Implementation of mode changes with the ravenscar profile

Alejandro Alonso, Juan Antonio de la Puente

Depto. Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid,
Ciudad Universitaria s.n.,E-28040 Madrid, Spain
e-mail: aalonso@dit.upm.es

Abstract

The use of concurrency in the development of safety-critical systems has not been recommended for a long time. Recently, a safe Ada tasking subset has been defined, known as the Ravenscar Profile. The imposed restrictions invalidated most of the proposed implementations for dealing with mode changes.

In this paper, it is described an implementation approach for mode changes, that is compliant with the Ravenscar Profile. It relies on the decoupling between jobs and tasks and on centralizing tasks activations in a system manager. The proposed ideas have been tested with the ORK, which is a kernel integrated with the GNAT compiler for developing and executing applications that follow the Ravenscar Profile.

1. Introduction

There are computer systems that have strict safety and hard real-time requirements. These are common in domain areas such as avionics, air traffic control, railway traffic control, nuclear power plants, and many others. Software development standards for safety-critical systems are usually very restrictive on the use of high-level programming languages and operating system constructs. Until recently, concurrency was one of such features that were not allowed in safety-critical systems.

In the 8th International Real-Time Ada Workshop it was defined a restricted tasking profile to be used in high-integrity real-time systems implemented with Ada [1] [2]. This profile has been later improved with minor alterations and some clarifications [3]. This profile is named Ravenscar Profile (RP). It defines a subset of the Ada tasking facilities that can be used to develop safe and efficient real-time systems. Industry has shown a notable interest in this profile, which is an indication of its importance.

The RP has been referenced in the ISO report *Guide for the use of the Ada Programming Language in High Integrity Systems* [4]. This report includes a number of recommendations to use the Ada language in the development of safety-critical systems. The RP is referenced in the section which describes recommendations for using tasking facilities.

The RP allows the implementation of real-time systems based on fixed-priority scheduling [5] [6]. This approach has reached the required maturity for being used in industrial applications. This method provides means for analytically guaranteeing the fulfillments of tasks' deadlines.

In the type of applications subject of this work is common that the activities to be executed depends on the state of the system and the environment. Changes in this state may require to adapt the functions performed by the application. To deal with this situation, a number of meaningful operational modes are identified. They are characterized by the set of activities to be executed and a system state during which it is active. If the system state changes, the execution mode may also be changed. Mode changes must be done in a safe way. In particular, they are required that data consistency is kept, and that hard deadlines of the activities are met during the mode change operation. A number of protocols that fulfil these requirements have been defined [7] [8].

Some of the proposed mode change implementations in Ada [9] [10] are not valid due to the restrictions imposed by the Ravenscar profile. In particular, the forbidden language facilities that invalidate these approaches are:

- Dynamic priorities.
- Select statements.

The reasons why mode change protocols are invalidated are:

- Dynamic priorities are not allowed, and hence it is not possible to change the priority of the task according to the job profile.
- In the published implementations of mode changes, it is common to rely on select statements for letting the tasks know that a mode change has been requested. However, as this statement is forbidden in the RP, the delay for letting the task know this event can be too long. This has a negative effect in the overall mode change delay.

The approach proposed in this paper for dealing with mode changes in the RP relies on the following two ideas:

- *Decoupling tasks from jobs*. The usual implementation is to associate a task to a job. Then, it executes the job activities in every system mode. As mentioned above, this is not possible under the RP, because of the lack of dynamic priorities. In the proposed approach, tasks and jobs are decoupled. This implies that a task can execute different job profiles at different modes and vice versa. The binding between a task and a job profile is performed when a mode changes. The relation is based on the priority: the priority of the task should be the same as the execution priority assigned to the job profile.
- *Centralized activation of tasks*. The most reasonable implementation of a periodic task is an endless loop which includes a delay until instruction for programming the activation times. Asynchronous events, such as a mode change request are implemented with a select instruction, which is forbidden under the RP. The proposed approach is based on programming the periodic behaviour outside of the task. The periodic task is blocked in a suspension object. When the delay expires, a manager unblocks the periodic task.

In the rest of this paper, a detailed description of the proposed approach for implementing mode changes with the RP is presented. Section 2 presents the underlying system model. Section 3 describes the general approach, while section 4 discusses the proposed implementation.

