MAST Real-Time View: A Graphic UML Tool for Modeling Object-Oriented Real-Time Systems

Julio L. Medina Pasaje, Michael González Harbour, José M. Drake
Grupo de Computadores y Tiempo Real, Universidad de Cantabria, SPAIN
{medinajl, mgh, drakej}@unican.es

Abstract[1]

This paper describes a methodology and a framework for building an analyzable real-time model of an object-oriented system that is developed using a UML CASE tool. The real-time model is formulated by a new UML view named "MAST_RT_View". This view allows the designer to gradually build the real-time model according to the phase of the development process, to feed data into the analysis tools, and to bring the relevant timing responses back into the model. The MAST_RT_View has three models: the processing capacity of the hardware/ software platform; the timing behavior of the application logical components; and the workload and the timing requirements of each real-time situation to be analyzed. This view is analyzable by the MAST set of tools, which includes worst-case schedulability analysis and discrete-event simulation for hard and soft timing requirements.

1. Introduction

MAST_RT_View is a new complementary view in the UML description of a system that models its real-time behavior. By using it, the designer is able to build models that can be used to analyze and predict crucial real-time properties of the system. This can be done in each phase of the development cycle, even before it is fully constructed. At the early phases, it allows making qualitative and quantitative analysis of response times using approximate models based on estimations. In the later phases however, it allows making a quantitative and exact validation by using accurate models based on data obtained from the actual execution of the generated code. In accordance to the software architecture "4+1 views" model [1] that is shown in Figure 1, the MAST_RT_View may be considered as an added component to the Process View.

UML-MAST is 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 [4][5] 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. The MAST suite provides abstract components for modeling:

- The hardware (computers, networks, devices, timers, etc.) and software (threads, process, servers, drivers) resources that constitute the platform of the system.
- The logical components (classes, methods, procedures, etc.) and synchronization primitives (mutexes, semaphores, etc.) of the application.
- The real-time situations that describe the system in those modes of operation with real-time constraints.

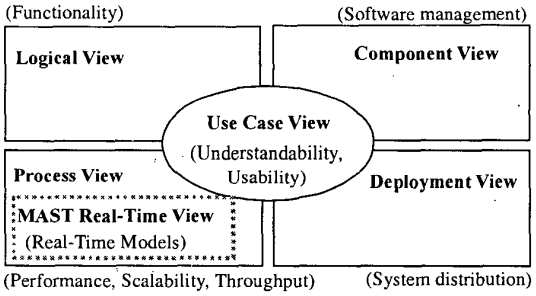

Figure 1. MAST_RT_View in the "4+1" paradigm

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. At its current state, it is able to model most real-time operating systems and languages (like POSIX and Ada).

[1] This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grants TIC'99-1043-C03-03 and 1FD 1997-1799(TAP)

0-7695-1420-0/01 $17.00 © 2001 IEEE

The following tools are already available now (✓) or are under development (-):

- ✓ Holistic analysis
- ✓ Offset-based analysis
- - Varying priorities analysis
- - Multiple event analysis
- ✓ Monoprocessor priority assignment
- ✓ Linear HOPA
- ✓ Linear simulated annealing priority assignment
- - Multiple event priority assignment
- - Monoprocessor simulation
- - Distributed simulation

The main goal of the UML-MAST methodology is to simplify the application of standard and known schedulability analysis techniques to object-oriented real-time systems. In this paper we describe only the main characteristics of the modeling technique. We do not deal with the analysis techniques themselves, which can be found in references [7][8][9][10][11] and [12].

One of our design emphasis has been to make the MAST_RT_View and the Logical View share the same modularity. The MAST_RT_View hosts the real-time model of the system as a whole, the model of each logical class, and the model of each method that is declared in its interfaces. The resulting real-time models of these logical elements are reusable, and currently we are working towards making the UML-MAST technology useful for the specification of real-time software components.

To analyze a real-time system, the designer must build its corresponding MAST_RT_View, and then the MAST set of tools can be invoked. To carry out this process the designer must know the principles of the analysis techniques and the main concepts involved, but with no need to know their theoretical details nor the specific analysis algorithms.

The MAST_RT_View is formulated as proposed by UML revision 1.3 and it is formally defined by the metamodel that is informally described in this paper. MAST_RT_View can be used in combination with most of the standard UML CASE tools. At present, we have developed these three components as well as other useful tools for Rational ROSE2000e [6].

The paper is organized as follows. In Section 2, we describe the basic structure of MAST_RT_View and the main abstract classes of the metamodel that formally describes it. In Section 3, the criteria used for mapping the concepts from the metamodel into UML components are enumerated and justified. Section 4 contains an example of the modeling and analysis of a simple application and shows the usage of a few concrete classes of the metamodel. Section 5 gives a brief description of the main differences between the model proposed in this paper and the "UML Profile for Schedulability, Performance and Time" [2][3] that is currently under development. Finally, Section 6 gives our conclusions.

2. The MAST_RT_View

The MAST_RT_View extends the standard UML description of a system with a real-time model that defines:

- The computational capacity of the hardware and software resources that constitute the platform.
- The processing requirements and synchronism artifacts that are relevant for evaluating the timing behavior of the execution of the logical operations.
- The real-time situations that include the workload and the timing requirements to be met.

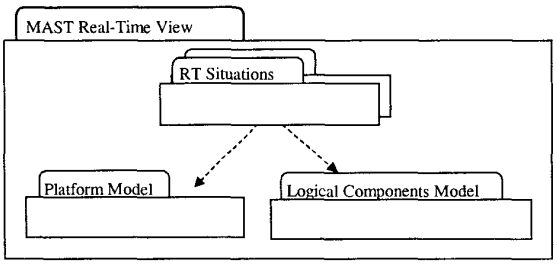

Figure 2. Modules of the MAST_RT_View

The MAST_RT_View collects all the necessary information to perform a schedulability analysis using the MAST automatic tools. It is a collection of UML entities such as packages, objects, instance classes, links, states, and transitions that are declared into several class and activity diagrams. The set of those components and the concrete values of their attributes constitute the real-time model of the system. The UML-MAST metamodel [6] formally documents the UML component types and the relations that can be accepted as valid in the view.

