Pergamon

Control Eng. Practice, Vol. 5, No. 7, pp. 999-1005, 1997

0967-0661/97 $17.00 + 0.00

PII:S0967-0661(97)00088-9

BUILDING SAFETY-CRITICAL REAL-TIME SYSTEMS WITH REUSABLE CYCLIC EXECUTIVES

J. Zamorano*, A. Alonso and J.A. de la Puente****

**Universidad Politécnica de Madrid, Facultad de Informática, E-28660, Boadilla del Monte, Spain (jzamora@fi.upm.es)*
***Universidad Politécnica de Madrid, ETSI Telecomunicación, E-28040, Madrid, Spain*

(Received January 1997; in final form March 1997)

Abstract: Many real–time systems have strict safety requirements, and concurrent processes cannot be used in their development. Thus, these safety–critical systems are developed using synchronous architectures based on cyclic scheduling. This paper describes a reusable cyclic executive implementation in Ada 95, based on a generic architecture for synchronous real–time systems. This generic architecture is described by means of an object–oriented design notation. New Ada characteristics, as well as exceptions and generics, are used to build the component blocks of this generic architecture. In order to develop real–time systems, guidelines for using these reusable components are also provided.

Keywords: Safety–critical, real–time systems, software engineering, real–time languages, system architectures.

1. INTRODUCTION

Many computer systems have strict safety requirements in addition to hard real–time requirements. Application areas for this kind of system include 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 language and operating–system constructs, and features such as dynamic data objects or concurrent threads or processes are often excluded (Holzapfel and Winterstein, 1988). This precludes the use of modern software architectures for real–time systems, such as those based on asynchronous threads with fixed–priority scheduling, for which operating–system standards and timing–analysis techniques have received widespread interest in recent years (Burns, 1994; Audsley *et al.*, 1995). Instead, most safety–critical systems are still being developed using synchronous architectures based on cyclic executives with static schedules, which has been the prevailing technique in industry for a long time (Burns and Wellings, 1996).

Synchronous architectures have, indeed, clear advantages when safety is a critical issue. Since there is no arbitrary interleaving of operations, the behavior of the system is fully predictable, and can be easily tested. No operating-system or run–time system support is required, and therefore the executable code can be kept small, which may be of interest in some on–board embedded systems with limited storage capacity. However, they also have serious problems, which have often been pointed out. The most important are the complexity of building cyclic schedules and the low abstraction level of the resulting software, which makes it difficult and costly to develop and maintain. These problems can be overcome using careful design and implementation techniques

as long as the system size stays small, but as real–time systems grow in size and complexity, raising the level of abstraction and enhancing the flexibility and maintainability of this kind of system becomes of primary industrial interest (Locke, 1992; Zamorano, 1995).

Reusable software components provide one promising technique for achieving these goals. Modern programming languages such as Ada 95 (ISO, 1995) and C++ (Stroustrup, 1991) provide a solid basis for the development of software components through the use of abstract data types and generic packages or templates. Components can be made very reliable by thorough testing, and make it easier to develop well–organized software architectures from a reduced set of building blocks, thus enhancing the flexibility and maintainability of the system.

A library of reusable components for avionics systems, built around two kinds of executives, a multithreaded one and a generic cyclic executive, was described in a previous work (Fernández and de la Puente, 1991; de la Puente *et al.*, 1992). The present paper develops the latter subject further, especially in two areas. The first is a more flexible approach to a reusable cyclic executive, using new Ada 95 features, and the second is the concept of a generic synchronous architecture which uses additional software components.

2. REQUIREMENTS FOR GENERIC CYCLIC EXECUTIVES

The cyclic executive is the core of the proposed synchronous architecture. The purpose is to make the executive into a generic, reusable component which can be tailored to a wide variety of real–time applications. Of course, the cyclic schedule itself has to be configurable. Basically, a cyclic schedule is a table which contains the full execution pattern for a major cycle. A major cycle consists of a fixed number of minor cycles which are executed periodically. Each minor cycle consists of a variable number of frames of variable duration. Each process or subprocess is executed as a procedure in one frame of one or more minor cycles. The system can have several operating modes with different schedules. Therefore, the following parameters should be configurable:

List of operation modes and, for each of them:

- Duration of minor and major cycle periods
- Number of minor cycles per major cycle
- Maximum number of frames per minor cycle
- Process (reference to procedure) that has to be executed in each frame.

In order to make the executive easily portable between different hardware architectures, the real–time clock and other possible timer software must be encapsulated as separate components. Full hardware independence is generally not feasible, though.

