MAST: Modeling and Analysis Suite for Real Time Applications

By: M. González Harbour, J.J. Gutiérrez García, J.C. Palencia Gutiérrez, and J.M. Drake Moyano
Departamento de Electrónica y Computadores, Universidad de Cantabria
39005 - Santander, SPAIN
{mgh, gutierjj, palencij, drakej}@unican.es

Abstract[1]

This paper describes a model for representing the temporal and logical elements of real-time applications, called MAST. This model allows a very rich description of the system, including the effects of event or message-based synchronization, multiprocessor and distributed architectures as well as shared resource synchronization. The model is directly obtainable from a description of the system design using a UML tool. A system representation using this model is analyzable through a set of tools that has been developed within the MAST suite, including worst-case schedulability analysis for hard timing requirements, and discrete-event simulation for soft timing requirements. Although the current model only includes fixed priority systems, it is conceived as an open model and is easily extensible to accommodate other kinds of systems.

1. Introduction

In this paper we describe the basic characteristics of MAST, a Modeling and Analysis Suite for Real-Time Applications. MAST is still under development and its main goal is to provide an open source set of tools that enables engineers developing real-time applications to check the timing behavior of their application, including schedulability analysis for checking hard timing requirements.

The schedulability analysis techniques have evolved a lot in the past decade, and in particular for fixed priority scheduled systems, such as those built with commercial operating systems or commercial languages. Although fixed priority schedulability analysis techniques were initially developed for single processor systems [7][6], today a full set of techniques exists for distributed real-time systems [9][10][17].

A model for describing real-time applications should represent not only the characteristics of the architecture of the distributed system, but also the hard real-time requirements that are imposed. Most of the existing analysis techniques for the scheduling of distributed hard real-time systems are based on a model that we call *linear*, which is representative of a large number of systems. In the linear model each task is activated by the arrival of a single event or message, and each message is sent by a single task. However, this linear model does not allow complex interactions among the responses to different event sequences, except for the shared resource synchronization, and so, the analysis is not applicable to systems in which these interactions exist.

Many real-time analysis techniques take into account complex synchronization and interactions between event sequences, but for scheduling mechanisms different than fixed priorities. For instance [2] is applied in statically scheduled systems, and [8] refers to the dynamic scheduling mechanisms used in the Spring kernel. The real-time model described in [14] is rather similar to the MAST model, but focuses mainly on the resource and data usage rather than on the relationships among the different processes or tasks in the system; besides, the schedulability analysis techniques used in that paper are not focused on priority scheduling.

The MAST model described in this paper uses a very rich representation of the real time system. It is an event-driven model in which complex dependence patterns among the different tasks can be established [4]. For example, tasks may be activated with the arrival of several events, or may generate several events at their output. This makes it ideal for analyzing real-time systems that have been designed using object-oriented methodologies, and event-driven architectures. Indeed tools have been developed for automatically obtaining a MAST model description of a real-time application from a standard UML

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TIC99-1043-C03-03

0-7695-1221-6/01 $10.00 © 2001 IEEE

description to which a real-time view of the system has been added.

The MAST suite includes schedulability analysis tools that use the latest offset-based techniques [9][10] to enhance the results of the analysis. These techniques are much less pessimistic than previous schedulability analysis techniques [17], which are also included in the toolset for completeness. The toolset also includes tools for assigning optimized priorities, and for the simulation of the timing behavior of the system.

The MAST toolset is open source and is fully extensible. That means that other research teams may provide enhancements. The first versions are intended for fixed priority systems, but dynamically scheduled systems may be added in the future.

The paper is organized as follows. In Section 2 we discuss the main features of the MAST suite. In Section 3 the most relevant aspects of the analysis tools are reviewed, and the current status of the project is described. In Section 4 we describe the general structure and the main elements of the MAST model for representing real-time applications. Section 5 contains an example of the modeling and analysis of a simple application. Finally, Section 6 gives our conclusions and discusses further work.

2. Main Features of MAST

MAST is an open model for describing event-driven real time systems, that is designed to be extensible so that it can support new characteristics or viewpoints of the system. It is designed to handle most real-time systems built using commercial standard operating systems and languages (i.e., POSIX and Ada). This implies fixed priority scheduled systems, but the system will be extended in the future to other scheduling algorithms, such as those with dynamic priorities. Within fixed priorities, different scheduling strategies are allowed, including preemptive and non preemptive scheduling, interrupt service routines, sporadic server scheduling, and periodic polling servers.

The MAST model is designed to handle both single-processor as well as multiprocessor or distributed systems. In both cases, emphasis is placed on describing event-driven systems in which each task may conditionally generate multiple events at its completion. A task may be activated by a conditional combination of one or more events. The external events arriving at the system can be of different kinds: periodic, unbounded aperiodic, sporadic, bursty, or singular (arriving only once).