Three complementary sections constitute this view:

- Platform Model: It models the processing capacity and the operative constrains of the hardware and software resources that execute the activities of the application. Some of those resources are: processors, threads, hardware devices, communication networks, etc.
- Logical Components Model: It describes the amount of processing resource capacity that is required for the execution of every functional operation. Examples of these operations are: methods declared in class specifications, synchronization primitives between threads, the data transfer process across a communication network, a specific task carried out by a hardware device, etc. This section of the model also declares the passive resources that the operations need for synchronization. The model of a logical component can be formulated in a generic way. This allows for the

same model to be used to describe its different invocations (potentially in different processing resources) during the application execution.

- Real-Time Situations: An RT_Situation models the system workload and the timing requirements that correspond to a particular execution mode. Every real-time situation is modeled by the set of transactions that concur in that situation. Each transaction is triggered by one or more external events, and represents the sequence of activities that may be executed as a response to them.

2.1. Real-time model of the platform

The RT Platform model describes the processing capacity of the hardware and software resources that execute the activities of the modeled system. The processing capacity available to the application is determined by the tools taking into account the processing capacity of the hardware and all the overheads associated with the different software resources (threads, scheduler, drivers, timers, etc.) running in the background, which must be adequately modeled. Figure 3 shows a portion of the UML-MAST metamodel with the basic abstract classes that are used to model the platform:

- Processing Resource: It models the processing capacity of a hardware component that executes some of the modeled system activities. At present, there are two specialized classes defined: the Processor class models a hardware device (a processor, a co-processor, an instrument, etc.) that executes a piece of application's code; the Network class models a communication system specialized for the transmission of messages among processors. The main attribute of a processing resource is the speed factor, which quantifies its processing capacity in a relative way. Also, it has other attributes that quantify the context switch overheads, the valid priority ranges, etc.

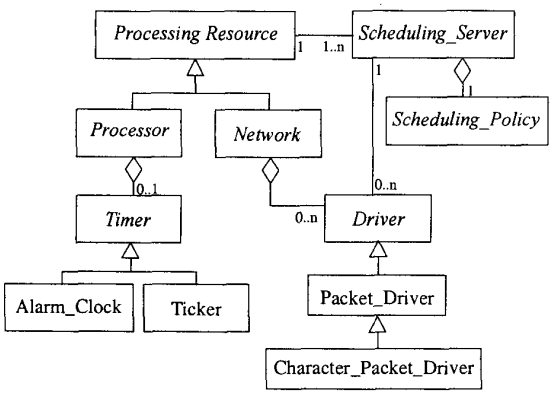

Figure 3. High-level metamodel of the platform

- Timer: It describes the specific features of a timing hardware device associated to a processor. It represents the different overhead models associated with the way the system handles timed events. At present, there are two concrete classes: the Alarm_Clock class models a timer that generates events only when required; the Ticker class models a timer that periodically causes interrupts with the associated overhead.
- Driver: It models the background operations executed in a processor as a consequence of the transmission or reception of a message or part of a message through a network. At present, there are two concrete classes: the Packet_Driver class represents a driver that is activated at each packet transfer (like for the CAN Bus), and the Character_Packet_Driver class models a driver activated by each character transfer (like for a serial port).
- Scheduling Server: It describes a schedulable entity in a processing resource, such as a single-threaded process. Each scheduling server has a link to the processing resource on which its activities are scheduled, as well as an entity that describes the scheduling policy.
- Scheduling Policy: It defines the scheduling policy of a scheduling server. Figure 4 shows the concrete scheduling policies that are usable in the current version. All these policies imply fixed priority scheduling and they are compatible (i.e., the effects of mixing them are predictable) within the same processing resource.

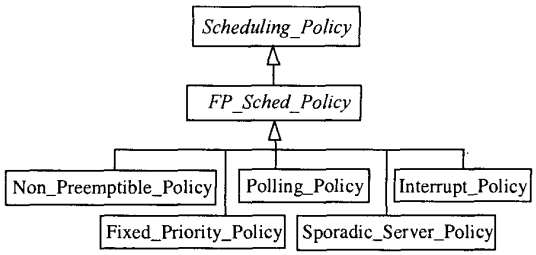

Figure 4. Fixed priority scheduling policy classes

The current version of the UML-MAST metamodel defines a few concrete classes derived from the previously described abstract classes: processor, network, timer, driver, scheduling server and scheduling policy. Each concrete class has a suitable set of attributes that qualify and quantify its behavior in detail.

2.2. Real-time model of the logical components

The Real-Time Model of the Logical Components describes the timing behavior of the functional operations whose execution times are relevant to satisfy the timing requirements defined in the real-time situations of the

system. The operations that are modeled in it can be classified into three types:

- The logical components (methods, procedures and functions) that are defined in the classes' specification of the Logical View of the system. This type of operation corresponds to a piece of code that can be executed by some processing resource.
- The physical transference of a message across a communication network whose timing is relevant to the temporal behavior of the system.
- Predefined tasks which, when invoked by the application, are executed by a co-processor, hardware device or peripheral component and whose timing is relevant to the temporal behavior of the system.

The model of each logical operation describes the two features that determine its temporal behavior:

- The normalized execution time, which is the amount of processing capacity that is required for executing the operation. It is described in a platform-independent way by means of three attributes: the worst-, average-, and best-case normalized execution times.
- The list of passive shared resources that must be accessed in a mutually exclusive way and, therefore, which may cause blocking delays during its execution.

The components of the real-time model in the MAST_RT_View are organized with the same architecture as the components of the Logical View. To make this possible, appropriate directives have been defined to model each component of the logical view in such a way that, if a logical component acting as a client makes use of a server defined as another logical component, the associated real-time model of the client component embodies the real-time model of the server.

