MAST: An Open Environment for Modeling, Analysis, and Design of Real-Time Systems

M.González Harbour, J.L.Medina, J.J.Gutiérrez, J.C.Palencia, and J.M.Drake

Departamento de Electrónica y Computadores, Universidad de Cantabria
Av. de los Castros s/n 39005 - Santander, SPAIN
{mgh, medinajl, gutierjj, palencij, drakej}@unican.es

ABSTRACT[1]

This paper describes the basic characteristics of MAST, a Modeling and Analysis Suite for Real-Time Applications. It is an environment that offers an open set of tools for modeling, analysis and design of real-time systems that is not oriented to any concrete design methodology. The MAST suite provides components for modeling the hardware resources like processors, networks, devices, or timers, and software resources like threads, processes, servers, or drivers, that constitute the platform of the system; the logical components of the application, i.e., classes, methods, or procedures, and synchronization primitives like mutexes, semaphores, or monitors; and finally the real-time situations that describe the system workload and the timing requirements that correspond to a particular execution mode. 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. 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 through future tools such as a discrete-event simulation for soft timing requirements. Likewise, the suite provides a set of software libraries for managing the model, building new analysis and design tools and storing and displaying the results generated with them. Also high-level UML-based representation techniques have been developed and are shown briefly.

1. INTRODUCTION

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 [14][10][13][9][5], today a full set of techniques exists for distributed real-time systems [17][18][24][7].

MAST defines an open model for describing event-driven real time systems [4], 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 modeling methodology 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 the user 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.

The MAST suite includes schedulability analysis tools that use the latest offset-based techniques [17][18] to enhance the results of the analysis. These techniques are much less pessimistic than previous schedulability analysis techniques for

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

distributed systems [24], which are also included in the toolset for completeness. The toolset also includes tools for assigning optimized priorities. In the future, it will include support for the simulation of the timing behavior of the system.The MAST toolset is open source (under the GNU general public license) and is fully extensible. That means that other research teams may provide enhancements. The first versions are intended for fixed priority systems, but support for dynamically scheduled systems is planned for the near future.

In order to overcome the complexity of the modeling activities, profiles for higher levels of abstraction are defined for some specific software methodologies or environments. The higher-level modeling components defined in each profile, implement the semantics of the specific methodology abstractions, and by using them the real-time model is not only more easily built, but closer to the development environment.

The paper is organized as follows. In Section 2 we describe the general structure and the main elements of the MAST model for representing real-time applications. In Section 3 we discuss the main components of the MAST suite, we review the most relevant aspects of the analysis tools, and we describe the current status of the project. In Section 4 we define profiles for higher levels of modeling abstraction, for some specific software methodologies or environments such as object-oriented, Ada, or component-based applications. In Section 5 we present some limitations found in the UML profile for performance schedulability and time proposed to the OMG for standardization [21]. Finally, Section 6 gives our conclusions.

2. 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 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.

Fig. 1 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.