The system model facilitates the independent description of overhead parameters such as processor overheads (including the overheads of the timing services), network overheads, and network driver overheads. This frees us from the need to include all these overheads in the actual application model, thus simplifying it and eliminating a lot of redundancy.

The model supports both hard and soft timing requirements, because it is very common to find systems with both kinds of requirements. For hard real-time requirements we consider deadlines and maximum output jitter requirements. Among the soft real-time requirements, MAST provides soft deadlines and maximum deadline miss ratios.

3. The MAST Tools

The MAST suite will contain the tools described in Figure 1. The MAST system description is specified through an ASCII description that serves as the input to the analysis tools. A parser converts the ASCII description of the system into a data structure that is used by the tools. The parser has been built using `ayacc` [18], which is an Ada-language equivalent of the popular `yacc` parser generator.

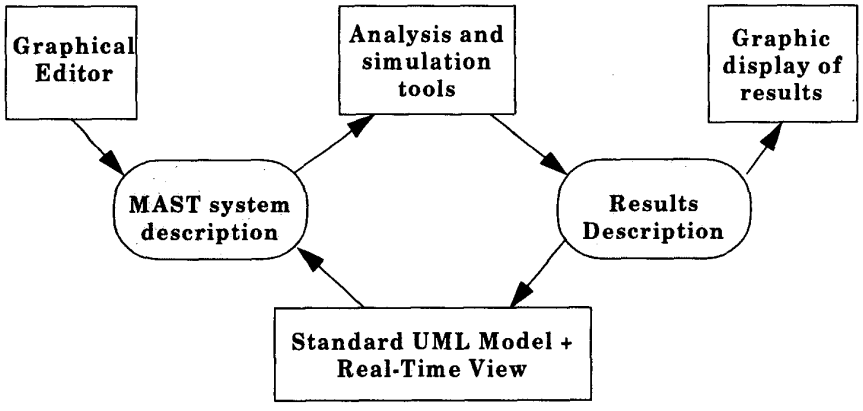

Figure 1. MAST toolset environment

This gives us a high degree of flexibility for adding new capabilities to the description language.

The data structure generated by the parser is built using object-oriented techniques, to make it easily extensible. The analysis tools operate on this data structure and are capable of using different kinds of worst-case schedulability analysis techniques to produce a set of results with the timing behavior of the system. Among the techniques provided we can mention the varying priorities analysis for event-driven single processor systems [6], and the holistic [17] and offset-based techniques [9][10] for multiprocessor or distributed systems. All the techniques used include analysis capabilities for arbitrary deadlines (such as pre- and post-period deadlines), handling of input and output jitter, periodic, sporadic and aperiodic events, and different fixed-priority compatible scheduling policies, such as preemptible and non preemptible, polling servers, sporadic servers, etc.

Blocking times relative to the use of shared resources and non-preemptible sections are calculated automatically. The analysis tool provides the user with capabilities to automatically calculate a suitable optimized set of priorities [3] and priority ceilings (for shared resources), that makes the system schedulable. It also allows checking the possibility of deadlocks.

The timing results of the worst-case analysis tools can be compared against the timing requirements to determine the schedulability. The toolset also includes discrete-event simulation tools that are able to simulate the behavior of the system to check soft timing requirements.

Using a UML tool, it is possible to describe a real-time view of the system [12] by adding the appropriate classes and objects that are necessary to describe the real-time behavior of the system. The application design is linked with the real-time view to get a full description of the system and its timing behavior and requirements. An automatic tool [1] has been developed within the Rose UML CASE tool [11]; it extracts the real-time description of the system from the UML description, generating the MAST description file. No special framework is needed with this approach, but the designer must incorporate the real-time view into the UML description. This methodology follows the approach that is being standardized as a real-time extension to UML [13].

Some of the tools that appear in Figure 1 are not yet available. For example, a graphical editor will be created to generate the system using the MAST ASCII description. This allows users not interested in following the UML approach to use the MAST analysis tools. A graphical display of results will also be available in the future.

The implementation language of the parser and analysis tools is Ada, which we consider best due to its full support of object-oriented features and its built-in constructs that facilitate producing reliable software.