The concrete classes used to model the timing behavior of the logical operations derive from the following three classes:

- Shared_Resource: It models a resource that is to be accessed in a mutually exclusive way. It represents an interaction point between concurrent operations and it is a potential cause of blocking delays. Figure 5 shows the shared resource classes that have been defined. The FP_Shared_Resource abstract class models resources that are to be used under some form of fixed priority protocols. There are two concrete classes defined:

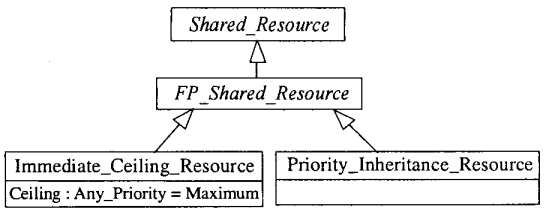

Figure 5. Shared_Resource classes

  - Immediate_Ceiling_Resource: It uses the immediate priority ceiling resource access control protocol.
  - Priority_Inheritance_Resource: It uses the basic priority inheritance access control protocol.
- Operation: It represents a piece of code to be executed by a processor, a task to be carried out by a hardware device, or a message that is sent through a communication network. An operation is always carried out by a single thread and during its execution it does not explicitly synchronize with any other concurrent thread. Of course, the scheduler may delay the execution of an operation if another operation is using the same processor or when access to a busy shared resource is requested. A Composite_Operation contains several other operations, and may have parameters. Each parameter represents a component of the model that is referenced in a symbolic way. When the operation is instantiated (within a transaction, a job or another operation's description), the actual component is assigned to it as the value of the parameter and then the operation model is completed. Figure 6 shows the metamodel section that describes the relevant operation classes. Figures 18 and 19 show some examples of operation declarations.

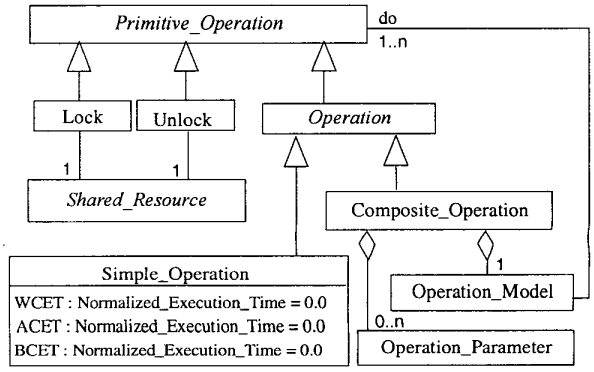

Figure 6. Operation classes

- Job: It models the set of concurrent activities that are carried out when a logical component (such as a method or a procedure) is called. A job can be composed of several activities that are scheduled on several scheduling servers, and may survive beyond the end of its launching method. Jobs are very versatile modeling components. They have been introduced to add modularity to the real-time model. They allow the real-time model of a complex logical procedure to be formulated as a function of the real-time models of the procedures that it invokes during its execution. The inner activities may be executed in several concurrent threads and may have a number of different kinds of synchronization and control flow relationships among them.

A Job is described as a complex structure that includes:
- The set of activities that may be executed when the procedure is invoked.
- The scheduling servers (i.e., threads) in which the activities are scheduled.
- The synchronization artifacts that control the execution of the activities.
- The states at which the execution suspends waiting for specific events to be received.
- The inner states that are relevant for describing the timing of the procedure's execution.
- The state in which control is returned to the launching thread.

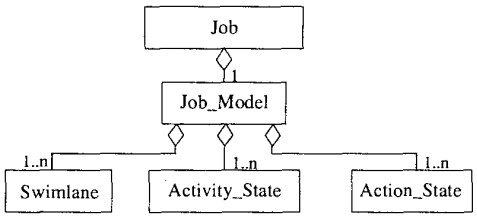

Figure 7. Metamodel of the Job class

The internal structure of a Job is described by means of a set of activity diagrams that as a whole constitute its Job_Model. Inside them, Activity_States represent the operations or subordinate jobs that may be invoked. The vertical Swim Lanes represent the scheduling servers in which the operations or the job's launching procedures are executed. The simple control flow relationships between activities are formulated as transitions and the complex ones, those that imply synchronization among threads or with external events, are formulated by means of control components such as branch, merge, fork, join, and signal receipt. Finally, the internal states that are relevant for the application timing are represented as Action_States and their names identify the associated timing requirements.
In job declarations it is possible to define parameters as typed attributes. A parameter of a Job represents an unassigned component (an operation, scheduling server, external event, timing requirement, etc.) which is used as part of the Job's description. By assigning suitable concrete values to job parameters, we may model the real-time behavior of different job instances using a single job description.

2.3. Real-time situations

Real-time situations are the hardware/software operating modes of the system for which real-time requirements have been defined. Each RT_Situation models a concrete workload of the system in which it is expected to have a specified real-time behavior.

A real-time situation is modeled as a set of Transactions. Each transaction describes the non-iterative sequence of activities that are triggered by a specific pattern of external events. Besides, it serves as the reference frame for specifying the timing requirements of the RT_Situation. Every transaction contains a description of all the activities that are carried out as a consequence of the arrival of an input external event, and also the ways in which they synchronize (by calling each other, signaling, rendezvous, etc.) Other activities that interact with the transaction's activities only by means of shared resources or processing resources are not included in it.

Although transactions are open (non-iterative) sequences of activities, it is usual (particularly in pipeline systems) that streams of external events activate a transaction repeatedly and therefore consecutive execution instances of its activities may overlap in time.

Even though the analysis of a RT_Situation is done independently of each other, it is convenient to include all of them together in the same MAST_RT_View because they share most of the components of the platform and logical models.

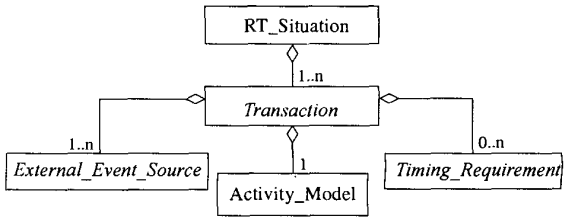

Figure 8. Metamodel of the RT_Situation class

Figure 8 shows the metamodel of an RT_Situation, described as an aggregation of the models of its transactions.

The model of a transaction is composed of a declaration and a description. A transaction is declared by means of an object that is an instance of the Transaction metamodel class. Each transaction object has two aggregated lists. The first list contains the external event sources that define the transaction's input event patterns. The second one contains all the timing requirements that are imposed on its timing behavior. The transaction description is an activity model that is formulated by one or more activity diagrams.

