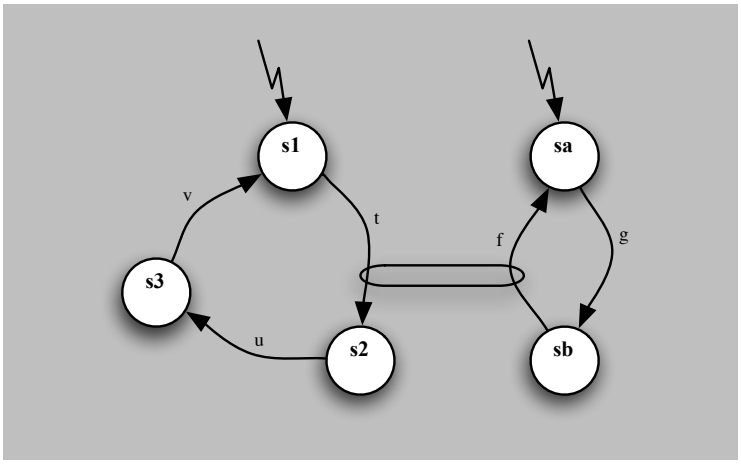


If machine $m1$ performs a transition t, it enters state $s2$. Notice that machine $m2$ does not change state. We show the new state like this: (The dot for $m1$ now appears in state $s2$.)

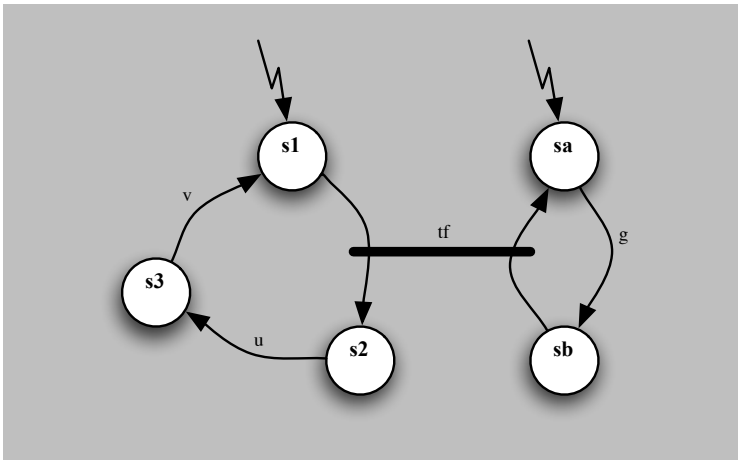


You can readily imagine the effect on the diagram of performing transition u, then g, then v.

Now let us couple the two FSAs to form a system, subject to the constraint that the transitions $m1.t$ and $m2.f$ must always occur together. We could represent this in a diagram by "tying the transitions together" with a loop, like this:



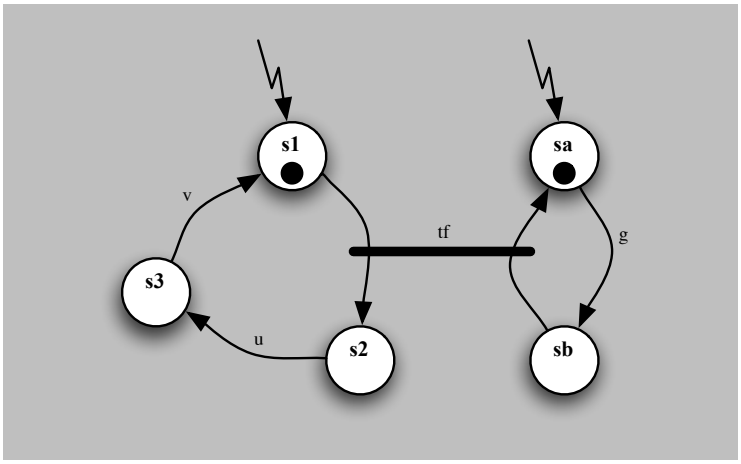
Petri realised that once two transitions are *synchronised* there can only be one name, so he changed the loop into a thick line with the transition name on it.
Let us name the synchronised t and f transition, as tf, like this:



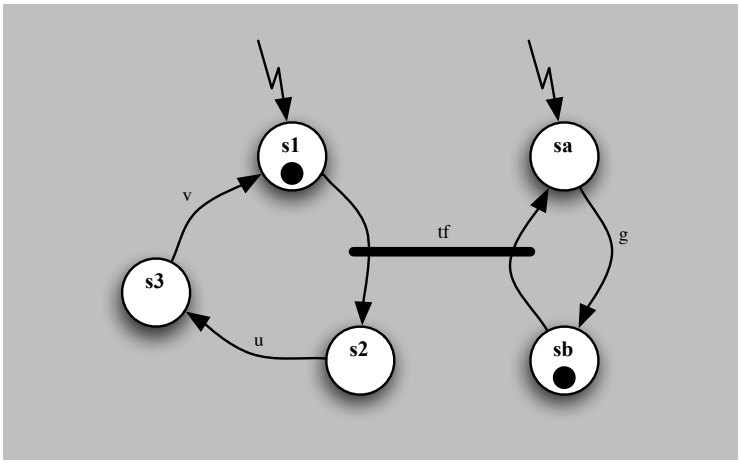
What does this diagram mean?

We begin by forcing the two coupled FSAs to their initial states, through the initial transition. The diagram will look like this:

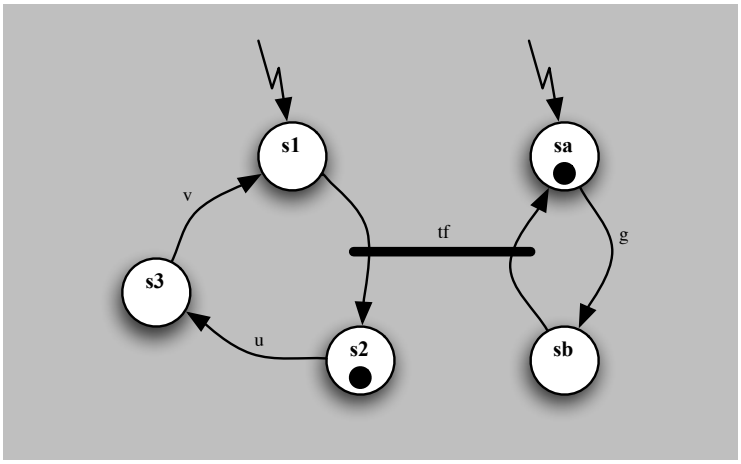
The black dots show the two machines are in states $s1$ and sa.



We cannot perform the combined tf transition, because machine $m2$ is not in state sb. The only transition we can perform is g, which causes machine $m2$ to enter state sb. The diagram now looks like this:

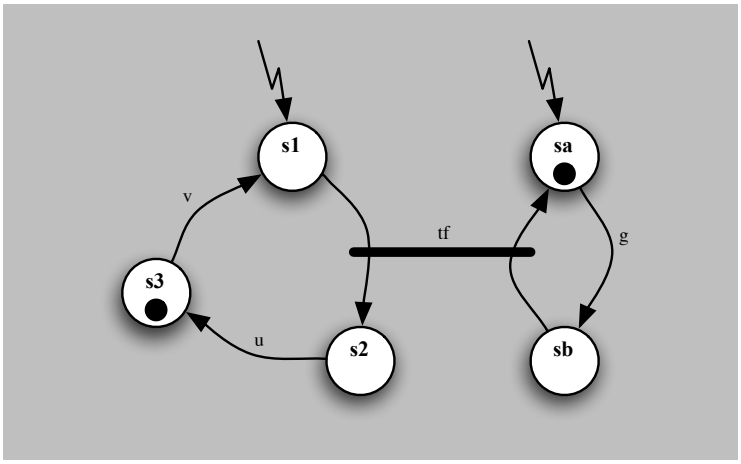


The states of the two machines are such that transition tf is now possible. If we take this synchronised transition, machine $m1$ enters state $s2$, and *simultaneously* machine $m2$ enters state sa. Here is the diagram now:



From here, there are two possibilities: we could take transition g, and the two machines would be in states $s2$ and sb; or we could take transition u and the machines would be in states $s3$ and sa.

Here is the diagram for the second possibility:



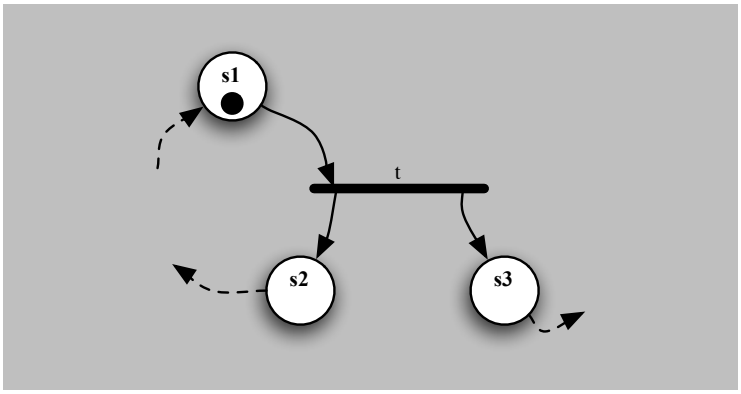
If we continue to explore this machine we will discover the complete behaviour of the machines. Assuming the machines start in their initial states, we can express this behaviour using a regular expression:

$$g\ (\ tf\ (\ g\ u\ v\ |\ u\ g\ v\ |\ u\ v\ g\ )\ )^*$$

6.1.2 Some special cases

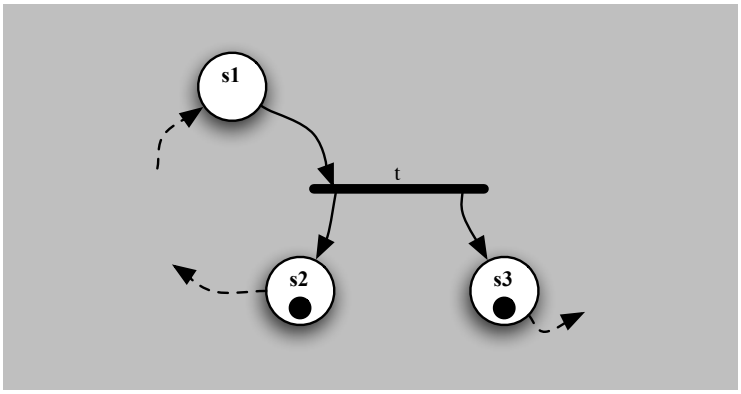
If we play with these diagrams, some interesting special cases come to mind. Consider this diagram:

There is one arrow from state $s1$ to transition t, but *two* arrows leading from t to states $s2$ and $s3$. What does this mean?



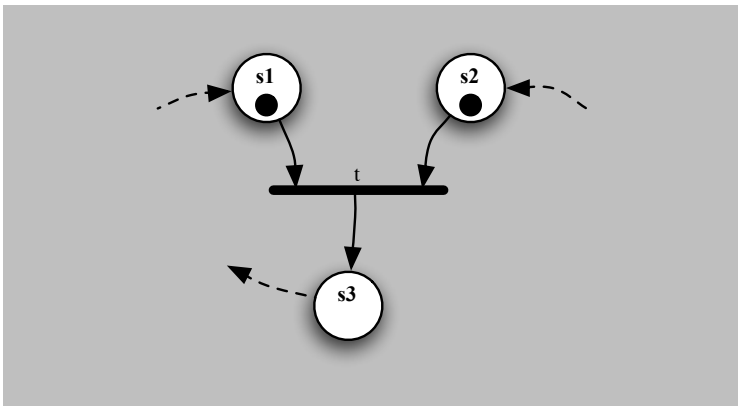
It seems sensible that when the transition t is taken, the system ends up in *two* states, $s2$ and $s3$, like this:

This construction is called *fork*.

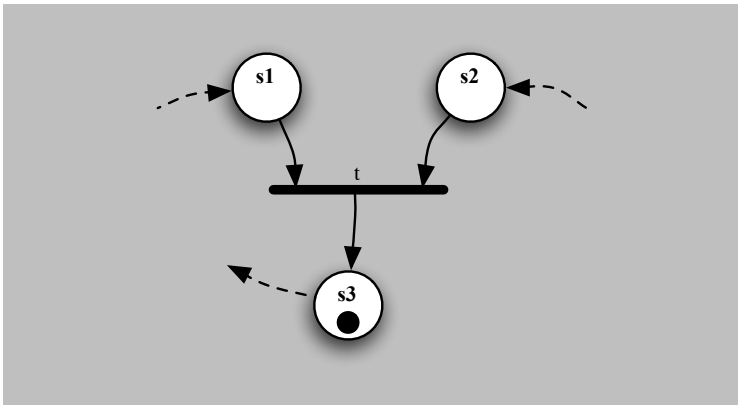


Here's another special case:

There are *two* states, $s1$ and $s2$, leading *to* the transition, t, but only one state, $s3$ afterwards.