4. The MAST Model

A real-time system is modeled in MAST as a set of transactions. Each transaction is activated from one or more external events, and represents a set of activities that will be executed in the system. Activities generate events that are internal to the transaction, and that may in turn

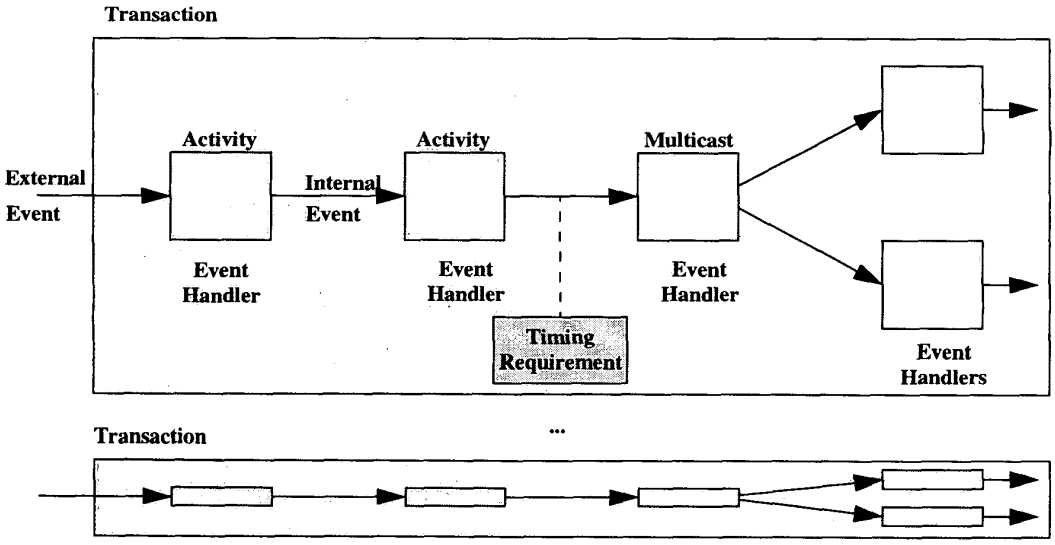

Figure 2. Real-Time System composed of transactions

activate other activities. Special event-handling structures exist in the model to handle events in special ways. Internal events may have timing requirements associated with them.

Figure 2 shows an example of a system with one of its transactions highlighted. Transactions are represented through graphs showing the event flow among the *event handlers*, which are represented as boxes in the graph. This particular transaction is activated by only one external event; after two activities have been executed, a multicast event handling object is used to generate two events that, in turn, activate the last two activities in parallel.

In the MAST model there are two kinds of event handlers:

- The *structural handler* just manipulates events and does not consume resources or execution time; the *Multicast* event handler in the figure above is an example of this kind of handler.
- The *Activity* represents the execution of an operation, i.e., a procedure or function in a processor, or a message transmission in a network.

The elements that define an activity are described in Figure 3. We can see that each activity is activated by one *input event*, and generates an *output event* when completed. If intermediate events need to be generated, the activity would be partitioned into the appropriate parts. Each activity executes an *Operation*, which represents a piece of code (to be executed on a processor), or a message (to be sent through a network). An operation may have a list of *Shared Resources* that it needs to use in a mutually exclusive way.

The activity is executed by a *Scheduling Server*, which represents a schedulable entity in the *Processing Resource* to which it is assigned (a processor or a network). For example, the model for a scheduling server in a processor is a task. A task may be responsible of executing several activities, and thus the associated operations (procedures). The scheduling server is assigned a *Scheduling Parameters* object that contains the information on the scheduling policy and parameters used. Some processing resources may contain references to *System Timers* or *Network Drivers*, which represent various overhead effects in the system.

In the following subsections we review in detail the particular classes and respective attributes, for the different elements currently defined in the MAST model. As we have mentioned, these classes may be easily extended to incorporate other features of real-time systems.

4.1. Processing Resources

They represent resources that are capable of executing abstract activities. This includes both conventional processors and communication networks. Each processing resource is identified through a name. Among its attributes we can mention the range of priorities valid for normal operations on that processing resource, and the speed factor. All the execution times of the operations are expressed