An external event source represents a channel for the arrival of an event stream, through which individual event instances may be generated. The external events model the interactions of the system with hardware timing devices or with external devices that use interrupts, signals, etc. They have a double role in the model: on the one hand they establish the rates and the arrival patterns of activities to be performed by the system. On the other hand, they provide references for defining global timing

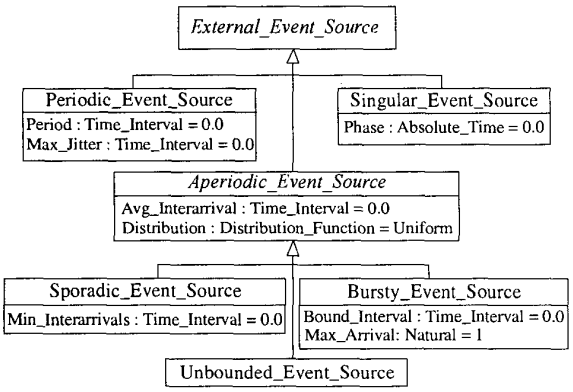

Figure 9. Metamodel of External_Event_Source

requirements. In the current version of the UML-MAST metamodel, a number of different arrival patterns are defined. Figure 9 shows them.

The timing requirements represent the temporal restrictions imposed on the time at which a particular state of the transaction must be reached. The different kinds of requirements that are defined in the current version are shown in Figure 10. A deadline requirement represents a maximum time value allowed for the associated state to be reached. It is expressed as a relative time interval. A local deadline is relative to the beginning of the previous activity, while a global deadline is relative to the external event source that is explicitly linked with it in its declaration. In addition, deadlines may be hard (they must be always met) or soft (they must be met only on average). Other types of timing requirements are the Max_Output_Jitter_Req (which limits the jitter with which the associated state may be reached) and the Maximum_Miss_Ratio (which is a kind of soft deadline that can not be missed more often than a specified ratio).

A transaction is described by means of an Activity Model that is aggregated to its declaration. The activity model is formulated as a set of activity diagrams that describe the sequence of activities (represented by activity states and transitions between them), relevant states (represented by action states) to which timing requirements will be associated, external event sources (represented by stereotyped input event states) and four types of control flow components: Fork, Join, Branch and Merge (represented by the corresponding UML icons). Activity models are also the elements where transaction activities are mapped to the scheduling servers of the platform. The swim lanes of the activity diagrams represent the scheduling servers, and each activity placed on one lane is assigned to the corresponding scheduling server. In Figure 20, you may see an activity diagram that is an example of a transaction description.

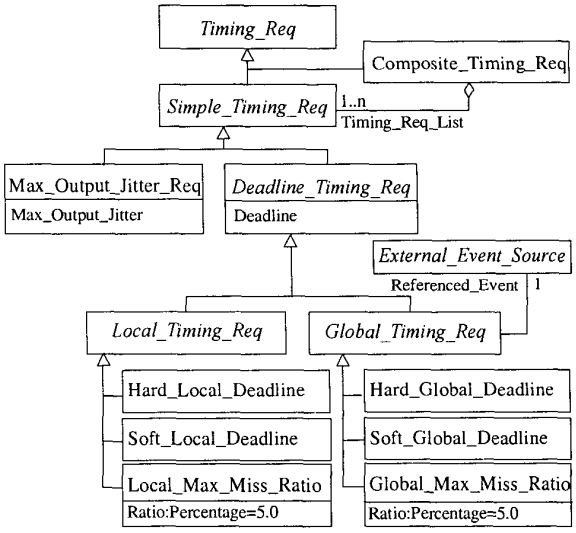

Figure 10. Metamodel of the Timing_Requirement classes

3. Description of the MAST_RT_View on a UML CASE tool

UML will be extended in the future to support real-time models (perhaps as it is proposed in [2]), but UML-MAST can be used by practitioners who need to design real-time systems now, using UML as it stands today (i.e. UML 1.3) and currently available UML tools (like ROSE'2000e). Therefore we have done our utmost to stay within the current UML standard, and not include extensions and alternatives that, while probably beneficial, are not supported by today's UML tools.

The MAST_RT_View is built using a set of objects (and making links between them) that are instances of the classes and associations defined in the UML-MAST metamodel. This is a UML-independent conceptual model with a number of abstract components that may be described in several different ways in the concrete MAST_RT_View models. Therefore, we have chosen a specific mapping between the objects derived from the UML-MAST metamodel and the general UML artifacts that are used in the MAST_RT_View. We use three types of UML diagrams and structures for representing our new view on a standard UML CASE tool:

- The three MAST_RT_View sections and the structure of the RT_Situations that describe the real-time operating modes of the system are incorporated by means of the package structure that is shown in Figure 11. This generic structure must be accurately implemented in order to allow the automatic tools to find the components of the model.

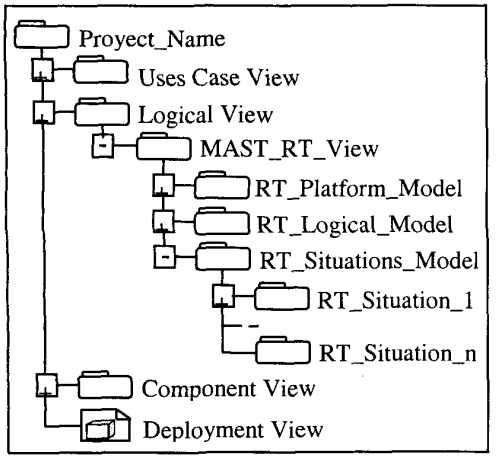

Figure 11. Package structure of MAST_RT_View

- Class diagrams incorporate the declarations of the model components and the relations between them. The classes of these diagrams are instances of the UML-MAST metaclasses. The metaclass from which a given class is derived is described by means of its stereotype. The initial values assigned to its attributes indicate the concrete values of the attributes of that instance and the links between the instanced components of the model are described by means of associations in the class diagrams. In strict UML we should use object diagrams instead of class diagrams for this purpose, but in practice, current UML tools almost never allow you to assign concrete values to object attributes, and this is a fundamental aspect of real-time modeling.
- UML activity diagrams carry out the description of jobs, composite operations and transactions instances. They are aggregated to the class declaring the described instance.

