Computer Science
University of Manchester

Implementing State Machines in Smalltalk

Trevor P. Hopkins

Technical Report UMCS-93-3-1

Implementing State Machines in Smalltalk[1]

Trevor P. Hopkins

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
tph@cs.man.ac.uk

Technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from `ftp.cs.man.ac.uk` in the directory `/pub/TR`. The files are stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

Abstract

Implementing software solutions using *finite state machines* (FSMs) is a useful technique in many application areas, including compiler implementation and network protocols. This report examines possible implementation techniques for FSMs in the Smalltalk object-oriented system, and identifies techniques which best support the goals of generality, flexibility and software reuse.

Keywords: Finite state machines, object-oriented design, Smalltalk.

Contents

Chapter 1

Introduction

A *Finite State Machine* (FSM) can be regarded as an 'object' which can be in one of a fixed number of internal states. External 'events' (regarded here as 'messages') can trigger internally-defined actions, which can include changing the internal state. Thus, the behaviour of the FSM depends on the message received, and on the current internal state.

The FSM approach is often utilized in both conventional and object-oriented systems; in object-oriented design, it is often recommended that the behaviours of classes are described as if they were FSMs. For example, Booch's design approach [1] recommends the use of FSMs to represent the abstract behaviours of classes. In this report, various techniques for the implementation of FSMs are considered, concentrating specifically on Smalltalk representations. The aims are: to identify techniques which adhere most closely to the tenets of object-oriented design, such as the avoidance of duplicated code and generality of design, and to move towards a reusable design framework for the use of FSMs in Smalltalk.

Possible uses of FSMs in Smalltalk (and other object-oriented systems) include: the implementation of compilers and other language processing tools; communications systems, such as the modelling of network protocols[1] and the interpretation of 'escape' character sequences in terminal emulators, and the design of interactive user interfaces.

It is assumed that the reader is familiar with object-oriented development in general terms, and with Smalltalk in particular. Also, there are two major variants of Smalltalk currently available: Objectworks\Smalltalk [3] from ParcPlace Systems, based on Smalltalk-80 [4], and the Digitalk products, such as Smalltalk/V for Windows [5] and Smalltalk/V for Presentation Manager [6]. There are also a number of Smalltalk-like systems; these include Opal [7], a language used with the Gemstone [8] object-oriented database. Unless otherwise indicated, the techniques discussed here are thought to be appropriate for any of these variants, although the techniques have only been tried using Objectworks\Smalltalk.

In this report, only event messages *without* arguments are considered, although all the techniques could be simply extended to allow messages with any number of arguments to be used.

[1]Many communication standards documents, such as those for ISO OSI protocols, define the behaviour of protocol entities in terms of state transition tables. Object-oriented design approaches to network protocols [2] provided much of the motivation for the work described here.

Chapter 2

Implementation Techniques

The flexibility of Smalltalk allows for a surprising variety of approaches to the implementation of FSMs. Some of these techniques are considered here, starting with the 'case' statement approach most likely to be familiar to programmers in conventional block-structured languages. In later sections, approaches where individual states are mapped to methods (section 2.2), to blocks (sections 2.3 and 2.4) and to different classes (sections 2.5 and 2.6) are considered.

2.1 Mapping States using Case Statements

Perhaps the most immediately obvious way of implementing a FSM would be to devise a class with a single instance variable representing the current state (here imaginatively called currentState). This could be a number (typically, a non-negative integer), or perhaps a Symbol. Symbols will be used throughout this discussion, since this allows the possibility of sensible names for states. When a particular message arrives, it is necessary to decide which action to perform; the obvious approach is to use some kind of 'case' statement.

In Smalltalk, it would be natural to use an idiom of the form:

```
someMessage
        currentState = #state0 ifTrue: [
            "process someMessage in state 0"
            currentState := #state1.    "update current state"
            ↑self].
        currentState = #state1 ifTrue: [
            " ...... and so on."
```

Stylistically, it might be better to define separate 'auxiliary' methods for each combination of message and method, particularly if reasonably complex actions are to be performed in each case.

```
someMessage
        currentState = #state0 ifTrue: [↑self someMessageInState0].
        currentState = #state1 ifTrue: [↑self someMessageInState1].
                ......
```

Also, it may be that the same actions are to be performed when a particular message is received, for several states. In this case, an idiom of the following form may be more appropriate:

```
someMessage
        ( #(state2 state3 state7) includes: currentState) ifTrue: [
            "process someMessage in states 2,3 and 7"
             currentState := state4.
             ↑self].
```

Often, some messages should not be accepted when in a particular state. Another error possibility is that the FSM gets into a state which is unexpected. These cases can be easily handled by adding a 'default' expression at the end of every method:

```
someMessage
        " ........ remainder of method here."
        "fall-though case; this should not happen"
        ↑self error: 'Message someMessage received in state', currentState printString.
```

Clearly, a single method for handling all error cases could be used. An alternative approach would be to retain a block for the exception code using another instance variable. The use of blocks for handling exception cases is considered further in section 2.4.1.

2.1.1 An Example

Suppose that a FSM with three states is required, with two 'events' (messages) which can change the states. The expected transitions are illustrated in figure 2.1.

For this example, a class SimpleStateMachine is defined, with a single instance variable currentState. On instance creation, currentState is set to the Symbol state0.

```
    Object subclass: #SimpleStateMachine
        instanceVariableNames: 'currentState '
        classVariableNames: ''
        poolDictionaries: ''
```