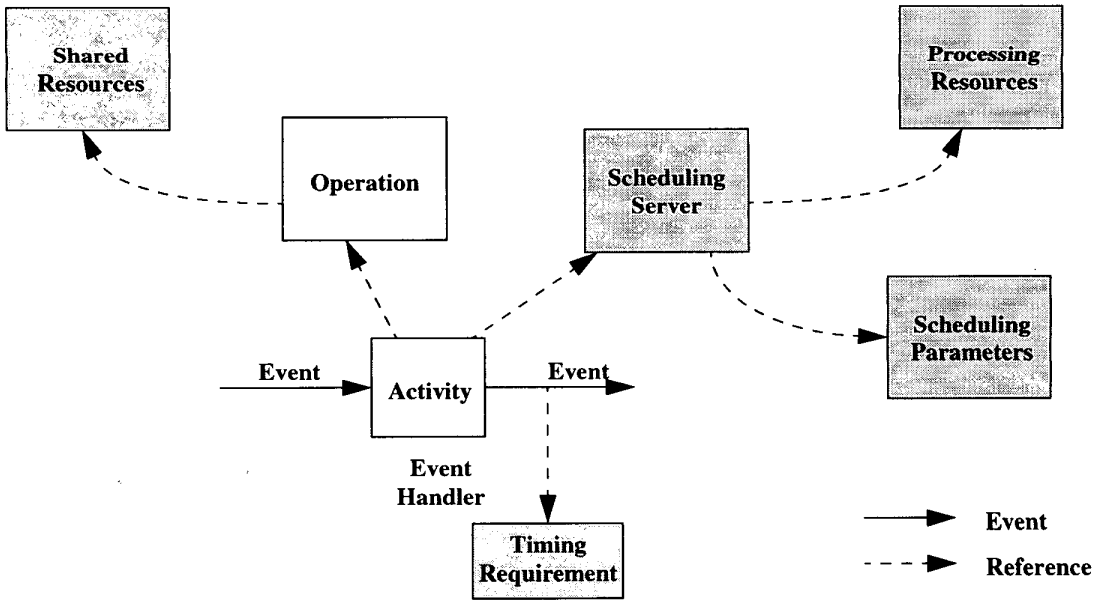

Figure 3. Elements that define an activity

in normalized units. The real execution time is obtained by dividing the normalized execution time by the speed factor.

There are two classes of processing resources currently defined: *Processors* and *Networks*. These are abstract classes and the only concrete classes that are currently defined as extensions of them are, respectively:

- *Fixed Priority Processor.* It represents a processor scheduled under a fixed-priority scheme. In addition to the mentioned attributes, it has: a range of priorities valid for activities scheduled by interrupt service routines; the context switch overheads (worst, average, and best), the interrupt service overheads; and a reference to the system timer (see below) that influences the overhead of the *System Timed Activities* (see Section 4.10).
- *Fixed Priority Network.* It represents a network that uses a priority-based protocol for sending messages. There are networks that support priorities in their standard protocols (i.e., the CAN bus [16], or the token ring [15]), and other networks that need an additional protocol that works on top of the standard ones (i.e., serial lines, ethernet). In addition to the common attributes, it has the following additional attributes: packet send overhead, because of the protocol messages that need to be sent before or after each packet; transmission kind (Simplex, Half Duplex, of Full Duplex); maximum packet transmission time, which represents a blocking time in the overhead model of the network, because packets are assumed to be non preemptible; minimum packet transmission time, which represents the shortest period of the overheads associated to the transmission of each packet; and a list of drivers (see below) that contain the processor overhead model associated with the transmission of messages through the network.

4.2. System Timers

They represent the different overhead models associated with the way the system handles timed events. There are two classes:

- *Alarm Clock.* This represents systems in which timed events are activated by a hardware timer interrupt. The timer is programed to generate the interrupt at the time of the closest timed event. The attributes are the overheads of the timer interrupt.
- *Ticker.* This represents a system that has a periodic ticker, i.e., a periodic interrupt that arrives at the system. When this interrupt arrives, all timed events whose expiration time has already passed, are activated. Other non-timed events are handled at the time they are generated. In this model, the overhead introduced by the timer interrupt is localized in a single periodic interrupt, but jitter is introduced for all timed events, because the time resolution is the ticker period. The attributes are the overheads and period of the ticker interrupt.

4.3. Network Drivers

They represent operations executed in a processor as a consequence of the transmission or reception of a message or a message packet through a network. We define two classes:

- *Packet Driver.* Represents a driver that is activated at each message transmission or reception. Its attributes are: the packet server, which is a reference to the scheduling server that is executing the driver (which in turn has a reference to the processor, and the scheduling parameters); and references to the packet send and receive operations that are executed each time a packet is sent or received, respectively.
- *Character Packet Driver.* It is a specialization of a packet driver in which there is an additional overhead associated to sending each character, as happens in some serial lines. Its attributes are those of a packet driver plus the character server, the character send and receive operations, and the character transmission time.

4.4. Scheduling Parameters

They represent the scheduling policies and their associated parameters. There is an abstract class defined for fixed priority scheduling parameters, for which the common attribute is the priority used for scheduling. The concrete classes defined are:

- *Non Preemptible Fixed Priority Policy.*
- *Fixed Priority Policy.*
- *Interrupt Fixed Priority Policy.* Represents an interrupt service routine.
- *Polling Policy.* Represents a scheduling policy in which there is a periodic server task that polls for the arrival of its input event. Thus, execution of the event may be delayed until the next period. Its additional attributes are the polling period and the polling overhead.
- *Sporadic Server Policy.* Represents the sporadic server scheduling algorithm as defined in the POSIX standard [5]. Its additional attributes are the background priority, the initial server capacity, the replenishment period, and the maximum number of simultaneously pending replenishment operations.