The UML-MAST framework is a "pseudo Add-In" that enhances the use of a UML graphical design and development tool by including a modeling framework that incorporates the package structure of the MAST_RT_View inside the system model. It also configures into the CASE tool the necessary menu options for providing access to the following services:

- Insert and initialize a new MAST_RT_View, generating the package structure in the tool directory.
- Make available the list of stereotypes corresponding to the UML-MAST metamodel classes.
- Install a wizard tool that assists the operator in inserting model components with a minimum of typing.
- Provide a checking tool that makes structural and syntactic analyses of the MAST_RT_View.
- Compile the MAST_RT_View generating the MAST-File description.
- Invoke the MAST toolset and configure the analysis tools to be used.
- Invoke the updater tool that returns the analysis results from the MAST tools to the CASE tool model.

As it is shown in Figure 12, the analysis of the MAST_RT_View is invoked from inside the UML CASE tool. The compiler translates the MAST_RT_View diagrams and data into the input text file compatible with the MAST suite. After the analysis, the updater tool gets the results back from MAST output text file into the MAST_RT_View.

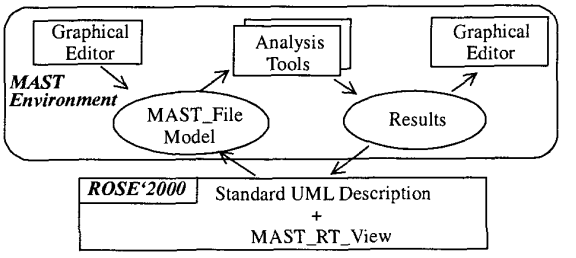

Figure 12. Analysis and simulation tools environment

The UML-MAST Framework has been implemented for the Rational ROSE'2000e UML CASE tool. It has been developed using the available libraries that are supplied with the Rational Rose customization tools.

4. An example of a MAST_RT_View application: Teleoperated robot

The following example shows the application of MAST_RT_View to the real-time modeling and analysis of a simple distributed system for the teleoperated control of a robotized cell [6]. The system platform is composed of two processors interconnected through a CAN bus.

The first processor is a teleoperation station (Station); it hosts a GUI application, where the operator commands the robot and where information about the system status is displayed. The second processor (Controller) is an embedded microprocessor that implements the controller of the robot servos and its associated instrumentation.

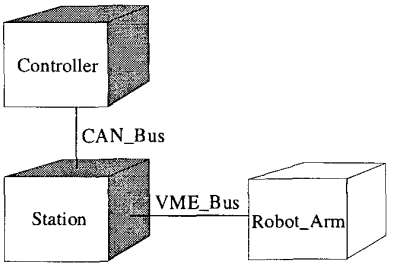

Figure 13. Deployment diagram

4.1. Software architecture: logical design

The software architecture is described by means of the class diagram shown in Figure 14. The software of the Controller processor contains three active classes and a passive one which is used by the active classes to communicate. Servo_Controller is a periodic task that is triggered by a ticker timer with a period of 5 ms. The Reporter task periodically acquires, and then notifies about, the status of the sensors. Its period is 100 ms. The Command_Manager task is aperiodic and is activated by the arrival of a command message from the CAN bus.

The software of processor Station has the typical architecture of a GUI application. The Command Interpreter task handles the events that are generated by the operator using the GUI control elements. The Display Refresher task updates the GUI data by interpreting the status messages that it receives through the CAN bus. Display_Data is a protected object that provides the embodied data to the active tasks in a safe way. Both processors have a specific communication software library and a background task for managing the communication protocol.

Teleoperated_Control is the RT_Situation to be analyzed. It contains three asynchronous transactions. Each one interferes with the others by sharing the processing resources (Station, Controller and CAN_Bus) and by accessing the protected objects. All three transactions have hard real-time requirements.

The Control_Process transaction executes the Control_Servos procedure with a period and a deadline of 5 ms. The Report_Process transaction transfers the sensors and servos status data across the CAN bus, to refresh the display with a period and deadline of 100 ms. Finally, the Command_Process transaction has a sporadic triggering pattern, but its inter-arrival time between events is bounded to 1 s.

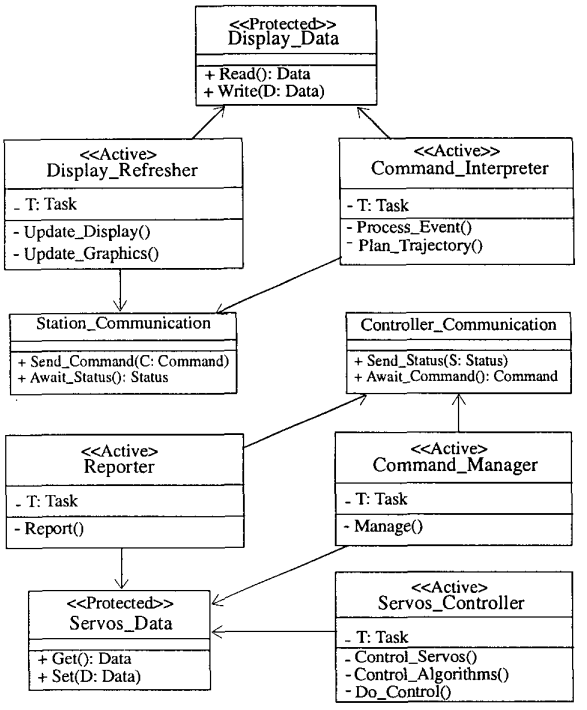

Figure 14. Software architecture of the example

Figure 15 shows the functional description of the Report_Process transaction, using a sequence diagram.

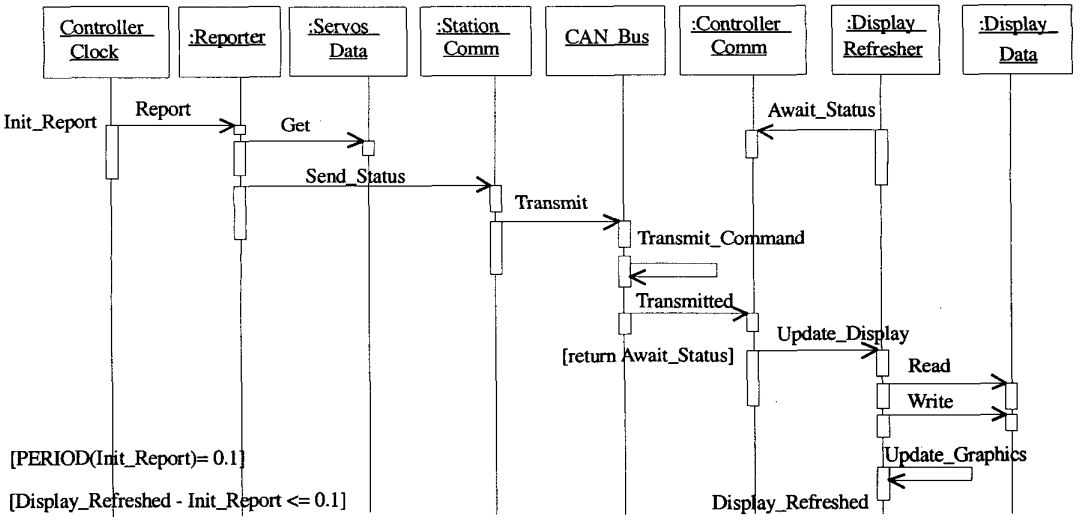

Figure 15. Sequence Diagram of the Report_Process Transaction

4.2. MAST real-time view

As it has been said, the MAST_RT_View has three sections, which will be described next.

