Programming Simple Reactive Systems in Ada: Premature Program Termination

A.J. Wellings, A. Burns, A.L.C. Cavalcanti and N.K. Singh
Department of Computer Science, University of York
Heslington, York YO10 5GH, UK
(andy.wellings, alan.burns, ana.cavalcanti, neeraj.singh)@york.ac.uk

Abstract

Reactive systems are systems that respond to stimuli from the environment within the time constraints imposed by the environment. This paper identifies an ease-of-use issue with Ada for developing small reactive systems. The problem is that Ada defines program termination solely in terms of whether all tasks have terminated. There are some advantages in adopting a purely interrupt-driven design in the implementation of small reactive system. With such programs, there are no tasks other than the environment task, which typically terminates when it finishes executing the main program. We argue that this is not the expected behaviour. To avoid this unexpected premature program termination, this paper proposes simple changes to the program termination conditions in the language so that the environment task of an active partition terminates when (1) all its dependent tasks have terminated, (2) the partition has no active timing events, and *(3) no handlers are attached to interrupts that are to be serviced by the partition. However, this would be a non-backward compatible change, and some programs that currently terminate would not terminate with the new rules if they still have attached interrupt handlers or outstanding timing events.*

1 Introduction

Reactive systems respond to stimuli from the environment within the time constraints imposed by the environment [3]. Hence they are typically event-triggered systems where the absence of an expected event can also be considered as an event itself (a timeout event). In this short paper, we illustrate an ease-of-use problem when programming small reactive systems in Ada.

Simple designs of a reactive system are based on (centralised) state information that is manipulated by atomic operations. In these cases, the kernel of the system is typically implemented as a deterministic automaton. Given a set of available inputs (or events), the automaton selects a transition that can be performed and executes the associated code implementing an atomic operation; it is then ready for its next reaction – this is illustrated in Figure 1. More complex reactive systems may be structured hierarchically and allow parallel or distributed execution of operations. Esterel [2] is a language that targets the implementation of potentially complex reactive systems. Here, however, we are concerned with single processor systems that are usually embedded. Our motivating example has been that of a cardiac pacemaker [4].

2 Reactive Systems and Ada

In embedded Ada applications, interaction with the environment is via interrupt handling and the various timeout mechanisms supported by the language. With systems for which environmental events must be polled, Ada supports abstractions that allow periodic tasks to be implemented, or periodic timing events to be generated. Here, we propose to use a combination of the Ada interrupt handling facility and the *Ada.Real_Time.Timing_Events* package to implement deterministic automata, which in turn can be used to support simple reactive applications. *Hence, we are interested in applications for which there are no tasks, other than the environment task that executes the main program.*

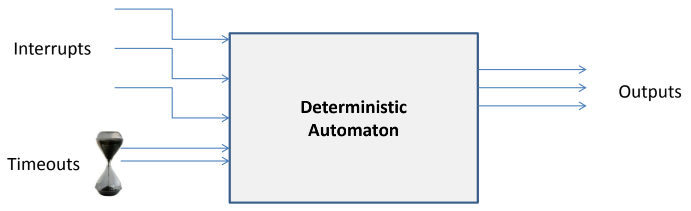

Figure 1. A Simple Reactive System

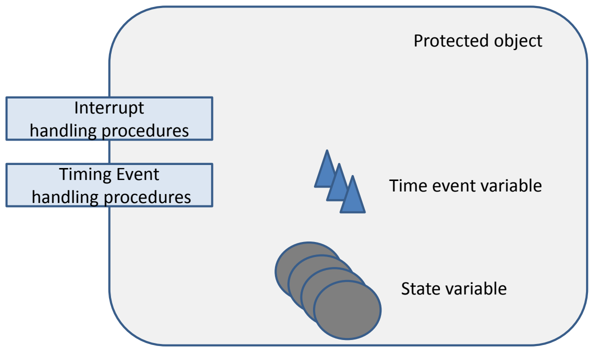
Protected object
Interrupt handling procedures
Timing Event handling procedures
Time event variable
State variable