Since the aim is the development of safe systems, some provision for fault tolerance must be made. A dynamic recovery policy will be adopted, based on the concept of recovery groups (de la Puente *et al.*, 1992). A recovery group is a set of related processes such that, if one of them fails, then the whole group is considered to have failed. In this case, the whole group is (gracefully) aborted, and a single recovery process is started that provides degraded functionality for the group. Should the recovery process itself fail, the whole system is taken to a safe stop.

One kind of failure that must be catered for in hard real–time systems is the deadline overrun. It has been assumed that every process has a specified deadline and a computation time budget. Whenever one of these intervals is violated, the process is immediately aborted, and all the processes that belong to its recovery group are tagged as faulty. The recovery process of the recovery group will be scheduled during subsequent minor cycles, preventing the other processes from overrunning in the first minor cycle.

3. CYCLIC SCHEDULER STRUCTURE

The main component of this kind of synchronous architecture is the cyclic scheduler. It is an iterative procedure that allows the explicit multiplexing of a periodic process set in one processor. A major schedule defines the process sequence that executes during a major cycle. This process sequence is cyclically repeated: once it ends, the major schedule executes repeatedly until the operating mode of the system is changed.

The major schedule is divided into one or more minor schedules, which describe the process sequence that executes during each minor cycle. The instant at which a minor cycle ends, and the next one begins, is synchronized by the ticks of a minor cycle clock. The synchronization enforced by the minor cycle clock ticks guarantees that the real–time requirements of the process are fulfilled.

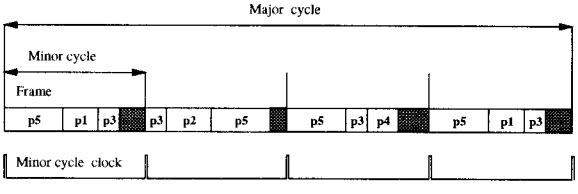

Fig. 1. Cyclic scheduler structure

Each tick of the minor cycle clock indicates the end of a minor cycle. At that time, the process

sequence corresponding to this minor cycle must have finished its execution. If this is not so, a `minor_cycle_overrun` occurs.

Each minor schedule is divided into one or more frames. Within each frame, only one process of the sequence corresponding to the minor schedule executes. When the frame ends, the corresponding process must have finished its execution, otherwise `frame_overrun` occurs.

3.1 *Scheduler exceptions*

The natural way of dealing with errors or anomalous situations is through exceptions. An exception is an event that leads to the temporary suspension of the normal program execution. Ada allows code exception handlers to capture predefined exceptions or exceptions that are defined by the programmer. When an exception is activated in a unit, its execution is terminated and an exception handler takes over the control of the execution. The latter can be coded at the end of the block or subprogram body, package or task it belongs to.

Exceptions simplify logic, and make the use of components easier. They are the most natural way of dealing with abnormal, erroneous, inconsistent or unexpected conditions that may occur during the execution of component operations.

There are predefined exceptions, that are implicitly activated by the run–time system when the associated event happens. The programmer can define new exceptions that will be explicitly activated through the statement `raise`. Predefined exceptions can also be explicitly activated.

The appearance of overruns in real–time systems is an error that can be conveniently treated with the exception mechanism. In this way, if an overrun in the execution time is produced in the minor cycle, the corresponding exception is activated in the process.

By using this mechanism, the cyclic scheduler's main loop structure can be coded in a clean and simple way, as shown in Figure 2.

However, restrictions that apply to the Ada language in safety–critical system development do not allow the use of access types or task operations, such as the `delay` statement. Asynchronous transfer of control (ATC) cannot be used, neither can the access to subprogram types included in the new language standard.

Hence, an interface component is needed to activate exceptions in an asynchronous way, and to call variable procedures. Furthermore, components are needed to isolate the executive from timer device–dependency.

```
Next_Minor_Cycle:= clock;
Scheduler_Cycle:
loop
  Major_Cycle:
  for Current_Cycle in Minor_Cycle_Index'range loop
    begin
      Next_Minor_Cycle:= Next_Minor_Cycle +
        Current_Period;
      select
        delay until Next_Minor_Cycle;
        raise Minor_Cycle_Overrun;
      then abort
        Minor_Cycle:
        for Current_Frame in Frame_Index'range loop
          exit Minor_Cycle when
            Current_Frame.Code = null;
          begin
            select
              delay Current_Frame.Computation_Time;
              raise Frame_Overrun;
            then abort
              -- Access to subprogram
              Current_Frame.Code.all;
            end select;
          exception
            when Frame_Overrun =>
              -- Recovery actions
          end;
        end loop Minor_Cycle;
      end select;
      -- A background process could be
      -- run during this time.
      delay until Next_Minor_Cycle;
    exception
      when Minor_Cycle_Overrun =>
        -- Recovery actions
    end;
  end loop Major_Cycle;
  -- check for mode change now
end loop Scheduler_Cycle;
```