4.2.1. Platform model. It describes the processing capacity of the three processing resources of the system: The Station and Controller processors and the CAN Bus network.

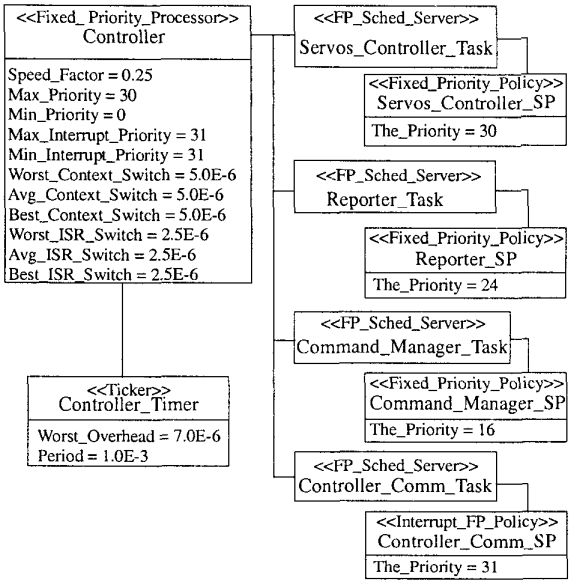

Figure 16. Platform model: the Controller processor

Figure 16 shows the model of the Controller processor and the models of its four threads. Controller has a timer, based on a periodic ticker that is used for timing the periodic tasks declared in its software. It is an embedded processor that executes an application developed in ADA'95 running on the minimum real-time kernel MaRTE OS [13]. The processing capacity of the processor is modeled by the UML-MAST component named Controller that has the Fixed_Priority_Processor stereotype. The activities executed in this processor are executed by four threads. Each of them is modeled by means of an FP_Sched_Server component and the associated FP_Sched_Policy component. The specific values of the Controller processor parameters are:

Max_Priority= 30 Min_Priority= 1	Range of priorities allowed by the Ada compiler and MaRTE OS.
Max_Interrupt_Priority= 31 Min_Interrupt_Priority=31	Range of priorities allowed for hardware-interrupt handlers.
Worst_Context_Switch=5.0E-6 Avg_Context_Switch=5.0E-6 Best_Context_Switch=5.0E-6	Estimated context switch times between the threads of the application.
Worst_ISR_Switch= 2.5E-6 Avg_ISR_Switch= 2.5E-6 Best_ISR_Switch= 2.5 E-6	Estimated context switch times between a thread and an interrupt service routine and viceversa.
System_Timer= "Controller_Timer"; Worst_Overhead = 7.0E-6 Period = 7.0E-6	Link to the timer model.
Speed_Factor= 0.25	The Controller processor has 25% of the processing capacity of the reference processor.

Figure 17 shows the real-time model of the CAN_Bus network. It is a communication channel between the Station and the Controller processors. It is a packet-oriented half-duplex channel. Each packet has a frame header with 47 bits and a data field with 1 to 8 bytes. The bus transfer rate is lower than or equal to 1 Mbit/s.

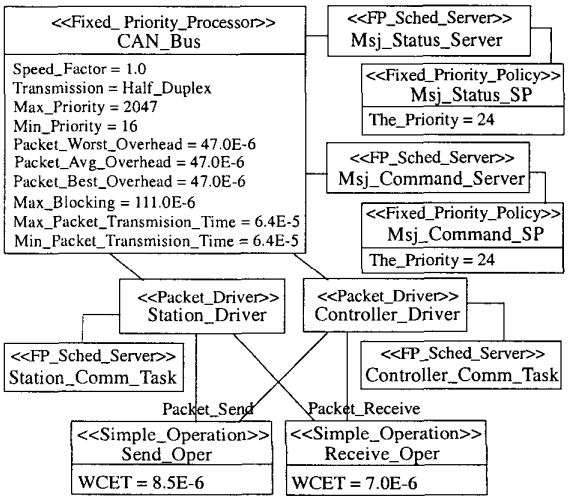

Figure 17. Model of the CAN bus

The packet transfer is prioritized, so a packet is never transferred if its priority is lower than the priority of any other packet that is waiting for being transferred.

The real-time model of the CAN bus is described by means of objects derived from the Packet_Driver, Fixed_Priority_Network and the FP_Sched_Server classes.

The UML-MAST CAN_Bus component has a Fixed_Priority_Network stereotype and it describes the transfer capacity of the channel. The attributes of this object that characterize its real-time behavior are:

Max_Priority= 2047 Min_Priority= 16	Priority range allowed for the messages on the CAN bus.
Packet_Worst_Overhead=47.0E-6 Packet_Avg_Overhead=47.0E-6 Packet_Best_Overhead=47.0E-6	This is the overhead due to the non-data bits of the standard CAN Bus Message Frame format. Non-data bits represent 47 bits per Data Frame (i.e., per packet).
Transmission= Half_Duplex	The transmission mode of the CAN bus is half-duplex.
Max_Blocking= 111.0E-6	The longest network's blocking time due to a packet transfer of maximum length.

Max_Packet_Transmission_Time = 6.4E-5 Min_Packet_Transmission_Time = 6.4E-5	Maximum and minimum times required for the transmission of the data bit field of a single packet frame (8 bytes/packet).
Speed_Factor= 1.0	The transmission time values refer to an implementation of the ISO 11898 standard for CAN operating at 1 Mbit/s.

