Modeling and Schedulability Analysis in the Development of Real-Time Distributed Ada Systems

By: J. Javier Gutiérrez, José M. Drake, Michael González Harbour, Julio L. Medina
Departamento de Electrónica y Computadores
Universidad de Cantabria
Avda. de los Castros s/n, 39005- Santander, SPAIN
{gutierjj, drakej, mgh, medinajl}@unican.es

Abstract[1]

The paper proposes a model for specific Ada structures that can be integrated into our methodology for modeling and performing schedulability analysis in the development phases of distributed real-time applications written in Ada 95 and using its Annexes D and E. This methodology is based on independently modeling the platform, the logical components used, and the real-time situations of the application itself (real-time transactions, workload or timing requirements). The specific models presented in the paper provide support for the automated analysis of local and remote access to distributed services; hence, if a procedure of a remote call interface is invoked from a component assigned to a remote node, the corresponding communication model (with marshalling, transmission, dispatching, and unmarshalling of messages) is implicitly integrated into the overall model that is being analyzed.

Keywords: *Ada, Real-Time, Distributed Systems, Modeling, Schedulability Analysis*

1. Introduction

Several industrial environments such as avionics, space, or robotics consider Ada as a first choice among the programming languages to implement real-time systems, because of its features and reliability. The Real-Time Systems Annex (D) of the standard [1] allows users to develop single-node applications with predictable response times. Furthermore, there are a few implementations of the Distributed Systems Annex (E), that support partitioning and allocation of Ada applications on distributed systems [2]. One of them is GLADE, which was initially developed by Pautet and Tardieu [3] and is currently included in the GNAT project, developed by Ada Core Technologies (ACT) [5]. GLADE is the first industrial-strength implementation of the distributed Ada 95 programming model, allowing parts of a single program to run concurrently on different machines and to communicate with each other. Moreover, the work in [3] proposes GLADE as a framework for developing object-oriented real-time distributed systems.

Unfortunately, Annexes D and E are mutually independent and, consequently, the distributed real-time systems environment is not directly supported in the Ada standard [4]. Our research group has been working on the integration of real-time and distribution issues in Ada 95. We have proposed a prioritization scheme for remote procedure calls in distributed Ada real-time systems [6], and in [7] we focused on defining real-time capabilities for the Ada 95 distributed systems in order to allow the development of this kind of applications in a simpler and potentially more efficient way than with other standards like real-time CORBA.

For the development of distributed applications it is necessary to have strategies for modeling the real-time behavior of the Ada components (reusable modules), and also to have tools for analyzing the schedulability of the entire application. Furthermore, in the real-time distributed applications development, we also need to address issues like the modeling of the communications, or the assignment of priorities to the tasks in the processors and to the messages in the communication networks. On these premises, a methodology for modeling and performing schedulability analysis of real-time distributed applications was proposed in [10]. This methodology is based on modeling basic Ada logical components as building blocks and uses the MAST suite for performing the analysis [8][9].

This paper proposes a model for some of these basic Ada structures that are commonly used in real-time distributed applications. With them, the suitability of MAST to precisely model and analyze real-time distributed Ada 95

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grants TIC99-1043-C03-03 and 1FD 1997-1799 (TAP)

applications is shown. Although we draw out the details of these basic Ada structures, we do not deal with the MAST description neither with the analysis techniques themselves, which can be found respectively in references [8][9][10] and [11][12]. The objective is also to show through several simple examples the feasibility of the methodology to model critical aspects of real-time applications like synchronization between concurrent tasks using protected objects, interrupt handling, and the communication between Ada partitions.

The paper is organized as follows. Section 2 presents the conceptual environment in which real-time analysis and modeling are considered, and summarizes the basic structure of the UML real-time view in which our models are hosted. In Section 3, the structure of the Ada application as a container of the different basic structures is presented. Sections 4 to 6 discuss and justify the feasibility of the approach used for mapping the basic Ada structures into analyzable real-time models. Finally, Section 7 gives our conclusions.

2. Real-Time Analysis and UML RT View

The real-time models are based on concepts and components defined in the Modeling and Analysis Suite for Real-Time Applications (MAST). This suite is still under development at the University of Cantabria [8][9] and its main goal is to provide an open-source set of tools that enable real-time systems designers to perform schedulability analysis for checking hard timing requirements, optimal priority assignment, slack calculations, etc. Figure 1 shows a diagram of the toolset and the associated information. At present, MAST handles single-processor, multiprocessor, and distributed systems based on different fixed-priority scheduling strategies, including preemptive and non-preemptive scheduling, interrupt service routines, sporadic server scheduling, and periodic polling servers.