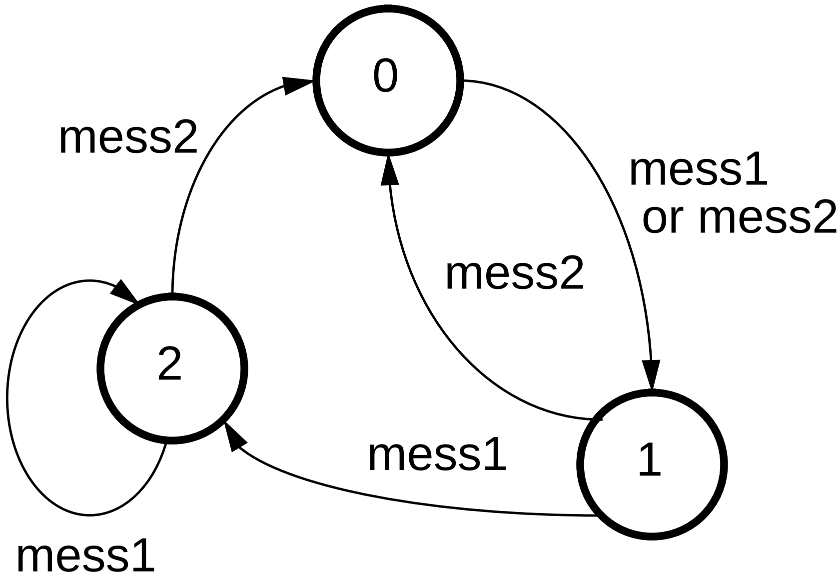

Figure 2.1: A simple three-state FSM

Two messages, mess1 and mess2, which change the state are provided:

```
mess1
        currentState = #state0 ifTrue: [
            Transcript cr; show: 'mess1 received: go to state 1'.
            ↑currentState := #state1].
        (#(state1 state2) includes: currentState) ifTrue: [
            Transcript cr; show: 'mess1 received: got to state 2'.
            ↑currentState := #state2].

mess2
        currentState = #state0 ifTrue: [
            Transcript cr; show: 'mess2 received: go to state 1'.
            ↑currentState := #state1].
        (#(state1 state2) includes: currentState) ifTrue: [
            Transcript cr; show: 'mess2 received: got to state 0'.
            ↑currentState := #state0].
```

This completes the implementation of the FSM outlined previously, as can be seen from the following example:

```
| fsm |
fsm := SimpleStateMachine new.   "fsm in state 0"
fsm mess1.                       "fsm in state 1"
fsm mess1.                       "fsm in state 2"
fsm mess1.                       "fsm in state 2"
fsm mess2.                       "fsm in state 0"
fsm mess2.                       "fsm in state 1"
fsm mess2.                       "fsm in state 0"
```

2.1.2 Advantages and Disadvantages

Clearly, there is a need for a different class for every application of a FSM; this limits the available code re-use. There is a possibility of much duplicated code within a single class, although the introduction of ‘auxiliary’ methods for each case can reduce this. Other possibilities for reducing duplication include the provision of a single method for error handling. Also, some code could be placed in an abstract superclass, with different applications being represented by different concrete subclasses. If consistently named ‘auxiliary’ methods are used, then the class browser and other tools can provide some assistance with structuring and reviewing the code.

Another criticism is that the ‘case’ statement style is not really part of the object-oriented approach [9]. Instead, the object-oriented style recommends the use of multiple objects with different but compatible behaviours, perhaps instances of different classes, possibly – but not necessarily – related by inheritance. Furthermore, Smalltalk does not have a ‘case’ statement in the language or standard library, so it is necessary to use multiple `ifTrue:ifFalse:` statements, which are bulky and prone to error.

It is interesting to note that it is essentially this approach which Booch uses in his Smalltalk example ([1], chapter 8). However, since his example has internal concurrency, the events are distributed using the *dependency* mechanism.

2.2 Mapping States to Methods

Another approach which is perhaps more in line with object-oriented principles is to directly implement a table-lookup mechanism, which is of course a widely-used general technique for FSMs. The instance variable currentState retains the current state as before; again, it is assumed to be a Symbol. Since it would be most likely that all instances of the class should have the same behaviour, it is reasonable to retain the transition table by a *class* variable. This class variable (here called TransitionTable) retains a Dictionary[1] whose keys are the names (i.e. Symbols) of states which can be held by the currentState instance variable, and whose values are further instances of class Dictionary. These map the names (again, Symbols) of messages which could be received in each state into the name of the method to be executed specifically for that combination of message and state.

The methods corresponding to the incoming messages simply need to extract from the TransitionTable structure the appropriate method name, and use perform: to execute that method.

2.2.1 An Example

For this example, consider a class TableStateMachine which defines a single instance variable currentState, and a single class variable TransitionTable.

```
Object subclass: #TableStateMachine
    instanceVariableNames: 'currentState '
    classVariableNames: 'TransitionTable '
    poolDictionaries: ''
```

The TransitionTable class variable is initialized as follows:

```
initialize
        TransitionTable := Dictionary new.
        TransitionTable at: #state0 put:
            ((Dictionary new)
                at: #mess1 put: #mess1inState0;
                at: #mess2 put: #mess2inState0;
                yourself).
        TransitionTable at: #state1 put:
            ((Dictionary new)
                at: #mess1 put: #mess1inState1;
                at: #mess2 put: #mess2inState1;
                yourself)
```

The currentState instance variable is initialized to state0 on instance creation. Messages such as mess1 are implemented as follows:

[1]If positive integers were used to represent the current state, then an Array or OrderedCollection of message names could be used instead.

mess1

```
self perform: ((TransitionTable at: currentState) at: #mess1)
```

Note that there is no handling of error conditions, such as unexpected messages or invalid states, in this example. This could easily be added using the at:ifAbsent: method from class Dictionary.

Finally, the individual methods which correspond to particular combinations of messages and states are implemented as follows:

mess1inState0

```
Transcript cr; show: 'Message mess1 in state 0 => go to state 1'.
currentState := #state1.
```

mess1inState1

```
Transcript cr; show: 'Message mess1 in state 1 => go to state 0'.
currentState := #state0.
```

Methods mess2inState0 and mess2inState1 print the current state in the Transcript, without changing the current state.

Clearly, this example forms a FSM with just two states; receiving the message mess1 in either state toggles the state, while mess2 simply reports the current state.

