

Implementation of state machines with tasks and protected objects

Dr. Bo I. Sandén
Colorado Technical University
4435 N. Chestnut Street
Colorado Springs, CO 80907-3896
USA
Phone: (719) 590-6733
E-mail: bsanden@acm.org
<http://artemis.coloradotech.edu/~bsanden>

This article has previously appeared in Ada User Journal 20:4 (Jan 2000), 273-288.

Abstract

State machines are common in real-time software. They are also a standard way of describing the life of an object in object-oriented analysis and design. Their use is fairly straightforward but non-trivial. The examples in this article show implementations of state machines and associated activities by means of tasks and protected objects. They cover a range of synchronization and communication needs between activity and state machine.

1. Introduction

The software engineering community is currently very interested in software architecture and design patterns. By studying, describing and reusing existing architectures we can avoid reinvention, and leverage past experience to produce better designs. By choosing among a set of applicable architectures we can often find solutions without going back to first principles. By encouraging designers to examine different architectures we may also discourage them from reusing favorite architectures in situations where they are ill suited, and from slavishly following design methods that always produce certain architectural styles.

Design patterns are discussed at two levels. The more conceptual and aesthetic discussion is based on the work of Architect Christopher Alexander. His ambition is to design buildings that are true to the nature of their surroundings and purposes. It is easy to recognize and be awed by such buildings, but difficult to describe their quality. Alexander attempts to express this "quality without a name" through patterns which he defines as "recurring solution[s] to a common problem in a given context and system of forces".

Although software-pattern authors often refer to Alexander, they are usually not concerned with fitting a program into natural surroundings but instead with the internal engineering of the software. For this reason, most software design patterns exist at a more practical level and may be defined as "description[s] of communicating classes and objects that are customized to solve a general problem in a particular context" [4].

Like architectures, patterns are vehicles for design reuse. Design patterns provide analysts, designers and programmers with a vocabulary to talk about often-used constructs. These con-

structs may be of many different kinds but are generally more abstract than programming idioms.

To encourage designers to reuse, existing architectures must be readily available. This article attempts to illustrate this on a small scale by presenting a set of architectural solutions for state machines and associated activities in a concurrent environment. Due to the many possible execution scenarios, concurrent programs are more difficult to debug and test than sequential ones. For this reason, it is particularly important to arrive at a simple and transparent architecture and to get the design right the first time.

The solutions are intended as ready-made designs for a set of problems identified during analysis. The analyst describes an entity by means of a state machine that associates events with state transitions. Given such a state machine, a design solution can then be found in the set presented here. To account for most common cases we need a range of different solutions that involve a protected object and/or a number of tasks. This may make these solutions somewhat too varied to fit the mold of patterns. They are intended for real-time applications where events either occur as interrupts or are detected by means of polling.

1.1 State machines

State machines, usually in the generalized form of Statecharts [6] have become a standard way of describing the behavior over time of the instances of a class. They are included in most approaches to object-oriented analysis and design and play an important role in UML, the emerging industry standard.

In a real-time system, an object whose dynamic behavior can be described by means of a single state machine typically controls an electro-mechanical or electronic device that can be described as a finite automaton. Everyday examples range from a toaster to an automated garage door, an answering machine and a cruise control system or a window elevator for a car.

In the solutions in this article, a state machine is represented either as a protected object or as a task. Internally, in a protected object, the state is represented *explicitly*. This means that a state variable, usually of an enumerated type, is declared in the private part. The state variable then typically appears in barriers and other conditional expressions. In a task, the state is represented either explicitly as the value of a local state variable, or *implicitly* by the position of the program counter in the text [13]. Both representations are illustrated later in this article.

1.2 Concurrency and real time computing

Ada 95 is one of very few industry-strength languages that combine object orientation and concurrency in a real-time environment. Java is a contender with its built-in support for threads and synchronized objects, but still needs a standard adaptation for real-time applications where interrupt handling is required.

A central theme in architectures and patterns for concurrent software is the distinction and interaction between active and passive objects, particularly *shared* objects that need provisions for exclusive access by one active object at a time. Generally speaking, active and shared,

passive objects translate into tasks and protected objects in Ada 95. The distinction is less clear in Ada 83, where tasks are used both for active objects and to enforce mutual exclusion in passive objects. This makes Ada 95 ideally suited for expressing concurrent patterns and architectures.

Especially on a single processor, it may be important to ensure that a concurrent solution is not slower than a sequential one. More tasks certainly do not guarantee better use of the processor, since additional context switches add to the CPU load. The solutions presented here illustrate a restrictive use of tasking without unnecessary context switches. A state machine is generally implemented as a protected object, while tasks implement activities. The more general, restrictive design philosophy behind this is discussed elsewhere as *entity-life modeling* [3, 12, 13, 14, 15, 16].