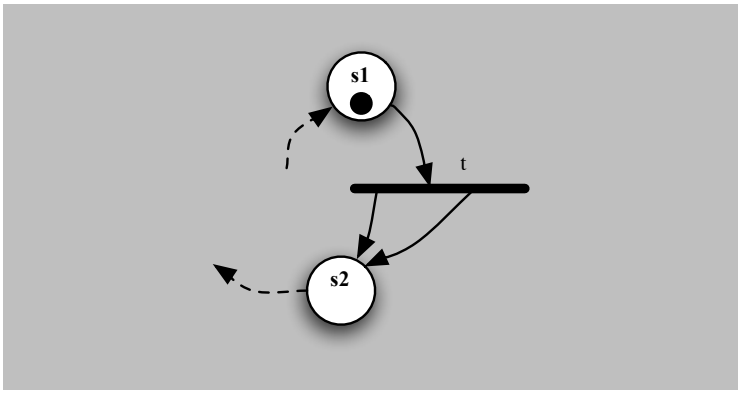
Logically, the result of taking transition t should be that the system ends up in state $s3$, like this:
This operation is called *join*.



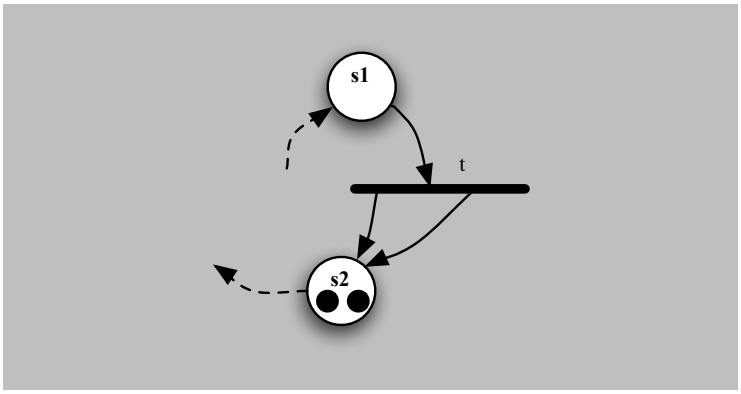
The two special cases we have just looked at show that dots are not always "conserved" by a transition — more dots can go into a transition than come out, and vice versa.

Here's another special case:

What does it mean to have *two* arrows going from a transition to a state?

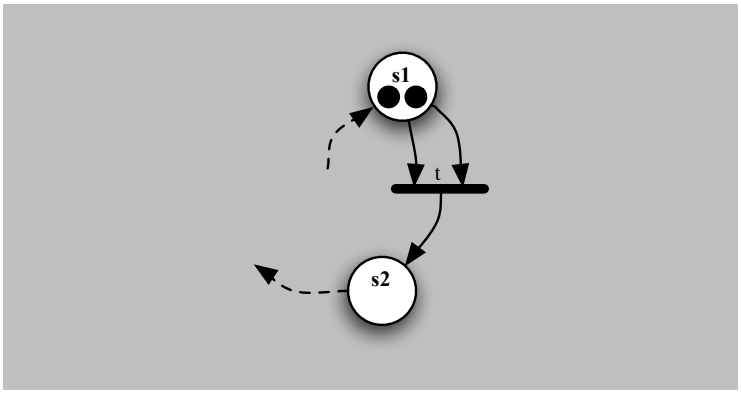


If we are to be consistent with our earlier decision, when the transition t is taken, one dot will go in, two will come out, and we will end up in state $s2$ *twice*, like this:

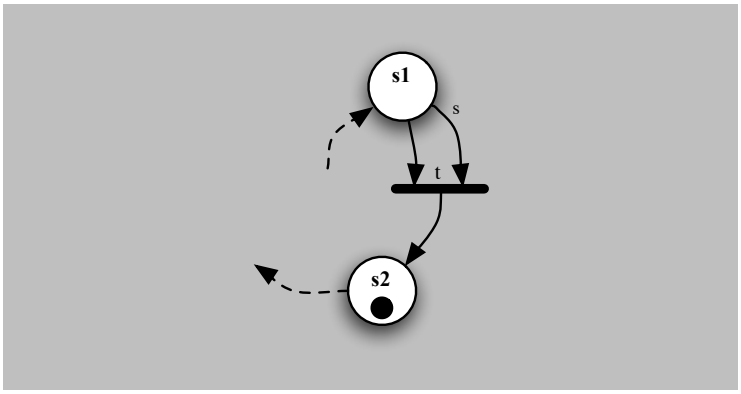


Symmetry suggests that this diagram:

with two arrows leading into a transition, will require state $s1$ to contain *two* dots before the transition t can be taken.

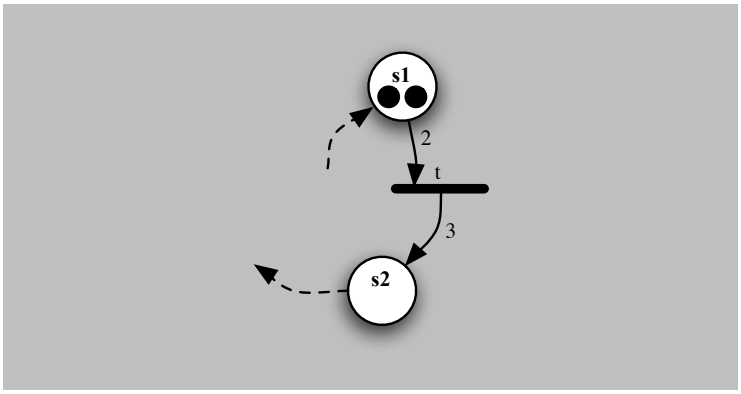


After taking the transition, t, the diagram will look like this:

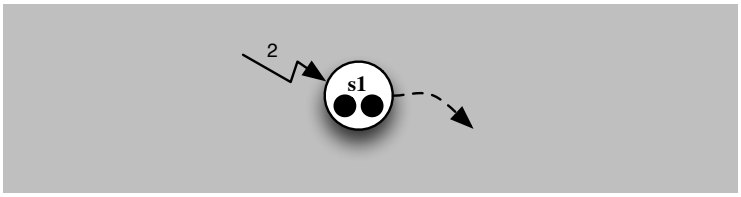


We can avoid drawing multiple parallel arrows by writing a *weight* alongside each arrow, like this:

We will require two dots (because of the two written beside the arrow) in state $s1$ *before* we can take transition t. *After* we take transition t, we remove two dots from $s1$, and add three dots (because of the three written beside the arrow) to state $s2$. By convention, arrows without numbers are taken to have a weight of one.



When we allow multiple dots in each state, we need to provide a way to initialise a state to contain multiple dots. We do it by writing a number beside the lightning-strike, to show how many dots are created. By convention, we omit the number if only one dot is created.



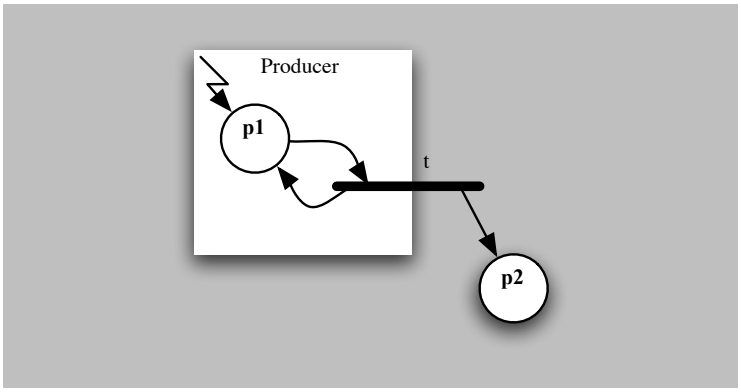
As a matter of historical interest these last cases, where we permit a state to have more than one dot, were not considered by Petri in his thesis. Later researchers generalised the notation to handle multiple dots, and called the notation "Place-transition nets".

6.2 Some examples

6.2.1 Producer

Consider this network:

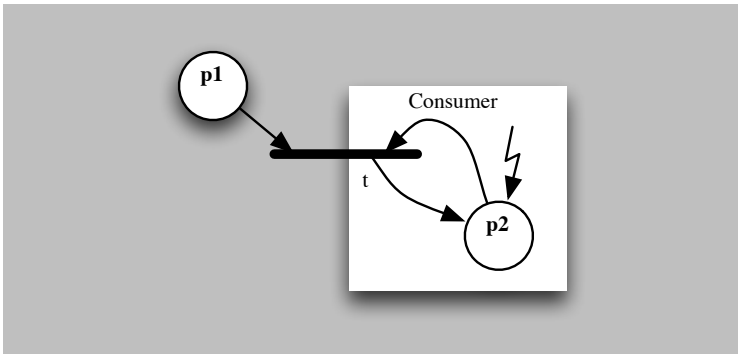
The diagram enclosed within the box can be considered as a *producer* of dots — it will produce a continuous stream of dots to place $p2$.



6.2.2 Consumer

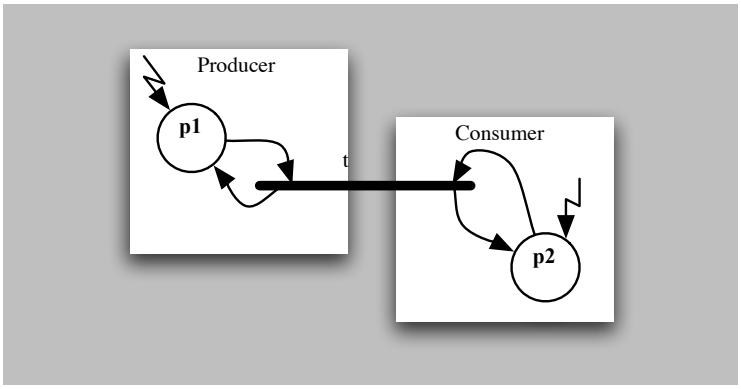
Consider this network:

The diagram enclosed within the box can be considered as a *consumer* of dots — However many dots appear in place $p1$, they will all be consumed.



6.2.3 Producer-Consumer

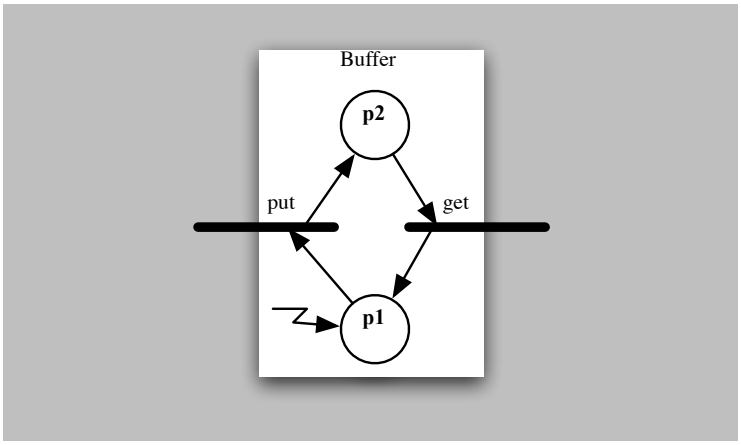
We can couple these two networks together, to model a simple producer-consumer system, like this:



6.2.4 Simple buffer

Consider this diagram:

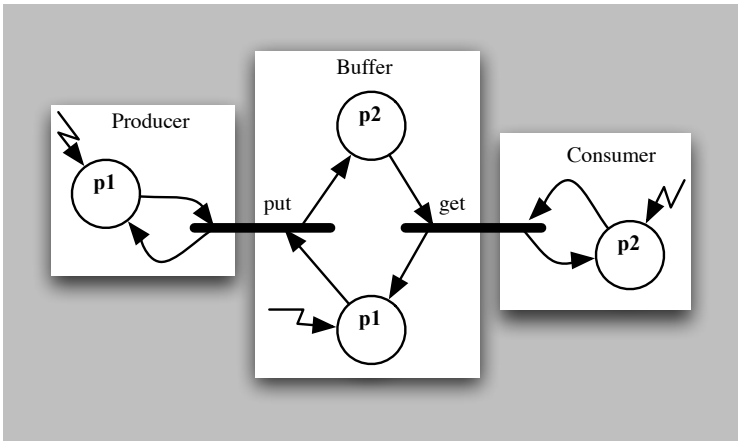
Whenever a dot is fed into the put transition, the buffer stores it. Later, when the buffer is able to execute a get transition, the dot is delivered, and the buffer is ready to receive another dot. Notice that the buffer can only ever hold one dot at a time.



6.2.5 Producer-buffer-consumer

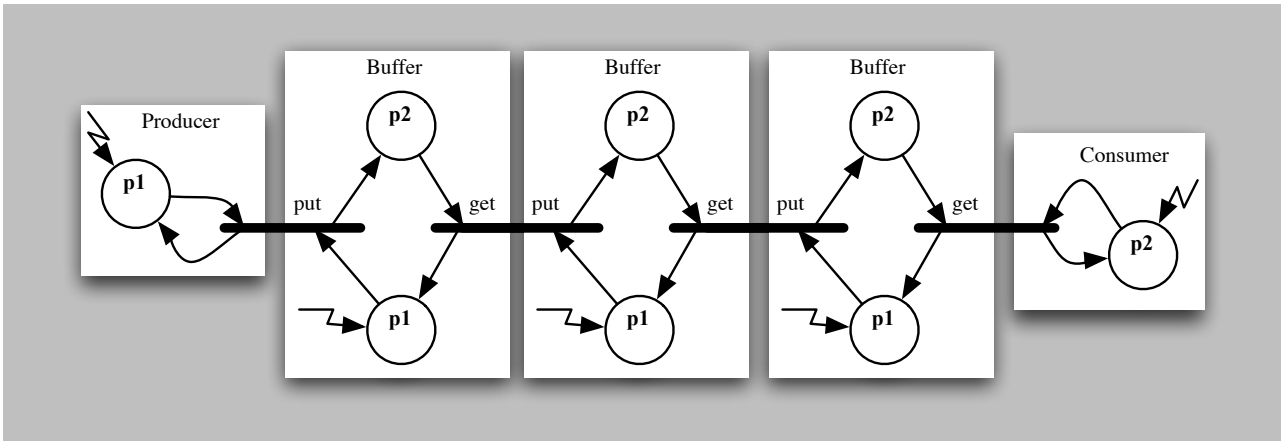
We can couple the producer, buffer, and consumer together, to make a complete system, like this:

The producer can only generate one dot, and pass it to the buffer, before it must stop. The consumer must wait until the buffer has a dot available, before it can proceed. When the buffer is empty again, the producer is able to generate a new dot.