2.2.2 Using doesNotUnderstand:

One possible problem with this approach is that a separate method is required for every message which is to be understood. All these methods are only trivially different. Code reuse could be improved by removing all the methods corresponding to possible state transitions, and re-implementing the doesNotUnderstand: method, as follows:

doesNotUnderstand: aMessage

```
| entry |
entry := TransitionTable at: currentState.
(entry includesKey: aMessage selector)
    ifFalse: [ ↑super doesNotUnderstand: aMessage]
    ifTrue: [ ↑self
        perform: (entry at: aMessage selector)
        withArguments: aMessage arguments]
```

Now, when a message (such as mess1) is sent to an instance of TableStateMachine, it is not understood, since there is no corresponding method. Under these circumstances, the Smalltalk virtual machine arranges to send a doesNotUnderstand: message to the same object, with an instance of class Message as the argument; this argument Message represents the message which was not understood. The new implementation of doesNotUnderstand: checks whether the message name (selector) is one of those acceptable in the current state: if it is, the corresponding method is executed using perform:withArguments:; otherwise, the normal error handling mechanism is invoked. Finally, note that the arguments (if any) to the incoming message are made available to the actual method executed.

It should be noted that this reduction in code size does not come for free; the doesNotUnderstand: mechanism is typically rather slow compared with a normal method lookup. Nevertheless, this might be acceptable in many applications.

As before, it is possible to add additional checks for invalid states, as shown below:

```
doesNotUnderstand: aMessage
        | entry |
        entry := TransitionTable
            at: currentState
            ifAbsent: [↑self error: 'invalid state: ', currentState printString].
        (entry includesKey: aMessage selector)
            ifFalse: [ ↑super doesNotUnderstand: aMessage]
            ifTrue: [ ↑self
                perform: (entry at: aMessage selector)
                withArguments: aMessage arguments ]
```

Note that it is possible to replace the use of includesKey: with further use of the at:ifAbsent: method; this might be marginally more efficient.

2.2.3 Advantages and Disadvantages

Since once again there is a separate method for every combination of state and incoming message, the browsers provide effective support for the programmer. Also, inspecting the class variable TransitionTable also conveniently provides much information during code development.

A disadvantage is that again, different classes are needed for each particular application; however, as before, some functionality could be put in an abstract superclass.

Another point is the use of the perform:withArguments: and doesNotUnderstand: mechanisms. These might be less efficient than simple case statements. However, this may well not be a limiting factor on performance in many cases.

2.3 Mapping States to Blocks (1)

As a variant on the case shown in the previous section, it is possible to arrange that the transition table could map incoming messages to *blocks* rather than to message names. Furthermore, since we are now able to provide a single class, instances of which can be customized for many applications, all instances now *cannot* share a single transition table. Instead, an instance variable (transitionTable) is used to retain this data structure.

For this to work satisfactorily, it is essential to allow the code in the block the same access to both the receiver (i.e. the FSM) and to the incoming message which invoked the block. This could be arranged by allowing the blocks two block arguments: one being the receiver (i.e. self), and the other being the Message received. However, it seems slightly more convenient to provide three block arguments: self, the Message name (selector) and the Message arguments as an Array.

2.3.1 An Example

Here, the example from section 2.2.1 is recast to map messages to blocks. Class TableStateMachine now has two instance variables: one for the current state and one for the transition table:

```
Object subclass: #TableStateMachine
    instanceVariableNames: 'currentState transitionTable '
    classVariableNames: ''
    poolDictionaries: ''
```

The same approach using doesNotUnderstand: is used to map incoming messages to the appropriate block of code:

```
doesNotUnderstand: aMessage
    | entry |
    entry := transitionTable
        at: currentState
        ifAbsent: [↑self error: 'invalid state: ' , currentState printString].
    (entry includesKey: aMessage selector)
        ifFalse: [↑super doesNotUnderstand: aMessage]
        ifTrue: [↑(entry at: aMessage selector)
                value: self
                value: aMessage selector
                value: aMessage arguments]
```

Now, the transition table must be specified when an instance of TableStateMachine is created. The code below sets up a two-state FSM, with messages to toggle and enquire about the state:

```
| fsm table |
fsm := TableStateMachine new.
table := Dictionary new.
table at: #state0 put:
   ((Dictionary new)
      at: #mess1 put: [ :slf :mes :arg |
        Transcript cr; show: 'Message ', mes printString, ' in state 0 => go to state 1'.
        slf currentState: #state1 ];
      at: #mess2 put: [ :slf :mes :arg |
        Transcript cr; show: 'In state: ', slf currentState printString  ];
      yourself).
table at: #state1 put:
   ((Dictionary new)
      at: #mess1 put: [ :slf :mes :arg |
        Transcript cr; show: 'Message ', mes printString, ' in state 1 => go to state 0'.
        slf currentState: #state0  ];
      at: #mess2 put: [ :slf :mes :arg |
        Transcript cr; show: 'In state: ', slf currentState printString  ];
      yourself).
fsm setTransitionTable: table.
```

2.3.2 Advantages and Disadvantages

An advantage of this approach is that there is now a single class for FSMs, which can be reused for many cases. Here, the re-use is by effectively by *parameterization*, where the parameters are blocks. Of course, a user of this class still needs to provide much code, but as blocks, rather than as new methods in subclasses. This has the undesirable side-effect of severely limiting the usefulness of the browsers and other tools, which are not really able to give the programmer much help.

Another issue is that the use of the `doesNotUnderstand:` mechanism is now essential, and the previously expressed reservations about the performance implications of this should be reiterated. However, note that blocks are being evaluated, rather than methods being activated using `perform:`; this may prove to be somewhat less efficient, in some versions of Smalltalk.

2.3.3 More Compact Table Descriptions

With this approach, there is a slightly odd style for writing code in blocks; it is necessary to use the block arguments in an unusual way. For example, instead of accessing `self` directly, it is necessary to use the block argument `slf`. Also, it is somewhat inconvenient to have to access the arguments to messages by indexing into an `Array`: the `(arg at: 1)` approach. Furthermore, it is necessary to send messages such as `currentState:` and `currentState` (which simply set and return the `currentState` instance variable), since there is no direct access to the instance variables of the FSM.