2. Solutions

2.1 Components

The state machine solutions involve the following components, each of which may have one or more possible implementations.

Event captor: The capture of events is not explicitly included in the solutions. It is assumed that each event, except timing events, can be made to cause a call to a protected procedure in a protected object or an entry call to a task. (A timing event is defined by a certain period of time having passed since another event; see section 3.) The following scenarios are typical:

Events detected by means of polling are captured by one or more periodic, sampler tasks. (The number of samplers depends on whether the periods are harmonic and on other considerations to do with the separation of concerns.)

For events emanating from other subsystems, tasks within those subsystems are responsible for the necessary procedure/entry calls. (In a distributed situation, one or more tasks at the node where the state machine resides are responsible for receiving inter-node messages and making the necessary calls.)

Events that create interrupts are mapped to interrupt handling procedures in protected objects.

State machine: As a standard solution, a protected object represents the state machine. Any actions, which are by definition instantaneous, are included in the protected operations. In simple cases, some activities can be combined with the state machine implementation in a task. If there are timing events the state machine may also need to be a task, particularly if there are time-outs, which can be expressed in delay clauses in select statements.

Another case when the state machine is implemented as a task is when it needs exclusive access to shared resources in competition with other tasks. This situation is be-

yond the set of solutions discussed here and is covered by the shared resource pattern (also known as the resource-user thread pattern) [3, 16, 17].

Activities: An activity is here assumed to be a series of periodic actions by the software. (The period need not be constant.)

An activity may be limited to single states or may start in one state and end in another. (The latter is the case when the activity is specified for a superstate with multiple sub-states. It then spans whatever substates are entered on a particular visit to the superstate.) Activities may overlap. (For example, an activity specified for a superstate may overlap with activities specified for substates.)

Passive interface: The state machine or its activities may reference external interfaces or objects encapsulating data structures. These interfaces or objects are passive in the sense that the state machine or an activity calls their operations; they do not call the state machine or the activities. The purpose of the calls may be to obtain the current state of a subsystem: the speed of a vehicle, a flow rate, a pressure, a temperature, etc. It may also be to operate on an actuator such as the garage-door or window-elevator motor.

Building on these components, the following cases are identified:

The standard case with the state machine implemented as a protected object and any activities as tasks is discussed in 2.2. This basic case further assumes that any communication between the state machine and an activity once it has started occurs at the same periodicity as the activity. This includes starting and stopping the activity.

A simplification is possible if each activity is limited to single states. In that case, we can combine the state machine and the activities in a task. This is illustrated in 2.3.

The case where an activity may have to be stopped or receive parameters before the end of a period is discussed in 2.4. In the example there, it is the period itself that is changeable. If the period is to be shortened, the wait for the end of the current period must be cut short.

Section 2.5 discusses the case where it is advantageous to let an activity task run at all times but its output is suppressed in certain states. The reason may be a need to keep the periodicity consistent (as for example every minute on the minute) across intervals when the activity's output is not needed.

2.2 State machine implemented as protected object; activities implemented as periodic tasks

The activities are implemented as one or more periodic tasks. The state machine is implemented as a protected object that provides a means for activities to query the current state or detect a state change. An activity queries the state at appropriate points in its cycle, typically upon return from a delay.

2.2.1 Example: Cruise Control

This cruise control problem is a particularly popular example of real-time software design [1, 5, 7, 8, 9, 15, 18]. The car driver determines the state of cruising and may select the car's current speed at a given time as the target cruising speed by manipulating various controls such as a cruise control lever, the brake and the gear stick. We shall assume that these devices cause interrupts.

The state of cruising is determined by a state machine with a periodic activity that controls the throttle when cruising is activated. In the following, the protected unit `Cruise_FSM` that represents the state machine is first presented in some detail, then the activity `Throttle` that periodically adjusts the physical throttle and communicates with `Cruise_FSM`

2.2.1.1 Protected unit `Cruise_FSM`

Many ways to describe a cruise control system in a state transition diagram are represented in the literature [5, 18]. Fig. 1 shows a fairly simple example, the cruise control for a car with manual transmission (Exercise 5.8 in [13]). There is a `Set` switch and a `Resume` switch. Pushing `Set` causes the car to coast. The current speed at the time when the button is released is taken as the new target cruising speed. The event `Unset` in the diagram is when the button is released. Similarly, the driver can cause the car to accelerate by pushing `Resume`. The speed at the time of the `Unresume` event becomes the new target speed.