2. System model

The system is composed by a set of jobs, which are passive entities that define how certain related activities have to be executed. System execution is structured as a number of modes. An execution mode can be viewed as a certain system state that requires the execution of a specific set of activities. The system evolution can be modeled by a finite-state machine, where the nodes are modes and the arcs defines allowed mode changes and are labelled with the condition that has to occur for firing the change.

A job is described by means of profiles, which include all the information required for executing a job, such as activation pattern, period, firing event, offset, deadline, priority and the procedure that has to be executed at each activation. In general, a job will be fully described by stating its profile for each of the system modes, as shown in figure 1.

The set of activities to be executed during a mode can be reflected in a table with at most one profile for each job. An example of this table is shown in figure 2. There can be jobs that are not active for a particular mode, and hence there is no corresponding entry in this mode table. In the context of this work, it is assumed that all tasks have different priorities. Although this restriction can be removed, it helps to simplify the presentation of the approach.

		Job_1				
Mode ID	*Type*	*Event*	*T*	*D*	*PR*	*Activity*
Mode_A	*Per*	-	*T_1A*	*D_1A*	*10*	*Act_1_A*
Mode_B	*Per*	-	*T_1B*	*D_1B*	*10*	*Act_1_B*
. . .	…	…	…	…	…	…
Mode_G	Per	-	*T_1G*	*D_1G*	*15*	*Act_1_G*

Figure 1. Job definition. Profiles for each mode.

		Mode_A				
Job ID	*Type*	*Event*	*T*	*D*	*PR*	*Activity*
Job_1	*Per*	-	*T_1A*	*D_1A*	*10*	*Act_1_A*
Job_2	*Spor*	*Event_1*	*T_2A*	*D_2A*	*11*	*Act_2_A*
. . .	…	…	…	…	…	…
Job_N	Per	-	*T_NA*	*D_NA*	*8*	*Act_n_A*

Figure 2. Mode definition. Job profiles to be executed in the mode.

Tasks are the active entities in the model and they are in charge of executing job profiles. The code of a task is completely generic and its implementation has no direct relation with the functionality of a particular application. It is on run-time when a job profile is assigned to a particular task. Then the task can execute the activities defined by the profile. It should be created one task for each of the priority values used by any job profile in any of the modes.

There is a system manager which is in charge of

- Managing mode changes, following an appropriate protocol.
- Binding job profiles to tasks, during a mode change according to jobs and modes descriptions. The bind is based on profiles priorities.
- Activating tasks at the beginning of a new period or when an appropriate event occurs.

3. System design

The proposed approach for dealing with mode changes is based on decoupling tasks from jobs. In most published of mode change implementations, a task always executes the code associated with a certain functionality. When a mode change is carried out, tasks profiles are updated, which implies to set the appropriate activity to be executed and parameters, such as period and priority.

Profiles are related to tasks in a dynamic way. In particular, the assignment is made at the beginning of a mode, i.e. when a mode change is executed. In this way, it is possible to achieve the effect that different activities of a certain job can be executed with different priorities. Instead of changing the priority of the task that executes the job, it is

changed the task that executes it. As a result, the activities of the job in different modes are executed with different and appropriate priority.

As mentioned above, the prohibition to use select statements in the RP makes it difficult for a task to detect that a mode change is taking place. This is because when the task is delayed, it is not possible to awaken it until the timer expires. This can lead to long delays to perform a mode change and to complex implementations. The proposed alternative is to change the usual activation implementation by deferring to a single entity the activation management.

The manager is the component in charge of executing the mode changes and tasks activation. In particular, its responsibilities are:

- Manage mode changes: The manager changes the mode when there is a request from any of the tasks. Once this event occurs, it waits to the appropriate time and performs the change according to a predefined mode change protocol.
- Bind job profiles and tasks: One of the activities to perform during a mode change is to bind tasks and job profiles, following the mode settings and the corresponding tables. This binding is based on the priority of the profile. In particular, a profile is assigned to the task that executes with its priority. If there is no profile with a certain priority, then the corresponding task will not be executed for the duration of the mode. The relation between tasks and profiles for a particular mode is illustrated in figure 3..
- Activate tasks: The manager is also in charge of activating tasks. This operation is done by calling the suspension object that is associated to each task. The task has previously called a suspend operation. When the activation time of the task is reached, then the manager unlocks it. For this purpose, the manager maintains a list with the activation times for all the active tasks. Then, it executes a delay until instruction for the closest activation time.
- The activation of sporadic tasks could be done in a similar way. It should be necessary to export a procedure for letting to know when a meaningful event has occurred. Then, the appropriate sporadic task could be activated. For this purpose, it seems necessary to add an additional manager task for dealing with this functionality.