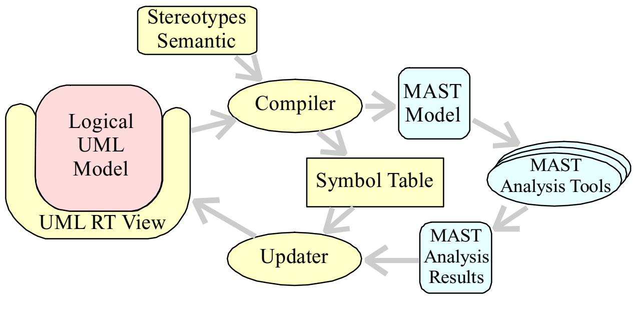


Figure 1. Components of the real-time analysis process

The main goal of the methodology is to simplify the use of well-known schedulability analysis techniques during the object-oriented development of real-time systems with Ada. The methodology extends the standard UML logical description of a system with a real-time model that is defined as an additional view [10]. In the schedulability analysis process, the UML model is compiled to produce a new description based on MAST components. This description includes the information of the timing behavior and of all the interactions among the different components. It also includes the description of the implicit elements introduced by the semantics of the Ada language components that influence the timing behavior of the system. All these elements are specified by means of UML stereotypes. The generated MAST description is the common base on which the real-time analysis toolset may be applied. Finally, the analysis results may be returned into the UML real-time view as a report for the designer. This process is illustrated in Figure 1.

The real time model (UML RT View) is composed of three complementary sections [10]:

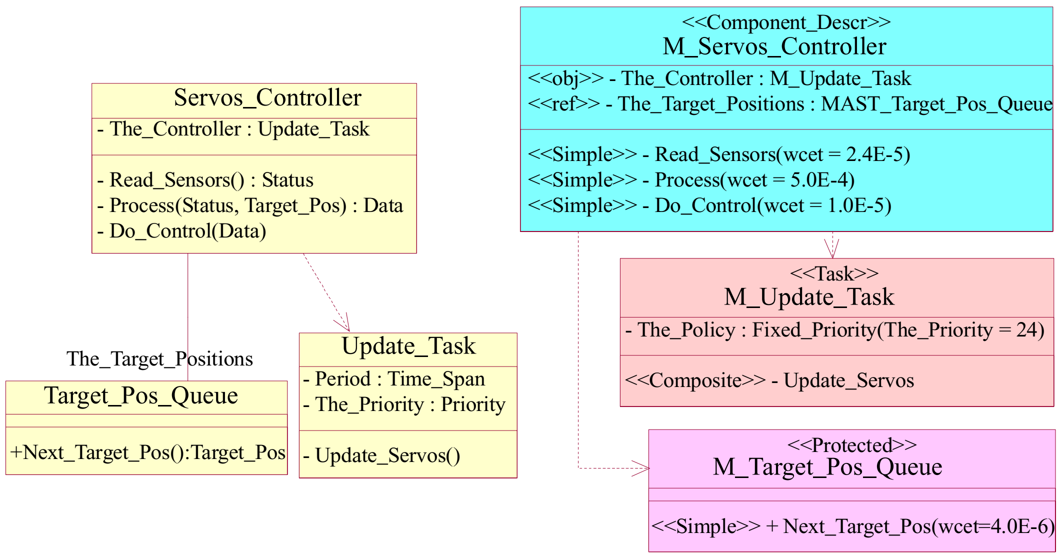
- **The platform model**: it models the hardware and software resources that constitute the platform in which the application is executed. It models the processors, the communication networks, and their configuration.
- **The logical components model**: it describes the real time behavior of the logical Ada components that are used to build the application. A component may model packages (with libraries or tagged type descriptions), tasks, the main procedure of the application, etc.
- **Real-time situation model**: it models a mode of operation of the system and describes the hardware and software components that take part in it, the workload that is required and the timing requirements that are set. The real-time situation is the framework for the schedulability analysis tools operation.

Even though this modeling and schedulability analysis methodology is language independent and is useful for modeling a wide range of real-time applications, the semantics of the high-level modeling components defined and the syntax and naming conventions proposed are particularly suitable and certainly adapted to represent systems conceived and coded in Ada.

3. The Structure of the Ada Application

Instances of the Component class are used to model the real-time behavior of packages and tagged types, which are the basic structural elements of an Ada architecture:

- Each Component object describes the real-time model of all the procedures and functions included in a package or Ada class.