6.2.6 Three-place buffer

We can make a three-place buffer by coupling three of our one-place buffers together, like this:



6.3 Place-transition nets — formally

Now that you understand the origins and principles of Petri-nets, we can provide a more formal definition. A place-transition net is a five-tuple:

$$N = (P, T, A, W, I)$$

where:

P — is a set of *places* (In FSAs, places are called *states*)

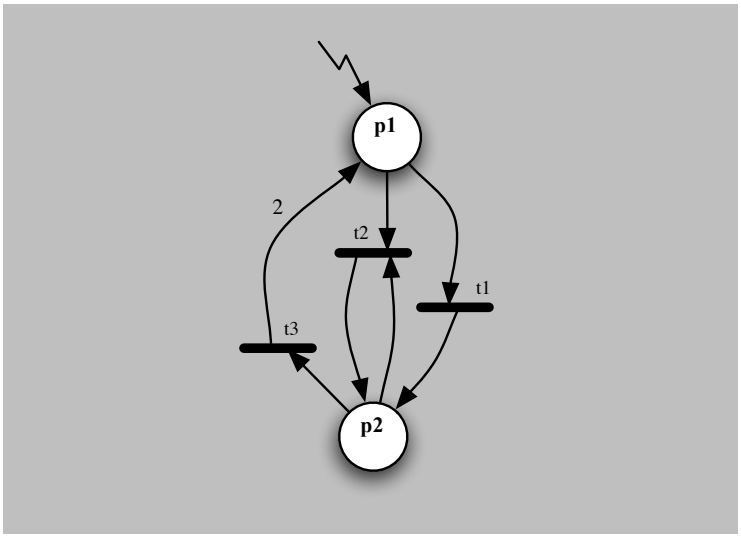
T — is a set of *transitions*

A — is a set of *arcs* between places and transitions. An arc that begins at a place *must* end on a transition. An arc that begins on a transition *must* end on a place. We can express this mathematically by writing: $P \cap T = \emptyset$ and $A \subseteq (P \times T) \cup (T \times P)$

W — is a set of *weights*. A weight is associated with an arc, and indicates the number of dots that must pass along the arc when its associated transition is taken. Clearly, a weight must be a non-negative integer. Mathematically, we can write: $W : A \to \mathcal{N}_0$

I — Is an *initialisation*, that shows the number of dots that will be present in each place after the machine is initialised. The initialisation is the number that must be placed beside the *initial transitions* in the network. We can represent this mathematically as a function: $I : P \to \mathcal{N}$

Consider this place-transition network:



We see: $P = \{p1, p2\}$
$T = \{t1, t2, t3\}$
$A = \{(p1, t1), (t1, p2), (p1, t2), (p2, t2), (t2, p2), (p2, t3), (t3, p1)\}$

$$W : \left\{ \begin{array}{l} (p1, t1) \rightarrow 1 \\ (t1, p2) \rightarrow 1 \\ (p1, t2) \rightarrow 1 \\ (p2, t2) \rightarrow 1 \\ (t2, p2) \rightarrow 1 \\ (p2, t3) \rightarrow 1 \\ (t3, p1) \rightarrow 2 \end{array} \right\}$$

$$I : \left\{ \begin{array}{l} p1 \rightarrow 1 \\ p2 \rightarrow 0 \end{array} \right\}$$

Notice that when an arc has a weight of one, we omit the "1" beside the arc. Similarly, if an initialisation generates one dot, we omit the "1" beside the initial transition.

6.3.1 Notion of marking

A *marking* of a Place-transition net is a record of the number of dots in each place. Clearly, a place must always have a non-negative number of dots. Formally, we can define the marking M like this:

$$M : p \rightarrow \mathcal{N}_0$$

When a place transition network is intialised, the initial marking is determined by the *initialisation* specified for the network.

Formally: $M_0 = I$

6.3.2 Pre-set and post-set

The *pre-set* of a place p, written $\bullet p$, is the set of all transitions, t, that have an arc to p. Similarly, the pre-set of a transition t, written $\bullet t$, is the set of all places, p, that have an arc to t. We can express both of these definitions formally, like this:

$$\bullet y = \{x \mid (x, y) \in A\}$$

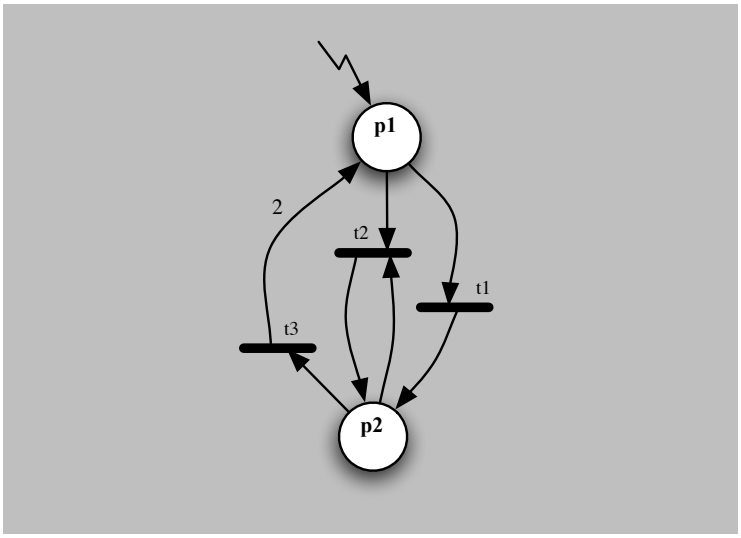
The *post-set* of a place p, written $p\bullet$, is the set of all transitions, t, that have an arc from p. Similarly, the post-set of a transition t, written $t\bullet$, is the set of all places, p, that have an arc from t. We can write this formally as:

$$x\bullet = \{y \mid (x, y) \in A\}$$

The pre-set and post-set are useful for defining the dynamics of a place-transition network.

Consider this network:

We see:
$\bullet p1 = \{t3\}$
$\bullet t2 = \{p1, p2\}$
$p1\bullet = \{t1, t2\}$
$t2\bullet = \{p2\}$



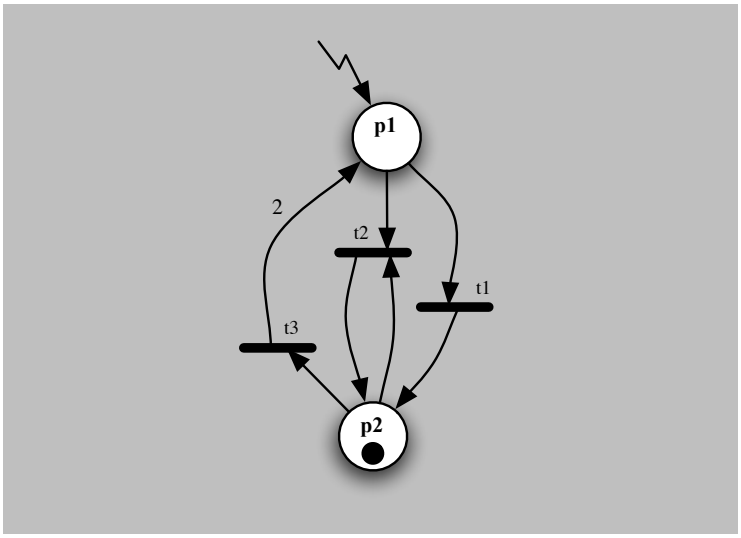
6.3.3 Enabled transitions

A transition in a place-transition network is *enabled* when there are enough dots in each place leading *to* that transition to meet the weight-requirements of each arc to the transition. A network can have zero or more enabled transitions. Formally, the set of transitions that are enabled when the network marking is M, is given by E(M), where:

$$E(M) = \{t \in T \mid M(p) \geq W(p,t)\ \forall p \in \bullet t\}$$

Consider this network:

Transition $t3$ is enabled because there is one dot in place $p2$. Transition $t2$ is not enabled, because it requires dots in both $p1$ and $p2$.



6.3.4 Firing a transition

The state of the network changes when an enabled transition is selected to *fire*. If there are multiple enabled transitions, one is selected according to a *selection policy*. If there are no enabled transitions, the network has reached *deadlock*.

When a transition fires, dots are removed from the places preceding the transition (in accordance with the weight on each ingoing arc), and dots are added to the places following the transition (in accordance with the weight on each outgoing arc).

Formally, for an enabled transition, $t \in E(M_n)$, the marking, M_{n+1}, of the network after firing transition t is given by:

$$M_{n+1} = \delta(M_n, t)$$

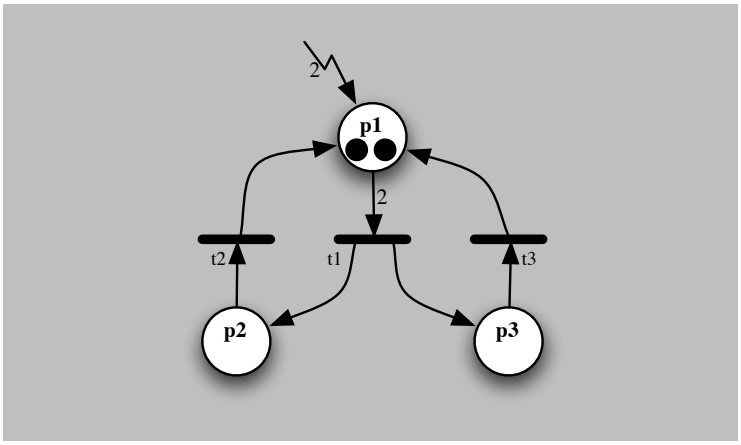
where, $\forall p \in P$:

$$\delta(M,t) = \left\{ p \rightarrow M(p) - \left\{ \begin{array}{r} W(p,t) : p \in \bullet t \\ 0 : \text{otherwise} \end{array} \right\} + \left\{ \begin{array}{r} W(t,p) : p \in t\bullet \\ 0 : \text{otherwise} \end{array} \right\} \right\}$$

Consider this place-transition network, that has just been initialised. The initial marking M_0, will be:

$$M_0 : \left\{ \begin{array}{l} p1 \rightarrow 2 \\ p2 \rightarrow 0 \\ p3 \rightarrow 0 \end{array} \right\}$$

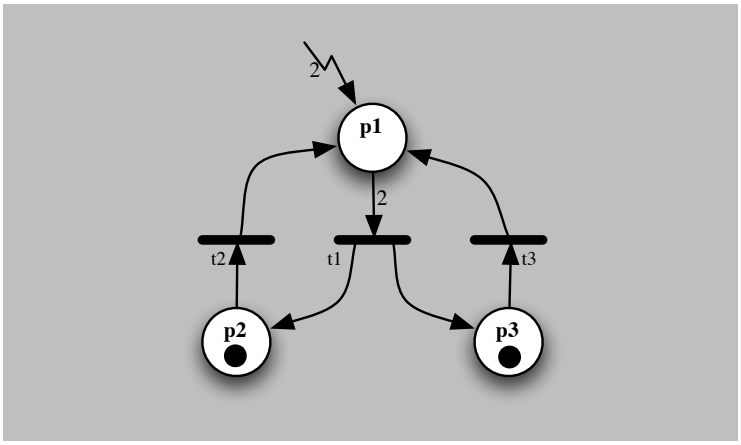
The only enabled transition is $t1$, which requires two dots for it to fire.



If we fire transition $t1$, the marking changes to M_1, which looks like this:

$$M_1 : \left\{ \begin{array}{l} p1 \rightarrow 0 \\ p2 \rightarrow 1 \\ p3 \rightarrow 1 \end{array} \right\}$$

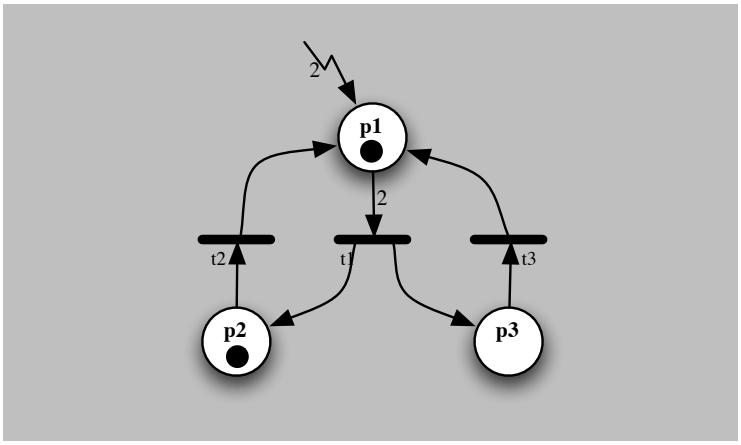
There are now two enabled transitions, $t2$ and $t3$, each of which requires only one dot to fire it.