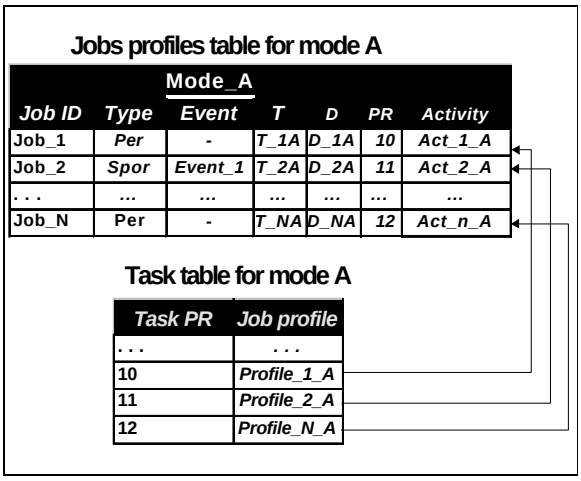

Figure 3. Binding between tasks and profiles for a particular mode.

In order to implement tasks activation, one suspension object is defined for each of them. When a task is not active, it is suspended on its object. When it is time to activate them, the manager calls the unlock procedure. This operational scheme is illustrated in figure 4.

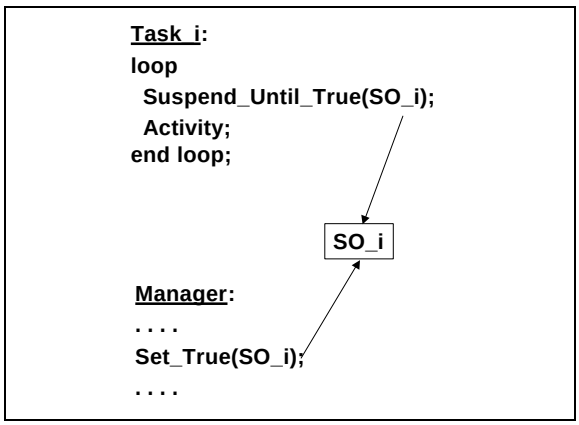

Figure 4. Tasks activation pattern.

The manager can also be used for detecting missed deadlines. In addition to the activation tables, it could have a deadlines table. Then the manager task is delayed until the closest activation time or deadline. In the second case, if the corresponding task is not blocked in the suspension object, then it has not met its deadline. This functionality and the activation of sporadic tasks is under development.

4. Mode change implementation

4.1 Global declarations

There are a number of declarations that are system-wide. They are specific for an application and include:

- Enumeration type with the system modes
- Enumeration type with the jobs identifier.
- Valid priority range. It includes the priority values that are used by any profile. This range indicates the number of tasks and suspension objects that should be created.
- Initial system mode.

4.2 Supporting data structures

The data described above is stored in a set of data structures, which are abstract data types. Best way to implement them is as generic units that depends on the global declarations. As generic units are not fully recommended for high integrity systems [4], they have been implemented as packages, although with a loss of reusability.

The main data structures used are the following:

- Job profile: This structure includes all the components of a job profile. Procedures are provided for setting and getting each element. Internally it is implemented as a record. The declaration of the record follows the structure illustrated in figure 1.
- System jobs table: This data structure relates profiles with jobs and modes. The provided operations serve to set and get a profile that corresponds to a job and a mode.
- Priority profile table: this table serves to associate profiles and priorities. This table is used for relating a task with the job profile it has to execute in a particular mode.

The package P_System_Tables includes the system tables. Its implementation relies on the previously described abstract data types. It provides an interface where the global tables are hidden to the users of the package.

```
with P_System_Declarations;
with P_Job_Profile;

package P_System_Tables is

 procedure Add_Job_Mode_Profile
   (The_Profile : P_Job_Profile.T_Job_Profile;
    The_Mode_ID:P_System_Declarations.T_System_Modes;
    The_Job_ID  : P_System_Declarations.T_Job_ID);

  function Get_Job_Mode_Profile
   (The_Mode_ID:P_System_Declarations.T_System_Modes;
    The_Job_ID  : P_System_Declarations.T_Job_ID)
    return P_Job_Profile.T_Job_Profile;

-- Priority-Profile table

  procedure Set_Priority_Profile_Table
 (The_Mode_ID:P_System_Declarations.T_System_Modes);

  function Get_Priority_Profile
     (The_Priority : P_System_Declarations.T_Priority_Range)
     return P_Job_Profile.T_Job_Profile;

end P_System_Tables;
```