4.2.2. Real-time model of the operations. The RT Logical Model describes the components that affect the execution time of the operations used in the system. Figure 18 shows the model elements that describe the temporal behavior of the Servos_Data class defined in the logical view.

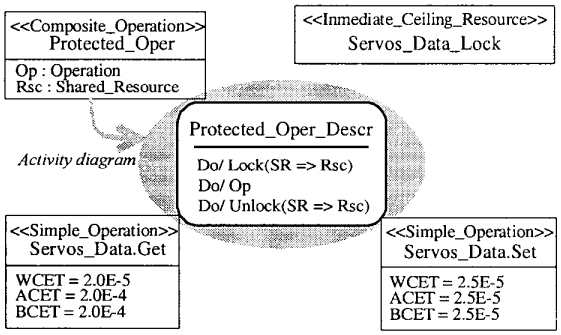

Figure 18. RT-model of the Servos_Data protected object

Servos_Data is an example of a passive class implemented by an Ada protected object. The implicit synchronization primitive of each protected object is modeled by a Shared_Resource component. Protected_Oper is a parametric composite operation that is used for modeling each read/write locking procedure of a protected object. The Op parameter represents the timing model of the executed operation excluding the blocking effects. The Rsc parameter represents the Shared_Resource of the protected object in which the operation is executed. The activity diagram aggregated to the Protected_Oper composite operation describes the sequence of primitive operations that it implies: Lock the shared resource, execute the operation and unlock the resource. Servos_Data_Lock is the shared resource that corresponds to the Servos_Data logical component. The timing behavior of the Get and Set operations is modeled by the corresponding UML-MAST Simple_Operation instances, which describe the required processing of each method. In Figure 19, the three procedures of the Servos_Controller active class are modeled. The Control_Algorithm and Do_Control procedures are simple and only require a declaration. Unlike them, the Control_Servos procedure is a composite operation and it requires a declaration and a description. This description is formulated in the aggregated activity diagram.

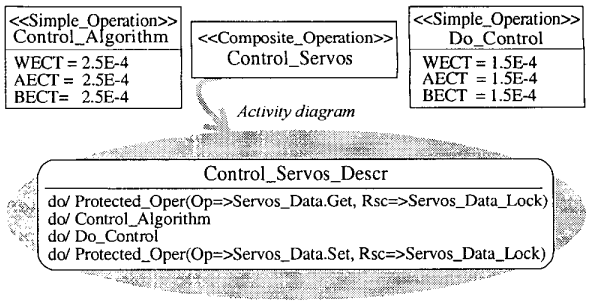

Figure 19. RT-model of the Control_Servos task

4.2.3. Model of the real-time situations. As it has been said, Teleoperated_Control is the only defined real-time situation. It is formulated by means of the three mentioned transactions (Control_Process, Report_Process and Command_Process). Figure 20 shows the model of the Report_Process transaction, whose functional behavior has been described by the sequence diagram in Figure 15.

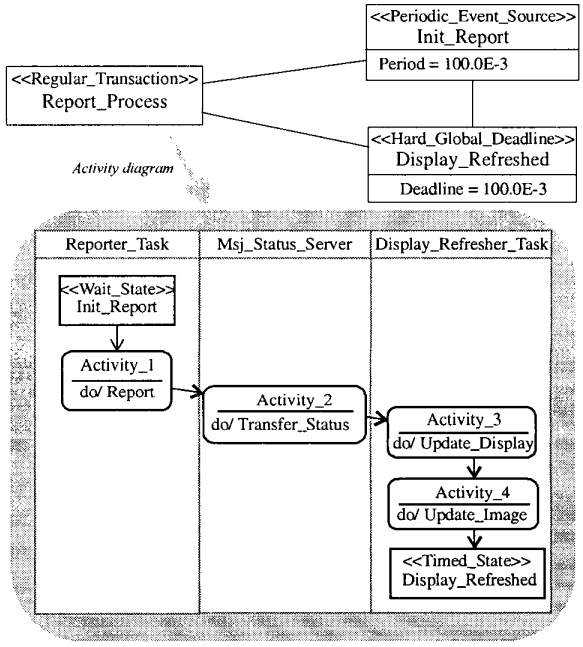

Figure 20. RT_Situation model: Report_Process transaction

The transaction is declared by means of a class diagram, which has the Regular_Transaction instance. It also has an aggregated list with its only input external event (Init_Report), and another list with the only timing requirement (Display_Refreshed) that it has declared. The external event source Init_Report is an instance of the Periodic_Event_Source class and its attribute value indicates that it has a period of 100 ms. The timing requirement Display_Refreshed is an instance of the

concrete class Hard_Global_Deadline and its deadline attribute is also 100 ms. This means that the transaction state Display_Refreshed (notice that the timing requirement and the action state have the same name) must be reached before the beginning of the next cycle.

The transaction is described by means of an activity diagram representing its composing activities and the scheduling servers (i.e., the threads) in which they execute. These scheduling servers are represented by swim lanes.

4.3. Real-time analysis and scheduling design of the system

The Teleoperated robot example has been analyzed using the MAST suite of tools, which let us calculate the blocking times, ceilings for PCP_Resources, transaction slack and system slack. We have chosen the Offset Based Analysis method, which is the least pessimistic. Once the analysis is done, we find out that the Controller processor is not schedulable. At first sight, it seems strange because in the qualitative analysis this processor only has (without blocking phenomena) 47% workload utilization. A more detailed analysis of the Controller processor load shows that the Control_Servo_Trans transaction is the one that uses more processor time and also is the most critical. Although its activities are executed at the highest priority, they present too much preemption and the transaction does not meet its deadline. We find that the main cause of the excessive preemption is the background activities that are executed at the hardware interrupt priority and which can therefore preempt all other transaction activities. An analysis of the driver of the CAN_Bus shows that during the message transfer the driver wastes 31% of the processor time, because in the worst case, each 111 μs interval a packet is received and the driver uses 34 μs managing it.