Fig. 2. Cyclic scheduler loop

4. A GENERIC ARCHITECTURE FOR SYNCHRONOUS REAL–TIME SYSTEMS

Given this set of requirements, a generic architecture for synchronous real–time systems can be devised. A generic architecture is a template for building a system or subsystem from component blocks, interconnected by a set of standard elements. The generic architecture proposed here is made up of the following components:

- Structures: these are modules in which the basic data structures of the system are defined in a portable way. These basic structures allow processes, modes, recovery groups and schedules to be defined. They are independent of the development system and the execution architecture.
- Executive: this is a subsystem, consisting itself of some simpler components that isolate hardware, compiler and run–time system dependencies. The main ones are a cyclic scheduler, a run–time system interface, and two interval timers.

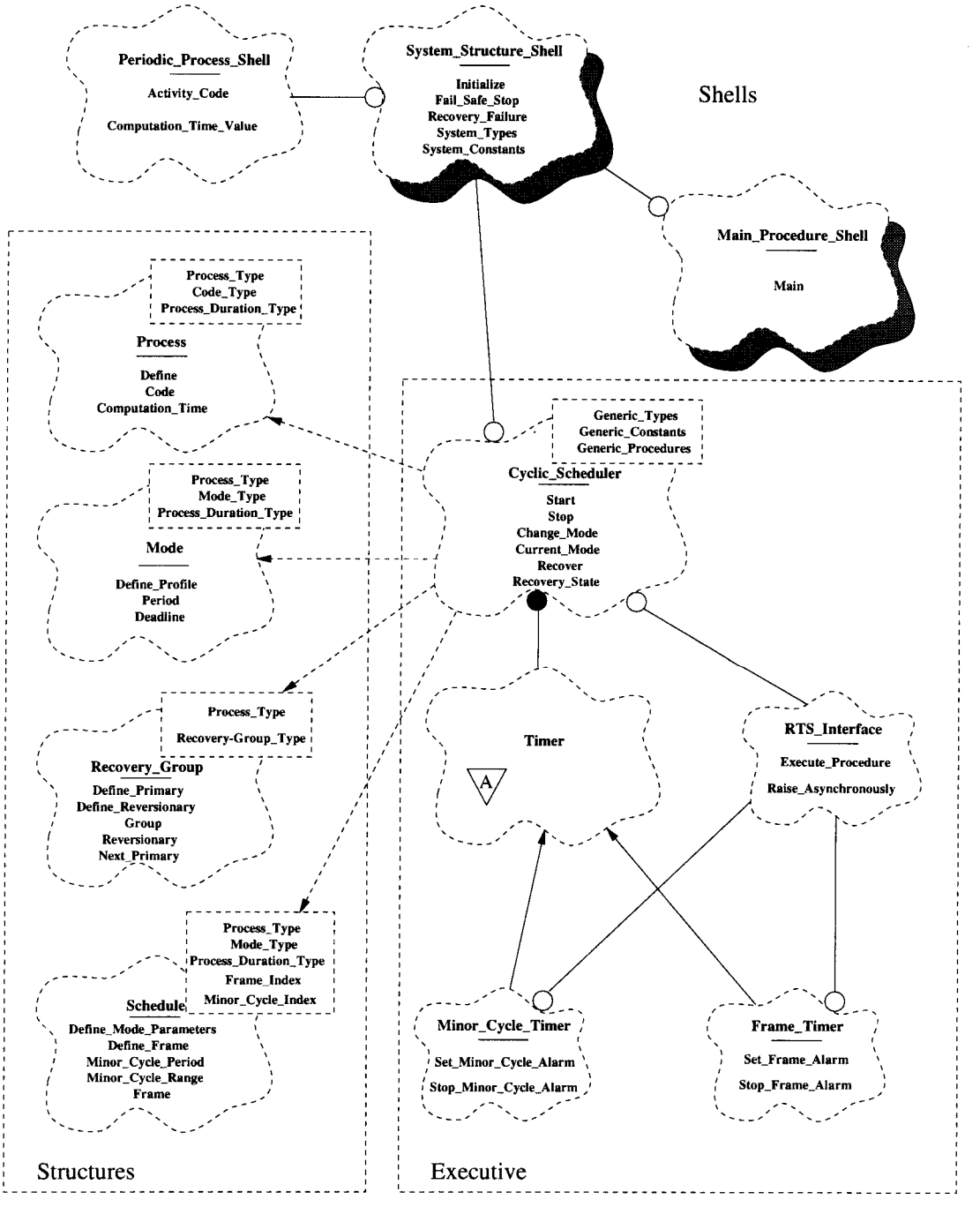

Fig. 3. Logical view of cyclic executive software architecture

- Shells: these are generic templates for building application processes and defining the system structure.

The system structure as a whole is represented as a global component including all the system parts.

The basic idea behind this generic architecture is that all the components can be made available as reusable software components coded in an appropriate programming language. The components are generic and thus configurable, and can be adapted to a particular design. The representation of the overall architecture as a global component makes it easier to build a system from its constituent blocks.