Cruise control is suspended as soon as the driver touches the brake or the clutch. This is indicated by the `Brake` or `Clutch` event, and takes the cruise control system to the `Passive` state. In `Passive`, if the driver pushes and releases the `Resume` button, the car goes back to cruising at the earlier target speed, provided the current speed is above a certain minimum ("Min" in the figure).

Events that cause neither a state transition nor an action have been left out of the diagram.

The state machine is implemented as the protected unit `Cruise_FSM`. In addition to the internal states shown in Fig. 1, it must also deal with the following states visible to the `Throttle` task:

`Constant_Acceleration`, corresponding to the internal state `Accelerating`

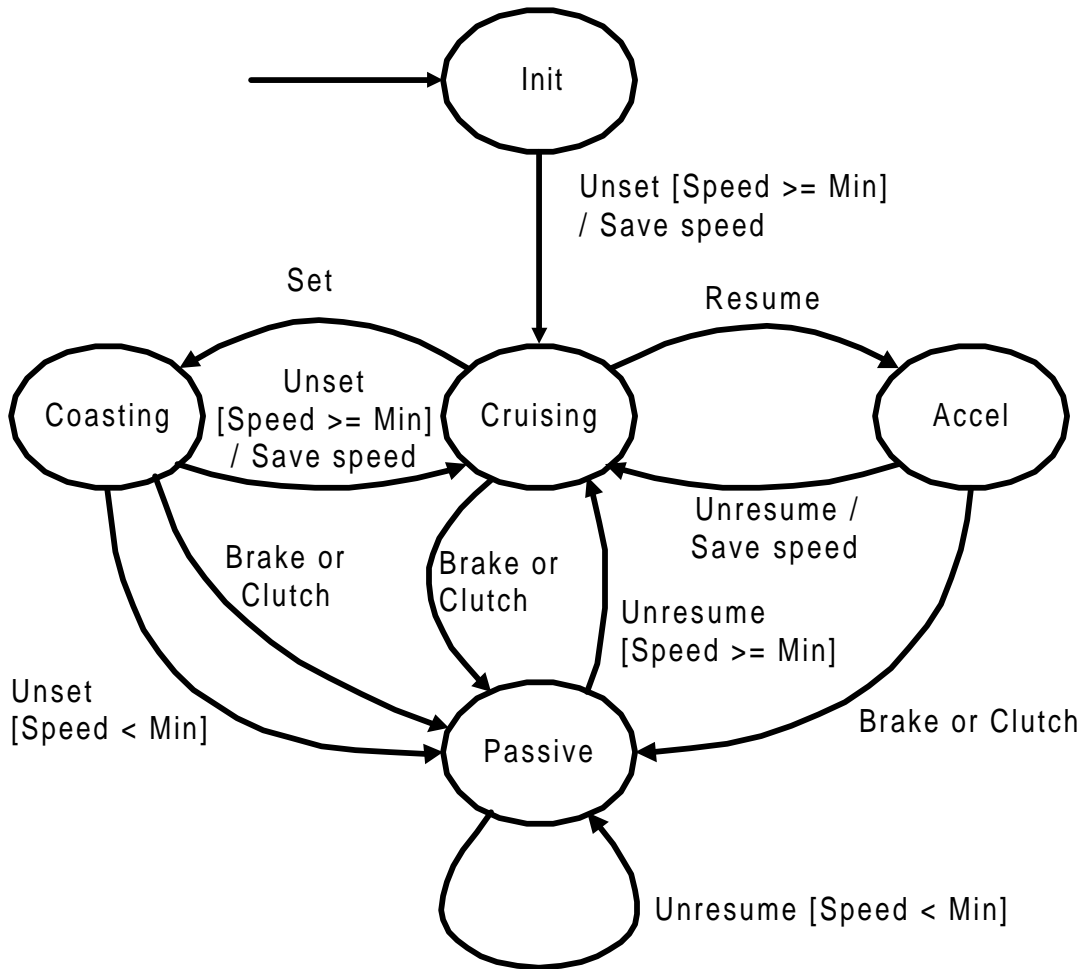
`Constant_Speed`, corresponding to the internal state `Cruising`

`Inactive`, corresponding to the internal states `Init`, `Coasting` and `Passive`.

`Cruise_FSM`'s protected functions `State` and `Speed` return the current external state and the current target speed, respectively.

`Driveshaft` is a passive interface. The function `Driveshaft.Speed` returns the current speed.