In this case, the solution is easy because the transfer rate of messages across the CAN bus is very low. We can reprogram the hardware bus controller to reduce the bus transfer speed by a factor of 8 (the initial transmission rate of 1 Mbit/s is reduced to 125 Kbits/s). This reduction does not affect the required communication capacity between the processors but reduces the overhead of the driver on Controller processor by the same factor of 8. In the real-time model this change is achieved by setting the Speed_Factor attribute of CAN_Bus to 0.125. After this redesign the RT_Situation becomes schedulable. The following table shows the most relevant results obtained from the schedulability analysis tools. In this table, we have compared the worst-case response times of each of the three most relevant events of the RT_Situation with their associated deadlines.

Transaction/Event	Slack	Worst resp.	Deadline
Control_Servos_Trans	101.56%		
End_Control_Servos		3.05 ms	5 ms
Report_Process	189.84%		
Display_Refreshed		39.1 ms	100 ms
Execute_Command	186.72%		
Command_Processed		359 ms	1000 ms

Although the overall schedulability is interesting information, it does not tell the designer whether the system is barely schedulable, or it has margin enough for error or change. In order to get a better estimation of how close the system is from being schedulable (or not schedulable), the MAST toolset is capable of providing the transaction and system slacks. These are the percentages by which the execution times of the operations in a transaction can be increased yet keeping the system schedulable. The table of results shows the transaction slacks obtained for the example. There, we can see that the execution time of every activity of the Report_Process transaction can be increased by 189.84% and the system will still be schedulable.

5. UML-MAST and the OMG real-time UML profile

In June of 2001, a group of OMG member companies elaborated a "UML Profile for Schedulability, Performance and Time" [2][3]. Its purpose is making a standard for qualitative and quantitative modeling of real-time systems behavior using object-oriented methods. Its adoption will be useful to facilitate communication of design artifacts between developers in a standard way, to extend the component technology to real-time systems and to enable interoperability among different analysis and design tools. At present, the UML-MAST methodology does not follow the nomenclature and the UML representation rules suggested for that profile, because they have been developed in parallel. Both approaches share the same modeling philosophy and the same domain viewpoint, but there are a number of substantial differences that make the UML-MAST methodology more flexible and capable of modeling more complex systems than the schedulability analysis sub-profile described in [2]. Some of the most important restrictions of this sub-profile are:

- The execution engine is required to be a processor, thus excluding the communication resource and its scheduling properties from the model.
- Each scheduling job is associated with a single execution engine, and thus it cannot model a distributed job.
- Each scheduling job has a single trigger, while the equivalent UML-MAST transactions can be activated

by several external events. Besides, a trigger can only generate events for a single scheduling job.

- Each SAction (roughly equivalent to a piece of a transaction) must be associated with a single Schedulable Resource (equivalent to a scheduling server). This makes it impossible to model multithreaded or distributed methods.
- The timing attribute of an SAction is too limited, because it does not allow expressing variable execution times. It is known that offset-based analysis can take advantage of best-case as well as worst-case execution times [10][11].
- The constructs for event handling among SActions are limited to Join and Fork primitives. UML-MAST also supports Merge and Branch primitives, which can be handled by existing schedulability analysis techniques [9].
- In UML-MAST it is possible to model the resource usage and the timing structure of a method, which can then be used in many different parts of the application, executed with different scheduling parameters or in different execution engines, whereas in the OMG profile the logical and timing structure of that method would need to be replicated for every instance. This independent modeling allows reuse of the model, and is a fundamental feature from a software engineering point of view.

In addition to these restrictions, another important difference is that UML-MAST proposes a new view of the system that contains all the elements that are relevant to the real-time analysis, while the OMG profile inserts all these elements within the different parts of the system description.

The OMG real-time profile is at its elaboration phase and therefore, based upon the experience with UML-MAST, we plan to submit proposals for relaxing some of its restrictions, making it possible to have the same modeling power.

6. Conclusions

The UML-MAST suite provides a methodology for modeling and analyzing real-time systems that are being designed using object-oriented representation techniques.

The UML-MAST methodology is independent of the methodology that is used for designing the application, because it only consists of adding to the representation of the system a new complementary UML view that provides an analyzable real-time model of that system. The main characteristic of this methodology is that it allows modeling each relevant aspect of the system with independent real-time modeling components. Therefore it is possible to independently model the logical operations, the real-time situations, the hardware platform, and the operating system details. Consequently, UML-MAST makes it easy to evolve and refine the system model according to the evolution of the development process.

By adding the MAST_RT_View we are able to use a set of tools that implement the most efficient fixed priority schedulability analysis techniques not only for single-processor systems but also for distributed real-time systems.

The UML-MAST framework developed for Rational ROSE is available as open source software and is distributed under the GNU General Public License. It can be found at: `http://ctrpc17.ctr.unican.es/umlmast`. The MAST toolset, which is usable without ROSE or a UML environment, is also accessible from that page.

7. References

[1] P. Kruchten: "The 4+1 View Model of Architecture". IEEE Software . Nov. 1995.

[2] B. Selic and A.Moore: "Response to the OMG RFP for Scheduling, Performance and Time: Revised Submission". OMG document number: ad/2001-06-14.

[3] B. Selic: "A generic framework for modeling resources with UML". Computer. Vol. 33 nº 6, pp. 64 – 69. June, 2000.

[4] 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.

[5] J.M.Drake, M. Gonzalez Harbour, J.J.Gutierrez and J.C. Palencia: "Description of the MAST Model". http://ctrpc17.ctr.unican.es/mast.

[6] J.M. Drake and J.L. Medina: "UML-MAST Metamodel, specification, example of application and source code". http://ctrpc17.ctr.unican.es/umlmast.

[7] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. González Harbour, "A Practitioner's Handbook for Real-Time Systems Analysis". Kluwer Academic Pub., 1993.

[8] J.W.S. Liu: "Real-Time Systems". Prentice Hall, 2000.

[9] J.J. Gutiérrez García, J.C. Palencia Gutiérrez, and M. González Harbour, "Schedulability Analysis of Distributed Hard Real-Time Systems with Multiple-Event Synchronization". Euromicro Conference on Real-Time Systems, Stockholm, Sweden, 2000.

[10] 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.

[11] 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.

[12] K. Tindell, and J. Clark, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems". Microprocessing & Microprogramming, Vol. 50, Nos.2-3, pp. 117-134, 1994.

[13] M.González Harbour and M. Aldea: "MaRTE OS: An Ada kernel for Real-Time Embedded Applications". Iint. Conf. on Reliable Software Technologies, Ada-Europe'01. Leuven, May, 2000