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.

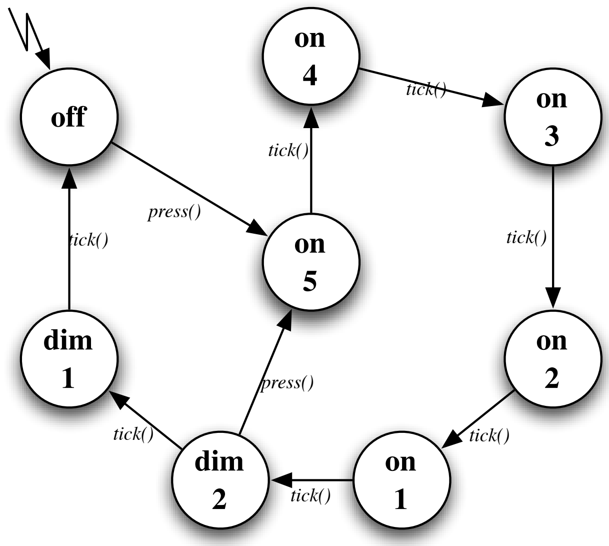
To solve this problem, you have decided to install a computer-controlled timer, that will work like this (assume the light is initially off):

- 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:

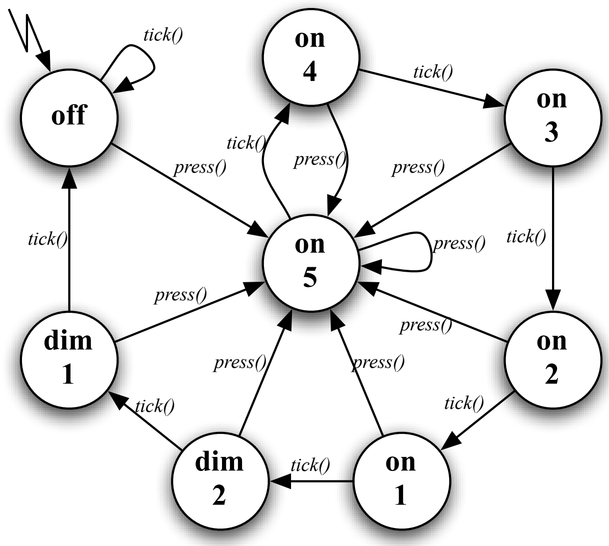


Note that the starting state is off, as indicated by the lightning-strike. If a $press()$ event occurs, the machine makes a transition to the $on5$ state. When a $tick()$ event occurs, the machine changes to the $on4$ state. A $press()$ event will return it to the $on5$ state, but a $tick()$ event will 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:



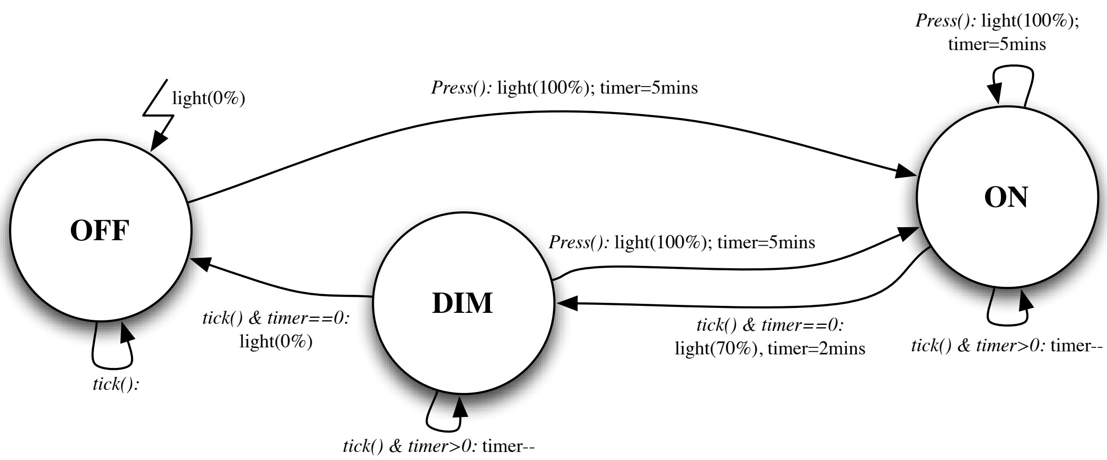
Despite the fact that the problem appeared trivial, we have ended up with a *large* diagram. It could have been far worse! Consider what the diagram would have looked like if ticks had occurred once every *second*, instead of once per minute: there would have been more than 400 states!

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.