Let us (arbitrarily) choose to fire transition $t3$. (Instead, we could have chosen to fire transition $t2$.) The marking changes to M_2, which looks like this:

$$M_2 : \left\{ \begin{array}{l} p1 \rightarrow 1 \\ p2 \rightarrow 1 \\ p3 \rightarrow 0 \end{array} \right\}$$

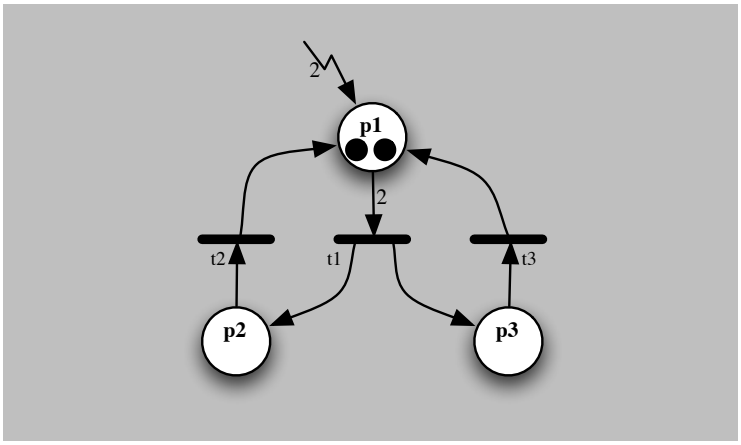
There is now only one enabled transitions, $t2$. $t1$ is not enabled, because the weighting requires two dots to be in place $p1$ before it can fire.



If we now fire transition $t2$, the marking changes to M_3, and looks like this:

$$M_3 : \left\{ \begin{array}{l} p1 \rightarrow 2 \\ p2 \rightarrow 0 \\ p3 \rightarrow 0 \end{array} \right\}$$

Clearly this is, the same as M_0, the marking we started with.



We can express the permissible sequence of transition-firings for this with the regular expression:

$$(t1\ ((t2\ t3) \mid (t3\ t2))\ )^*$$

6.3.5 Firing policy

In all our examples in these notes we will use the firing policy *Randomly choose and fire an enabled transition*. However, other policies are possible, for example:

- We could choose one or more transitions to fire simultaneously. This policy models reality, where in practice multiple events can occur simultaneously.
- We could adopt a simple policy, of always firing the first enabled transition that we find. This policy might be useful if we were writing a software simulation tool.

6.4 More properties of place-transition networks

Let us examine some of the implications of the behaviour of place-transition networks.

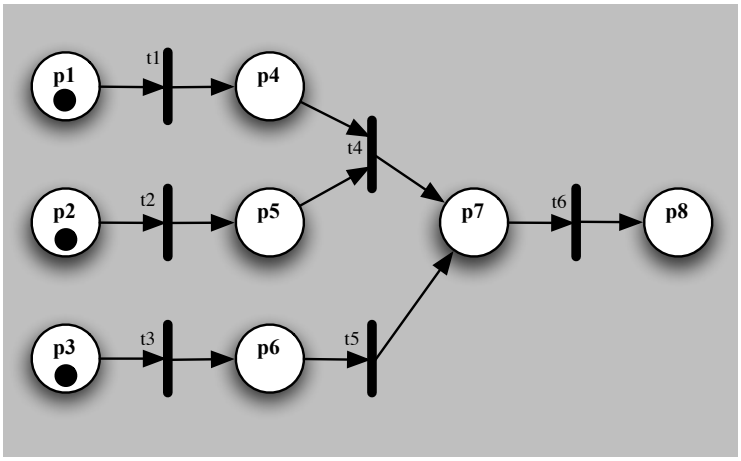
6.4.1 Causality

The construction of a place-transition network tells us a lot about the order in which transitions are permitted to occur.

Consider the network shown here:

We can see from the diagram that:

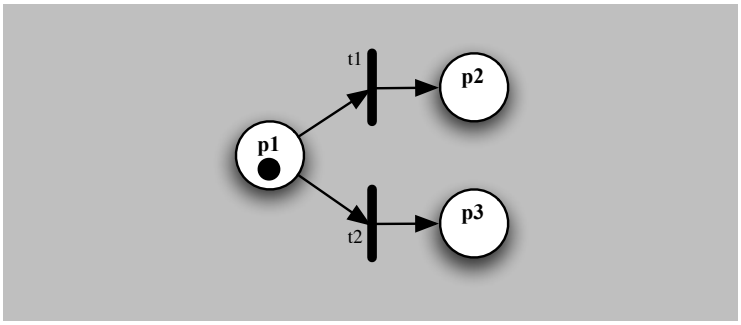
- transition $t3$ must fire before transition $t5$ can fire;
- transitions $t1$ and $t2$ must *both* fire before transition $t4$ can fire; and
- *either* transition $t4$ or $t5$ must fire before transition $t6$ can fire.



6.4.2 Non-determinism

As we have seen earlier, a place -transition network can have a marking where more than one transition is enabled and could fire.

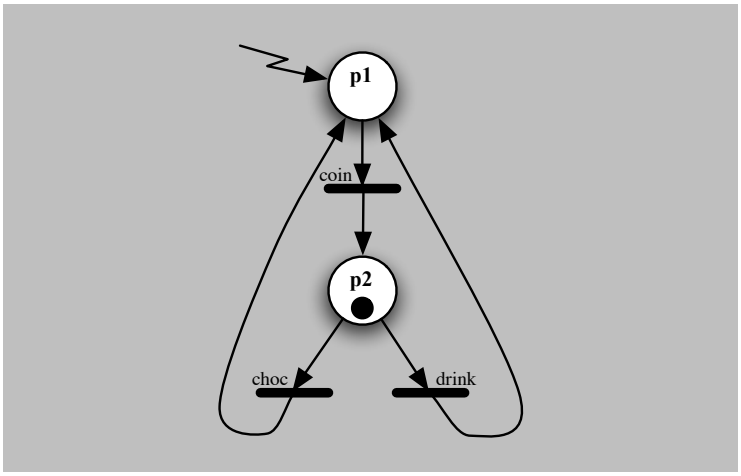
Consider this network:



Place $p1$ contains a dot, and thus transitions $t1$ and $t2$ are both enabled. We can only fire one of the transitions, since firing requires a dot to be removed from place $p1$. Which one will fire? The answer is that perhaps $t1$ will fire, perhaps $t2$ will fire — the choice is unpredictable.

We say that the situation is *non-deterministic*. Non-determinism allows a network to represent unpredictable behaviour, such as is often observed in real-world applications. Consider a simple(!) vending machine that accepts a coin and then either dispenses a chocolate or a drink, according to the wish of the user. The events that drive this machine are $coin$, $choc$ and $drink$.

Here is a place-transition network for our vending machine:



The dot in place $p2$ shows the situation after a coin has been inserted, but the customer has not yet made a choice of product. The customer could choose $choc$ or she could choose $drink$, but she *cannot* choose both. Clearly the transitions $choc$ and $drink$ are both enabled, but only one of them can fire, because there is only one dot.

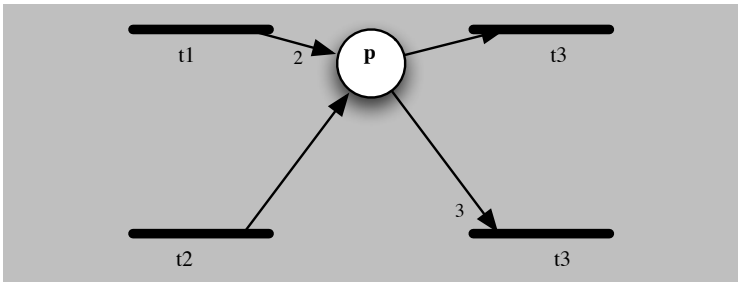
6.4.3 Capacity constraints

We can ensure that a place, p, never contains more than n dots by implementing a *capacity-constraint* for that place. We do this by building a shadow-place, p' that is initialised to n dots. Then we add transitions to the existing network containing p such that every transition t that leads *to* p has an arc (with the same weight) leading *from* p', and every transition u that leads *from* p also has an arc (with the same weight) leading *to* p'.

The effect of the extra place and arcs is to build a "dot-conserver" that only allows dots to *enter* p if there are sufficient dots already available in p'. Dots are conserved because whenever a dot is *removed* from p, it is returned to p' for future use. We have already seen this mechanism in action in the earlier buffer example.

Consider this network:

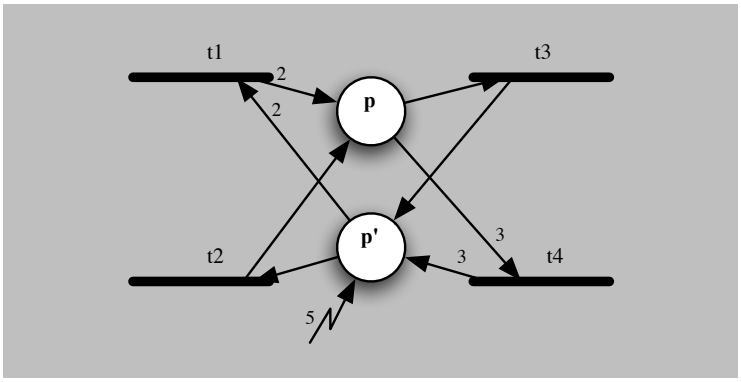
As you can see, there are four transitions connected to place p. We wish to implement a capacity constraint on place p, that restricts the number of dots it can hold to 5.



To implement the constraint, we add a new place p', and arrange that it will be initialised to contain 5 dots. Then we add new arcs to/from p' from/to each existing transition, as described earlier.

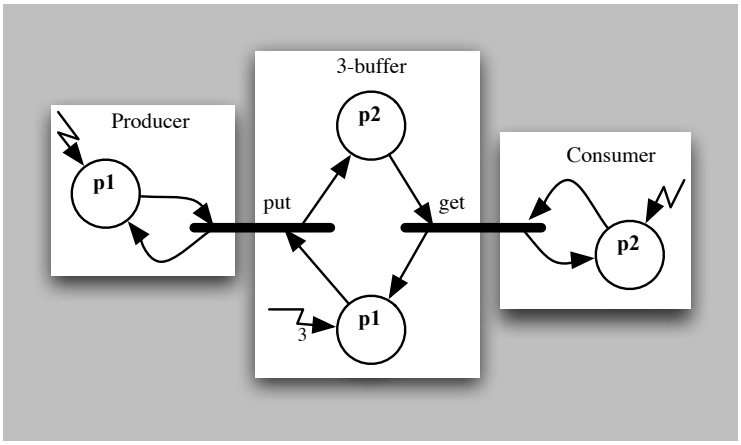
For example: the original network had an arc from $t1$ to p, with a weight of 2, so we add a new arc from p' to $t1$, with a weight of 2. There was an arc from p to $t3$, with w weight of 1, so we add a new arc from $t3$ to p', with a weight of 1.

After doing similar things for the remaining two arcs, we end up with this network:



Here is an alternative implementation of the producer-buffer-consumer example shown earlier, with a three-place buffer:

It works by initialising the buffer to contain *three* dots. It is therefore possible for the buffer to accept three items from the producer, before the producer must stop.



6.5 Reachablility analysis

Given place-transition network, we are interested in answers to these questions:

- Is a particular marking, M, reachable from the initialisation, I? If it is, what is the shortest sequence of transitions that will take the network to that marking?
- Is a particular marking, M, *always* reachable from any other marking of the network?
- Does the network suffer from *deadlock* — a marking from which it is impossible to make further transitions?

6.5.1 Reachable markings

Let us first sharpen up the definition of a reachable marking. Consider a place-transition network that currently has a marking M. What markings can be reached by taking *exactly one* transition?

We can find this by firing every possible transition that is *enabled* for this marking (i.e. in the set $E(M)$), and collecting the results into a set. The set of markings reachable from M in one step can be expressed formally by the function $R(M)$:

$$R(M) = \{\delta(M, t) \mid t \in E(M)\}$$

If we repeatedly apply R, until *every* possible sequence of transitions has been explored, we will find the set of *all* markings that are reachable starting from a marking M. We can express this formally as the function $R^*(M)$:

$$R^*(M) = \{M\} \cup \bigcup_{m\prime \in R(M)} R^*(m\prime)$$

If we apply the R^* function to the initialisation, I, of a place-transition network, we will find a set that contains *all* of the markings, that can be reached by *any* possible sequence of transition-firings in that network. We call this set I^*: Formally:

$$I^* = R^*(I)$$

6.5.2 Constructing a DFA equivalent to a PT-network

It is difficult to directly implement in software a place-transition network, because it often appears to be in multiple states at one time (there are dots in multiple places). We can convert a PT-network to an equivalent DFA using an algorithm similar to that used for converting a non-deterministic finite-state automaton (NFA) to a deterministic finite-state automaton (DFA).