Figure 2. The Implementation of a Simple Reactive System in Ada

Our proposed approach is to encapsulate all the procedures for handling interrupts and timing events within the *same* library-level protected object, as illustrated in Figure 2. The advantages of this approach (for small systems) are:

- All event-handling procedures execute atomically with respect to one another – this is guaranteed by the language's semantics for protected objects.
- The worst-case execution time needed to respond to each event is well defined – there can be no preemptions (hence, no cache refills during execution, etc) and the code to be executed is clearly identifiable from the procedure that handles the event.
- The response time for each event is predictable – there can be no preemptions and protected procedures cannot self suspend for any reason.

There are, perhaps, two disadvantages of this implementation model.

1. Interrupts that are unable to be delivered are assumed to be queued by the hardware – this is usually the case and certainly required for a reactive system that must keep up with the environment.
2. It is not possible to control the order of handling when multiple events are available – this is inevitable when interrupts come in from various sources; typically the hardware priority of the device will determine the order in which the processor processes the outstanding interrupts. Of course, if events are being polled for, the order of polling will allow some control. More significantly, there is no order specified for the servicing of timing events that occur at the same time.

These disadvantages may not be relevant for (many) applications. In the next section, we consider a significant case study where these issues do not pose a serious problem.

3 An Illustrative Example

In this section we summarise an Ada implementation of our motivating example. A full description of the Cardiac Pacemaker is given in [4] along with details of the implementation in Safety Critical Java and Ravenscar Ada.

Time Intervals	Purpose	Time in milliseconds
Length of a P wave (T_P)	Time during which intrinsic atrium activity must be sensed	110
Duration of pulse (T_{pulse})	the time for which the pulse current must be maintained	1
Length of a QRS complex (T_{QRS})	Time during which intrinsic ventricular activity must be sensed	100
Atrioventricular interval (AVI)	Provides time for ventricle to fill following an atrial contraction	150
Ventriculoatrial interval (VAI)	Ensures an atrial pulse following a ventricular pulse	850
Postventricular atrial refractory (PVARP)	Ensures atrium doesn't falsely sense ventricular activity	350
Mode Switching Interval (MSI)	Time between two atrial events used to change modes	500

Table 1. Example Timing Intervals in a Single Heart Beat

A pacemaker system is a small electronic device that helps the heart to maintain a regular beat. The conventional pacemakers serve two major functions: *pacing* and *sensing*. The pacemaker's actuator paces by the delivery of a short, intense electrical pulse into the heart. The pacemaker sensor uses the same electrode to detect the intrinsic activity of the heart. So, the pacemaker's pacing and sensing activities are dependent on the behavior of the heart.

Essentially, the pacemaker monitors (senses) the intrinsic activity of the heart and in the absence of such activity forces the heart to beat by the delivery of the electric current (pacing). The sensing and pacing activities can be performed in both chambers of the heart (atrial and ventricle). The control requirements can be complex, and it is beyond the scope of this paper to consider them in detail (see [4]).

Fig. 3 depicts the scenarios for sensing and pacing activities. The Ventriculoatrial Interval (VAI) is the maximum time the pacemaker should wait after sensing ventricle activity (either intrinsic or paced) for some indication of intrinsic activity in the atrium. If none is present, the pacemaker should pace in the atrial chamber. The Atrioventricular Interval (AVI) is the maximum time the pacemaker should wait after sensing atrial activity (either intrinsic or paced) for some indication of intrinsic activity in the ventricles. If none is present then the pacemaker should pace in the ventricle chamber. After every pace in the ventricle chamber, there is some sensed activity in the atrial, but this is not true intrinsic heart activity and should be ignored. The Postventricular Atrial Refractory Period (PVARP) indicates the length of time that such activity should be ignored. Sensed atrial activity is called a P wave, and sensed ventricular activity is called a QRS complex. A T wave follows a QRS complex and represents the recovery of the ventricles. A P wave is sensed when the amplitude of the signal is greater than a threshold P_{th} for T_p time units. Similarly, a QRS complex is detected when the amplitude of the signal is greater than a threshold QRS_{th} for T_{QRS} time units. For safety, it is imperative to ensure that pulsing does not occur during a T wave.