Servos_Controller
- The_Controller : Update_Task
- Read_Sensors() : Status
- Process(Status, Target_Pos) : Data
- Do_Control(Data)
The_Target_Positions
Target_Pos_Queue
+Next_Target_Pos():Target_Pos
Update_Task
- Period : Time_Span
- The_Priority : Priority
- Update_Servos()
<<Component_Descr>>
M_Servos_Controller
<<obj>> - The_Controller : M_Update_Task
<<ref>> - The_Target_Positions : MAST_Target_Pos_Queue
<<Simple>> - Read_Sensors(wcet = 2.4E-5)
<<Simple>> - Process(wcet = 5.0E-4)
<<Simple>> - Do_Control(wcet = 1.0E-5)
<<Task>>
M_Update_Task
- The_Policy : Fixed_Priority(The_Priority = 24)
<<Composite>> - Update_Servos
<<Protected>>
M_Target_Pos_Queue
<<Simple>> + Next_Target_Pos(wcet=4.0E-6)

(a)Logical model of the control pattern. (b)MAST real-time model of the control pattern.

Figure 2. Software pattern of a servos controller artifact

- Each Component object declares all other inner Component objects (package, protected object, task, etc.) that are relevant to model its real-time behavior. It preserves the same visibility and scope of the original Ada structures.

A Component object only models the code that is included in the logical structure that it describes. It does not include the models of other packages or components on which it is dependent.

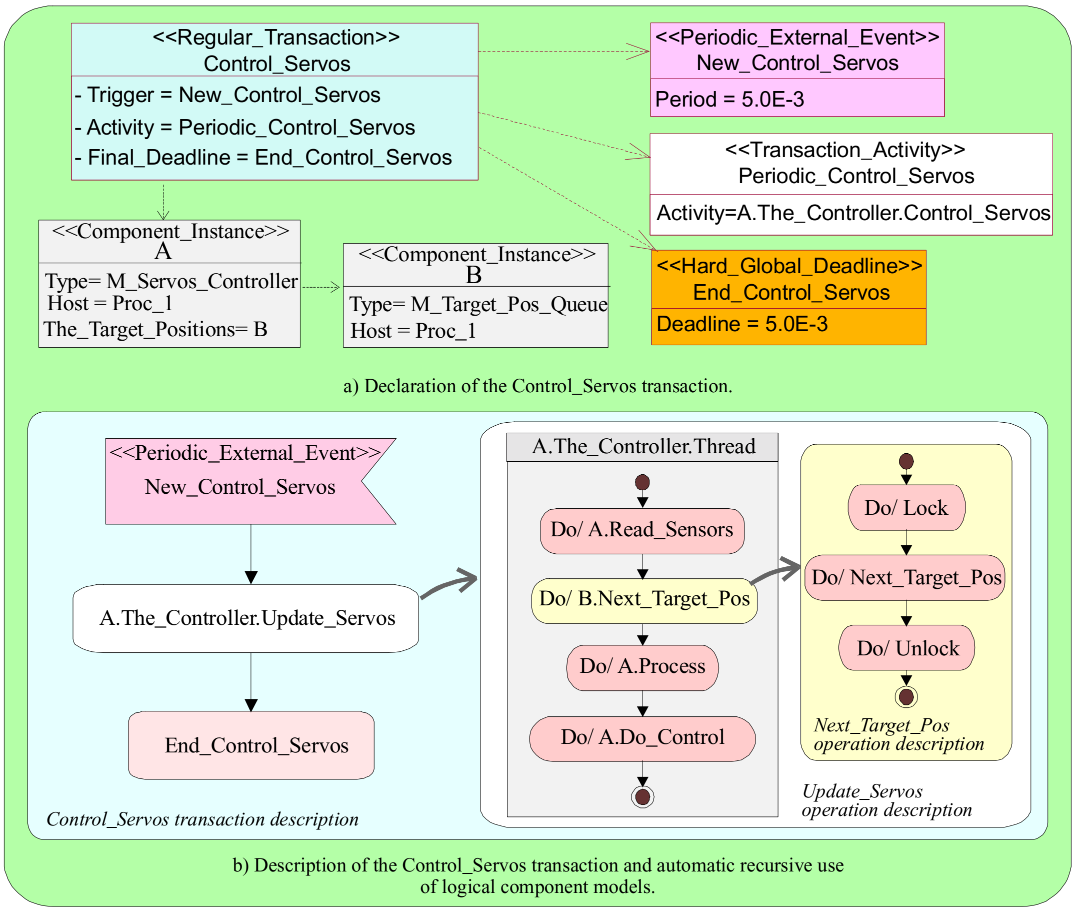
Figure 2 presents the Ada logical model of a simple software pattern and its MAST real-time model, showing the structural parallelism existing between them. This pattern implements the periodic control of a set of servos belonging to a robot or a machine tool (see example in Section 5). It is composed of the active class Servos_Controller and the passive class Target_Pos_Queue, which represents the asynchronous communication mechanism between Servos_Controller and the software components that produce the servos target positions. The class Servos_Controller is a container component designed to group the domain-specific procedures Read_Sensors, Process, and Do_Control, together with an instance of the Periodic_Task class, which concurrently drives the periodic control of the servos. The private procedure Update_Servos holds the sequence of operations that are to be executed in each activation of the periodic task.