Fig. 2 The cruise control state machine



In this implementation of a the cruise-control state machine, each interrupt has a handling procedure that simply makes a call to the procedure `State_Transition` with the current interrupt event as a parameter. The advantage of this is that `State_Transition` can be implemented in the traditional way as an outer case statement over states and for each state, an inner case statement over events. This is a very direct translation of the diagram in Fig. 1. (A slightly shorter implementation would be to instead include a case statement in each interrupt procedure over the states in which a particular interrupt can occur as in 3.1.)

```
type Event_Type is (Set, Unset, Resume, Unresume, Brake_or_Clutch);
```

```
Minimum_Speed : constant Float := ... ;
```

```
protected Cruise_FSM is
```

```
    function State return External_State_Type;
```

```
    function Speed return Float;
```

```
private
```

```
    Internal_State : Internal_State_Type := Init;
```

```

Target_Speed : Float;

procedure Set;
procedure Unset;
procedure Resume;
procedure Unresume;
procedure Brake_or_Clutch;

pragma Attach_Handler (Set, ...);
etc.

procedure State_Transition (E : Event_Type);
end;

protected body Cruise_FSM is
  function State return External_State_Type is
  begin
    case Internal_State is
    when Init | Coasting | Passive => return Inactive;
    when Cruising => return Constant_Speed;
    when Accelerating => return Constant_Acceleration;
    end case;
  end;

  function Speed return Float is
  begin
    return Target_Speed;
  end;

  procedure Set is
  begin
    State_Transition (Set);
  end;

  etc. for the other interrupts

  procedure State_Transition (E : Event_Type) is
  Speed : Float;
  begin
    case Internal_State is
    when Init =>
      case E is
      when Unset =>
        Speed := Driveshaft.Speed;
        if Speed >= Minimum_Speed then
          Target_Speed := Speed;
          Internal_State := Cruising;
        end if;
      when others => null;          -- All other events are
                                     -- ignored
    end case;
  end;

```

```

when Coasting =>
  case E is
  when Unset =>
    Speed := Driveshaft.Speed;
    if Speed >= Minimum_Speed then
      Target_Speed := Speed;
      Internal_State := Cruising;
    else
      Internal_State := Passive;
    end if;
  when Brake_or_Clutch =>
    Internal_State := Passive;
  when others => null;
  end case;
when Cruising =>
  case E is
  when Set =>
    Internal_State := Coasting;
  when Resume =>
    Internal_State := Accelerating;
  when Brake_or_Clutch =>
    Internal_State := Passive;
  when others => null;
  end case;
when Accelerating =>
  case E is
  when Unresume =>
    Target_Speed := Driveshaft.Speed;
    Internal_State := Cruising;
  when Brake_or_Clutch =>
    Internal_State := Passive;
  when others => null;
  end case;
when Passive =>
  case E is
  when Unresume =>
    if Driveshaft.Speed >= Minimum_Speed then
      Internal_State := Cruising;
    end if;
  when others => null;
  end case;
end case;
end State_Transition;
end Cruise_FSM;

```

2.2.1.2 The activity task Throttle

An outline of the periodic activity task Throttle is shown below. The throttle-control activity executes in Constant_Speed and in Constant_Acceleration. Throttle includes the control law that determines the pull on the accelerator wire. Although the program excerpt below suggests that

the pull depends only on the difference between target speed and current speed, the control law can be as complex as necessary and include an integral component.

```

task body Throttle is
  . . .
begin
  loop
    case Cruise_FSM.State is
    when Constant_Acceleration =>
      <Apply constant pull>
    when Constant_Speed =>
      Difference :=
        Cruise_FSM.Speed- Driveshaft.Speed;
      <Compute pull based on Difference, etc.>
    when others => <Release accelerator wire>
    end case;
    delay ....;
  end loop;
end Throttle;

```

To avoid Throttle's busy wait for the state to be either Constant_Acceleration or Constant_Speed, Cruise_FSM can provide an entry that serves as a parking place for Throttle:

```

entry Hold_Throttle when State = Constant_Acceleration
  or else State = Constant_Speed is
begin
  null;
end;

```

Throttle would then change as follows:

```

begin
  loop
    Cruise_FSM.Hold_Throttle;
    loop
      case Cruise_FSM.State is
      when Constant_Acceleration =>
        <Apply constant pull>
      when Constant_Speed =>
        Difference :=
          Cruise_FSM.Speed - Driveshaft.Speed;
        <Compute pull based on Difference, etc.>
      when others =>
        <Release accelerator wire>
        exit;
      end case;
      delay ....;
    end loop;
  end loop;
end Throttle;

```

In both the above solutions, the delay in Throttle affects the response time to events that turn cruising off, such as braking. How faster reaction to these events can be achieved is described in 2.4.