There are four possible scenarios for pacing and sensing activities, which are given in Fig. 3.

- Scenario A – shows a situation in which the pacemaker paces after standard time intervals (VAI and AVI) in both chambers. This is the reaction when no intrinsic heart activity is detected.
- Scenario B – shows a situation in which the pacemaker paces in the atrial chamber after VAI, while the ventricular pacing is inhibited due to a sensing of intrinsic activity from the ventricle.
- Scenario C – shows a situation in which intrinsic atria activity is sensed, pacing inhibited in the atrial chamber but occurs in the ventricular chamber after AVI (due to a lack of intrinsic ventricular activity).
- Scenario D – represents the case where both pacing activities are inhibited due to a sensing of intrinsic activities in both chambers.

Table 1 summarises the main timing requirements for a particular pacemaker, and Figure 4 illustrates the basic requirements that must be met. The required pacing activities are not regular enough to be controlled by a periodic activity. They are essentially aperiodic and time-triggered depending on the presence or absence of intrinsic heart activity. There are no interrupts generated other than that needed to support Ada's timing events.

There are clearly several software architectures that could be adopted. Here we use Ada's timing events to control both the sensing and pacing activities as this eliminates the needs for tasks and, therefore, reduces the size of the Ada run-time

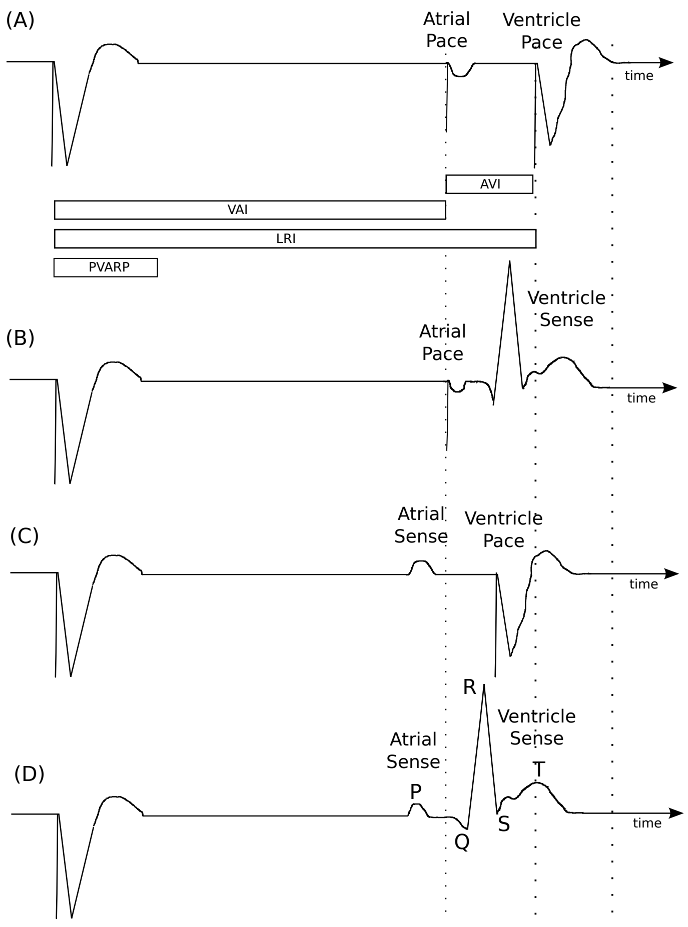

Figure 3. Example Pacing Scenarios

Time goes left to right, and a flat line indicates no heart activity. A spike above the lines indicates intrinsic activity and a spike below the line indicates activity as a result of the action of the pacemaker. A rounded spike indicates activity in the atrial and a sharp spike indicates activity in the ventricle.