Figure 2(b) shows the MAST model corresponding to the control software pattern. The MAST components shown in that figure model the temporal behavior and the interactions between their corresponding logical classes. The M_Servos_Controller component has three Simple operations; each of them models the temporal behavior of its corresponding logic procedure. Their wcet (worst case execution time) argument represents the amount of processing required to complete the operation, and is expressed as a normalized execution time, making its model independent of the processor that executes the code. M_Servos_Controller has two attributes with different stereotypes: Attribute The_Controller has the <<obj>> stereotype, which means that for each instance of M_Servos_Controller, a component of the M_Update_Task type is also instantiated; The_Target_Positions has the <<ref>> stereotype, which indicates that for each instance of M_Servos_Controller a reference to a M_Target_Pos_Queue instance must be specified.

The declaration of the Control_Servos transaction is displayed in Figure 3. This transaction uses an instance of the Servos_Controller pattern, named as A, and it has its The_Target_Positions attribute referencing B, which is an instance of M_Target_Pos_Queue.

4. The Concurrency of Ada Tasks

The <<Task>> components model the Ada tasks. Each task component instance has an aggregated Scheduling_Server, which is associated with the processor where the component instance is allocated. Synchronization between tasks is only allowed inside the operations stereotyped as <<Entry>>. The model implicitly handles the overhead due to the context switching between tasks.



<<Regular_Transaction>>
Control_Servos
- Trigger = New_Control_Servos
- Activity = Periodic_Control_Servos
- Final_Deadline = End_Control_Servos
<<Periodic_External_Event>>
New_Control_Servos
Period = 5.0E-3
<<Transaction_Activity>>
Periodic_Control_Servos
Activity=A.The_Controller.Control_Servos
<<Component_Instance>>
A
Type= M_Servos_Controller
Host = Proc_1
The_Target_Positions= B
<<Component_Instance>>
B
Type= M_Target_Pos_Queue
Host = Proc_1
<<Hard_Global_Deadline>>
End_Control_Servos
Deadline = 5.0E-3
a) Declaration of the Control_Servos transaction.
<<Periodic_External_Event>>
New_Control_Servos
A.The_Controller.Update_Servos
End_Control_Servos
Control_Servos transaction description
A.The_Controller.Thread
Do/ A.Read_Sensors
Do/ B.Next_Target_Pos
Do/ A.Process
Do/ A.Do_Control
Do/ Lock
Do/ Next_Target_Pos
Do/ Unlock
Next_Target_Pos operation description
Update_Servos operation description
b) Description of the Control_Servos transaction and automatic recursive use of logical component models.

Figure 3. Transaction Control_Servos with a model instance of a control pattern

The example described in Figure 2 shows the MAST model of an active Ada class, which declares the task M_Update_Task. For each instance of the task component, a new Scheduling_Server is associated with the corresponding host processor. Hence, in Figure 3 A.The_Controller, which is of the type M_Update_Task, is instantiated together with the instance of the A component. The Scheduling_Server A.The_Controller.Thread is also instantiated with a Fixed_Priority scheduling policy and a priority of 24. Figure 3(b) shows that the activities included in the Update_Servos operation are scheduled within this thread.

5. The Contention in the Access to Protected Objects

A <<Protected>> MAST component models the real-time behavior of an Ada protected object. It implicitly models the mutual exclusion in the execution of the operations declared in its interface, the evaluation of the guarding conditions of its entries, the priority changes implied by the execution of its operations under the priority ceiling locking policy, and also the possible delay while waiting for the guard to become true. Even though the methodology that we propose is not able to model all the possible synchronization schemes that can be coded using protected entries with guarding conditions in Ada, it does allow to describe the usual synchronization patterns that are used in real-time applications. Therefore, protected object-based synchronization mechanisms like handling of hardware interrupts, periodic and asynchronous task activation, waiting for multiple events, or message queues, can be modeled in an accurate and quantitative way.

The operations involved in the declaration of a protected MAST component are implicitly modeled with mutual exclusion between them by attaching an implicit shared resource to them. Each operation in this component implicitly locks and unlocks the shared resource before and after the operation activities. The model of the operation declared as <<Guarded>> is more complex and it needs an activity diagram to describe its behavior.