This inconvenience could be overcome by insisting that all the blocks used in the FSM are defined in *instance* methods of class TableStateMachine, or perhaps in a user-specialized subclass. This would allow direct access to self, so that this would no longer need to be passed as an argument to the blocks. Also, the code inside the blocks would have direct access to the currentState instance variable, and to any other instance variables a user might define in a subclass. The code of creating the blocks can then be put in an initialize method, as follows:

```
initialize
        table := Dictionary new.
        table at: #state0 put:
           ((Dictionary new)
              at: #mess1 put: [ :mes :arg |
                 Transcript cr; show: 'Message ', mes printString, ' in state 0 => go to state 1'.
                 currentState := #state1 ];
              at: #mess2 put: [ :mes :arg |
                 Transcript cr; show: 'In state: ', self currentState printString ];
              yourself).
        table at: #state1 put:
           ((Dictionary new)
              at: #mess1 put: [ :mes :arg |
                 Transcript cr; show: 'Message ', mes printString, ' in state 1 => go to state 0'.
                 currentState := #state0 ];
              at: #mess2 put: [ :mes :arg |
                 Transcript cr; show: 'In state: ', self currentState printString ];
              yourself).
```

Another criticism of the above code is that the amount of text required to create instances of class Dictionary is rather large, thus introducing much clutter and possible confusion. However, by providing the '−>' method in class Object to answer an Association formed by the receiver and argument,[2] and by re-implementing the ',' (concatenation) method in classes Dictionary and Association,[3] it is possible to substantially reduce the coding clutter:

[2]This is a standard method in some Smalltalk systems, including Objectworks\Smalltalk.
[3]Thanks to Mario Wolczko for suggesting this trick.

```
initialize
    table :=
     (#state0 ->
       ((#mess1 -> [ :mes :arg |
           Transcript cr; show: 'Message ', mes printString, ' in state 0 => go to state 1'.
           currentState := #state1 ]),
       (#mess2 -> [ :mes :arg |
           Transcript cr; show: 'In state: ', currentState printString  ]))),
      (#state1 ->
        ((#mess1 -> [ :mes :arg |
           Transcript cr; show: 'Message ', mes printString, ' in state 1 => go to state 0'.
           currentState := #state0  ]),
        (#mess2 put: [ :mes :arg |
           Transcript cr; show: 'In state: ', currentState printString  ]))).
```

Note that blocks used in the way suggested here (with access to instance or other variables in another context) are no longer *pure*; some Smalltalk implementations are able to optimize the performance of pure blocks. This means that this approach might be slightly slower in some cases. Yet another variation would be for the blocks to return the next state, with the assignment to the currentState instance variable in the doesNotUnderstand: method. In this case, the blocks would again be pure, and perhaps a better performance would be possible.

2.4 Mapping States to Blocks (2)

Instead of retaining an instance variable which explicitly identifies a state, and accessing into a Dictionary (retained by an instance or class variable) to find the current mapping between messages and actions, it is possible to represent the current state *implicitly*, as a single Dictionary of the current mapping between messages and blocks. When the state is changed, then a new (or, more likely, a previously constructed) Dictionary is used. The doesNotUnderstand: mechanism can be used in a similar way to that outlined in section 2.3.

Clearly, this approach has less redundant information retained as instance variables, and there is less overhead in accessing the appropriate block. A variety of approaches could be used to retain the different blocks, including temporary variables in the method defining the blocks, a class variable, and so on.

2.4.1 Handling Errors

Here, the question of handling errors such as unexpected messages and states is re-examined, in the context of FSM implementations which define the behaviour in each state in terms of blocks. Of course, it will always be possible to invoke the default implementation of doesNotUnderstand: from class Object when unexpected messages arrive, but this approach might be a little heavy-handed.

Instead, it is possible to retain with a separate instance variable an exception block, which is executed if the current state (the Dictionary of blocks) does not contain a key corresponding to the message which has just been received. This could then invoke the normal doesNotUnderstand: mechanism (which might be a sensible default), or simply ignore the message (also a possible default action). Of course, the programmer is able to add any desired behaviour here.

For a more sophisticated approach, it may be desirable to have different error blocks invoked when receiving a message in different states. When an explicit state variable (currentState) is used, as in section 2.3, then an instance variable referring to a Dictionary of blocks would support state-dependent error handling. Since it may sometimes happen that many states do not require special error handling, the use of a DictionaryWithDefault[4] would reduce the amount of effort required.

However, when the current state is implicitly represented by a single Dictionary of blocks, as in this section, it is possible to define an instance variable retaining the error block for this state. This could be updated in particular states, as part of the state changing code, to specialize the error handling. Alternatively, there could be two instance variables: currentErrorBlock and defaultErrorBlock. The latter is always initialized on instance creation. If the former is nil, then the default is used when an error occurs, otherwise the current error block is used directly.

[4] Class DictionaryWithDefault is not a standard library class, but can be added with little effort. Typically, it is a subclass of Dictionary, and retains a default value using an additional instance variable. When accessing, if the key supplied is not found, then the default value is returned.

2.4.2 An Example

Class DynamicStateMachine defines two instance variables: messages retains a Dictionary mapping message names to blocks, while exceptionBlock refers to a block executed under error conditions.

```
Object subclass: #DynamicStateMachine
    instanceVariableNames: 'messages exceptionBlock '
    classVariableNames: ''
    poolDictionaries: ''
```

Once again, the doesNotUnderstand: message is overridden; this implementation either executes a block extracted from the messages Dictionary, or the block retained by exceptionBlock.

```
doesNotUnderstand: aMessage
        ↑(messages
            at: aMessage selector
            ifAbsent: [exceptionBlock])
                value: self
                value: aMessage selector
                value: aMessage arguments
```

In this example, a simple two-state FSM is created, with the states represented by two instances of Dictionary retained by variables mesSet1 and mesSet2. In the first state, message mess1 toggles to the other state while mess3 reports the current state; mess2 is not allowed, and invokes the exception block. In the second state, the roles of mess1 and mess2 are reversed. The system starts in state 1.

```
| fsm mesSet1 mesSet2 |
fsm := DynamicStateMachine new.
mesSet1 := (Dictionary new)
    at: #mess1 put: [ :s :m :a |
        Transcript cr; show: m printString, ' => Go to state 2'.
        s messages: mesSet2];
    at: #mess3 put: [ :s :m :a |
        Transcript cr; show: m printString, ': In state 1'.];
    yourself.
mesSet2 := (Dictionary new)
    at: #mess2 put: [ :s :m :a |
        Transcript cr; show: m printString, ' => Go to state 1'.
        s messages: mesSet1];
    at: #mess3 put: [ :s :m :a |
        Transcript cr; show: m printString, ': In state 2'.];
    yourself.
fsm messages: mesSet1.
fsm exceptionBlock: [ :s :m :a |
    Transcript cr; show: 'Message ', m printString, ' ignored.'].
```

2.4.3 Advantages and Disadvantages

One interesting aspect of this particular implementation technique is that it is possible to alter the state table dynamically, depending (for example) on the messages received. This means that the system no longer represents a *finite* state machine.