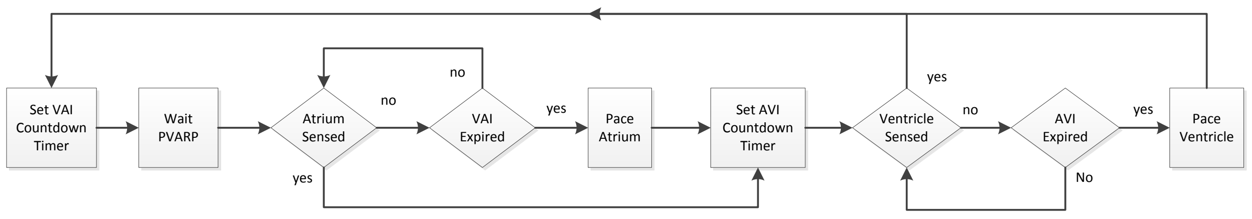

Figure 4. The Required Pacing Cycle

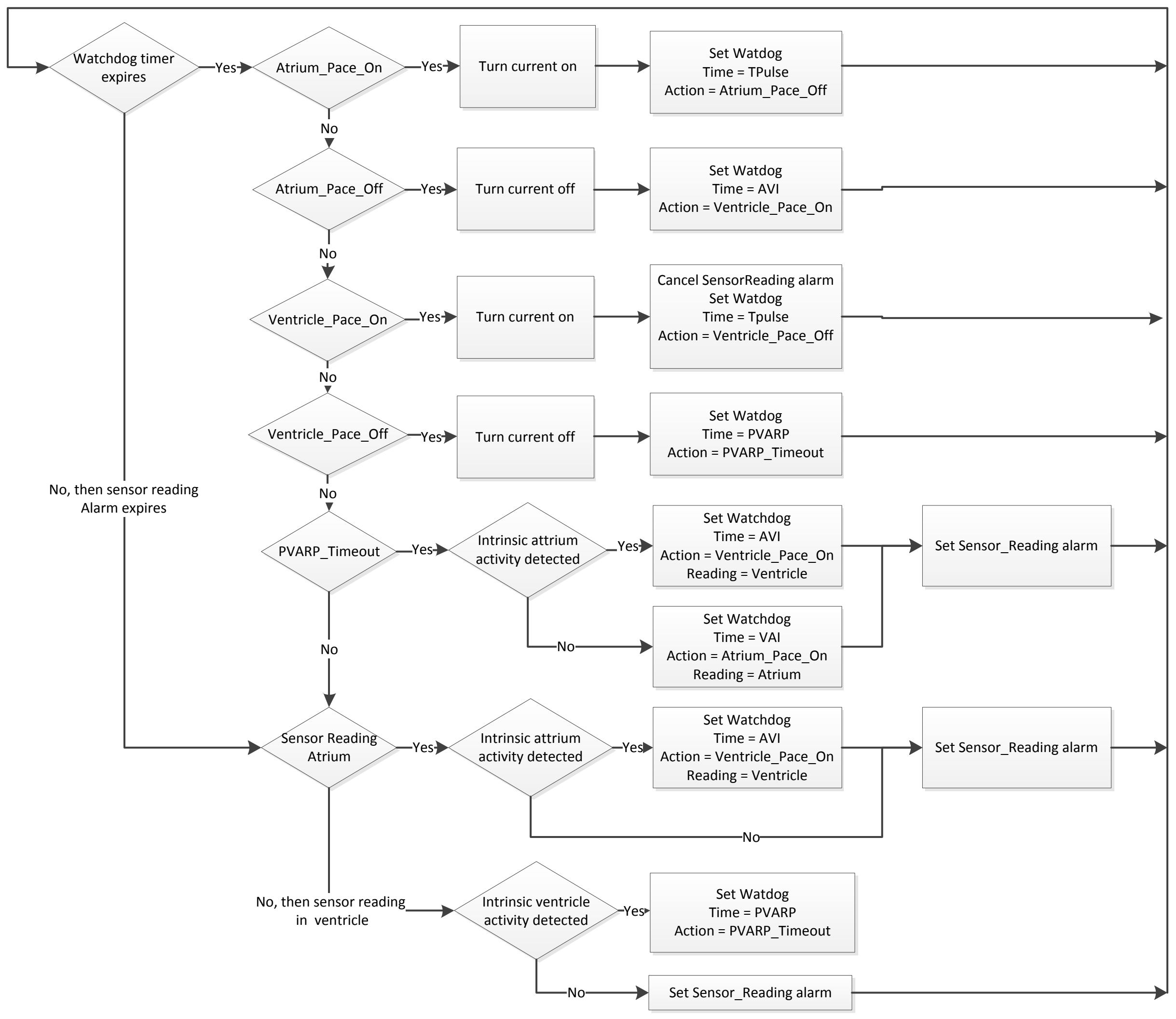

Figure 5. Cardiac Pacemaker Architecture in Ada

support needed. There are two main timing events: the first (`Watchdog`) is used to detect the absence of intrinsics activities and to control the pacing current, and the second (`Sensor_Readings`) is used to initiate the reading of sensors. A single protected object (`Timer`) is used for encapsulating the handlers for these timing events.

The details of the Ada approach are shown in Figure 5. The algorithm follows closely that given in Figure 4 which informally defines the requirements. The protected object code is given below. The full code for the application be found at `http://www.cs.york.ac.uk/circus/hijac/case.html`.

```
with Ada.Real_Time.Timing_Events; use Ada.Real_Time.Timing_Events;
with System; use System;
package Timers is
  type Sensor is (Atrium, Ventricle);
  protected Timer is
    pragma Priority(Priority'Last);
    procedure Atrium_Pace_On(E : in out Timing_Event);
    procedure Atrium_Pace_Off(E : in out Timing_Event);
    procedure Ventricle_Pace_On(E : in out Timing_Event);
    procedure Ventricle_Pace_Off(E : in out Timing_Event);
    procedure PVARP_Countdown(E : in out Timing_Event);
    procedure Sensor_Read(E : in out Timing_Event);
  private
    Reading : Sensor := Atrium;
    IntrinsicV_Sensed : Boolean := False;
  end Timer;

  General_Timeouts : Timing_Event;
  Sensor_Readings : Timing_Event;
end Timers;


-- various with and use clauses

package body Timers is

  protected body Timer is
    procedure Atrium_Pace_On(E : in out Timing_Event) is
    begin
      Pace_A_On; -- turns pace current on
      Set_Handler(General_Timeouts, Clock+Pulse_Duration, Atrium_Pace_Off'Access);
    end;

    procedure Atrium_Pace_Off(E : in out Timing_Event) is
    begin
      Pace_A_Off;  -- turns pace current off
      Set_Handler(General_Timeouts, Clock+AVI, Ventricle_Pace_On'Access);
      Reading := Ventricle;
    end;