As an example of the capacity of the MAST methodology to model the real-time behavior of synchronization artifacts

```
protected Synchro is
  pragma Locking_Policy(Ceiling_Locking);
  pragma Priority(28);
  procedure Fire;
  entry Await;
private
  Event: Boolean:=False;
end Synchro;
```

```
protected body Synchro is
  procedure Fire is
  begin
    Event:=True;
  end Fire;
  entry Await when Event is
  begin
    Event:=False;
  end Await;
end Synchro;
```

```
-- Ada task that generates the event
..
Synchro.Fire;
..
```

```
-- Ada task that waits for the event
 ..
 Synchro.Await;
 ..
```

Figure 4. Ada synchronization artifact

using protected objects in Ada, Figure 4 shows the Ada code of one such artifact, in which a task suspends itself executing Synchro.Await until another task activates it by executing Synchro.Fire. The Ada code precisely establishes the semantics of the modeled mechanism.

There is no unique MAST model for the Synchro object. Instead, the model has to be built for each usage pattern. The model described by Figure 5 shows two tasks belonging to the same transaction that synchronize through this object. Consequently they have the same period, and we can assure that there is no more than one task in the Await entry queue. The model shown is built specifically for making the schedulability analysis, so it does not model the actual response but its worst-case behavior instead.

The M_Synchro_1 component declares an attribute of the type Named_State, which is required for synchronizing the operations it offers. The composite operation Fire models the sequence of basic operations that will be executed in the worst case by the calling thread: the code of the Fire operation (Fire_Oper), the evaluation of the Await entry guard (Await.GE), and the execution of the Await code itself (Await.Exec), which will be executed only if there is a task queued in the Await entry. Furthermore, the model establishes that after the execution of the Fire operation the Event state will be reached. As any <<Guarded>> operation, Await is characterized by three elements: the activity model that describes its behavior, the <<Simple>> (or in other cases <<Composite>>) operation that describes the amount of processing required to execute its code (Await.Exec), and the guard evaluation operation that models the amount of processing required to evaluate the guard (Await.GE). The activity description of the Await operation starts by the initial evaluation of the guard, then it suspends until the Event state is reached, and in the end it executes the entry code. The model obtained is pessimistic, because the execution of the Await code is included in both threads. Even though this situation will never take place, it avoids making the model too optimistic, which would yield incorrect results.