Figure 3 shows the *logical* view (Kruchten, 1995) of the cyclic executive. The Booch Lite notation (Booch, 1994) is used for describing this object–oriented software architecture. This logical view is built from the functional requirements of the executive. The system is decomposed into a set of object classes that exploit the principles of abstraction, encapsulation, and inheritance. The Ada programming language provides these modern software engineering principles.

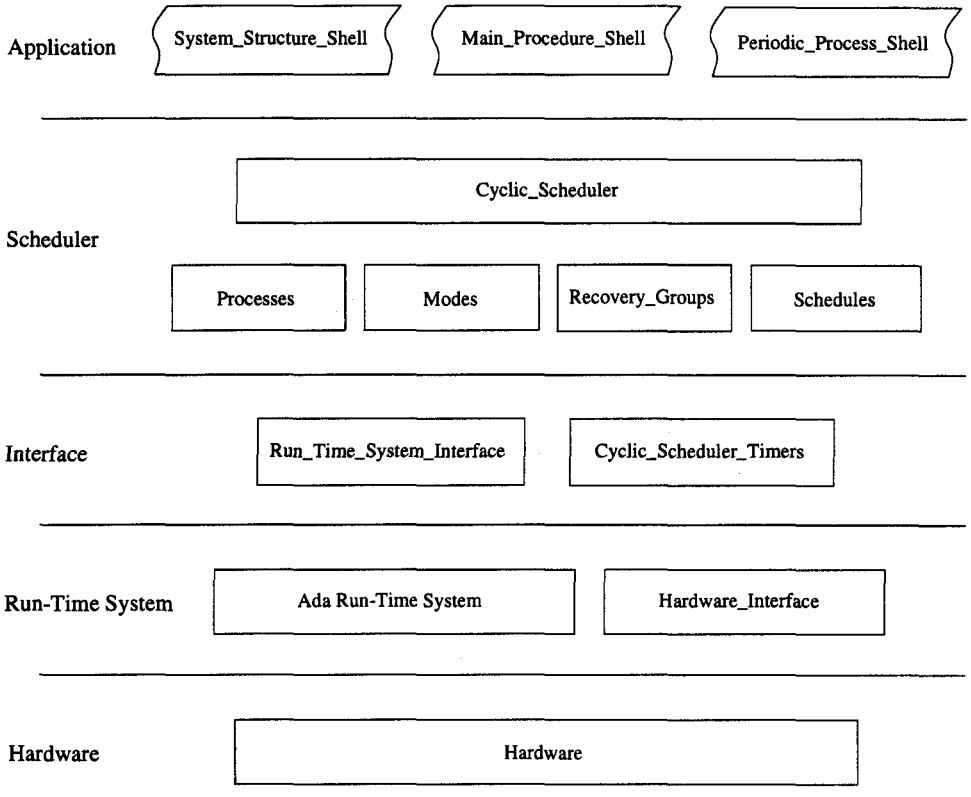

Fig. 4. Development view of cyclic executive software architecture

Another interesting view is the *development* view (Kruchten, 1995), which describes the static software organization in its development environment. The software modules are organized in a hierarchy of layers, as shown in Figure 4. The lower layers are related to the hardware and run–time system. On these layers, the first cyclic executive layer is built up. This layer is made up of the executive interface components. The next layer is an instance of the `Cyclic_Scheduler` component; that is, the specific cyclic executive of the application. Inside this component, instances of the components that deal with executive basic objects are created. These basic objects are the processes, modes, recovery groups and schedules. The application is located at the top layer, that includes the periodic processes, the component that defines the system structure and the main procedure of the application.