The following example creates a state machine capable of responding to *any* message. On the first occurrence of a message, the state machine is modified to add a new response to that message, which is then activated when that message is next received.

```
| fsm |
fsm := DynamicStateMachine new.
fsm exceptionBlock: [ :s :m :a |
    Transcript cr; show: 'New message added: ', m printString.
    s addMessage: m block: [ :ss :mm :aa |
        Transcript cr; show: 'Existing message received: ', mm printString]].
```

Note that this example uses a method called addMessage:block:, which simply adds a new entry to the current Dictionary of blocks.

As in the previous case, a single class is available which can be reused for many applications, by parameterization using blocks. This version is more flexible than that described in section 2.3, since it can be easily used for non-finite state machines; however, the applications for such machines may be rather limited. Also, since all the user-written code is in blocks, the usual class browser is not very useful. As in previous cases, it is straightforward to make copies of the Dictionary of blocks, and then modify these copies, as a convenient way of implementing lots of similar states.

Note that it would also be possible to use a similar approach, but with a single Dictionary mapping messages to methods, which are executed using perform:, in the style of section 2.2. However, this has few advantages, since a shared class variable cannot be used for the transition table, and is not considered further here.

As in the previous approach (section 2.3), it is possible to insist that all blocks are defined in instance methods, thus allowing immediate access to *all* instance variables. Furthermore, the ‘abbreviated’ Dictionary creation methods (section 2.3.3) can also be used. The example shown above could then be recoded:

```
initialize
        | mesSet1 mesSet2 |
        mesSet1 :=
          (#mess1 -> [ :mes :arg |
             Transcript cr; show: 'Go to state 2'.  messages := mesSet2]),
          (#mess3 -> [ :mes :arg |
             Transcript cr; show: 'In state 1'.]).
        mesSet2 :=
          (#mess2 -> [ :mes :arg |
             Transcript cr; show: 'Go to state 1'.  messages := mesSet1]),
          (#mess3 -> [ :mes :arg |
             Transcript cr; show: 'In state 2'.]).
        messages := mesSet1.
        exceptionBlock := [ :mes :arg | Transcript cr; show: 'Message ignored.'].
```

2.5 Mapping States to Classes (1)

Yet another approach is to have each state of the FSM represented by an instance of one of several different *classes*; all these classes can conveniently all be subclasses of a single abstract superclass. Each class implements its own collection of methods for all of the messages an instance can receive, corresponding to the particular state which that class represents. Then, a single instance of another class, not related by inheritance, is used to represent the entire FSM. This is similar to an approach discussed by Meyer ([10], chapter 12), in an example of user interface control.

2.5.1 An Example

In this example, a FSM with three states is constructed, with two messages increment and decrement. This implements a simple counter which counts either up or down, modulo-3. The actual state machine is represented by an instance of class ClassStateMachine:

```
Object subclass: #ClassStateMachine
    instanceVariableNames: 'state '
    classVariableNames: ''
    poolDictionaries: ''
```

Methods to set and access the state instance variable are provided. The actual states are represented by instances of classes StateZero, StateOne and StateTwo, which are all subclasses of class AbstractState:

```
Object subclass: #AbstractState
    instanceVariableNames: 'fsm '
    classVariableNames: ''
    poolDictionaries: ''
```

Again, methods to set and access the instance variable fsm are provided. Note that the object representing the current state and the object representing the entire state machine have references to each other. For convenience, an instance creation (class) method newIn: is provided, which creates a new 'state' object already referring to a FSM.

The three concrete classes each provide their own implementation of the increment and decrement methods. For class StateZero, the following definitions are used:

```
increment
        Transcript cr; show: 'In state 0 => go to state 1'.
        fsm state: (StateOne newIn: fsm)

decrement
        Transcript cr; show: 'In state 0 => go to state 2'.
        fsm state: (StateTwo newIn: fsm)
```

Similar definitions are provided for classes StateOne and StateTwo. Note that these methods create new instances of StateOne and StateTwo, and install the two-way relationship between the state and FSM objects, by sending the messages state: and newIn:. Finally, ClassStateMachine must itself implement the messages increment and decrement, and simply forward them on to the object currently representing the state; for example:

```
increment
        state increment
```

2.5.2 Advantages and Disadvantages

Since a separate class is used to represent the behaviour of the FSM in each of its states, the browsing and other programming tools can give much support. This might be important if the behaviour in each state was rather complex.

However, the class representing the FSM itself (ClassStateMachine) has to provide a large number of essentially trivial methods, which simply forward the same message to the object representing the current state. This could require lots of methods. It is possible to use the doesNotUnderstand: mechanism to automatically forward messages to the 'state' object:

```
doesNotUnderstand: aMessage
        | selector |
        selector := aMessage selector.
        (state respondsTo: selector)
            ifTrue: [state perform: selector withArguments: aMessage arguments]
            ifFalse: [super doesNotUnderstand: aMessage]
```

Note that the respondsTo: message is used to ensure that the 'state' object can understand the message to be sent to it; this means that the normal error handling mechanism can be invoked if an unexpected message is received. Finally, note that this approach means creating (and destroying) an object on every state transition; together with the overheads of doesNotUnderstand: and respondsTo:, this might reduce the available performance.

2.6 Mapping States to Classes (2)

In the approach discussed in section 2.5, there seems to be a rather artificial distinction between the FSM and the object representing its state. This also gives rise to some stylistically poor coding practices. As an alternative, consider an approach where a single object is created, which is an instance of the class corresponding to the initial state of the FSM. To change the state of the FSM, the class of this object must be changed, without destroying its identity or any other property. This removes the distinction between the FSM and its state.

A convenient way to change the class of an object efficiently is to use the changeClassToThatOf: method. This is implemented as a primitive operation, and changes the class of the receiver to that of the object supplied as an argument.[5] There are restrictions on the structure of the classes which can be changed in this way; however, provided that all classes corresponding to different states have the same number of instance variables – this is readily achieved by having any instance variables defined in the abstract superclass – this does not become a problem.