4.5. Scheduling Servers

They represent schedulable entities in a processing resource. If the resource is a processor, the scheduling server is a process, task, or thread of control. There is only one class defined, named *Regular*. Its attributes are the

name, a reference to the scheduling parameters, and a reference to the scheduling resource.

4.6. Shared Resources

They represent resources that are shared among different tasks, and that must be used in a mutually exclusive way. Only protocols that avoid unbounded priority inversion are allowed. There are two classes, depending on the protocol:

- *Immediate Ceiling Resource.* Uses the immediate priority ceiling resource protocol. This is equivalent to Ada's *priority ceiling*, or the POSIX *priority protect* protocol. Its attributes are the name, and the priority ceiling (which may be computed automatically by the tool, upon request).
- *Priority Inheritance Resource.* Uses the basic priority inheritance protocol. Its only attribute is the name.

4.7. Operations

They represent a piece of code to be executed by a processor, or a message that is sent through a network. They all have the following common attributes: execution time (worst, average, and best), in normalized units (for messages, this represents the transmission time); and overridden scheduling parameters, which represents a priority level above the normal priority level at which the operation would execute; the overridden priority is in effect only until the operation is completed.

The following classes of operations are defined:

- *Simple.* Represents a simple piece of code or a message. Additional attributes are: the list of shared resources to lock before executing the operation, and the list of shared resources that must be unlocked after executing the operation. These lists need not be equal.
- *Composite.* Represents an operation composed of an ordered sequence of other operations, simple or composite. The execution time attribute of this class cannot be set, because it is the sum of the execution times of the comprised operations.
- *Enclosing.* As the composite operation, it represents an operation that contains other operations as part of its execution, but in this case the total execution time must be set explicitly; it is not the sum of execution times of the comprised operations, because other pieces of code may be executed in addition. The enclosed operations still need to be considered for the purpose of calculating the blocking times associated with their shared resource usage.

4.8. Events

Events may be internal or external, and represent channels of event streams, through which individual event instances may be generated. An event instance activates an instance of an activity, or influences the behavior of the event handler to which it is directed.

Internal events are generated by an event handler. Their attributes are the name and associated timing requirements imposed on the generation of the event. See the description of the timing requirements below.

External events model the interactions of the system with external components or devices through interrupts, signals, etc., or with hardware timing devices. They have a double role in the model: on the one hand they establish the rates or arrival patterns of activities in the system. On the other hand, they provide references for defining global timing requirements. The following external event classes are defined for representing different arrival patterns:

- *Periodic.* Represents a stream of events that are generated periodically. Its attributes are the period; maximum jitter, i.e., the maximum amount of time that may be added to the activation time of each event instance; and the phase, which is the instant of the first activation if it had no jitter (after that time, the following events are periodic, possibly with jitter).
- *Singular.* Represents an event that is generated only once. Its only attributes are the name and the phase, or instant of the first activation.
- *Sporadic.* Represents a stream of aperiodic events that have a minimum interarrival time. They have the following attributes: minimum interarrival time, which is the minimum time between the generation of two events; the average interarrival time; and the distribution function of the aperiodic events (which can be *Uniform* or *Poisson*).
- *Unbounded.* Represents a stream of aperiodic events for which it is not possible to establish an upper bound on the number of events that may arrive in a given interval. They have the following attributes: average interarrival time, and distribution function.
- *Bursty.* Represents a stream of aperiodic events that have an upper bound on the number of events that may arrive in a given interval. Within this interval, events may arrive with an arbitrarily low distance among them (perhaps as a burst of events). They have the following attributes: bound interval, which is the interval for which the amount of event arrivals is bounded; the maximum number of events that may arrive in the bound interval; the average interarrival time; and the distribution function.

4.9. Timing Requirements

They represent requirements imposed on the instant of generation of the associated internal event. There are different kinds of requirements:

- *Deadlines*. They represent a maximum time value allowed for the generation of the associated event. They are expressed as a relative time interval that is counted in two different ways:
  - *Local Deadlines*: they appear only associated with the output event of an activity; the deadline is relative to the arrival of the event that activated that activity.
  - *Global deadlines*: the deadline is relative to the arrival of a *Referenced Event* that is an attribute of the deadline.

  In addition, deadlines may be hard or soft:
  - *Hard Deadlines*: they must be met in all cases, including the worst case
  - *Soft Deadlines*: they must be met on average.