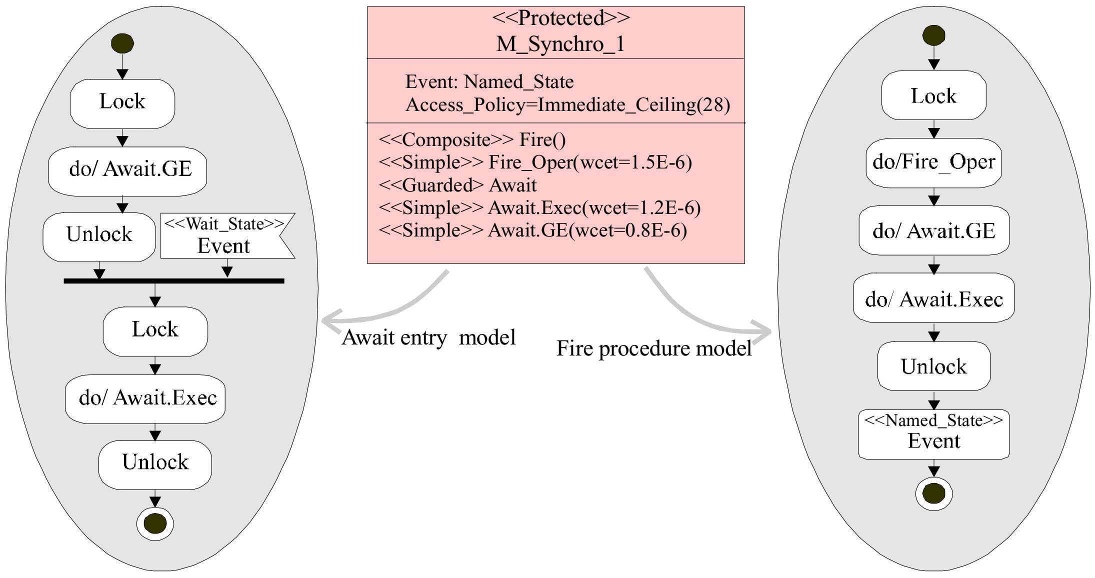


Figure 5. MAST model of an Ada synchronization artifact

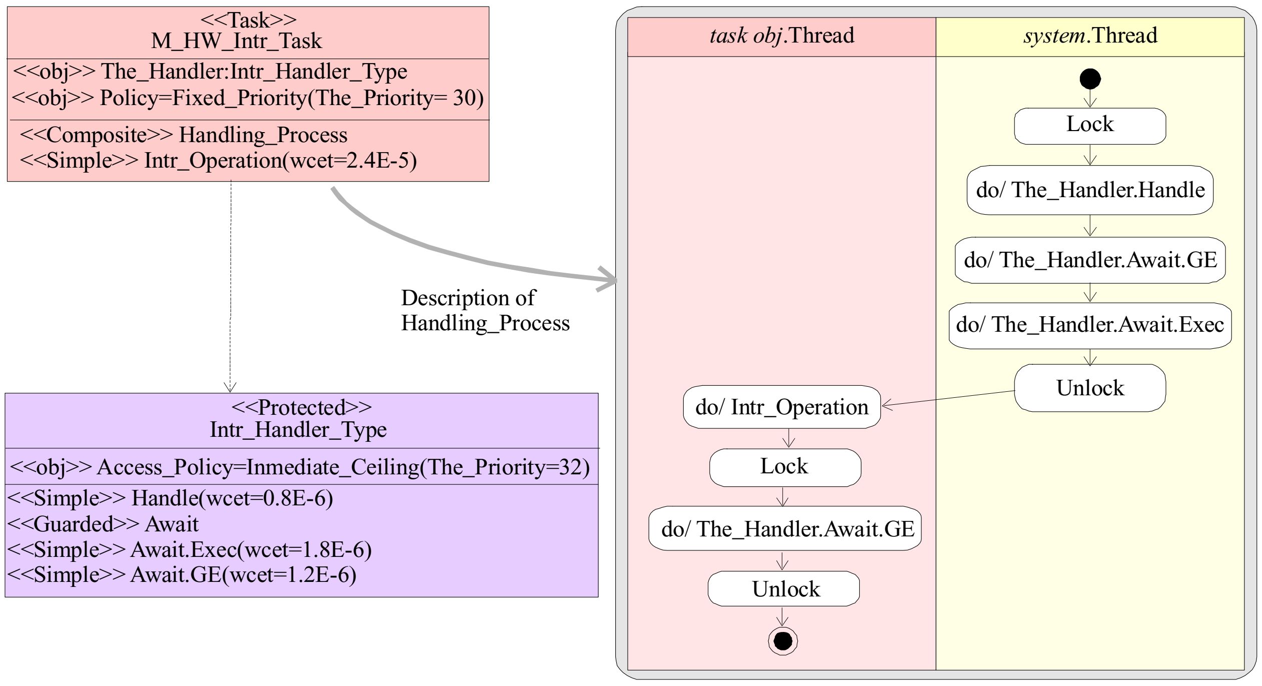
```
task type HW_Intr_Task;

task body HW_Intr_Task is
   The_Handler : Intr_Handler_Type;
   procedure Intr_Operation is
   begin
     --....
   end Intr_Operation;
begin
   loop
      The_Handler.Await;
      Intr_Operation;
   end loop;
end HW_Intr_Task;
```

```
protected  type Intr_Handler_Type is;
    entry Await
private
    procedure Handle;
    pragma
       Attach_Handler(Handle,Ada.Interrupts.names.xxx);
    pragma Interrupt_Priority(32);
    Arrived:Boolean:=False;
end Intr_Handler_Type;
protected type body Intr_Handler_Type is
    entry Await when Arrived is
    begin
      Arrived:=False;
    end Await;
    procedure Handle is
    begin
      Arrived:=True;      --Reset HW Interrupt controller
   end Handle;
end Intr_Handler_Type;
```

Figure 6. Handler of a hardware interrupt

In our second example of an Ada protected object, Figure 6 and Figure 7 describe the Ada code and the MAST model of an asynchronous mechanism for the execution of a procedure triggered by a hardware interrupt, but executed by a software task running at a user-specifiable priority. The actual behavior can be obtained from the analysis of the Ada code shown in Figure 6, and its proposed MAST model is shown in Figure 7. Just like the Ada code, the model has two components. The M_HW_Intr_Task <<Task>> adds to the model a scheduling server in which the Intr_Operation operation is executed. In this case, the operations of the The_Handler object (which is the protected object implementing the actual interrupt handler running at the hardware interrupt priority) are executed by the implicit scheduling server modeling the behavior of the interrupt hardware. This scheduling server (named System.Thread) is declared associated to the processor in the platform model. The operations Lock and Unlock are shown in Figure 7 only for illustrative purposes. They are implicitly added by the <<Protected>> semantics of the Interrupt_Handler_Type component and thus they don't have to be introduced by the modeler. The evaluation of the guard Await.GE is placed after the handling on the interrupt because this will be the last action of the handler task before suspending itself waiting for the next interrupt.