2.6.1 An Example

As an example, consider a FSM represented by an abstract class AbstractClassStateMachine, which has three states represented by concrete subclasses StateZero, StateOne and StateTwo. None of these classes defines any instance variables. Each of these concrete classes defines two methods: increment and decrement. On the receipt of increment, the FSM moves to the next higher state (modulo 3); decrement moves to the next lower state. For class StateZero, these two methods are implemented as shown:

```
increment
    Transcript cr; show: 'Change from state 0 to state 1'.
    self changeClassToThatOf: StateOne new.

decrement
    Transcript cr; show: 'Change from state 0 to state 2'.
    self changeClassToThatOf: StateTwo new.
```

Corresponding methods are provided in classes StateOne and StateTwo. Clearly, this implements a ‘counter’ which increments and decrements as required, as illustrated by the following code:

[5] There are detailed technical reasons why the more obvious approach, using a method such as changeClassTo: with a Class provided as an argument, might be much less efficient.

```
| fsm |
fsm := StateZero new.
fsm increment.      "fsm is an instance of StateOne."
fsm increment.      "fsm is an instance of StateTwo."
fsm decrement.      "fsm is an instance of StateOne."
fsm decrement.      "fsm is an instance of StateZero."
fsm decrement.      "fsm is an instance of StateTwo."
```

2.6.2 Using Class Instance Variables

In methods which change the state, such as the increment and decrement methods in the example, it can be seen that a new instance of the 'target' class is created, simply to permit the use of the changeClassToThatOf: primitive. This object is then immediately discarded. To avoid this unnecessary object creation, it is possible to create a single 'prototype' instance of each class, which can be used repeatedly. A convenient way of storing the 'prototype' objects is by using a *class instance variable*.[6]

```
AbstractClassStateMachine class
        instanceVariableNames: 'prototypeInstance '
```

A class initialization method is of course required, which must be separately executed for *all* subclasses before they can be used. A class method to return the 'prototype' instance is also required.

```
initialize
    "AbstractClassStateMachine withAllSubclasses
            do: [ :eachClass | eachClass initialize]."
    prototypeInstance := self new.

prototype
    ↑prototypeInstance
```

2.6.3 Use of become:

The method changeClassToThatOf: is implemented as a primitive operation in the Smalltalk virtual machine, and is only available in Objectworks. Some other versions of Smalltalk provide different primitives with similar behaviour. For example, some versions of Smalltalk from Digitalk (such as Smalltalk/V-Windows and Smalltalk/V-Presentation Manager) provide a class:

[6]Class instance variables are defined in the *metaclasses*, and are thus only accessible in the class methods of the defining class. They differ from class variables in their scope: a class variable is shared by the defining class, all instances of this class, all subclasses (direct and indirect) of the class, and all instances of all these classes. A class *instance* variable is only accessible in class methods, and is *not* shared by subclasses; the subclasses all have a private copy of the variable.

method and corresponding primitive, which can set the class of the receiver directly to the class provided as an argument. However, the use of this primitive may be undesirable or slow. It seems *very* unlikely that the functionality of changeClassToThatOf: could easily be added (as a user-defined primitive, for example) to a version of Smalltalk where it is not already provided by the supplier.

In systems where changeClassToThatOf: or a similar method is not available, it is possible to use the become: method to perform a similar function. The become: method is also primitive, and is available in Smalltalk systems from both suppliers; it allows the object supplied as an argument to take on the identity of the receiver object.[7] So, provided that the objects representing the state of the FSM do not have any instance variables whose values must be retained during state changes, the new state can be entered by creating a new instance of the appropriate class and use become: to ensure that it takes on the identity of the object representing the existing state:

```
increment
    Transcript cr; show: 'Change from state 0 to state 1'.
    self become: StateOne new.
```

Note that there might be some concern over the efficiency of the become: method, compared with that of changeClassToThatOf:. Certainly, in some implementations of the Smalltalk virtual machine (such as those without an object table), the use of become: might be very expensive indeed.

Finally, it should be noted that class instance variables are really only supported in Objectworks, although it does appear to be possible to add them to Digitalk products. Alternatively, it would be possible to use a class variable retaining a Dictionary which maps classes to 'prototype' instances.

2.6.4 Advantages and Disadvantages

As in many previous cases, this approach is really only able to represent finite state machines, unless one is willing to invoke a compiler to generate new classes on the fly (see section 3.3). Of course, this is unlikely to be a problem in many application areas.

Since each state is represented by a distinct class, with different responses to the same event (in different states) represented by methods of the same name in different classes, it is possible to take great advantage of the capabilities of the browser and associated tools. Furthermore, for FSMs which have a large number of states, and where there are many states which have common responses to events, it is possible to introduce intermediate abstract subclasses, which represent states which react in the same way to some messages. This is not a complete solution, however, as in the absence of multiple inheritance, there are problems when there are multiple partially-overlapping sets of methods for each state. Also, a different collection of subclasses will be required for each application, so that there is little opportunity for code reuse. Nev-

[7] The semantics of become: varies slightly between suppliers. Digitalk's version operates as described here; in Objectworks, the identity of receiver and argument are interchanged.

ertheless, since each application will necessarily need to describe its own behaviour in detail, there well not be many natural places where code reuse is feasible.

When comparing this technique with previous approaches, there is the question of the relative efficiency of the basic mechanisms used. For example, the trade-offs between using doesNotUnderstand: or perform: for every message send, and using changeClassToThatOf: or become: every time the state is changed, may be difficult to determine. In any case, it is likely to be sensitive to the implementation technique used by the Smalltalk virtual machine, so could well vary between versions. Of course, the absolute performance of FSMs may well not be the limiting factor in many implementations.

Finally, using this technique gives the unusual effect of, when inspecting an instance, the class as reported in the title bar does not always correspond to the class reported when selecting self. This is of course a natural outcome of using an approach which changes the behaviour (i.e. class) of objects without altering their identity; the tools assume that the class of an object is fixed at instance creation time.