2.2.2 Generalization

The solution where an activity has its own task extends to the case where there are multiple activities. If the activities do not overlap in time, they can be implemented in a single task, although separating them may be preferable if the activities are unrelated. Activities that overlap in time need separate tasks.

Design methods based on real-time structured analysis typically only allow an activity to be enabled and disabled [5]. But it is quite reasonable that an activity occasionally needs to continue with a mere change of parameters. The cruise control example illustrates the solution in this case, which is to let the activity periodically query the parameters by calling functions such as `Cruise_FSM.State`. The case where it is insufficient to let this happen on a periodic basis, and a change of parameters must have a more immediate effect is discussed in section 2.4.

2.3 State-machine and activities combined in one task

If each activity is limited to a single state, it may be advantageous to combine the state machine and the activities in a task. This is illustrated here by means of the software in a bicycle odometer.

2.3.1 Example: The odometer problem

A particular bicycle odometer has four states: Distance, Speed, Mileage and Time. In each state, it repeatedly displays one quantity: distance traveled, speed, total mileage or elapsed time, with a frequency particular to each quantity [15].

The odometer has two buttons, A and B. By pressing A, the biker changes the odometer's state cyclically from Distance to Speed, etc., back to Distance. Pressing B in the Distance or Time state resets a reference distance and a reference time, respectively. B has no effect in the other states. Internally, an enumerated type is used to indicate which button was pressed:

```
type Button_Type is (A, B);
```

A protected unit `Buttons` serves as the interrupt handler for the interrupts from A and B:

```
protected Buttons is  
    entry Interrupt (Button : out Button_Type);  
    -- Interrupt is called by the Odometer task  
private  
    procedure A_Int;  
    procedure B_Int;  
    pragma Attach_Handler (A_Int, ....);  
    pragma Attach_Handler (B_Int, ....);
```

```

    Occurred : Boolean := False;
    Which_Occurred : Button_Type;
end;

protected body Buttons is
    entry Interrupt (Button : out Button_Type)
        when Occurred is
    begin
        Button := Which_Occurred;
        Occurred := False;
    end;
    procedure A_Int is
    begin
        Occurred := True;
        Which_Occurred := A;
    end;
    procedure B_Int is
    begin
        Occurred := True;
        Which_Occurred := B;
    end;
end;
end;

```

The state machine and the activities are combined in a task, Odometer, as shown below. Odometer calls Wheel, which is a protected object that receives an interrupt every time the wheel has completed a revolution. The protected functions Wheel.Distance and Wheel.Speed return the current distance and speed, respectively.

```

task body Odometer is
    Ref_Dist: Distance_Type := Wheel.Distance;
    Ref_Time, Next : Time := Clock;
    Button : Button_Type;
begin
    loop
        -- Distance state
        Next := Clock + Dist_Delay;
        D: loop
            select Buttons.Interrupt (Button);
                case Button is
                    when A => exit; -- Exit to Speed state
                    when B => Ref_Dist := Wheel.Distance;
                        -- Reset the reference distance
                end case;
            or delay until Next;
                Display (Wheel.Distance - Ref_Dist);
                Next := Next + Dist_Delay;
            end select;
        end loop;
        -- Speed state
        Next := Clock + Speed_Delay;
    end loop;
end;

```

```

S: loop
    select Buttons.Interrupt (Button);
        case Button is
            when A => exit; -- Exit to Mileage state
            when B => null; -- Ignore B button
        end case;
    or delay until Next;
        Display (Wheel.Speed);
        Next := Next + Speed_Delay;
    end select;
end loop;
-- etc. for the Mileage and Time states
end loop;
end Odometer;

```

Implementing the state machine as a task allows us to represent the state implicitly as in the above solution. *Implicit state representation* means that each state corresponds to a portion of the program text [13]. For example, while the odometer is in the state Distance, control remains within the loop D. When a state transition occurs, the exit statement transfers control to the construct related to the Speed state. That way, testing a state variable becomes unnecessary.

Clearly, it is equally possible to implement the odometer program with an explicit state variable, State, say, of an enumerated type that takes the values Distance, Speed, etc. Such an implementation would contain a loop with a case statement over the different values of State.