    procedure Ventricle_Pace_On(E  : in out Timing_Event) is
      Set : Boolean;
    begin
      Pace_V_On; -- turns pace current on
      if IntrinsicV_Sensed then
        IntrinsicV_Sensed := False;
      end if;
      Cancel_Handler(Sensor_Readings, Set);
      pragma assert(Set);
      Set_Handler(General_Timeouts, Clock+Pulse_Duration , Ventricle_Pace_Off'Access);
    end;

    procedure Ventricle_Pace_Off(E  : in out Timing_Event) is
    begin
      Pace_V_Off; -- turns pace current off
      Set_Handler(General_Timeouts, Clock+PVARP , PVARP_Countdown'Access);
      Reading := Atrium;
    end;

    procedure PVARP_Countdown(E : in out Timing_Event) is
      res : Float;
    begin
      res := Read_Atrium_Data; -- measure intrinsic activity
      if res > 0.3 then
        -- Intrinsic activity sensed in atrium;
        Set_Handler(General_Timeouts, Clock+AVI, Ventricle_Pace_On'Access);
        Reading := Ventricle;
      else
        Reading := Atrium;
        Set_Handler(General_Timeouts, Clock+VAI, Atrium_Pace_On'Access);
      end if;
      Set_Handler(Sensor_Readings, Sensor_Period, Sensor_Read'Access);
    end;


    procedure Sensor_Read(E : in out Timing_Event) is
      res : Float;
    begin
```

```
      if Reading = Atrium then
        res := Read_Atrium_Data;
        if res > 0.3 then
           -- Intrinsic activity sensed in Atrium
          Set_Handler(General_Timeouts, Clock+AVI, Ventricle_Pace_On'Access);
          Reading := Ventricle;
        end if;
        Set_Handler(Timers.Sensor_Readings, Sensor_Period, Timer.Sensor_Read'Access);
      else -- reading ventricle
        res := Read_Ventricle_Data;
        if res >= 0.9 then
          -- Intrinsic activity sensed in ventricle;
          Set_Handler(General_Timeouts, Clock+PVARP , PVARP_Countdown'Access);
          Reading := Atrium;
          IntrinsicV_Sensed := True;
        else
          IntrinsicV_Sensed := False;
          Set_Handler(Timers.Sensor_Readings, Sensor_Period, Timer.Sensor_Read'Access);
        end if;
      end if;
    end Sensor_Read;
  end Timer;
end Timers;
```