Chapter 3

Concluding Remarks

Section 2 introduced several techniques for FSM implementation in Smalltalk, but which one should be used? The choice is by no means clear, although some of the techniques described earlier in section 2 show evidence of poor coding style. This question is discussed further in section 3.1.

There is also some relationship between the FSM approach suggested here, and some more unusual features in certain object-oriented programming languages. This is touched upon in section 3.2. Finally, the issue of FSM applications dynamically changing their behaviour by the use of the `Compiler` is discussed in section 3.3.

3.1 Which Approach to Use?

The approaches outlined previously can be broadly divided into two kinds. Firstly, there are those techniques which create separate sets of methods for each application and for each state of the FSM. These have generally poor code re-use, but the class browser and other programming tools are able to provide useful assistance to the programmer. Alternatively, there are those techniques which allow the use of the same class for every application, but which customize the behaviour using blocks. Here, the browser is much less supportive, but avoids some code duplication.

For applications of only moderate complexity, it would seem that the approach described in section 2.4 is appropriate. This allows a single class to be re-used by adding user-defined blocks. Furthermore, it the most flexible, in the sense that self-modifying state machines can be readily created.

However, for the most complex and sophisticated applications, one of the other approaches might be preferred, so that the full power of the class browsers and other programming tools can be utilized. For example, the multiple class approach (sections 2.5 and 2.6), or the approach using a class variable to map messages to methods (section 2.2) might be more suitable. The latter approach is perhaps especially useful when only part of the behaviour of a class can sensibly be described by a FSM. Note however, that there is very little opportunity for code reuse in this case.

3.1.1 A Preferred Approach

Perhaps the overall best approach is a compromise: using the techniques outlined in sections 2.3.3 and 2.4, construct an abstract superclass where the state is implicitly represented as a Dictionary of blocks, retained by the instance variable messages. This Dictionary is set up in initialize methods, defined (differently) in subclasses.

```
Object subclass: #DynamicStateMachine
    instanceVariableNames: 'messages exceptionBlock '
    classVariableNames: ''
    poolDictionaries: ''
```

When defining a new subclass, the user then has the choice of either putting all the code for each state and message combination in blocks, or simply sending further messages to self in the blocks. In the latter case, the browser is able to support the proliferation of methods reasonably effectively. The instance variables referencing the current state and the exception handling blocks are immediately accessible, as are any user-defined instance variables.

3.1.2 A Larger Example

This example uses the preferred approach identified in section 3.1.1. The application considered is a FSM which counts the number of 'words' and 'lines' in a String[1] (or other SequenceableCollection of Characters). This is much in the style of the unix `wc(1)` command, although the definition of 'word' and 'line' used here is different. Here, 'words' are defined to be sequences of *alphabetic* characters, separated by any number of any other characters; the separating characters include punctuation and 'white space' characters, as well as digits. 'Lines' are separated by any one of several 'end-of-line' characters; here, they are assumed to be 'carriage return' and 'line feed'. Of course, a String with no characters has no words and no lines.

The FSM for this application is shown in figure 3.1. This FSM has three states, and starts in state startOfLine. In each state, three different messages can be received: eol, corresponding to the end of the line; letter, corresponding to an alphabetic character, and other, for the punctuation and other separating characters. There are two counters initialized to zero: one for lines and the other for words. On some state transitions, either or both of these counters will be incremented.

The FSM is implemented as a class inheriting from DynamicStateMachine, called WordLineCounter. It defines two additional instance variables for the line and word counts. Class variables are used to retain Arrays containing the characters considered to be part of a 'word', and those considered to mark the end of a line.

```
DynamicStateMachine subclass: #WordLineCounter
    instanceVariableNames: 'wordCount lineCount '
    classVariableNames: 'CharactersInWords EndOfLine '
    poolDictionaries: ''
```

[1] Thanks to Ian Piumarta for suggesting this example.

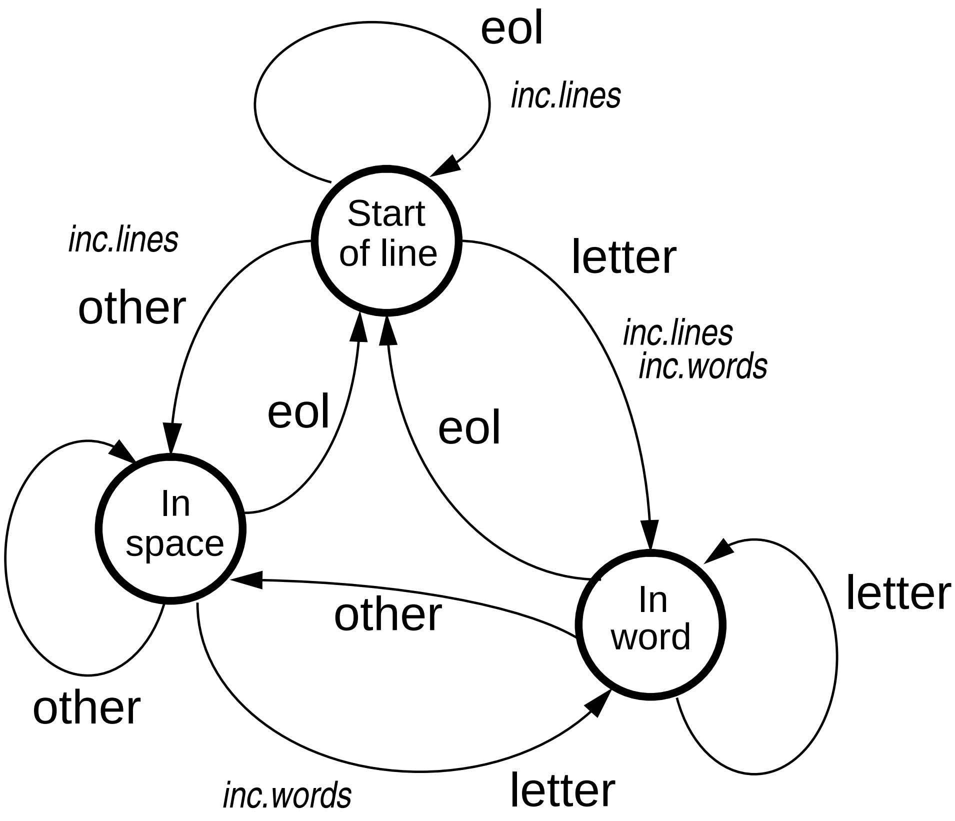

Figure 3.1: FSM for counting words and lines

The basic operation of the FSM is defined by the three instances of Dictionary defined in the initialize method. These define the response to the messages in each of the states.