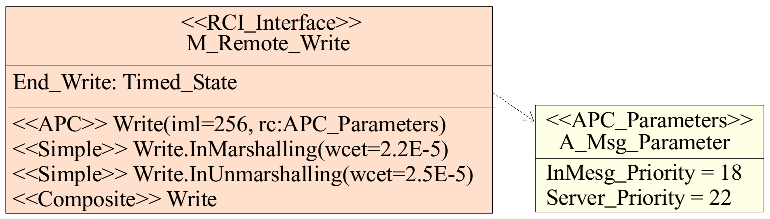


<<Task>>
M_HW_Intr_Task
<<obj>> The_Handler:Intr_Handler_Type
<<obj>> Policy=Fixed_Priority(The_Priority= 30)
<<Composite>> Handling_Process
<<Simple>> Intr_Operation(wcet=2.4E-5)
Description of
Handling_Process
<<Protected>>
Intr_Handler_Type
<<obj>> Access_Policy=Inmediate_Ceiling(The_Priority=32)
<<Simple>> Handle(wcet=0.8E-6)
<<Guarded>> Await
<<Simple>> Await.Exec(wcet=1.8E-6)
<<Simple>> Await.GE(wcet=1.2E-6)
task obj.Thread
system.Thread
Lock
do/ The_Handler.Handle
do/ The_Handler.Await.GE
do/ The_Handler.Await.Exec
Unlock
do/ Intr_Operation
Lock
do/ The_Handler.Await.GE
Unlock

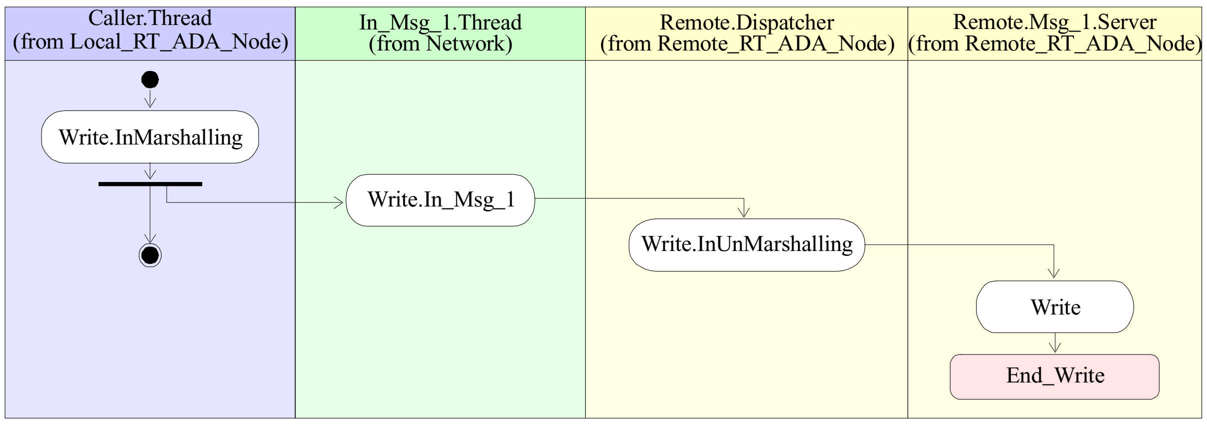
Figure 7. MAST model of the handler of a hardware interrupt

```
package Remote_Write is
   procedure Write(D: in Data_Type);
   pragma Remote_Call_Interface;
   pragma Asynchronous(Write);
end Remote_Write;
```

(a) Ada code of a remote call interface



(b) MAST model of the remote call interface.



(c) Implicit activities diagram for APC Write.

Figure 8. Example of an Ada remote call interface and its MAST model

6. The Real-Time Communication between Ada Distributed Partitions

The model supports in an implicit and automated way the local and remote access to the APC (Asynchronous Procedure Call) and RPC (Remote Procedure Call) procedures of a Remote Call Interface (RCI), as described in Annex E of the Ada standard. The declaration of an RCI includes the necessary information for the marshalling of messages, their transmission through the network, their management by the local and remote dispatchers and the unmarshalling of messages to be able to be modeled and included automatically by the tools.

Figure 8 shows an example of an Ada remote call interface. In order to support an Ada RCI, a platform must have an implementation of the Partition Communication Subsystem (PCS) as it appears in Annex E, and moreover it is necessary to have the System.RPC package, which includes the communication between different active partitions (which in turn can be allocated to different processors), the necessary stubs, and the marshalling and unmarshalling operations for the local and remote processing nodes. In the work presented in this paper, we are modeling an implementation of the Distributed Systems Annex of Ada according to the criteria described in [6] and [7] which allows a predictable scheduling of distributed applications. Processors with this distributed implementation have been modeled by an <<RT_Ada_Processor>> component in the platform model of the system. Each processor of this type has a background thread called Dispatcher and a set of server threads to execute the remote procedures with the scheduling policies and priorities that are requested by the application.