The main program sets up the first timing event. From that point on, every timing event handler will set up at least one other timing event.

```
-- with clauses omitted
procedure Pacemaker is  --DDDR
begin
  Set_Handler(Timers.Sensor_Readings, Clock+PVARP, Timer.Sensor_Read'Access); -- sets initial event
end Pacemaker;
```

The resulting program design is efficient, with handlers only executing when control is needed. (We observe, however, that there is a problem with this solution, which we discuss in the next section.) The handlers for timing events in Ada are called from the Ada run-time clock interrupt handling code. Hence, no Ada tasks are actually required. As all code is run at interrupt-level, it is imperative to keep the handling code as simple and as short as possible so that the computation can be completed before another timing event needs to be set. The `Atrium_Pace_On`, `Atrium_Pace_Off`, `Ventricle_Pace_Off` and `PVARP_Timeout` handlers are mutually exclusive events. Hence the requirements are for:

- `Atrium_Pace_On` to be completed before the pulse duration (Tpulse) has expired (1 millisecond – see Table 1) when `Atrium_Pace_Off` needs to be called. The code in the handler is simply one actuator operation and the setting of one timing event.
- `Atrium_Pace_Off` to be completed within safety margins for the the pulse duration (and before the AVI (150 milliseconds)). The code in the handler is simply one actuator operation and the setting of one timing event.
- `Ventricle_Pace_On` to be completed before the pulse duration (Tpulse) has expired (1 millisecond – see Table 1) when `Ventricle_Pace_Off` needs to be called. The code in the handler is simply one actuator operation and the setting of one timing event and the canceling of another.
- `Ventricle_Pace_Off` to be completed within safety margins for the the pulse duration (and before the PVARP duration (350 milliseconds)). The code in the handler is simply one actuator operation and the setting of one timing event.
- `PVARP_Timeout` to be completed before the VAI duration (850 milliseconds). The maximum code in the handler is simply one sensor operation and the setting of two timing event.

The sensor reading handlers are similarly simple, mainly consisting of one sensor reading operation and at most two settings of timing events. The asynchronous relationship between the watchdog and sensor-reading timing events means that it is possible for one event to want to fire whilst the other event is being handled. Hence, the response time of each handler must include an interference time equal to the maximum handling timing of the other event.

4 The Premature Termination of Simple Reactive Programs in Ada

The implementation of our motivating example, as presented so far, does not work. This is because the program terminates immediately the main program finishes executing; there are no tasks to keep the program alive. Hence, we had to add an delay in the main procedure to keep the program alive, as illustrated below. (The alternative is to introduce a low priority idle task, or to call `Suspend_Until_True` on a suspension object.)

```
-- with clauses omitted
procedure Pacemaker is  --DDDR
begin
  Set_Handler(Timers.Sensor_Readings, Clock+PVARP, Timer.Sensor_Read'Access); -- sets initial event

  while True loop
      null;
      delay until clock + Milliseconds(50000);
  end loop;
end Pacemaker;
```

This is clearly not very elegant. The reason is that the Ada programming language specification defines the following.