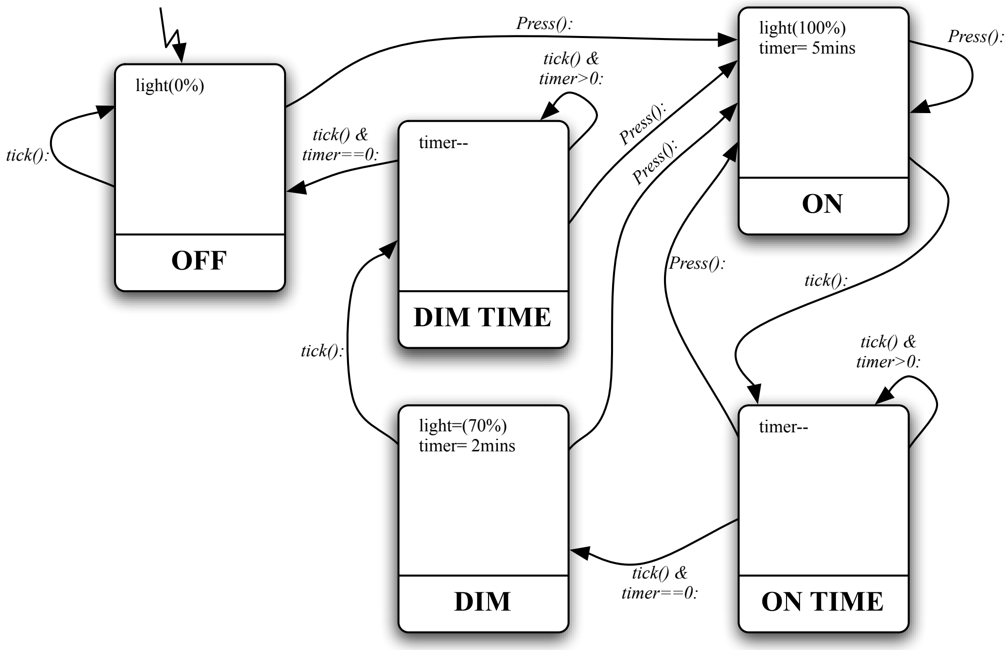
7. For each state, there is a case to handle the processing for that state. Each state can perform whatever *exit-actions* are appropriate. The *last* thing to do, after handling a case, is to assign a new value to the `state` variable, thus causing a transition to a new state. To make it clear that 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.

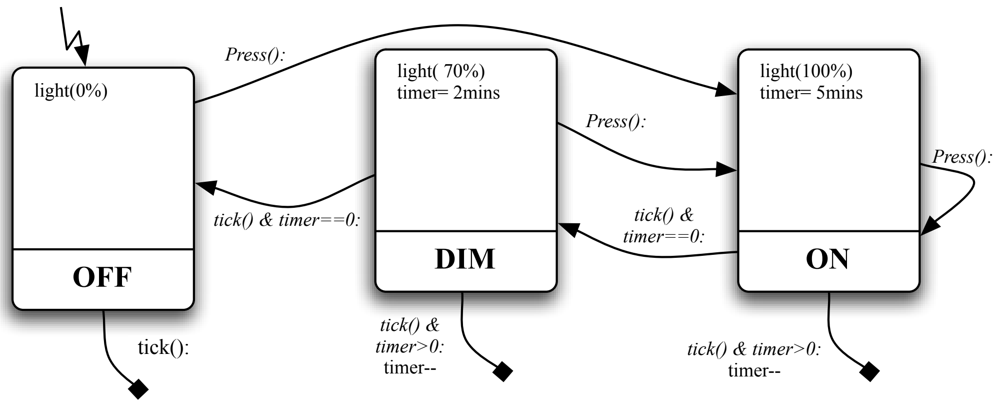
It is clear that we could build a FSM that behaved like both of these models at the same time: It would have *exit-actions*, like the Mealy machine, and *entry-actions*, like a Moore machine.

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.

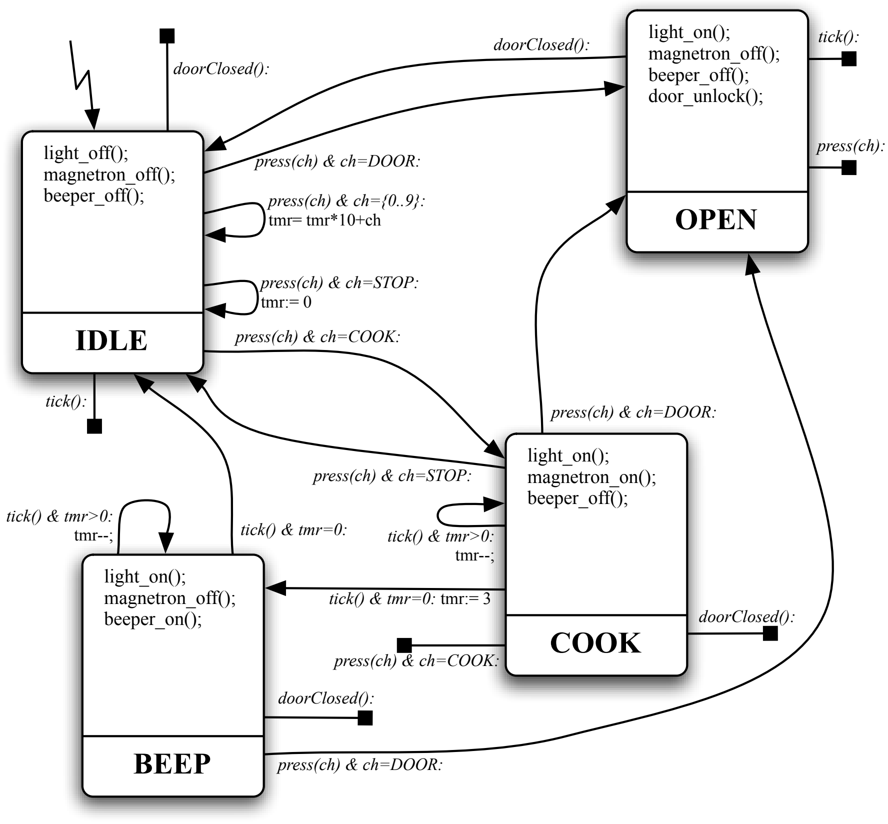


Figure 5.11: FSM for the microwave oven

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!