initialize

```
| startOfLine inWord inSpace |
super initialize.
wordCount := lineCount := 0.
startOfLine :=
    (#eol -> [ :mess :args | self incrementLineCount]),
    (#letter -> [ :mess :args |
        self incrementLineCount.
        self incrementWordCount.
        messages := inWord]),
    (#other -> [ :mess :args |
        self incrementLineCount.
        messages := inSpace]),
    (#currentState -> [ :mess :args | #startOfLine]).
inWord :=
    (#eol -> [ :mess :args | messages := startOfLine]),
    (#letter -> [ :mess :args | "do nothing" ]),
    (#other -> [ :mess :args | messages := inSpace]),
    (#currentState -> [ :mess :args | #inWord]).
inSpace :=
    (#eol -> [ :mess :args | messages := startOfLine]),
    (#letter -> [ :mess :args |
        self incrementWordCount.
        messages := inWord]),
    (#other -> [ :mess :args | "do nothing" ]),
    (#currentState -> [ :mess :args | #inSpace]).
messages := startOfLine.        "Counter starts in this state."
```

Each of the Dictionarys is built using the ‘comma’ method, as discussed in section 2.4.3. Methods are provided to increment the word and line counters. Note that a fourth message is provided, which simply returns a Symbol identifying the current state; this is not used in this example.

To provide a convenient ‘user’ interface to the WordLineCounter, the nextChar: method identifies the incoming characters, and sends the appropriate message to self.

nextChar: aCharacter

```
(EndOfLine includes: aCharacter) ifTrue: [↑self eol].
(CharactersInWords includes: aCharacter) ifTrue: [↑self letter].
↑self other
```

The following code illustrates the operation of the FSM; with the example string shown, it reports 16 words and 3 lines.

```
| string counter |
counter := WordLineCounter new.
string := 'Now is the time
for all good men
to come to the aid of the party.'.
Transcript cr; show: string.
string do: [ :eachChar | counter nextChar: eachChar].
Transcript cr; show: 'Words: ', counter wordCount printString.
Transcript space; show: 'Lines: ', counter lineCount printString.
```

Note that, if the individual behaviours in each state were rather complex, they could be implemented as separate methods in this class.

3.2 Relationships with other Language Constructs

Several other object-oriented programming languages, particularly those concerned with correct behaviour when concurrency is used, permit the message (public, client) interface to be restricted dynamically, under the control of the programmer. Messages corresponding to methods which are currently not available may be blocked, awaiting a change of state of the receiver object; clearly, this is really only appropriate if message-sending is at least sometimes potentially concurrent. Alternatively, sending a message when the corresponding method is unavailable can raise a exception; this interpretation still permits a serial semantics for message sending. The latter case clearly corresponds to the FSMs discussed in this report; perhaps the approach where the class of the FSM object is changed depending on the state (section 2.6) is the most natural implementation from this viewpoint.

The approach described in section 2.6, where the class of the object representing the FSM changes in order to represent the change of state (and thus behaviour), has much in common with the approach used in languages like Self [11] and actors [12]. In Self, *dynamic* inheritance could be used in a straightforward way to change the behaviour of an object depending on the 'state' currently supported, by updating a 'parent' slot. Similarly, in actors, the more general *delegation* approach could be used. This leads to the view that the FSM style of design may be a very natural approach for such languages; certainly, communicating state machines has been used as a model for concurrent systems.

3.3 Dynamic Generation of Code

In section 2.4.3, it was suggested that the technique of implicitly representing the state of a FSM by a Dictionary of blocks made it possible to implement state machines which dynamically changed their behaviour. While this appears to be the most convenient way of creating machines with potentially non-finite behaviour, it is in principle possible to provide this facility in other ways. In particular, as long as it is possible to compile new code while the system is running (which is the normal style of operation of class Compiler), then it will be possible to generate new methods, blocks or classes 'on-the-fly'. Of course, there are stylistic and software engineering reasons why the use of the Compiler in this fashion might be undesirable: certainly, an application which generates its own code under some circumstances is likely to be much harder to design, test, debug or reason about. Also, some suppliers have 'run-time' licence arrangements which preclude the use of the Compiler in delivered applications.

Acknowledgements

The author would like to thank the following for stimulating discussions on the topics covering in this report: Carlos Camarão de Figueiredo, Lih Shiew Guo, Benedict Heal, Ian Piumarta and Mario Wolczko. Mario Wolczko and Benedict Heal also read closely and provided much useful feedback on early drafts of this report.

Bibliography

[1] Grady Booch, *Object-Oriented Design with Applications*, Addison-Wesley, 1991.

[2] *An Investigation into the Feasibility of applying the Object-oriented Approach to the Development of OSI Communication Software*, Lih Shiew Guo, M.Sc Thesis, Computer Science Department, University of Manchester, October 1990.

[3] *Objectworks\Smalltalk Version 4.0 Reference Manual*, ParcPlace Systems, 1991.

[4] David Robson and Adele Goldberg, *Smalltalk-80: the Language and its Implementation*, Addison-Wesley, 1984.

[5] *Smalltalk/V for Windows: Tutorial and Reference Manual*, Digitalk Inc., 1991.

[6] *Smalltalk/V for Presentation Manager: Tutorial and Reference Manual*, Digitalk Inc., 1991.

[7] *Programming in Opal*, Servio Corporation, Alameda, California, USA, 1991.

[8] *GemStone ODBMS for Sun SPARCstations*, Servio Corporation, Alameda, California, USA, 1993.

[9] Ralph Johnson and Brian Foote, *Designing Reusable Classes*, Journal of Object-Oriented Programming, June 1988, pp. 22-35.

[10] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.

[11] David Ungar and Randall B. Smith, *Self: The Power of Simplicity*, ACM SIGPLAN Notices, vol. 22, no. 12, December 1987, pp. 227-242.

[12] Gul A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.