Of course, full genericity is not feasible, as there are some hardware and run–time system dependencies. In the software architecture of the executive, these dependencies are isolated in the components of the Interface layer: `Run_Time_System_Interface`, `Minor_Cycle_Timer`, and `Frame_Timer`. The solution adopted is the development of reusable versions of these objects for a particular hardware architecture, and the provision of guidelines for the designer to adapt them to other hardware configurations.

5. IMPLEMENTATION IN ADA 95

The generic architecture proposed in the last section can be implemented in Ada 95 in a rather straightforward way. No tasking features of Ada 95 (including timing or delays) are used, since they are usually prohibited in safety–critical systems. Other features that are not used are dynamic storage allocation, recursive procedure calls, and unlimited loops (except in the executive main loop). However, exceptions are used, as they are a powerful way of structuring error–detection and recovery mechanisms. With these restrictions, the executive main loop that results is shown in Figure 5.

Some new characteristics of Ada 95, that have proved very useful in building reusable code, are:

- Enhanced generic parameters, including packages and access types.

  The generic package mechanism provides for parameterization of the abstraction represented by a package. Generic formal packages allow generics to be parameterized by other generics, which allows for the safer and simpler composition of generic abstractions. In this way, a hierarchy of related packages can be built up, avoiding an excessive number of formal parameters in the `Cyclic_Scheduler` specification.
- Enhanced access types, including access to static objects and procedures.

```
Set_Minor_Cycle_Alarm (Current_Period);
Scheduler_Cycle:
loop
  Major_Cycle:
  for Current_Cycle in Minor_Cycle_Index'range loop
    begin
      Minor_Cycle:
      for Current_Frame in Frame_Index'range loop
        exit Minor_Cycle when
          Current_Frame.Process = Null_Process;
        begin
          Set_Frame_Alarm (Current_Frame.Frame_Time);
          Call (Current_Frame.Process);
          Stop_Frame_Alarm;
        exception
          when Frame_Overrun =>
            -- Recovery actions
        end;
      end loop Minor_Cycle;
      -- A background process could be
      -- run during this time.
      wait_minor_cycle_tick;
    exception
      when Minor_Cycle_Overrun =>
        -- Recovery actions
    end;
  end loop Major_Cycle;
  -- check for mode change now
end loop Scheduler_Cycle;
```

Fig. 5. Cyclic scheduler loop coded in Safe Ada

As the code of a process is a parameterless procedure, this feature allows the storage into schedules of the process's code references. Later on, at run-time, it enables the cyclic executive to dynamically select and invoke the parameterless procedure that is to be executed during the next frame, as shown in Figure 2. However, access types cannot be used in Safe Ada, and thus a procedure `Call` has been developed for this purpose. It is included in the `Run_Time_System_Interface` component, and its use is shown in Figure 5.

- Hierarchical public and private packages.

  Ada packages provide abstraction and separation between implementation and specification. Hierarchical packages allow a closer control of visibility through the use of public or private child units, simplifying the development of large systems from component subsystems (de la Puente *et al.*, 1996). Structures are generic private child packages of the generic package `Cyclic_Scheduler`. In this way, the decomposition of the `Cyclic_Scheduler` implementation is hidden from the rest of the system. The *renaming-as-body* feature is used to export child operations.

- Standard interfaces to procedures written in other languages.

  This feature allows an Ada programmer to use commercial components and subsystems coded in other programming languages in a clean and portable way. Examples of software packages that can be used are mathematical libraries and device drivers. It also makes more portable the specifications of the packages `Run_Time_System_Interface`, `Minor_Cycle_Timer` and `Frame_Timer`.

These new Ada characteristics, together with exceptions and generics, allow a developer to build parameterized components with a high level of cohesion and loose coupling. In this way, its reusability potential is increased. The generic synchronous architecture can easily be built with this kind of component. Moreover, these characteristics enhance the flexibility, reliability, and maintainability of the application code itself.

6. BUILDING REAL-TIME SYSTEMS

The building of real-time systems based on this architecture is straightforward. The developer writes the application code according to the shells provided, together with the components, and provides actual parameters for the generic objects of the architecture. The following steps are required:

(1) Write a package using the `System_Structure_Shell` in order to define the structure of the system. This package includes the definition of types for identifying and defining the cyclic executive objects: processes, modes, recovery groups and schedules. Procedures for performing special functions, such as `Fail_Safe_Stop`, must be defined, as well as one schedule per operation mode.