> The execution of a program consists of the execution of a set of partitions. Further details are implementation defined. The execution of a partition starts with the execution of its environment task, ends when the environment task terminates, and includes the executions of all tasks of the partition. The execution of the (implicit) task body of the environment task acts as a master for all other tasks created as part of the execution of the partition. When the environment task completes (normally or abnormally), it waits for the termination of all such tasks, and then finalizes any remaining objects of the partition. (Ada Reference Manual Section 10.2 Program Execution).

On a single processor system, a program is considered to be a single active partition that terminates when its environment task terminates. Hence, when the environment task finishes executing the main program, it checks to see if there are any active tasks. If there are not, the program terminates.

Modifying Ada Program Termination Rules

In order to ensure that unexpected terminations of programs do not occur, the Ada language specification would need to be changed so that:

> The environment task of an active partition terminates when all its dependent tasks have terminated *and* the partition has no active timing events *and* there are no handlers attached to interrupts that are to be serviced by the partition.

This would require all interrupt handlers to be attached dynamically rather than via the static use of the `Attach_Handler` pragma. Alternatively, the language could be modified to allow dynamic detaching of a static handler. For the Ravenscar profile, where programs are not meant to terminate and additional pragma could be used to indicate that there are no tasks in the program.

With this new rule, our implementation described in the previous section would be correct. However, such a change would not formally be backward compatible. Some programs that currently terminate would not terminate with the new rules if they still have outstanding timing events or attached interrupt handlers. Arguably, a program that terminate with an outstanding timing event has a dormant fault – it is almost the equivalent of mixing the terminate and a delay alternatives in a select statement (only at the program level rather than the task level). For interrupt handling, the current rules generate a race condition between the interrupts being handled and the program terminating. The new rules, however, could lead to the case where handlers are attached but all interrupts have been disabled. This would be equivalent of writing a program that deadlocks.

5 Conclusions

This paper has identified an ease-of-use issue with Ada for developing small reactive systems. The issue is that Ada defines program termination solely in terms of whether all tasks (application and environment) have terminated. There are some advantages in implementing small reactive systems as being interrupt driven – be they timing interrupts or other device interrupts. With such programs, there are no tasks other than the environment task, which typically terminates when it finishes executing the main program. This is not the expected behaviour. While the work-arounds are simple, they are a little inelegant.

To avoid this unexpected premature program termination, it is necessary to change the program termination conditions in the language so that the environment task of an active partition terminates when all its dependent tasks have terminated *and* the partition has no active timing events *and* no handlers that are attached to interrupts that are to be serviced by the partition. It is interesting to note that the initial version of the Real-Time Specification for Java had a similar problem with the way it specified program termination[1]. There, all asynchronous event handlers attached to environment-generated events (*happenings* in the RTSJ terminology) were treated as daemon Java threads. This resulted in purely reactive programs suffering from premature termination.

The paper has also illustrated that for time-driven reactive programs, the order of servicing timing events is undefined when more than one event is due at the same time. Most implementation probably service them in a FIFO order. There may be some merit is allowing a priority to be assigned each event. Also, allowing periodic timing events to be specified would remove the need to continually reset them.

Acknowledgements

The authors are grateful for Johan Nielsen for pointing up the backward incompatibility of our proposals, and for suggesting the use of suspension objects as another work-around.

References

[1] Greg Bollella, James Gosling, Benjamin Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[2] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293 –1304, sept 1991.

[3] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[4] N.K. Singh, A.J. Wellings, and A.L.C. Cavalcanti. The cardiac pacemaker case study and its implementation in Safety-Critical Java and Ravenscar Ada. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems - JTRES 2012.*, 2012.