4.3 Manager

The manager is encapsulated on a package. It provides operations for getting the current mode and requesting a mode change.

```
with P_System_Declarations;

package P_Mode_Manager is

  procedure Change_Mode (New_Mode :
      P_System_Declarations.T_System_Modes);
  function  Get_Mode return
      P_System_Declarations.T_System_Modes;

end P_Mode_Manager;
```

Within the body of this package, a number of state variables are defined. The most relevants hold the current mode and the next mode. They are only different when a mode change is pending. The main component of the package is the manager task, which is in charge of performing the activation and mode management activities. Its implementation is based on an endless loop. First, it is checked whether a mode change request has occurred. In that case, the mode change procedure is called.

The activation of the periodic activities is performed by calculating the closest activation time among all the jobs which are executed in the current mode. Then, a delay is executed. When the manager task is awaken, the appropriate task is activated, by calling to the unlock operation of the associated suspension object. Then, it calculates the next activation time and stores it in the appropriate table.

```
-- Task specification
task Manager_Task is
    pragma Priority (System.Any_Priority'Last);
  end Manager_Task;

  task body Manager_Task is
    The_Job      : P_System_Declarations.T_Job_ID;
    The_Priority : P_System_Declarations.T_Priority_Range;
    Activation   : Ada.Real_Time.Time;
  begin
    loop
      if Next_Mode /= System_Mode
        then Perform_Mode_Change;
      end if;

      Get_Closest_Activation
               (Activation, The_Priority, The_Job);

      delay until Activation;

      Ada.Synchronous_Task_Control.Set_True
     (P_Blocking_Objects.All_Locking_Objects(The_Priority));
      Activation_Time(The_Job):= Activation_Times(The_job)
           +  P_Job_Profile.Get_Period
                (P_System_Tables.Get_Job_Mode_Profile
                        (System_Mode, The_Job));
    end loop;
  end Manager_Task;
```

The mode change protocol implemented in this example is described in [7]. It is based on identifying an instant when the attributes of the tasks can be safely changed to the new ones. This can be ensured if tasks in the old mode do not affect the worst-case execution time of tasks in the new mode. This special situation occurs in an *idle period*, which is a time interval in which there are no real-time tasks

eligible for execution. In the implementation, there is a task that is activated when it is detected the mode change and it has lower priority than real-time tasks. Its main function is to update the priority profile table and the activation table.

4.4 Tasks and suspension objects

Every task has an associated suspension object, where it is blocked waiting for its activation. These objects are declared as follows:

```
with P_System_Declarations;
with Ada.Synchronous_Task_Control;
package P_Blocking_Objects is

 All_Locking_Objects : array
    (P_System_Declarations.T_Priority_Range) of
    Ada.Synchronous_Task_Control.Suspension_Object;

end P_Blocking_Objects;
```

Tasks are defined by means of a task type, with a discriminant that is their priority. Tasks of this type are defined for each of the used priorities of any job profile.

```
with P_System_Declarations;
with System;
package P_Execution_Objects is

 task type T_Execution_Object
  (The_Priority:P_System_Declarations.T_Priority_Range) is
     pragma Priority (System.Any_Priority(The_Priority));
 end T_Execution_Object;

 Task_Pr_5 : T_Execution_Object (5);
 Task_Pr_6 : T_Execution_Object (6);
end P_Execution_Objects;
```

The task implementation is an endless loop where the task first suspends itself in the corresponding object. When it is activated by the manager, it executes the activity that belongs to the profile that is associated with the task in the current mode.

```
 task body T_Execution_Object is
   The_Activity : P_Job_Profile.T_Activity;
 begin
    loop
    -- The task suspends in the suspension object
    Ada.Synchronous_Task_Control.Suspend_Until_True
    (P_Blocking_Objects.All_Locking_Objects(The_Priority));
    -- The activity is executed
    P_Job_Profile.Get_Activity
    (P_System_Tables.Get_Priority_Profile (The_Priority)).all;
   end loop;
 end T_Execution_Object;
```

In this implementation, activities are referred as an access to procedure. In this way, task code is completely independent of the profiles. However, this is not completely appropriate according to the accepted recommendations for building high integrity systems. Alternatively, it is possible to explicitly call to the appropriate activities within a case statement, whose index is the system mode. This solution is less reusable and elegant than the previous, but it is fully compliant with the recommendations for HIS [4].