These types, procedures and constants constitute the formal parameter list of the `Cyclic_Scheduler`. Therefore, in order to simplify the use of the `Cyclic_Scheduler`, a set of generic packages must previously have been instantiated. These packages are the actual parameters for instantiating the generic package `Cyclic_Scheduler`.

(2) Write the application process packages, which can be derived from the `Periodic_Process_Shell`. One package per application process must be created.

The process identifier, code, and computation time must be defined, as well as the recovery group and the process category (primary or recovery).

(3) Write the main procedure, derived from the `Main_Procedure_Shell`. This procedure starts the system in some specific operating mode.

The instantiation and type checking are automatically done by the compiler, so that the developer does not have to rewrite the basic components.

7. CONCLUSION

New Ada features, such as hierarchical packages and packages as generic parameters, facilitate the development of reusable components. They are very powerful tools for implementing decomposition and aggregation, which are fundamental in object–oriented analysis and design methods.

The approach described in this paper can lead to a simpler and safer development of real–time systems. A previous version of the architecture's components has been used for an airborne computer system of a modern fighter airplane built in part by CASA (Construcciones Aeronáuticas S.A.). Although some changes were required in order to cope with some restrictions in the Ada language that were imposed by the project, reusing the components was straightforward, and resulted in considerably less development time than initially forecast.

Also, the concept of recovery groups (de la Puente *et al.*, 1992) has been reused in an Ada cyclic executive for space real-time applications (André *et al.*, 1996) developed by TRIALOG with the support of the ESA/ESTEC (European Space Agency Technical Center).

This architecture could also be easily implemented in C++, although the class structure is not as well organized for large systems as in Ada; and the use of pointers must be carefully controlled in order to prevent some common run–time errors.

ACKNOWLEDGMENTS

This work has been supported by the Spanish Agency for Scientific and Technical Research (CICYT), projects TAP93–0001–CP and TIC96–0614.

8. REFERENCES

André, J.-M., A. Kung and P. Robin (1996). OX: Ada cyclic executive for embedded applications. In: *International Symposium on "On-board Real-time Software"*. ESA SP-375. pp. 241–245.

Audsley, N., A. Burns, R. Davis, K. Tindell and A. Wellings (1995). Fixed-priority pre-emptive scheduling: an historical perspective. *Real-Time Systems* **8**(2), 173–198.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications.* Second ed. Benjamin Cummings.

Burns, A. (1994). Preemptive priority based scheduling: An appropriate engineering approach. In: *Advances in Real-Time Systems* (S.H. Son, Ed.). Prentice-Hall. pp. 225–248.

Burns, A. and A.J. Wellings (1996). *Real-time systems and their programming languages.* Second ed. Addison-Wesley.

de la Puente, J., J. Zamorano, A. Alonso and J. L. Fernández (1992). Reusable executives for hard real-time systems in Ada. In: *Ada: moving towards 2000* (J. van Katwijk, Ed.). Lecture Notes in Computer Science 603. Ada Europe. Springer-Verlag. pp. 104–115.

de la Puente, J., A. Alonso and A. Alvarez (1996). Mapping HRT-HOOD designs to Ada 95 hierarchical libraries. In: *Reliable Software Technologies* (A. Strohmeier, Ed.). Lecture Notes in Computer Science 1088. Ada Europe. Springer-Verlag. pp. 78–88.

Fernández, J. L. and J. A. de la Puente (1991). Constructing a pilot library of components for avionic systems. In: *Ada: The Choice for '92* (D. Christodoulakis, Ed.). Lecture Notes in Computer Science 499. Ada Europe. Springer-Verlag. pp. 362–371.

Holzapfel, R. and G. Winterstein (1988). Ada in safety critical applications. In: *Ada in Industry* (S. Heilbrunner, Ed.). Cambridge University Press. pp. 228–248.

ISO (1995). *Ada 95 Language Reference Manual.* ANSI/ISO/IEC-8652:1995.

Kruchten, P. B. (1995). The 4+1 view model of software architecture. *IEEE Software* **12**(6), 42–50.

Locke, D. (1992). Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems* **4**(1), 37–53.

Stroustrup, B. (1991). *The C++ Programming Language.* Second ed.. Addison-Wesley.

Zamorano, J. (1995). Planificación estática de procesos en sistemas de tiempo real crítico. PhD thesis. Universidad Politécnica de Madrid.