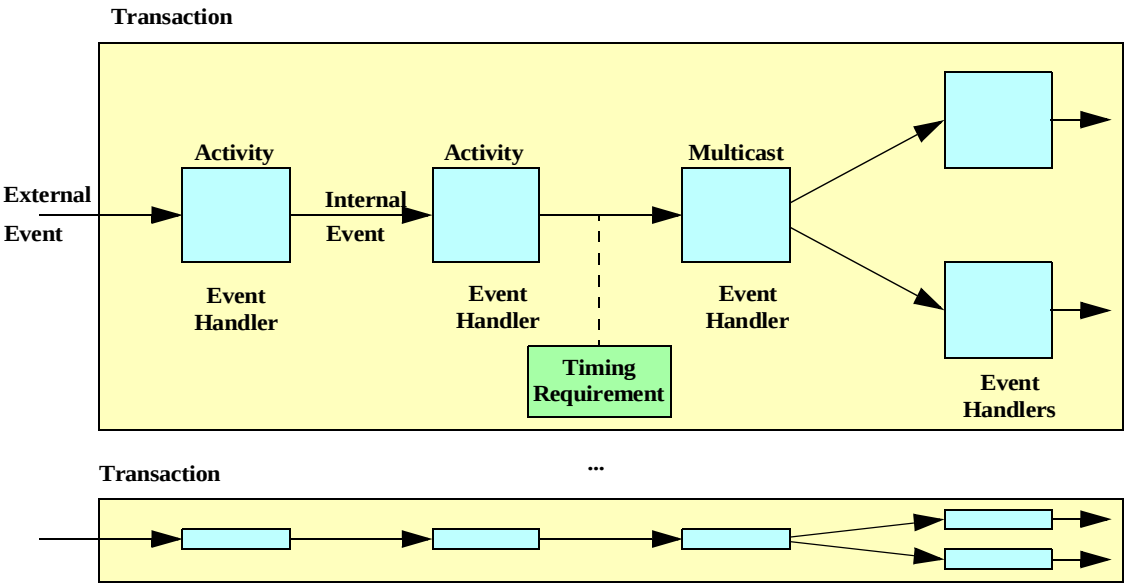

Fig. 1. Real-Time System composed of transactions

- 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 Fig. 2. 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.

2.1. 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 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:

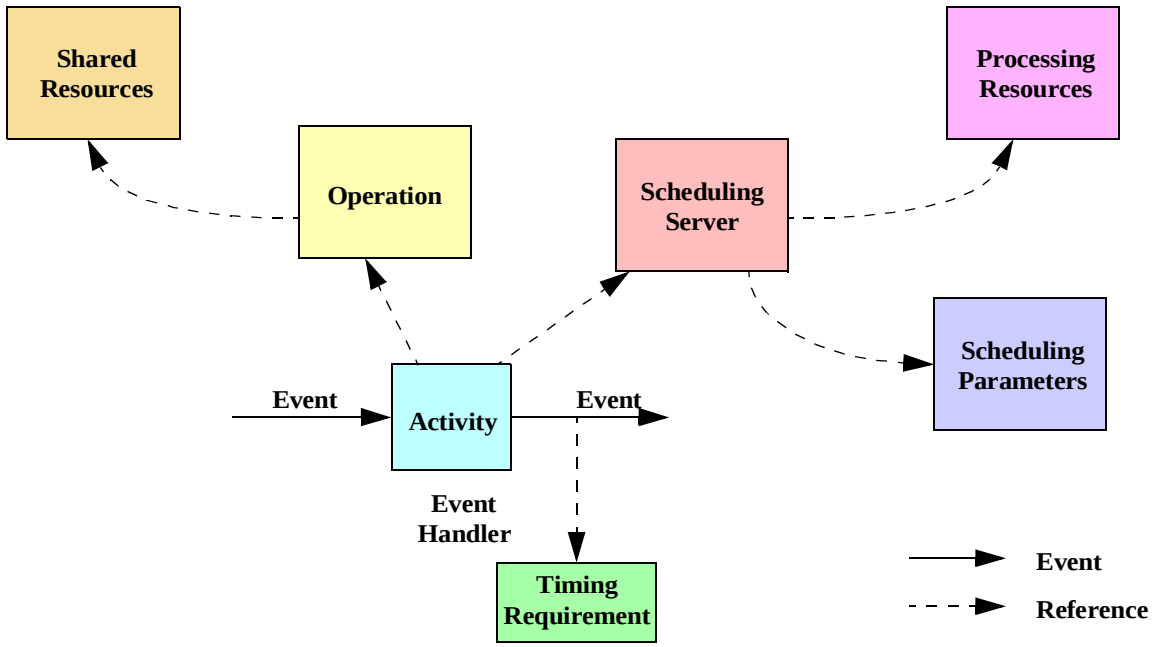

Fig. 2. Elements that define an activity

- *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 2.10).
- *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 [23], or the token ring [22]), 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.

2.2. 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.

2.3. 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.

2.4. 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 [1]. Its additional attributes are the background priority, the initial server capacity, the replenishment period, and the maximum number of simultaneously pending replenishment operations.