  This gives way to four kinds of deadlines:
  - *Hard Global Deadline*. Attributes are the value of the *Deadline*, and a reference to the *Referenced Event*.
  - *Soft Global Deadline*. Attributes are the value of the *Deadline*, and a reference to the *Referenced Event*.
  - *Hard Local Deadline*. The only attribute is the value of the *Deadline*.
  - *Soft Local Deadline*. The only attribute is the value of the *Deadline*.
- *Max Output Jitter Requirement*: Represents a requirement for limiting the jitter with which a periodic internal event is generated. Output jitter is calculated as the difference between the worst-case response time and the best-case response time of the activity that generates the associated event, relative to a *Referenced Event* that is an attribute of this requirement. Consequently, the attributes are the maximum output jitter, and the referenced event.
- *Max Miss Ratio*: Represents a kind of soft deadline in which the deadline cannot be missed more often than a specified ratio. Its attributes are the deadline and the ratio, or percentage representing the maximum ratio of missed deadlines. There are two kinds of Max Miss Ratio requirements: global or local.
- *Composite*: An event may have several timing requirements imposed at the same time, which are expressed via a composite timing requirement. It contains just a list of simple timing requirements.

4.10.Event Handlers

Event handlers represent actions that are activated by the arrival of one or more events, and that in turn generate one or more events at their output. There are two fundamental classes of event handlers. The *Activities* represent the execution of an operation by a scheduling server, in a processing resource, and with some given scheduling parameters. The other operations are just a mechanism for handling events, with no runtime effects. Any overhead associated with their implementation is charged to the associated activities. Figure 4 shows the different classes of event handlers.

- *Activity*. It represents an instance of an operation, to be executed by a scheduling server. Its attributes are its input and output events, the reference to the operation, and the reference to the scheduling server (which in turn

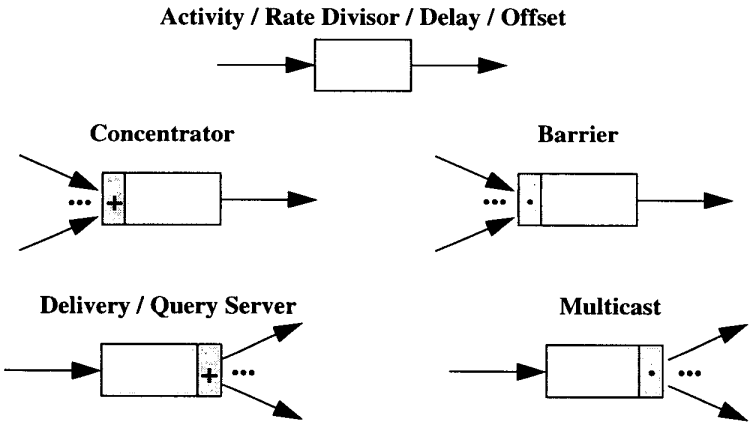

Figure 4. Classes of Event Handlers

contains references to the scheduling parameters and the processing resource). See Figure 3.

- *System Timed Activity.* It represents an activity that is activated by the system timer, and thus is subject to the overheads associated with it. It only makes sense to have a *System Timed Activity* that is activated from an external event, or an event generated by the *Delay* or *Offset* event handlers (see below). It has the same attributes as the regular activity.
- *Concentrator.* It is an event handler that generates its output event when any one of its input events arrives. Its attributes are its input and output events.
- *Barrier.* It is an event handler that generates its output event when all of its input events have arrived. For worst-case analysis to be possible it is necessary that all the input events are periodic with the same periods. This usually represents no problem if the concentrator is used to perform a "*join*" operation after a "*fork*" operation carried out with the *Multicast* event handler (see below). Its attributes are its input and output events.
- *Delivery Server.* It is an event handler that generates one event in only one of its outputs each time an input event arrives. The output path is chosen at the time of the event generation. Its attributes are its input and output events, and the delivery policy, which is used to determine the output path. It may be *Scan* (the output path is chosen in a cyclic fashion) or *Random*.
- *Query Server.* It is an event handler that generates one event in only one of its outputs each time an input event arrives. The output path is chosen at the time of the event consumption by one of the activities connected to an output event. Its attributes are its input and output events, and the request policy, which is used to determine the output path when there are several pending requests from the connected activities. It may be *Scan* (the output path is chosen in a cyclic fashion), *Priority* (the highest priority activity wins), *FIFO* or *LIFO*.
- *Multicast.* It is an event handler that generates one event in every one of its outputs each time an input event arrives. Its attributes are its input and output events.
- *Rate Divisor.* It is an event handler that generates one output event when a number of input events equal to the *Rate Factor* have arrived. Its attributes are its input and output events, and the rate factor, which is the number of events that must arrive to generate an output event
- *Delay.* It is an event handler that generates its output event after a time interval has elapsed from the arrival of the input event. Its attributes are its input and output events and the longest and shortest time intervals used to generate the output event.
- *Offset.* It is similar to the *Delay* event handler, except that the time interval is counted relative to the arrival of some (previous) event. If the time interval has already passed when the input event arrives, the output event is generated immediately. Its attributes are the same as for the *Delay* event handler, plus the reference to the appropriate event.