2.4 Aperiodic communication between activity and state machine

In some applications, it is insufficient to let the activity query the current state periodically. One example is when the periodicity itself must be changed. Assume that an activity takes action every 60 seconds. If the period is changed to 2 seconds, the 60-second suspension must be broken. For a solution to this problem assume that the protected unit FSM represents a state machine and let it provide an entry, `New_Event`, with a barrier that is set True when the event that affects the activity occurs. Then the following timed entry call in the activity task implements a delay that is curtailed when the event occurs:

```

select
    FSM.New_Event;
or
    delay <full-length delay>;
end select;

```

Asynchronous transfer of control provides a solution in case a lengthy computation must be interrupted. The activity uses the call `New_Event` as the trigger in an asynchronous select statement as follows:

```

select
    FSM.New_Event;
then abort
    <computation>;
    delay until <activation time>;
end select;

```

(The delay statement is intended to cover any time remaining after the end of the computation and before the next periodic activation.)

2.4.1 Example: The remote temperature sensor problem

In one solution of the remote temperature sensor (RTS) problem [10, 12, 13, 15, 17], there are 16 periodic tasks of the type Furnace. Each samples the temperature of one furnace and sends the reading in a data packet to a host computer. The sampling is done by sending a command to an A/D converter connected to a thermo-couple for each furnace. The A/D converter responds with an interrupt when the temperature is available. Each task has an individual sampling period, which can be changed by a message from the host. Each sampler makes a delay commensurate with its current frequency. It must be possible to break the delay in case a message arrives that considerably increases the frequency.

In this example, the state machine is reduced to a protected unit Sensor that is updated by messages from the host. (This is a degenerate state machine with a single state of importance but multiple events.) Sensor has an entry family New_Period (Furnace_Type) (Period : **out** Duration) with one instance for each Furnace task. Its barrier is set True whenever a message from the host has arrived for a given furnace.

Each Furnace task calls its New_Period entry initially to wait for its first activation, and then in a timed entry call as shown below. (My_Furnace is a discriminant that gives each Furnace task a unique number.)

```

task body Furnace is
Next : Time;                                -- Time for next sampling
Period: Duration;
begin
    Sensor.New_Period (My_Furnace) (Period);  -- Wait for initial
                                                -- activation
    Next := Clock;
    loop
        <request sample from A/D converter>
        <wait for A/D converter interrupt>
        <send data packet to host>
        Next := Next + Period;
        select
            Sensor.New_Period (My_Furnace) (Period);
        or
            delay until Next;
        end select;
    end loop;
end Furnace;

```

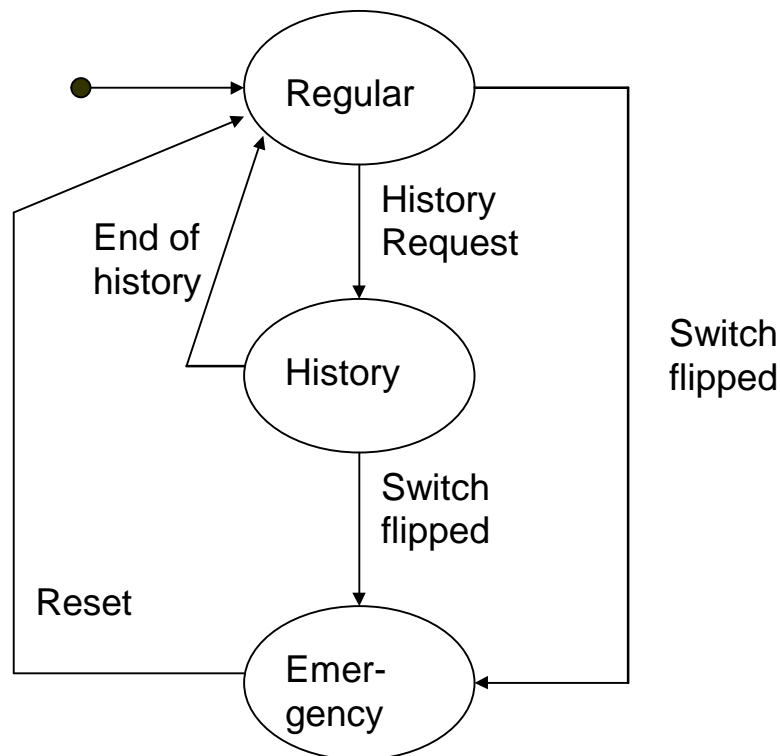
2.5 Suppression of activities in certain states

Some activities must continue at a given periodicity independent of state, but take effect only in certain states. In the buoy example below, an activity task produces output every 60 seconds, but the output is sometimes suppressed. This is accomplished by letting the activity task call a protected procedure that either transmits or suppresses the output depending on the current state.