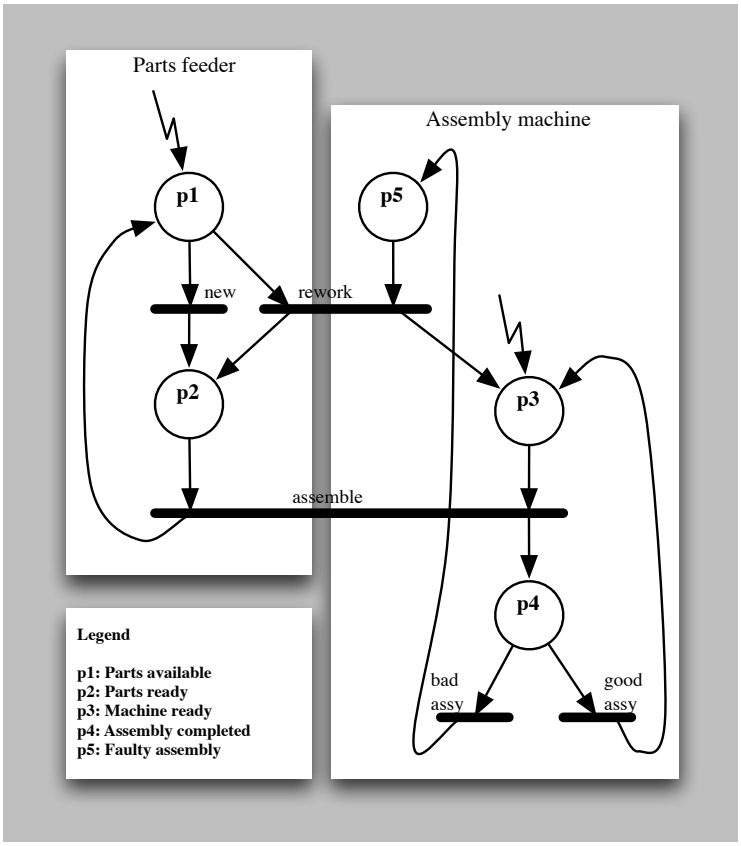
The states of the DFA are labelled to correspond to a particular marking of the PT-network. The transitions of the DFA are labelled with the names of the transitions of the PT-network.

Once we have constructed the DFA, it is then easy to answer questions about reachability and deadlock.

6.5.3 Example: Manufacturing machine

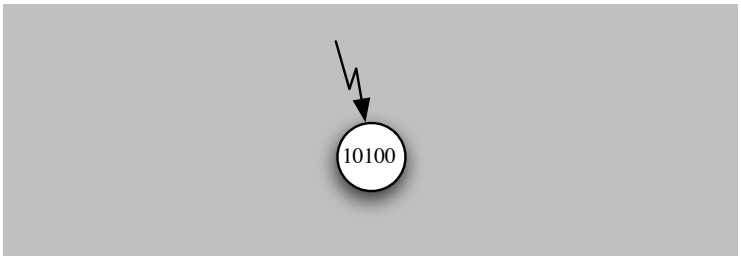
Consider this network, that represents a simple manufacturing machine:

When parts are available (p_1), the parts-feeder selects new parts, and enters p_2 (parts ready). When parts are ready (p_2) and the machine is ready, the parts are assembled ($assemble$), the parts feeder is then ready again (p_1), and the machine has a a completed product (p_4). If the assembled product is ok ($good\ assy$), the machine is immediately ready (p_3). If the machine has produced a $bad\ assy$, it has a faulty assembly (p_5). The faulty assembly and new parts are combined for $rework$, and fed back into the machine to be processed again.
You should take a few moments to play with this network, and you will probably discover that it can deadlock.

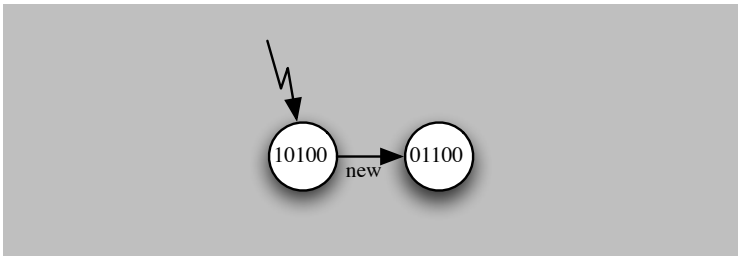


We will construct an equivalent DFA, where the states are labelled to correspond with each marking of the original place-transition network. Each state of the DFA will be labelled with a 5-digit number that represents the marking of the PT-network, written in the order $p1$, $p2$, $p3$, $p4$, $p5$.

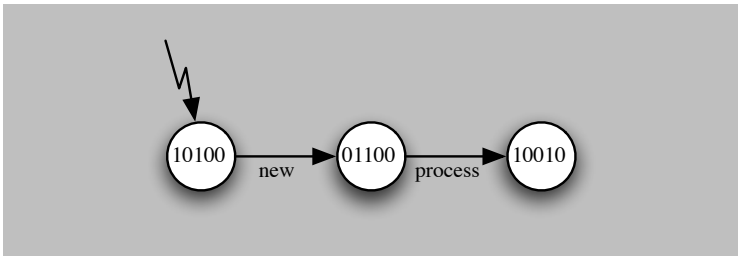
Immediately after the network is intialised, there are dots in places $p1$ and $p3$, so the marking can be represented by the number 10100. This gives the label for first state of our DFA:



With the network in this state, the only transition that is enabled is new. The marking after taking this transition is 01100, and the DFA becomes:

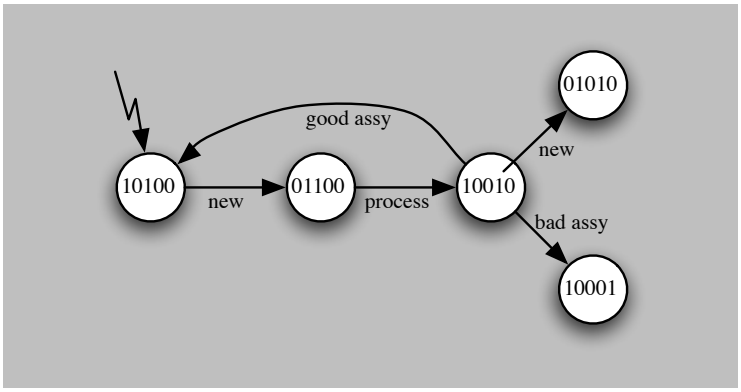


Once again, only one transition is enabled: $process$. If we now take this transition, the network now has the marking 10010. Our DFA becomes:



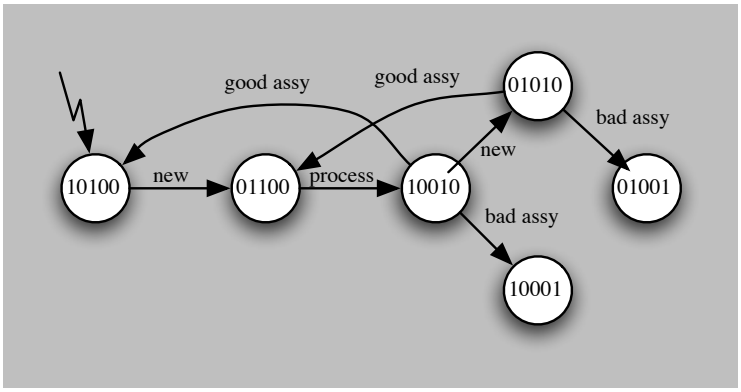
There are now three enabled transitions: new, $good\ assy$, and $bad\ assy$. Let us now investigate each of these in turn. If we fire the new transition, the marking becomes 01010. If we fire the $good\ assy$ transition, the marking becomes 10100. If we fire the $bad\ assy$ transition, the marking becomes 10001.

Thus the DFA becomes:



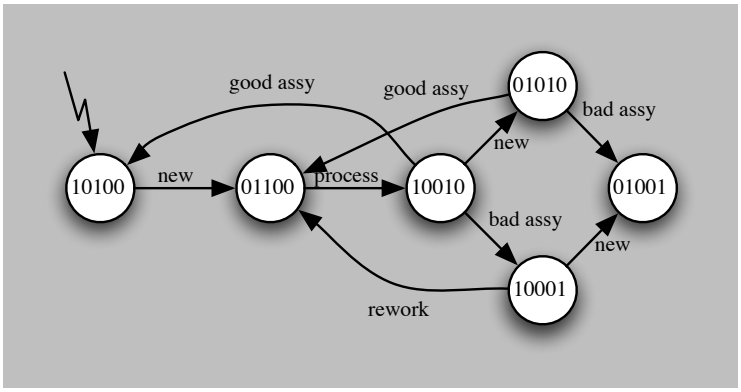
From the marking 01010 there are two enabled transitions: $good\ assy$, and $bad\ assy$. If we fire the $good\ assy$ transition, the marking becomes 01100. If we fire the $bad\ assy$ transition, the marking becomes 01001.

The DFA is now:



From the marking 10001 there are two enabled transitions: new, and $rework$. If we fire the new transition, the marking becomes 01001. If we fire the $rework$ transition, the marking becomes 01100.

The DFA is now:



There are no transitions enabled for the marking 01001, so our DFA is complete. We can conclude several things:

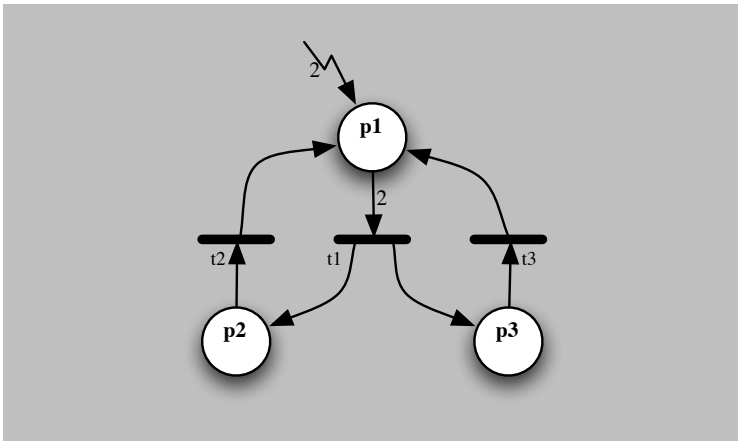
- We know it is impossible to reach the marking 01110, since that marking does not appear in the DFA.
- It is obvious that there is at most one dot in every place, since every state in the DFA only contains the digits 0 or 1.

- We can immediately see that the system will be deadlocked when it reaches the marking 01001, since there are no transitions leading *out* of this state. By inspecting the diagram, we see that there are two short sequences of firings that lead to deadlock: (new, $process$, new, $bad\ assy$), and (new, $process$, $bad\ assy$, new).

- It is obvious that the state-space of the original network is finite: in fact, there are exactly six unique markings (states).

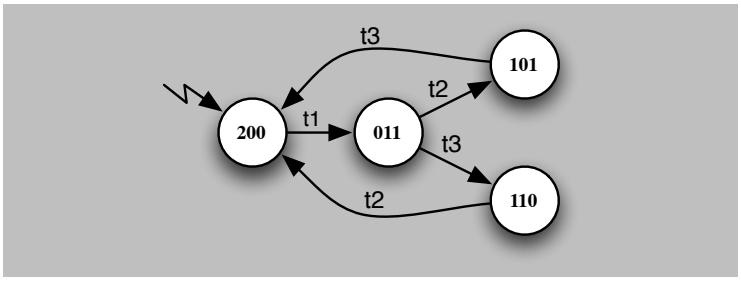
6.5.4 Example: Multiple dots in a place

Here is a network we examined earlier:



If we draw the DFA equivalent to this network, we find:

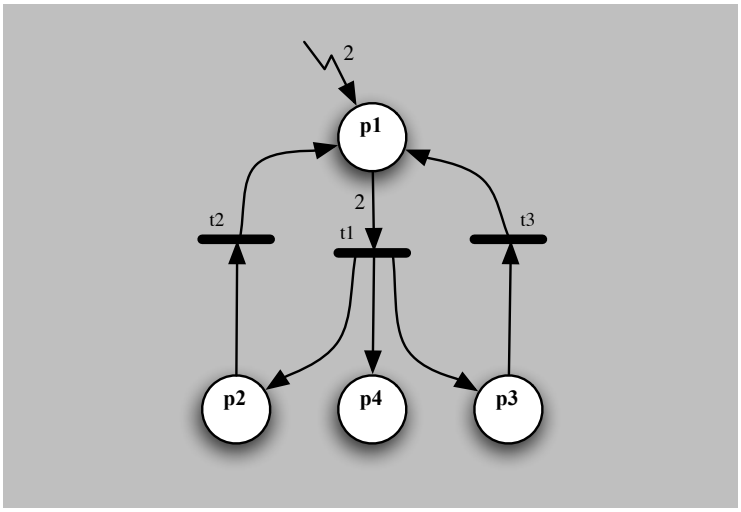
It is clear that the largest number of dots ever appear in this network is two.



6.5.5 Example: Unbounded number of dots

Here is a network that has an unbounded number of dots:

After a few moments tinkering with the network, it will be clear that the number of dots in place $p4$ always increases, because dots are never removed from it. There is thus no limit to the number of dots that will appear in place $p4$.