A job can be implemented as a package which includes the activities to be executed on each mode. It has to initialize the corresponding data in the system tables. It is possible to use global variables if it is necessary to keep perdurable data between activations. In the current implementation, the activities to be executed are parameterless procedures.

4.5 Discussion

For validation purposes, it has been used the Open Ravenscar Kernel (ORK) [11], which is an open-source real-time kernel that supports Ada 95 and C applications, and that is fully integrated with GNAT. An implementation of the proposed ideas was developed, compiled with ORK, and tested. In this way it was checked that it complies with the RP and that it works as desired. (More information about ORK can be found at http://www.openravenscar.org).

With respect to the performance, it is obvious that the use of the manager task and the proposed activation pattern adds some overhead. In particular, the two main effects are that context switches are doubled and the manager task interferes to the rest of tasks. The exact overhead has not already measured, although it seems to be low.

The final system can be analyzed, as it is possible to know which is the maximum interference of the manager task. The number of times it interferes with the execution of a task is equal to the number of other tasks that can be activated. With respect to the cost of changing mode, the main required execution time is that required to fill the priority profile table and the activations time table, which is internal to the mode manager package.

5. Conclusions

In this paper it has been presented an approach for an implementation of mode changes that is fully compliant with the Ravenscar Profile. In addition, this proposal can be implemented in a way that meets the recommendations for high integrity systems. The presented implementation has been tested with the ORK environment.

There is future work to be done. The implementation of sporadic tasks following this approach has been sketched, but has already to be fully implemented and tested. In addition, it is needed to have precise measurement of the overhead that this approach implies, in order to let the designers whether it is feasible to follow this approach when building a hard real-time system.

6. Acknowledgments

This work has been funded by Spanish CICYT, under contract TIC99-1043-C03.

The authors would like to thank to Juan Zamorano, Michael González Harbour, José F. Ruíz and Jorge Real for the fruitful discussions that have helped to improve and validate the proposed approach.

7. References

[1] A. Burns, B. Dobbing, and G. Romanski, "The Ravenscar profile for high integrity real-time programs," in Reliable Software Technologies — Ada-Europe'98 (L. Asplund, ed.), no. 1411 in LNCS, Springer-Verlag, 1998.

[2] Ada 95 Reference Manual: Language and Standard Libraries. International Standard ANSI/ISO/IEC-8652:1995, 1995. Available from Springer-Verlag, LNCS no. 1246.

[3] A. Burns, "The Ravenscar profile," Ada Letters, vol. XIX, no. 4, pp. 49–52, 1999.

[4] ISO/IEC/JTC1/SC22/WG9, Guidance for the use of the Ada Programming Language in High Integrity Systems, 2000. ISO/IEC TR 15942:2000.

[5] A. Burns, "Preemptive priority based scheduling: An appropriate engineering approach," in Advances in Real-Time Systems (S. Son, ed.), Prentice-Hall, 1994.

[6] A. Burns and A. J. Wellings, Real-Time Systems and Programming Languages. Addison-Wesley, 2 ed., 1996.

[7] K. Tindell and A. Alonso, "A very simple protocol for mode changes in priority preemptive systems," tech. rep., Universidad Politécnica de Madrid, 1996.

[8] J. Real, "Protocolos de cambio de modo en sistemas de tiempo real," Master's thesis, Universidad Politecnica de Valencia, January 2000. In Spanish.

[9] A. Alonso, J. A. de la Puente, and K. Tindell, "Components for the implementation of fixed priority real-time systems in ada," in Second International Workshop on Real-Time Ada Issues, ACM SIGAda, 1997. Ada Letters, (17)5.

[10] J. Real and A. Wellings, "Implementing mode changes with shared resources in ada," in 11th Euromicro Workshop on Real-Time Systems, IEEE Computer Society Press, 1999.

[11] J. A. de la Puente, J. F. Ruiz, and J. Zamorano, "An open Ravenscar real-time kernel for GNAT," in Reliable Software Technologies — Ada-Europe 2000 (H. B. Keller and E. Ploedereder, eds.), LNCS, Springer-Verlag, 2000.