2.5.1 Example: The buoy problem

A number of free-floating buoys provide navigation and weather data to air and ship traffic at sea [2, 11, 13, 15]. Under normal circumstances, each buoy makes a *regular transmission* of current wind, temperature and location information every minute on the minute. A passing vessel may request a *history transmission* of all data collected over the last 24 hours. This transmission takes several minutes and takes precedence over the regular transmission. Furthermore, a sailor in distress who physically reaches the buoy may engage an emergency switch and initiate an *emergency transmission* that continues until it is explicitly reset. The emergency transmission takes precedence over all other transmissions. The state diagram in Fig. 2 shows the three transmission states Regular, History and Emergency, and the transitions just described.

Fig 2. The buoy state machine



In a solution to this problem, the state machine is represented as a protected unit, Reporter. There are three activity tasks, Regular_Task, History_Task and SOS_Task, as follows:

Regular_Task prepares a regular transmission every 60 seconds, then calls Reporter.Regular_Msg, which sends the message if the buoy is in the proper state.

History_Task blocks on the entry Reporter.Hold_History waiting for a history request, then repeatedly calls Reporter.History_Msg with history data messages until the relevant history information has been exhausted. It then again blocks on Hold_History. The entry History_Msg ensures that history messages are transmitted while the buoy is in the proper state and are suppressed in the Emergency state.

SOS_Task repeatedly produces an SOS message and calls Reporter.SOS_Msg, which has a barrier that lets a call through only in the Emergency state.

The text of Reporter is as follows:

```
type State_Type is (Regular, History, Emergency);

protected Reporter is
  procedure Regular_Msg (...); -- Regular, periodic message
  procedure History_Request;   -- Request for history data
  entry History_Msg (...);    -- Send history message
  entry Hold_History;        -- Parking place for
                               -- History_Task
  entry SOS_Msg (...);       -- Send SOS message
                               -- if appropriate
  procedure Reset;           -- Emergency reset

private
  procedure Switch;           -- Emergency switch flipped
  pragma Attach_Handler (Switch, ...);
  State : State_Type := Regular;
end Reporter;

protected body Reporter is
  procedure Regular_Msg (...) is
  begin
    if State = Regular then <send message > end if;
  end;

  procedure History_Request is
  begin
    if State = Regular then State := History; end if;
  end;

  entry History_Msg (...) when True is
  begin
    if <end of messages> then
      if State = History then
        State := Regular;
      end if;
    end if;
  end;

```

```

        requeue Hold_History;
    elsif State = History then
        <send message>;
    end if;
end;

entry Hold_History (....) when State = History is
begin
    null;
end;

procedure Switch is
begin
    State := Emergency;
end;

entry SOS_Msg (....) when State = Emergency is
begin
    <send message>
end;

procedure Reset is
begin
    if State = Emergency then State := Regular; end if;
end;
end Reporter;

```

The advantage of this solution with separate activity tasks is that the regular broadcast stays on the same schedule from its original starting point independent of history and emergency broadcasts, even though actual transmissions are sometimes suppressed. If this is not a requirement, the state-machine representation and the activities can be combined in the manner of the odometer example in 2.3.1. In that simpler solution [13, 15], the starting point for the regular transmission schedule is reset at the end of each history and emergency transmission.

3. Timing events

A *timing event* is defined by a certain period of time having passed since another event, as for example, "s seconds have passed since state X was entered". A timing event is often conditional in that it serves to time out the wait for some other event; if that event occurs, the timing event is not supposed to happen.

Ada 95 is similar to most other real-time languages in that a timing event is created by a task executing a **delay** statement. The timed entry call, selective accept and asynchronous select statements provide conditional timing events that are automatically canceled if the timed-out event occurs. For these reasons, a state machine with timing events is conveniently implemented as a task as shown for the odometer problem in 2.3.1.

Unfortunately, a solution with a task representing the state machine is suitable only if it is known (or anticipated) at the outset that there is a timing event. But perhaps a state machine was originally implemented as a protected object and must later be modified to include a timing

event. Re-implementing it as a task may then be a disproportionately major change. An alternative solution is to let it remain a protected object and implement the timing event by means of a *timer task*. Such a timer task is quite similar to the activity tasks shown earlier.

3.1 Example: Automobile window elevator

In an automatic window elevator for a car, the driver lowers the window by holding down a lever on the car door [13, 15]. The window elevator control then enters a state *Opening*. Three events may occur in *Opening*:

Release: The driver releases the lever causing the window to stop at its current position
Bottom: The window is fully opened
T seconds: This timing event signifies that the lever has been held down for T seconds and causes the control to move to the state *Automatic_Opening*

In state *Automatic_Opening*, the window continues to open even if the lever is released.