The example in Figure 8 corresponds to an RCI with an asynchronous (APC) operation. The model of the Ada RCI is described by a MAST component with an <<RCI>> stereotype, and each APC operation is modeled by a set of four operations whose names are derived from the APC name:

- The operation with the <<Composite>> (or <<Simple>> in other cases) stereotype models the operation code. It corresponds to the model used when the operation is invoked locally.
- The <<APC>> stereotyped operation declares it as such and brings in the following arguments:
    1. Iml (Incoming message length) expressed in bytes, describes the use of the communication network that is required to send the incoming message.
    2. APC_Parameter.InMsg_Priority is the priority of the incoming message in the communication network.

3. APC_Parameters.Server_Priority is the priority for the execution of the procedure in the remote processor server.

As the last two attributes may vary on each invocation, they must be established as parameters in the MAST model, and their corresponding values must be assigned in the description of the transaction.

- The operation with the InMarshalling extension is simple, and models the amount of processing required for encoding the input arguments of the procedure into the appropriate network format.
- The operation with the InUnMarshalling extension is simple, and models the amount of processing required for decoding the input arguments of the procedure from the network format used.

Figure 8(c) shows the set of activities as well as the scheduling servers involved in the remote invocation of an APC operation (Write). These model elements are automatically included when a component invokes an APC operation of an <<RCI>> component allocated in another processor.

7. Conclusion

In this work we have presented some real-time models of Ada structures in order to integrate them into a methodology for modeling the real-time behavior of Ada applications. These models are specified with the appropriate level of detail to guarantee that, the schedulability analysis, the optimum priority assignment, and the slack calculations, can be applied.

Complex Ada components (packages, tagged types, protected objects, tasks, etc.), as well as other issues like the context switch between tasks, the background communication management, the timer interrupt service routines, the use of mutexes for accessing protected objects, or mechanisms to access remote interfaces (RCI), are modeled independently of the application in which they are used. This property makes it possible to use the Ada components as the basis for the support of a design methodology for real-time systems based on Ada reusable components.

The Ada structures presented in this paper have been integrated in a methodology that is currently being implemented in the UML-MAST toolset. The description and implementation of this toolset can be found at: `http://mast.unican.es`

References

[1] S. Tucker Taft, and R.A. Duff (Eds.) “Ada 95 Reference Manual. Language and Standard Libraries”. International Standard ISO/IEC 8652:1995(E), in Lecture Notes on Computer Science, Vol. 1246, Springer, 1997.

[2] L. Pautet and S. Tardieu, “Inside the Distributed Systems Annex”. Intl. Conf. on Reliable Software Technologies, Ada-Europe’98, Uppsala, Sweden, in LNCS 1411, Springer, pp. 65-77, June 1998.

[3] L. Pautet and S. Tardieu: “GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems. In Proceedings of the 3rd IEEE”. International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'00), Newport Beach, California, USA, March 2000.

[4] Luís Miguel Pinho, "Distributed and Real-Time: Session summary". 10th.Intl. Real-Time Ada Workshop, IRTAW'01, Ávila, Spain, in Ada Letters, Vol.XXI, Number 1, March 2001.

[5] Ada-Core Technologies, Ada 95 GNAT Pro Development Environment. http:// www.gnat.com/

[6] J.J. Gutiérrez García, and M. González Harbour: “Prioritizing Remote Procedure Calls in Ada Distributed Systems”. 9th International Real-Time Ada Workshop, ACM Ada Letters, XIX, 2, pp. 67-72, June 1999.

[7] J.J. Gutiérrez García, and M. González Harbour: “Towards a Real-Time Distributed Systems Annex in Ada”. 10th International Real-Time Ada Workshop, ACM Ada Letters, XXI, 1, pp. 62-66, March 2001.

[8] M. González Harbour, J.J. Gutiérrez, J.C. Palencia and J.M. Drake: “MAST: Modeling and Analysis Suite for Real-Time Applications”. Proceedings of the Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001.

[9] J.L. Medina, M. González Harbour, and J.M. Drake: “MAST Real-Time View: A Graphic UML Tool for Modeling Object-Oriented Real-Time Systems”. RTSS’01, London, December, 2001.

[10]J.L. Medina, J. Javier Gutiérrez, J.M. Drake, and M. González Harbour: “Modeling and Schedulability Analysis of Hard Real-Time Distributed Systems based on Ada Components”. 7th International Conference on Reliable Software Technologies, Ada-Europe 2002, Vienna, Austria, June, 2002.

[11]J.C. Palencia, and M. González Harbour, “Schedulability Analysis for Tasks with Static and Dynamic Offsets”. Proc. of the 19th IEEE Real-Time Systems Symposium, 1998.

[12]J.C. Palencia, and M. González Harbour, “Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems”. Proceedings of the 20th IEEE Real-Time Systems Symposium, 1999.