6.5.6 B-safety

A place-transition network is *B-safe* if there are never more than B dots in a place. To check this, we need to find all the markings that can be reached from the initial marking (which is given by $R^*(I)$), and then check that for every such marking m, the number of dots in any place is less-than or equal to B (i.e. $\forall p \in P, m(p >\leq B$). We can express this formally:

$$\forall m \in R^*(I), \forall p \in P, m(p) \leq B$$

or equivalently:

$$\forall m \in I^*, \forall p \in P, m(p) \leq B$$

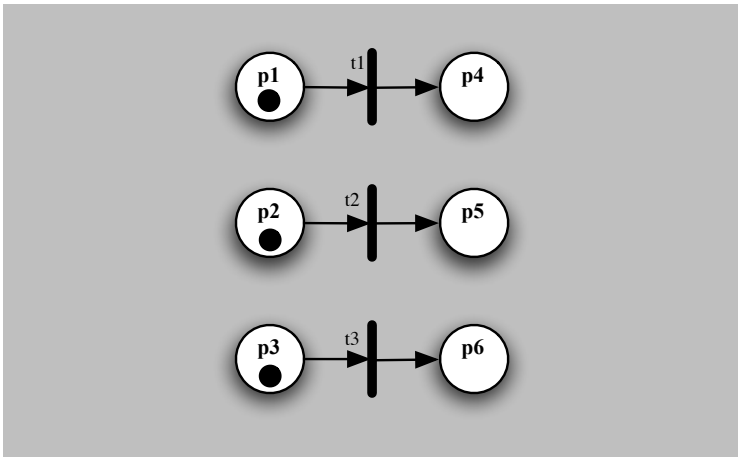
A network that is not B-safe is *unsafe*.

6.6 Concurrency — one last time

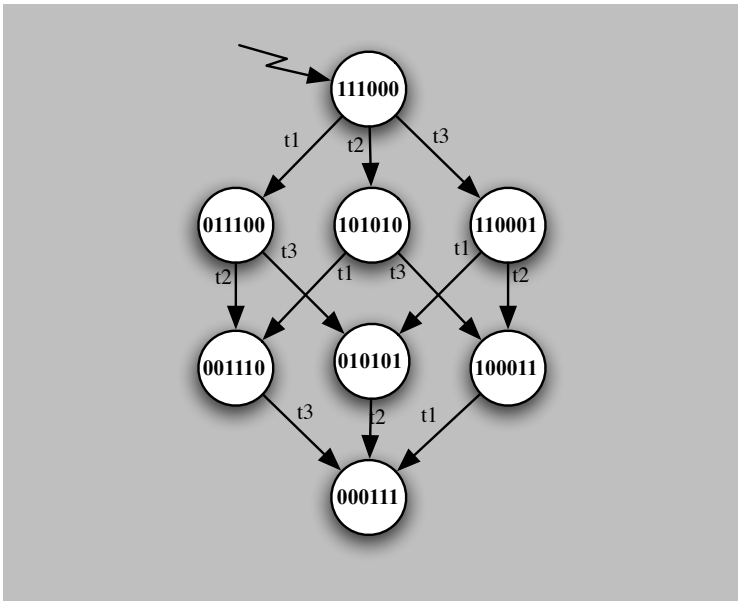
We have seen earlier that there can be situations when the marking of a place-transition network is such that there can be more than one enabled transition.

Consider this network:

There are clearly three enabled transitions, $t1$, $t2$, and $t3$. If we restrict ourselves to the simple policy of firing only one transition at a time, we could fire these three transitions in any order.

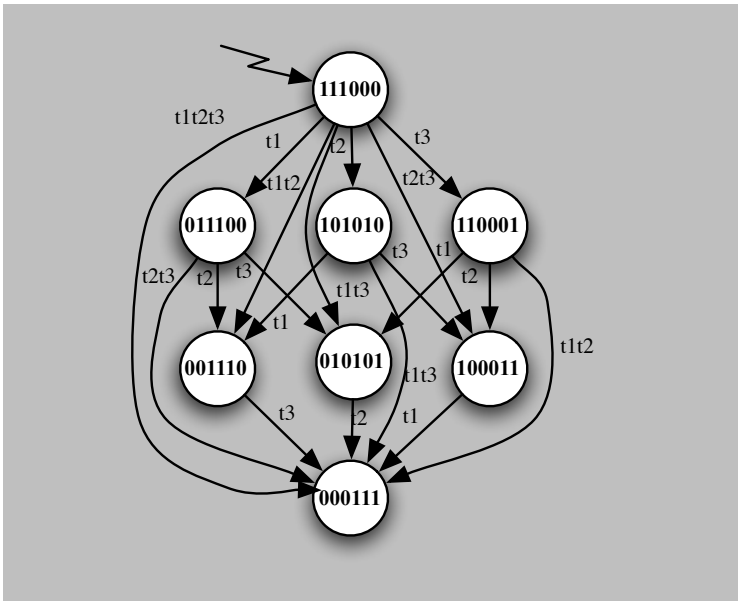


Here is the reachability DFA when we fire one transition at a time:



And here is the reachability DFA when we allow one, or two, or all three transitions to fire at once:

The diagram is quite a lot more complicated, due to the extra concurrent transitions. However, the important thing to notice is that the reachability DFA has exactly the same states as before — there are *no additional states*. This is the reason that we use the "single firing" policy — multiple firings don't change the states of the system that are reachable.



Chapter 7

Synchronisation

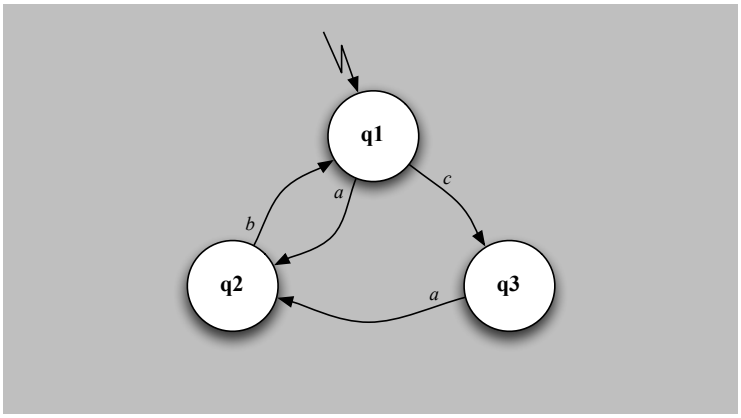
A Finite State Automaton (FSA) is a good way to visualise the behaviour of a complex system. It is a natural development of this idea to imagine constructing a complete system out of multiple FSAs. As we have seen, Petri-nets provide one way of visualising the behaviour of a complete system of FSAs. In this section we look at an alternative view.

7.1 A simple DFA

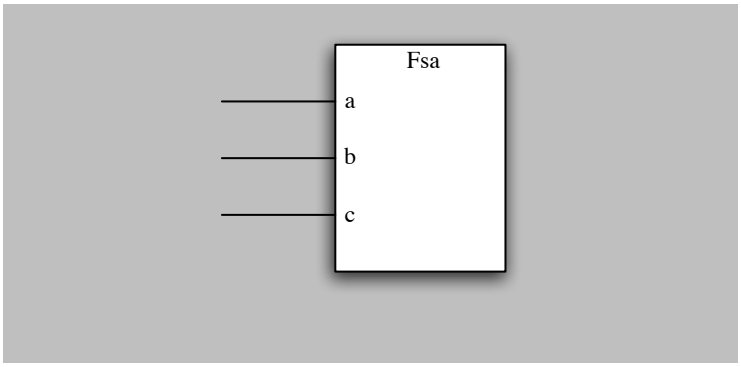
Consider the Deterministic Finite State Automaton (DFA) described by this transition table:

	a	b	c
q_1	q_2		q_3
q_2		q_1	
q_3	q_2		

and shown diagramatically on the right of this page.

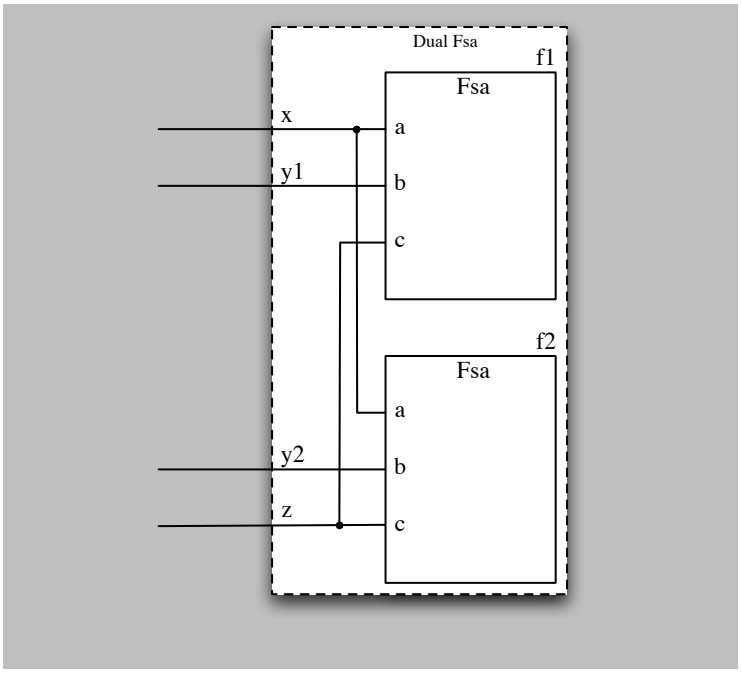


We can think of this machine abstractly as a black-box that responds to three events, a, b, and c, and can represent it as shown here:



7.2 Synchronisation of two FSAs

Suppose now that we construct a *DualFsa* consisting of two of these machines interconnected like this:



Notice that the effect of interconnecting the two FSA machines is to create a new DualFsa machine, that has inputs labelled x, $y1$, $y2$, and z.

What does this diagram mean? The line joining event x of the DualFsa to the a events of the two component FSAs indicates that when an x event is received, the DualFsa will cause each of the component FSAs to receive an a event.

Similarly, the line labelled z shows that a z event received by the DualFsa will cause c events to be received by each of the component FSAs.

A $y1$ event will cause a b event to be received by FSA f_1, and have no effect on FSA f_2). $y2$ behaves similarly for FSA f_2.

Consider what happens when an x event is received: each FSA will now receive an a event and, if all is well, each will make an internal transition.

If one of the FSAs is unable to make an a transition, it will "die". If any one of the internal FSAs dies, we regard the whole DualFsa as having died too. We do this because a half-dead component is just as useless as an all-dead one — its behaviour is no longer correct.

7.3 Synchronisation example one

Since each FSA has 3 states, the DualFsa we have constructed potentially has a state-space containing $3 \times 3 = 9$ states. (The cartesian product of the state-spaces of the two machines.)

Synchronising the events of multiple machines as we have done here forces the machines to make internal transitions at the same time. Since not all transitions are possible from every state, we generally find that the state-space of multiple synchronised state-machines is *smaller* than the cartesian product predicts.

Let us now construct the complete state-space for the DualFsa. We will represent each possible state as a tuple of states, showing the state of f_1 and of f_2. Since initialising each Fsa forces it into state q_1, the initial state of the DualFsa is $(q_1 q_1)$.

Suppose now that we apply an x event. This event is sent to f_1, which responds by entering state q_2. It is also sent to f_2, which behaves the same, so the overall effect is for the DualFsa to enter state $(q_2 q_2)$.

Suppose instead that we had applied a y_1 event. In this case, the event is sent only to f_1, which

is unable to handle the event in its present state (q_1), so it "dies". f_2, of course, does not change, since it did not receive an event. Since f_1 has "died", the DualFsa also "dies", so there is no new state.

Similar reasoning tells us that a y_2 event will cause f_2 to die, so there is no new state.

Suppose instead that, in the initial state, we received a z event. Since the event is sent to both f_1 and f_2, the next state will be (q_3q_3).

Thus the first row of the transition table for the DualFsa machine will show:

	x	y_1	y_2	z
(q_1q_1)	(q_2q_2)			(q_3q_3)

We now perform the same process, beginning in state (q_2q_2). If we apply an x event, f_1 and f_2 both die. If we apply a y_1 event, the DualFsa enters state (q_1q_2). If we apply a y_2 event, the DualFsa enters state (q_2q_1). If we apply a z event, the DualFsa dies.

So the Transition table now looks like this:

	x	y_1	y_2	z
(q_1q_1)	(q_2q_2)			(q_3q_3)
(q_2q_2)		(q_1q_2)	(q_2q_1)	

Repeating the process for state (q_3q_3), we find: If we apply an x event, the DualFsa enters state (q_2q_2). If we apply a y_1 event the DualFsa dies. If we apply a y_2 event the DualFsa dies. If we apply a z event the DualFsa dies.

So the Transition table now looks like this:

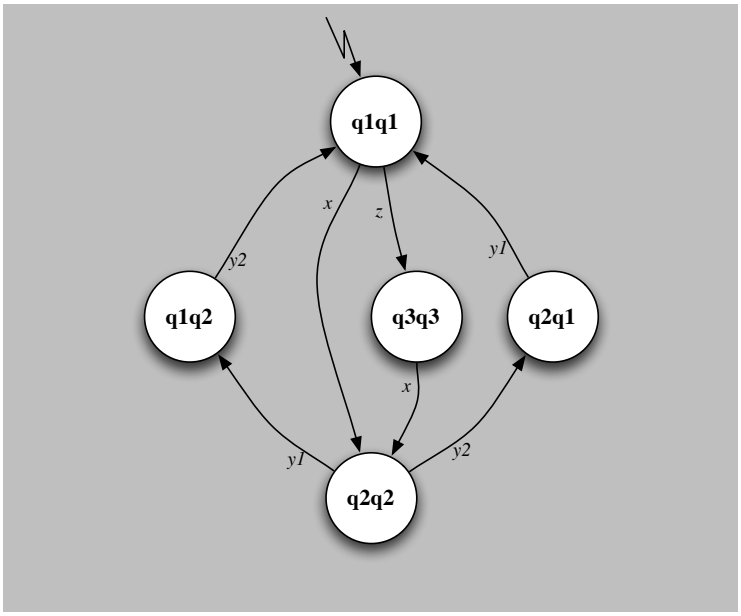
	x	y_1	y_2	z
(q_1q_1)	(q_2q_2)			(q_3q_3)
(q_2q_2)		(q_1q_2)	(q_2q_1)	
(q_3q_3)	(q_2q_2)			

Continuing in this fashion for states (q_1q_2) and (q_2q_1), we find the complete transition table is:

	x	y_1	y_2	z
(q_1q_1)	(q_2q_2)			(q_3q_3)
(q_2q_2)		(q_1q_2)	(q_2q_1)	
(q_3q_3)	(q_2q_2)			
(q_1q_2)			(q_1q_1)	
(q_2q_1)		(q_1q_1)		

The state diagram for the DualFsa is thus:

It is obvious that the DualFsa has only five states, rather than the nine states predicted by the cartesian product.