4.11.Transactions

The transaction is a graph of event handlers and events, that represents interrelated activities that are executed in the system. A transaction is defined with three different components: a list of external events, a list of internal events (with their timing requirements if any), and a list of Event handlers.

In addition, each transaction has a *Name* attribute. There is only one class of transaction defined, called *Regular* transaction.

5. An Example

The following example will show the aspect of the MAST file format that has been chosen to represent the timing behavior of real-time applications. The example is a simplification of the control system of a teleoperated robot. This is a distributed system with two specialized nodes: a local robot controller, and a remote teleoperation station,

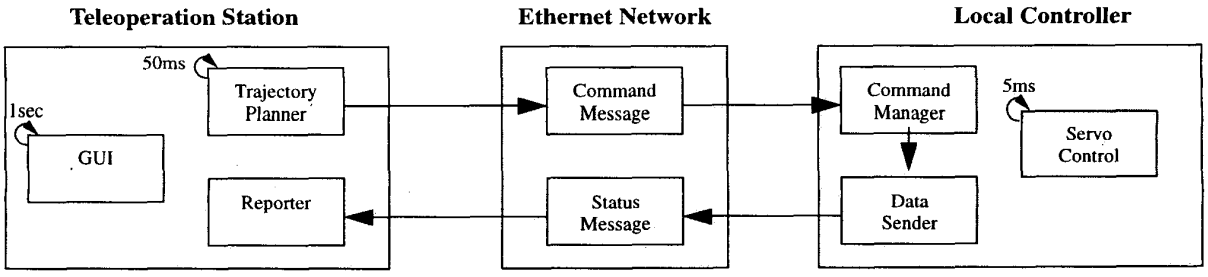

Figure 5. Architecture of the teleoperated robot controller

where the operator manipulates the controls, and gets information about the system status. Figure 5 shows a diagram of the software architecture. The system has three transactions; one of them, the main control loop, implies execution in different processing resources, and has a global end-to-end deadline. Communication is through an ethernet network used in master-slave mode to achieve hard real-time behavior.

Some parts of the MAST description of the system follow:

```
-- Processing Resources
Processing_Resource (
   Type => Fixed_Priority_Processor,
   Name => Local_Ctler,
   Worst_Context_Switch => 15,
   System_Timer =>
      Type => Alarm_Clock,
      Worst_Overhead => 10));
...
Processing_Resource (
   Type => Fixed_Priority_Network,
   Name => Ethernet,
   Transmission => Half_Duplex);

-- Scheduling Servers
Scheduling_Server (
   Type => Fixed_Priority,
   Name => Servo_Control,
   Server_Sched_Parameters => (
      Type => Fixed_Priority_Policy,
      The_Priority => 415),
   Server_Processing_Resource=> Local_Ctler);
...
-- Shared Resources
Shared_Resource (
   Type => Immediate_Ceiling_Resource,
   Name => Servo_Data);
...
-- Operations
Operation (
   Type => Simple,
   Name => Read_Servos,
   Worst_Case_Execution_Time => 74,
   Shared_Resources_List => (Servo_Data));
Operation (
   Type => Enclosing,
   Name => Servo_Control,
   Worst_Case_Execution_Time => 1019,
   Composite_Operation_List =>
      (Read_Servos,Write_Servos));
Operation (
   Type => Simple,
   Name => Command_Message,
   Worst_Case_Execution_Time => 4850);
...
-- Transactions
Transaction (
   Type => Regular,
   Name => Servo_Control,
   External_Events => (
      (Type => Periodic,
       Name => E1,
       Period => 5000)),
   Internal_Events => (
      (Type => Regular,
       Name => O1,
       Timing_Requirements => (
          Type => Hard_Global_Deadline,
          Deadline => 5000,
          Referenced_Event => E1))),
   Event_Handlers => (
      (Type => System_Timed_Activity,
       Input_Event => E1,
       Output_Event => O1,
       Activity_Operation => Servo_Control,
       Activity_Server => Servo_Control)));
...
```

In the MAST description we can see that we declare, in this order, the processing resources, the scheduling servers, the shared resources, the operations, and finally, the transactions. The timing requirements are embedded in the events described in the transactions. The timers (and also the network drivers) are embedded in the description of the processing resources. The scheduling parameters are embedded in the description of the scheduling servers. Finally, the events and event handlers are embedded in the description of the transactions.

If we feed the MAST worst-case analysis tools with this description we obtain the results of the worst case response times of all the activities, which we can compare with the deadlines to determine the system schedulability.