The protected unit *Window_Control* has a variable *State* that takes values such as *Opening*, *Automatic_Opening* and *Stopped*. The following are the interrupt handling procedures and entries that pertain to these states.

```
procedure Release is
begin
    if State = Opening then
        <stop window motor>;
        State := Stopped;
    end if;
end;

procedure Bottom is
begin
    <stop window motor>;
    State := Stopped;
end;

procedure Time_Out is
begin
    if State = Opening then
        State := Automatic_Opening;
    end if;
end;

entry Hold_Timer when State = Opening is
begin
    null;
end;
```

```

entry Stop_Timer when State /= Opening is
begin
    null;
end;

```

(This state-machine implementation is an alternative to that used for Cruise_FSM in 2.2.1.1. There is no procedure State_Transition with nested case statements over states and events. Instead, the procedure associated with each event completely defines its effect in each state.)

Here is the timer task:

```

task body Timer is
begin
    loop
        Window_Control.Hold_Timer;           -- Wait for right state
        select
            Window_Control.Stop_Timer;      -- Accepted when timer is
                                           -- no longer needed
        or    delay T;
            Window_Control.Time_Out;       -- Create time-out event
        end select;
    end loop;
end;

```

Even though the Time_Out call can be ignored in states where it does not apply, Timer must be canceled when no longer needed. It is insufficient to let it reach the end of its delay and then decide whether a time-out event is warranted, since, by then, the state machine may be back in a state where a time-out does apply. For this reason, the timer must be canceled immediately upon exit from Opening so it can be restarted whenever the state is reentered. If the timer were used in additional states, the barriers for Hold_Timer and Stop_Timer must be modified.

4. Conclusion

Providing sets of solutions to common design problems is a way to encourage architectural reuse, which is in line with the current interest in architectures and patterns in software engineering. Many designs involving a state machine can be implemented in a real-time concurrent environment based on a small number of ready-made designs. The solution set given here covers a number of basic but not always trivial cases of interaction between state machine and activities.

References

- [1] M. Awad, J. Kuusela, and J. Ziegler. Object-oriented technology for real-time systems, Prentice-Hall 1996
- [2] G. Booch. *Object-oriented development*. IEEE TSE, 12(2), Feb. 1986, 211-221.

- [3] J. R. Carter and B. Sandén. *Practical uses of Ada-95 concurrency features*. IEEE Concurrency 6:4 (Oct/Dec1998), 47-56.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley 1995
- [5] H. Gomaa. Software Design Methods for Concurrent and Real-time Systems. Addison-Wesley, 1993.
- [6] D. Harel. *Statecharts, a visual formalism for complex systems*, Science of Computer Programming, Aug 1986, 231 - 274
- [7] D. Hatley and I. A. Pirbhai. Strategies for Real-Time System Specification. Dorset House, 1987.
- [8] D.-W. Jones. *Cruising with Ada*, Embedded Systems Programming 7:11 (Nov. 1994), 18-44
- [9] S. J. Mellor and P. T. Ward. Structured Development for Real-Time Software. Vol. III Implementation Modeling Techniques. Yourdon Press, 1986.
- [10] K. W. Nielsen and K. Shumate. *Designing large real-time systems with Ada*. CACM, 30:8 (Aug. 1987) 695-715. Corrected in CACM 30:12 (Dec. 1987) 1073
- [11] B. I. Sandén. *The case for eclectic design of real-time software*, IEEE TSE 15 (March 1989), 360 - 362.
- [12] B. I. Sandén. *Entity-life modeling and structured analysis in real-time software design - a comparison*. CACM, 32:12 (Dec. 1989) 1458-1466.
- [13] B. I. Sandén. Software Systems Construction with Examples in Ada. Prentice-Hall, 1994.
- [14] B. I. Sandén, *Using tasks to capture problem concurrency*. Ada User Journal 17, 1 (March 1996), 25-36.
- [15] B. I. Sandén. A course in real-time software design based on Ada 95, Available from the ASSET repository as ASSET_A_825, 1996, <http://artemis.coloradotech.edu/~bsanden/DISA/rtcouse.html>
- [16] B. I. Sandén. *Modeling concurrent software*. IEEE Software, Sept. 1997, 93-100.
- [17] B. I. Sandén, *Concurrent design patterns for resource sharing*. Proc. TRI-Ada, St. Louis, MO, Nov. 1997, 173-183.
- [18] M. Shaw. *Comparing architectural design styles*, IEEE Software, November 1995, 27-41