The state diagram for the DualFsa allows us to answer questions such as these:

- Can the DualFsa ever reach deadlock?

  No. Every state that can be reached fron the initial state has always got at least one transition leading from it.

- Is the state $(q_1 q_3)$ reachable?

  No. There is *no* sequence of inputs, no matter how long, that will ever get the DualFsa into the state $(q_1 q_3)$.

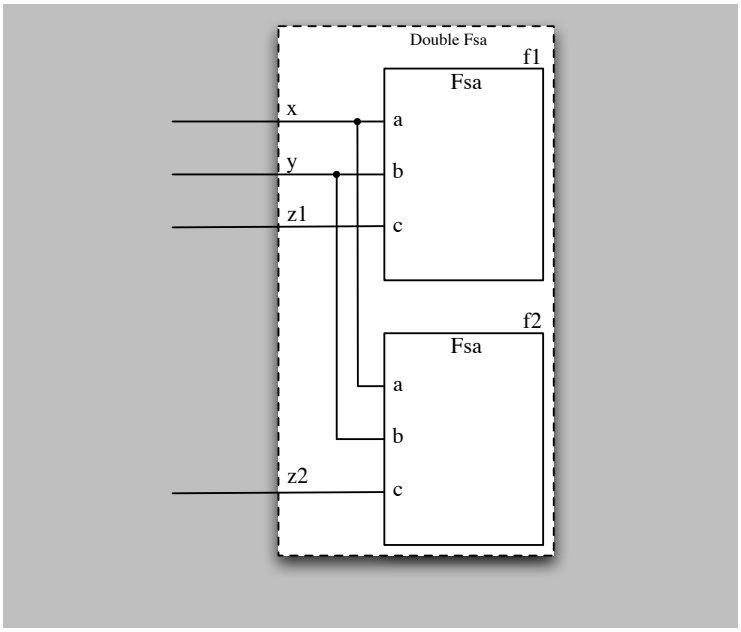
- What is the shortest sequence of events that will get the DualDfa into the state $(q_1 q_2)$?

  It is x, y_1.

These questions could not be easily answered just by looking at the intercconection diagram.

7.4 Synchronisation example two

Suppose now that we construct a *DoubleFsa* consisting of two of these machines interconnected in a different way, like this:

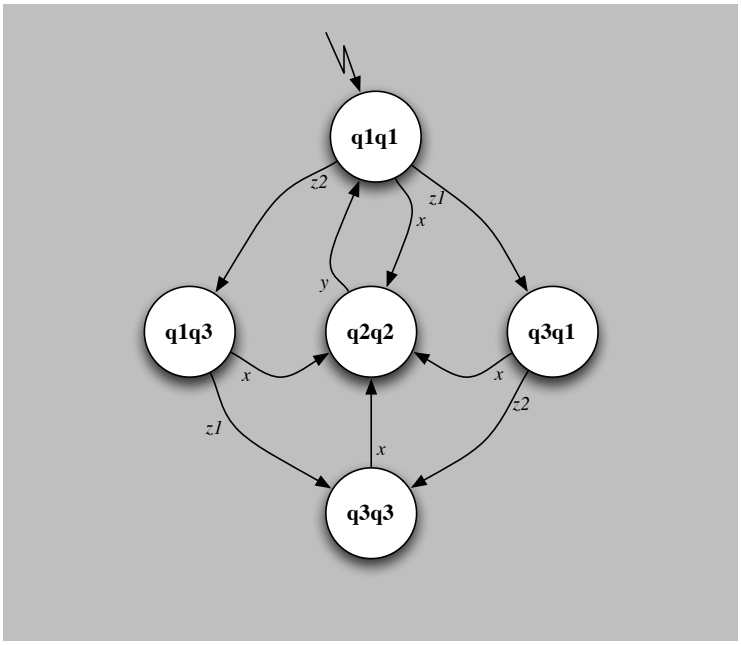


The DoubleFsa responds to four events, labelled x, y, z_1 and z_2.

Beginning from state (q_1q_1), we can evaluate all the reachable states for the DoubleFsa using the same process as describe above. Here is the resulting transition-table:

	x	y	z_1	z_2
(q_1q_1)	(q_2q_2)		(q_3q_1)	(q_1q_3)
(q_2q_2)		(q_1q_1)		
(q_3q_1)	(q_2q_2)			(q_3q_3)
(q_1q_3)	(q_2q_2)		(q_3q_3)	
(q_3q_3)	(q_2q_2)			

The resulting state-diagram looks like this:



With a different interconection, the resultant machine can also only reach five of the possible nine states in the cartesian product.

The state diagram for the DoubleFsa allows us to answer questions such as these:

- Can the DoubleFsa ever reach deadlock?

  No. Every state that can be reached fron the initial state has always got at least one transition

leading from it.

- Is the state $(q_1 q_3)$ reachable?

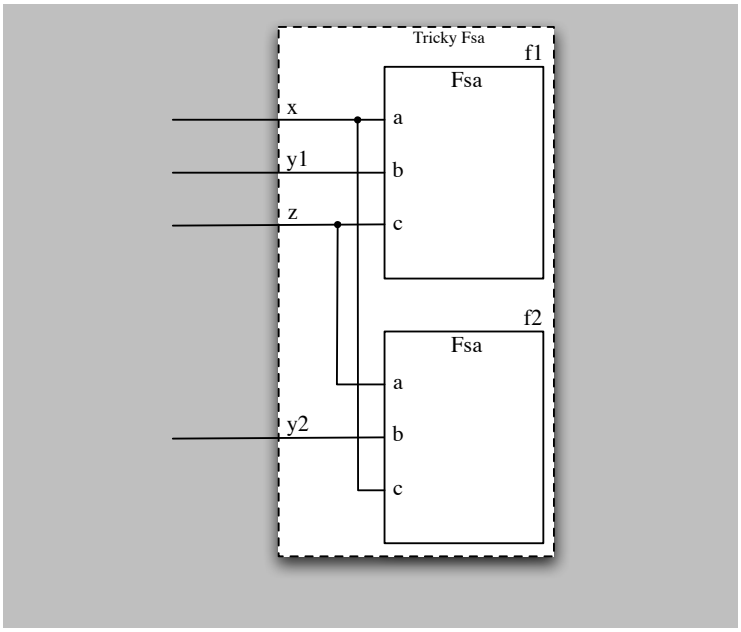
  Yes. We can get there from the initial state with an x event followed by a y_1 event.

- What is the shortest sequence of events that will get the DoubleFsa into the state $(q_1 q_2)$?

  There is no sequence, because the state $(q_1 q_2)$ is not reachable.

7.5 Synchronisation example three

Suppose now that we construct a *TrickyFsa* consisting of two of these machines interconnected in a complex way, like this:



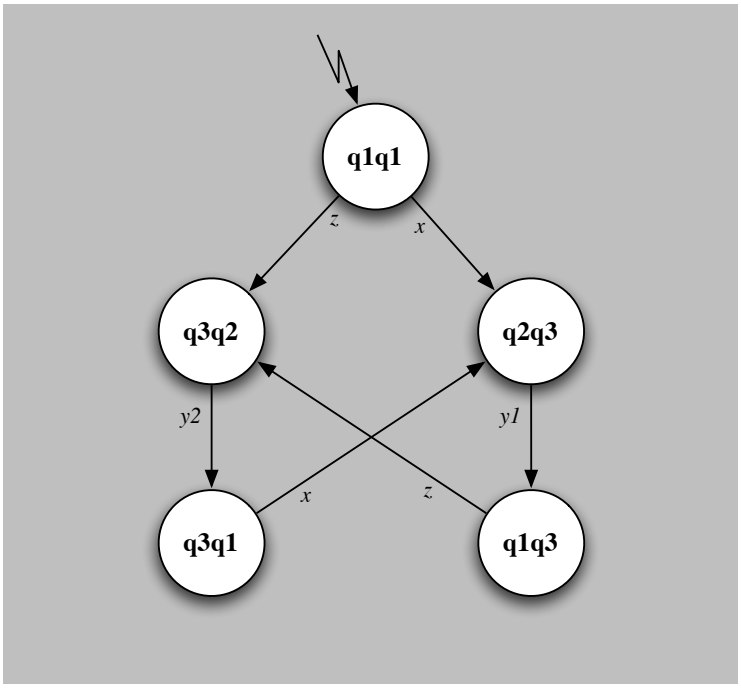
The TrickyFsa responds to four events, labelled x, y_1, y_2 and z.

Beginning from state $(q_1 q_1)$, we can evaluate all the reachable states for the TickyFsa using the same process as describe above. Working out exactly what happens now requires a good deal of care, since (say) an x event causes an a event in fsa f_1, but a c event in f_2. It is easy to make mistakes!
Here is the resulting transition-table:

	x	y_1	y_2	z
$(q_1 q_1)$	$(q_2 q_3)$			$(q_3 q_2)$
$(q_2 q_3)$		$(q_1 q_3)$		
$(q_3 q_2)$			$(q_3 q_1)$	
$(q_1 q_3)$				$(q_3 q_2)$
$(q_3 q_1)$	$(q_2 q_3)$			

The resulting state-diagram looks like this:

With a different interconection, the resultant machine can also only reach five of the possible nine states in the cartesian product.



Once again, the state diagram for the TrickyFsa allows us to answer questions:

- Can the TrickyFsa ever reach deadlock?

  No. Every state that can be reached fron the initial state has always got at least one transition leading from it.

- Is the state $(q_1 q_3)$ reachable?

  Yes. We can directly get there from the initial state with an x event followed by a y_1 event.

- What is the shortest sequence of events that will get the TrickyFsa into the state $(q_1 q_2)$?

  The state $(q_1 q_2)$ is not reachable.

Appendix A

State-machine languages

by Kevin Maciunas

In this chapter,we very briefly explore solutions to programming with state machines other than *hand coding* in a standard programming language.

Accomplishing this task involves using a new language. Our aim is to be able to express our state machine in a convenient and readable form.

A.1 Why a new language?

We've already seen how we can view (say) a graphical representation of a state machine and use that as a coding aid to produce a state machine implementation in our chosen programming language (Java and DLX Assembler examples were used).

As programmers, we do this all the time - jot down pseudo code, make use of drawings of data structures etc. Other programmers quickly grasp what our code is doing (or *trying* to do) because we recognise the *design patterns* in the code and can logically re-invent the appropriate pseudo code or diagrams in our heads.

If we focus on the Java versions of the state machines given earlier in this document, you'll see that this visualisation does not work for the state machine design paradigm! The natural reaction of any Java programmer to the code produced is one of revulsion or at best disbelief!

The problem with the encoding of the state machine into Java (or other "conventional" language) is that it is difficult to mentally reverse engineer the succinct state machine diagram from the code. Furthermore, if we are implementing a state machine which *changes* then we have little recourse but to throw away the old state machine and start from scratch with our new one.

Solving this problem has motivated many programming language designs, but the ones we'll look at here are so-called *pre-processors* — they aren't a complete programming language, but rather are an add-on to a language.

Languages like these have a long and distinguished career in Computer Science.

A.2 JLex

The language JLex is modelled after the language *lex* and takes a regular expression specified state machine and produces a Java recogniser for it. JLex is intended to build so-called *lexical analysers* — pieces of code that recognise sequences of characters. The "scanner" class from the standard Java library is an example of this.

JLex input files are typical of the kind of input file you'll find in these class of programming languages — the syntax comes from the 1970's from the language lex.

The JLex input file is divided into three parts, each separated by a line containing two "%%" characters on a line by themselves. The first part of the JLex file is simply copied to output. In this part of the JLex file, you would normally place miscellaneous bits of Java code that your generated code will use. The second part of the JLex file contains definitions. These take the form of *macros*. Normally, you'll find little more than handy definitions of regular expressions that match particular things, for example:

```
ALPHA=[a-zA-Z]
```

is a definition of a macro that matches a single alphabetic character. This permits us to write clearer regular expressions in the third part of the JLex input file. The third part of the JLex input file does the work. This is where we specify the Finite State Automaton that we are trying to build. In JLex, this is done by writing a series of regular expressions with associated (Java) actions that get performed when the regular expression is matched. This is particularly easy to write. For example: the following matches a Java identifier:

```
{ALPHA}({ALPHA}|{DIGIT}|_)* {system.out.println("Identifier="+yytext());}
```

(Assuming the definition of ALPHA given above and an equally obvious definition of the macro "DIGIT". The actual text matched by the regular expression is given by the method "yytext" (all JLex internal things are prefixed with "yy").

A.2.1 Using JLex

Once we have our particular problem specified as a series of pattern-action pairs, we just run JLex on the program. This *generates* a Java program (class) which implements the FSA.