Although the overall schedulability is an interesting piece of data, it does not tell the designer whether the system is barely schedulable, or has enough margin for error or change. In order to get a better estimation of how close we are to being schedulable (or unschedulable), the MAST toolset is capable of providing the transaction and system slacks. These are the percentages by which the execution times of the operations of a transaction (or of the system) can be increased while keeping the system schedulable (or decreased to make it schedulable if it wasn't). Table 1 shows the slack times obtained for the example, using both the holistic and offset-based analysis tools. We can see that the execution times can be increased by 34.28% and the system would still be schedulable with the offset-based analysis. The results of the holistic analysis are much

Table 1. Slacks calculated for the example

Slack for:	Holistic (%)	Offset-Based (%)
Transaction Servo Control	-100.00	204.69
Transaction Main_Loop	-24.22	47.66
Transaction GUI	-100.00	434.38
Whole system	-21.09	38.28

worse, and the execution times would need a reduction of 21.09% to achieve schedulability according to that technique. Since both techniques obtain upper bounds for the worst-case response time, we can discard the results of the holistic analysis and assure that the system is schedulable.

6. Conclusions and Further Work

The MAST suite defines a model capable of describing the timing behavior of a large set of real-time systems, including distributed systems and event-driven systems with complex synchronization schemes. The model is appropriate for UML system descriptions in which a real-time view of the system is added to the design, and then an automatic tool is used to generate the MAST description. This description, in text format, is used as the input to the MAST tools for hard and soft real-time analysis, that incorporate the latest developments in scheduling theory. Currently MAST is defined for fixed priority systems, but has been designed to be easily extensible to other kinds of systems, such as those scheduled with dynamic priority policies.

The MAST suite is still under development and there are some missing tools: worst-case analysis for systems with multiple-event synchronization, calculation of the possibility of deadlocks, event-driven simulation, and the graphical editor are not yet available. But the current set of tools is already useful to a large number of applications, and the rest of the tools will be available soon.

There is still some scheduling theory needed to eliminate some of the restrictions, and we plan to work on these in the near future. The most important missing pieces are: the calculation of remote blocking times in distributed systems when the blocking terms depend on each other; and the enhancement of the multiple-event analysis, to make it less pessimistic.

The MAST suite is open source software, distributed under the gnu license. It can be found at:
`http://ctrpc17.ctr.unican.es/mast`

References

[1] J.M. Drake, M. González Harbour J.L. Medina: "MAST Real-Time View: Graphic UML tool for modeling object-oriented *real time systems.*" Group of Computers and Real-Time Systems. University of Cantabria (Internal report), 2000.

[2] G. Fohler, "Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems". 16th Real-Time Systems Symposium, Pisa, Italy, 1995.

[3] J.J. Gutiérrez García and M. González Harbour. "Optimized Priority Assignment for Tasks and Messages in Distributed Real-Time Systems". Proceedings of 3rd Workshop on Parallel and Distributed Real-Time Systems, Santa Barbara, California, pp. 124-132, 1995.

[4] 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.

[5] IEEE Standard 1003.1d:1999, "Information Technology - Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]. Additional Realtime Extension". The Institute of Electrical and Electronics Engineers, 1999.

[6] 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.

[7] C.L. Liu, and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". Journal of the ACM, 20 (1), pp 46-61, 1973.

[8] L. Molesky, K. Ramamritham, C. Shen, J. Stankovic, and G. Zlokapa, "Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel". IEEE Workshop on Real-Time Operating Systems and Software, 1990.

[9] 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.

[10]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.

[11]T. Quatrany: "Visual Modeling with Rational ROSE 2000 and UML" Addison Wesley Longman, Reading, Mass., 2000.

[12]B. Selic: "A Generic Framework for Modeling Resources with UML". IEEE Computer, Vol. 33, N. 6, pp. 64-69. June, 2000.

[13]B. Selic, A. Moore, M. Bjorkander, M. Gerhardt, and B. Watson: "Response to the OMG RFP for Schedulability, Performance and Time" OMG document n. Ad/2000-08-04. August, 2000.

[14]A.D. Stoyenko, T.J. Marlowe, and P.A. Laplante, "A description Language for Engineering of Complex Real-Time Systems". Real-Time Systems Journal 11(3) 1996.

[15]J.K. Strosnider, T. Marchok, J.P. Lehoczky, "Advanced Real-Time Scheduling Using the IEEE 802.5 Token Ring". Proceedings of the IEEE Real-Time Systems Symposium, Huntsville, Alabama, USA, pp. 42-52, 1988.

[16]K. Tindell, A. Burns, and A.J. Wellings, "Calculating Controller Area Network (CAN) Message Response Times". Proceedings of the 1994 IFAC Workshop on Distributed Computer Control Systems (DCCS), Toledo, Spain, 1994.

[17]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.

[18]`http://www.ics.uci.edu/~arcadia/Aflex-Ayacc/aflex-ayacc.html`