Once we have our resulting Java class, we embed this in the Java application we are building. We don't look at the code, we just *use* it.

The virtue in using a tool like JLex becomes obvious once we need to change the FSA. It is a relatively simple task to modify the regular expressions and produce a new FSA. Contrast this with the hand crafted FSA approach.

JLex is typical of a large number of similar tools which are oriented to processing input — these tools were built by compiler-writers (usually) and while they *do* build FSAs, they are actually intended to build *lexical analysers*. The FSAs discussed earlier are in fact closer to *one* of the regular expressions JLex uses.

A.3 Ragel

Ragel is an example of a new class of languages intended to process state machines. Unlike JLex, it doesn't carry much "historical baggage" (much of the strangeness of JLex arises because it is intended to be somewhat backwards compatible with lex).

Ragel is also interesting because it is not a Java tool — Ragel (at the time of writing) can generate C, C++, Objective-C, D, Java and Ruby. Ragel is like JLex in that Ragel state machines can not only recognize byte sequences as regular expression machines do, but can also execute code at arbitrary points in the recognition of a regular language.

A.3.1 Using Ragel

If you understand how JLex is used, Ragel (while different) is not particularly difficult. To give a flavour of Ragel, here is a (C language) example, that converts a string to an integer.

```
/*
 * Convert a string to an integer.
 */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

%%{
    machine atoi;
    write data;
}%%

long long atoi( char *str )
{
    char* p= str;
    char* pe= str+strlen(str);
    int cs;
    long long val= 0;
    bool neg= false;

    %%{
        action see_neg{
            neg= true;
        }

        action add_digit{
            val= val*10 + (fc-'0');
        }

        main:= ('-'@see_neg | '+')? (digit @add_digit)+ '\n';

        # Initialize and execute.
        write init;
        write exec;
    }%%

    if( neg ){
        val= -val;
    }
    if( cs<atoi_first_final ){
        fprintf( stderr, "atoi: there was an error\n" );
    }
    return val;
}


#define BUFSIZE 1024

int main()
{
    char buf[BUFSIZE];
    while( fgets(buf,sizeof(buf),stdin) != 0 ){
        long long value= atoi(buf);
        printf("%lld\n",value);
    }
```

```
    return 0;
}
```

As you can intuit from the example, this Ragel example makes use of regular expressions (in the familiar Unix REGEXP form).

Unlike JLex, Ragel also permits you to specify the state machine via state charts (and other formats, see the manual) — which permits more natural expression of state machines when you are using the state machine for a task *other* than parsing input.

A.4 SMC — State-machine compiler

SMC (State Machine Compiler) is a modern state machine processing language and behaves similarly to JLex and Ragel. Like Ragel, SMC is a language-agnostic system. At the time of writing, SMC can output no fewer than 14 different languages.

Unlike Ragel, SMC input files are expressed in a more conventional states-and-transitions format. SMC can be used for producing FSMs for a very diverse range of application. By contrast, Ragel and JLex are designed to make smal language-parsers trivial to write.

A.4.1 Using SMC

Again, it is not the intention to provide full details on the use of SMC (of the three languages, SMC is by far the most complicated). Reproduced here is an example from the on-line manual, for a telephone exchange:

```
//
// The contents of this file are subject to the Mozilla Public
// License Version 1.1 (the "License"); you may not use this file
// except in compliance with the License. You may obtain a copy of
// the License at http://www.mozilla.org/MPL/
//
// Software distributed under the License is distributed on an "AS
// IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or
// implied. See the License for the specific language governing
// rights and limitations under the License.
//
// The Original Code is State Machine Compiler (SMC).
//
// The Initial Developer of the Original Code is Charles W. Rapp.
// Portions created by Charles W. Rapp are
// Copyright (C) 2000. Charles W. Rapp.
// All Rights Reserved.
//
// Contributor(s):
//
// Name
//  Telephone.sm
//
// Description
//  Runs a plain old telphone. That means the proper sounds at
//  the proper time.
//
// RCS ID
// $Id: TelephoneSrc.htm,v 1.4 2008/05/23 17:30:02 fperrad Exp $
//
```

```
// CHANGE LOG
// $Log: TelephoneSrc.htm,v $
// Revision 1.4  2008/05/23 17:30:02  fperrad
// - remove duplicated XML declaration: <?xml version="1.0" ...
//
// Revision 1.3  2008/05/20 18:32:42  cwrapp
// Committing release 5.1.0
//
// Revision 1.2  2006/06/04 19:50:44  cwrapp
// Updated the copyright.
//
// Revision 1.1.1.1  2005/06/16 00:31:38  cwrapp
// Initial import into CVS
//

%class Telephone
%start CallMap::Initialized

%%
Initialized
{
    Start OnHook{
        //Nothing
    }

    // Ignore all other transitions.
    Default nil{
        //Ignore
    }
}

//STATE//
OnHook
Entry{
    updateClock();
    startClockTimer();
}
Exit{
    stopTimer("ClockTimer");
}
{
    //We are handling the caller's side of the connection.
    OffHook Dialing/push(PhoneNumber::DialTone){
        clearDisplay();
        setReceiver("on hook", "Put down receiver");}
    }

    //Dialing errors.
    LeftOffHook LeftOffHook{
        //Nothing
    }

    //Time to update the clock's display.
    ClockTimer nil {
        updateClock();
        startClockTimer();
```

```
    }
}

//STATE//
Dialing //The number is being dialed.
{
    //Dialing successfully completed.
    DialingDone(callType: int, areaCode: String, exhange: String, local: String) Routing{
        routeCall(callType, areaCode, exchange, local);
    }

    InvalidDigit InvalidDigit {
        //Nothing special
    }
}

//STATE//
Routing //The call is now being routed.

{
    Emergency PlayingMessage{
        playEmergency();
    }

    NYCTemp NYCTemp{
        //Nothinbg special
    }

    Time Time{
        //Nothing special
    }

    DepositMoney DepositMoney{
        //Nothing special
    }

    LineBusy BusySignal{
        //Nothing special
    }

    InvalidNumber PlayingMessage {
        playInvalidNumber();
    }
}

//STATE//
NYCTemp
Entry{
    loop("ringing");
    startTimer("RingTimer", 10000);
}
Exit{
    stopLoop("ringing");
}
{
    RingTimer PlayingMessage{
```

```
        playNYCTemp();
    }
}

//STATE//
Time
Entry{
    loop("ringing");
    startTimer("RingTimer", 10000);
}
Exit{
    stopLoop("ringing");
}
{
    RingTimer PlayingMessage{
        playTime();
    }
}

//STATE//
DepositMoney
Entry{
    loop("ringing");
    startTimer("RingTimer", 5000);
}
Exit{
    stopLoop("ringing");
}
{
    RingTimer PlayingMessage{
        playDepositMoney();
    }
}

//STATE//
BusySignal
Entry{
    loop("busy");
}
Exit{
    stopLoop("busy");
}
{
    //Wait for on hook only.
}

//STATE//
PlayingMessage
{
    //If caller hangs up while a message is being played,
    //be sure to stop the playback.
    OnHook OnHook{
        stopPlayback();
        setReceiver("off hook", "Pick up receiver");
        clearDisplay();
    }
```

```
    Stop Initialized{
        stopPlayback();
        setReceiver("off hook", "Pick up receiver");
        clearDisplay();
    }

    PlaybackDone MessagePlayed{
        //Nothing special
    }
}

//STATE//
MessagePlayed
Entry{
    startTimer("OffHookTimer", 10000);
}
Exit{
    stopTimer("OffHookTimer");
}
{
    OffHookTimer LeftOffHook {
        //Nothing special
    }
}


//-----------------------------------------------------------------
//Error States.
//
//Let someone know the phone has been left off the hook.

//STATE//
LeftOffHook
Entry{
    startTimer("LoopTimer", 10000);
    loop("phone_off_hook");
}
Exit{
    stopTimer("LoopTimer");
    stopLoop("phone_off_hook");
}
{
    LoopTimer WaitForOnHook{
        //Nothing special
    }
    Default nil{
        //Nothing special
    }
}

//STATE//
InvalidDigit
Entry{
    startTimer("LoopTimer", 10000);
    loop("fast_busy");
```

```
}
Exit{
    stopTimer("LoopTimer");
    stopLoop("fast_busy");
}
{
    LoopTimer WaitForOnHook{
        //Nothing special
    }
    Default nil{
        //Nothing special
    }
}


//STATE//
WaitForOnHook //Stay in this state until the telephone is on-hook.
{
    Default nil{
        //Nothing special
    }
}

//STATE//
Default //Actions appear in every state
{
    //Ignore any dialings after a phone number has been
    //collected.
    Digit(n: String)  nil {
        //Ignore
    }

    //No matter when it happens, when the phone is hung
    //up, this call is OVER!
    OnHook OnHook{
        setReceiver("off hook", "Pick up receiver");
        clearDisplay();
    }

    Stop Initialized{
        setReceiver("off hook", "Pick up receiver");
        clearDisplay();
    }

    //Ignore the clock timer outside of the OnHook state.
    ClockTimer nil{
        //Ignore
     }
}

%%

//This map processes dialed digits. It either returns success
//when
<a name="PhoneNumber">%map PhoneNumber</a>
%%
```

```
DialTone
Entry{
    loop("dialtone");
    startTimer("OffHookTimer", 10000);
}
Exit{
    stopTimer("OffHookTimer");
    stopLoop("dialtone");
}
{
    //If the first digit is 1, then this is a long distance
    //phone call. Don't save this first digit.
    Digit(n: String) [equal(n, 1);] LongDistance{
        playTT(n);
        setType(Telephone.LONG_DISTANCE);
        saveAreaCode(n);
        addDisplay("-");
    }

    //Check for 911.
    Digit(n: String) [equal(n, 9);] OneOneStart{
        playTT(n);
        saveExchange(n);
    }

    Digit(n: String) Exchange{
        playTT(n);
        setType(Telephone.LOCAL);
        saveExchange(n);
    }
}

//Collect the area and then move on to the local number.
LongDistance
Entry{
    startTimer("OffHookTimer", 10000);
}
Exit{
    stopTimer("OffHookTimer");
}
{
    Digit(n: String) {!isCodeComplete();] nil{
        playTT(n);
        saveAreaCode(n);
        resetTimer("OffHookTimer");
    }

    Digit(n: String) Exchange{
        playTT(n);
        saveAreaCode(n);
        addDisplay("-");
    }
}

//Check if this is a 911 call.
```

```
OneOneStart
Entry{
    startTimer("OffHookTimer", 10000);
}
Exit{
    stopTimer("OffHookTimer");
}
{
    Digit(n: String) [equal(n, 1);] NineOne{
        playTT(n);
        saveExchange(n);
    }

    Digit(n: String) Exchange{
        playTT(n);
        setType(Telephone.LOCAL);
        saveExchange(n);
    }
}

//Almost there.
NineOne
Entry{
    startTimer("OffHookTimer", 10000);
}
Exit{
    stopTimer("OffHookTimer");
}
{
    Digit(n: String) [equal(n, 1);] pop(DialingDone){
        playTT(n);
        setType(Telephone.EMERGENCY);
        saveExchange(n);
    }

    Digit(n: String) LocalCall{
        playTT(n);
        setType(Telephone.LOCAL);
        saveExchange(n);
        addDisplay("-");
    }
}

//Collect the three digit exchange.
Exchange
Entry{
    startTimer("OffHookTimer", 10000);
}
Exit{
    stopTimer("OffHookTimer");
}
{
    Digit(n: String) [!isExchangeComplete();] nil{
        playTT(n);
        saveExchange(n);
        resetTimer("OffHookTimer");
```

```
    }

    Digit(n: String) LocalCall{
        playTT(n);
        saveExchange(n);
        addDisplay("-");
    }
}

//Process a local call.
LocalCall
Entry{
    startTimer("OffHookTimer", 10000);
}
Exit{
    stopTimer("OffHookTimer");
}
{
    Digit(n: String) [!isLocalComplete();] nil{
        playTT(n);
        saveLocal(n);
        resetTimer("OffHookTimer");
    }

    Digit(n: String) pop(DialingDone){
        playTT(n);
        saveLocal(n);
    }
}

Default
{
    //If an invalid digit is dialed, give up collecting
    //digits immediately.
    Digit(n: String) [!isDigitValid(n);]  pop(InvalidDigit){
        clearDisplay();
    }

    //Caller has stopped dialing and left the phone
    //off hook.
    OffHookTimer pop(LeftOffHook){
        clearDisplay();
    }

    //Pass this event up.
    OnHook pop(OnHook){
        clearDisplay();
    }
    Stop pop(Stop) {
        clearDisplay();
    }

    //Ignore the clock timer outside of the OnHook state.
    ClockTimer nil{
    }
}
```

%%

The state machine for this telephone exchange is reasonably complex — 7 states and 16 transitions — and as you might appreciate by simply looking at the code above, it is quite succinctly expressed in SMC. Coding this "by hand" would be a daunting prospect!

A.5 Summary

We've looked (very briefly and superficially) at three languages which permit us to directly express state machines in the language. The clarity which arises from this is not to be ignored. Our hand coded examples are, by comparison, quite difficult to understand.

The three languages each have their own strengths and weaknesses and for any particular application you would need to evaluate the suitability of the language.

Unlike hand coding our state machines, some of these languages result in code that is, perhaps, less efficient. For larger state machines, however, the advantage in using one of these is the expressive clarity and correctness which ensues.