2.5. 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.

2.6. 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.

2.7. 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 without any contention or protocol overheads); and overridden scheduling parameters, which represents a priority level above the normal priority level at which the operation would execute; the regular overridden priority is in effect only until the operation is completed, but there is also a permanent overridden priority that remains in effect until the end of the activity or until explicitly changed.

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.

2.8. 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 inter-arrival 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.

2.9. 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.

2.10. 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. Fig. 3 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 contains references to the scheduling parameters and the processing resource). See Fig. 2.
- *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

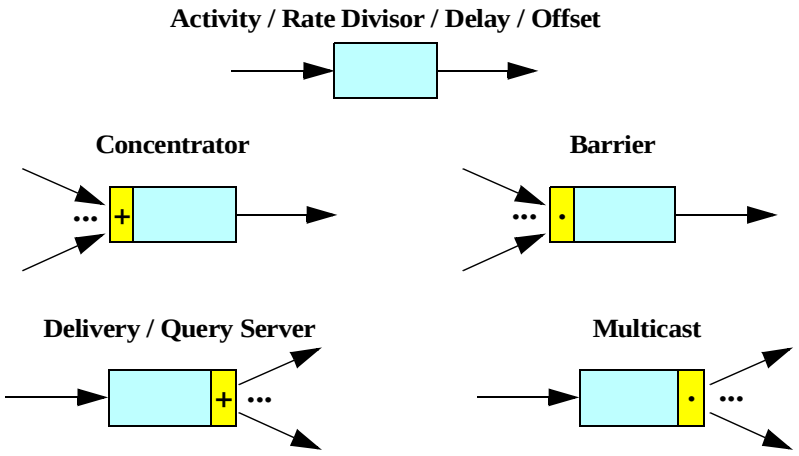

Fig. 3. Classes of Event Handlers

- *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.

2.11. 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.

3. THE MAST TOOLS

The main components of the MAST suite appear in Fig. 4. 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` [2], which is an Ada-language equivalent of the popular `yacc` parser generator. 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.

Before describing the tools that we can apply to analyze the system model, we must consider the different kinds of systems supported by the tools. Restrictions and consistency are checked in order to invoke the tools properly. MAST distinguishes between the following kinds of systems according to the criteria imposed by the algorithms that are implemented by the tools:

- Single-Processor vs. Multi-Processor. Some tools support just one processing resource, and others support several processing resources, including multiprocessor and distributed systems.
- Referred to the structural architecture of the application, the systems can be classified as follows. Each category includes the previous ones:

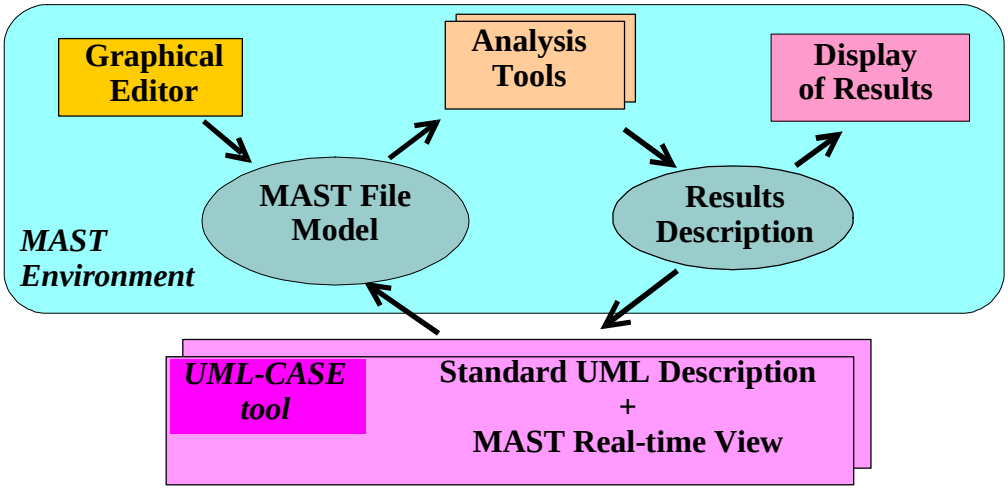

Fig. 4. MAST toolset environment

- Simple Transaction Only: the system is composed of transactions with only one event handler holding an activity. In this case we distiguish between activities with or without permanent overridden priorities, respectively called many priorities or one priority transaction.
- Linear Transactions: the system contains transactions built with linear event handlers, including Rate Divisors, Offsets, and Delays.
- Non-Linear Transaction: the transactions of the system can include any of the event handlers, including multiple-event triggering.

The set of analysis tools is capable of using the following kinds of worst-case schedulability analysis techniques to produce a set of results with the timing behavior of the system:

- Non-Linear. This technique is currently under implementation and can be applied to Multi-Processor and Non-Linear Transaction systems [7]. It is based on transforming the system into an equivalent model on which the existing RMA techniques for analyzing distributed real-time systems can be applied.
- Offset Based. It corresponds to the technique presented in [18], and it is applicable to Multi-Processor and Linear Transaction systems.This is the best technique available at the moment for linear distributed systems.
- Offset Based Unoptimized. As the previous technique, it is applicable to Multi-Processor and Linear Transaction systems. It was presented in [17]. This technique obtains better results than the Holistic analysis, but does not fully exploit the benefits of the precedence constraints in the transaction, as opposed to the full offset-based technique.
- Holistic. This is the classic technique for distributed systems [24], that support Multi-Processor and Linear Transaction systems. Although it is clearly less accurate than the offset-based techniques, we provide it for completeness and comparison purposes.
- Varying Priorities. The technique developed in [5] can be applied to Single-Processor and Simple Transaction systems, in which there may be permanent overridden priorities. It supports tasks that are decomposed into several serially executed sub-tasks of different priorities.
- Classic Rate Monotonic. This is the classic response time analysis developed by Joseph and Pandya [9], and then extended to arbitrary deadlines by Lehoczky [13]; it supports only Single-Processor and Simple Transaction systems with no permanent overridden priorities.

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. There is an option to automatically calculate the priority ceilings for shared resources of type *Immediate_Ceiling_Resource*. Table 1 shows a summary of the level of support of each tool.

Table 1: Kinds of systems supported by the different analysis tools

Technique	Single-Processor	Multi-Processor	Simple Trans., one priority	Simple Trans, many priorities	Linear Trans.	Non-Linear Trans.
Classic Rate Monotonic	☑		☑			
Varying Priorities	☑		☑	☑		
Holistic	☑	☑	☑	☑	☑	
Offset Based Unoptimized	☑	☑	☑	☑	☑	
Offset Based	☑	☑	☑	☑	☑	
Non-Linear	☑	☑	☑	☑	☑	☑

The MAST toolset provides an option to automatically calculate an optimized set of priorities. MAST includes three priority-assignment algorithms [6]:

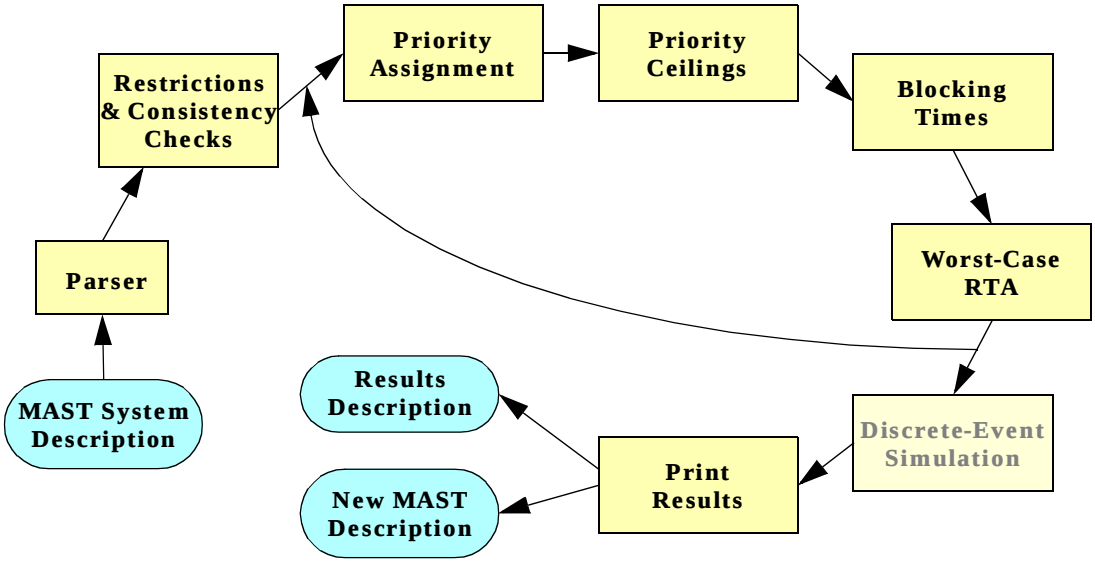

Fig. 5. Internal steps in the schedulability analysis tools

- Single processor. If deadlines are at or before the end of the period, a deadline-monotonic assignment is made.
- HOPA. It is an optimization algorithm based on the distribution of the end-to-end or global deadlines of each transaction among the different actions that compose that transaction.
- Simulated Annealing. It is a general optimization technique for discrete functions that had been previously used for solving similar problems.

All the priority assignment techniques operate by iterating over the analysis. Fig. 5 shows the different steps performed by the tools, where this iteration is made explicit.

The timing results of the worst-case analysis tools can be compared against the timing requirements to determine the schedulability.

Finally, there is an interesting option in the MAST schedulability analysis tools, related to the calculation of slacks. We can calculate slacks of isolated operations, transactions, processing resources, and the overall system. The slack is a number with the following meaning: if positive, it is the percentage by which all the execution times of all the operations involved in the real-time situation (operation, transaction, processing resource, or system) may be increased while still keeping the system schedulable; if negative, it is the percentage by which these execution times have to be decreased to make the system schedulable; if zero, it means that the system is just schedulable.

The toolset will include in the near future a discrete-event simulation tool that will be able to simulate the behavior of the system to check soft timing requirements.

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.

As it is described in the next section, using a UML tool it is possible to describe a real-time view of the system [15] 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 [15] has been developed within the Rose UML CASE tool [19]; 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.

4. HIGHER-LEVEL MODELING PROFILES

As seen in the previous sections, the MAST environment offers a complete set of primitive modeling components. They are simple and very close to the hardware and software artifacts that are relevant to the timing behavior of the system. For a moderately complex system the model is composed of hundreds or thousands of primitive components, which means that validating the model can become quite difficult. In order to overcome the modeling complexity, profiles for higher levels of modeling abstraction have been defined for some specific software methodologies or environments. The higher-level modeling components defined in each profile implement the semantics of the specific abstractions of the associated methodology, and by using them the real-time model is not only more easily built up but it is also closer to the development environment. The usage of CASE tools adapted to the composition of models from the modeling components defined in these profiles, have proved to be useful in building the MAST analysis model of the system automatically. At present there are two profiles already defined:

- UML_MAST: Profile for modeling real-time systems that are designed using object-oriented representation techniques and UML-CASE tools. It includes a CASE tool support for building models and generating the MAST file.
- ADA_MAST: Profile for modeling real-time distributed applications, built using basic Ada logical components. The methodology has been designed to facilitate the modeling of systems written for an Ada 95 platform that satisfy both Annex D and Annex E of the Ada 95 standard.

And a third is under development:

- CBSE_MAST: Profile for modeling real-time systems based on software components. The methodology makes it possible to build the model of the system by aggregation of the components models and the model of the CBSE platform used.

The next subsections present these profiles in some more detail.

4.1. UML_MAST Profile

The modeling components defined in this profile [15][3] are suitable for the representation of systems that are being developed using object-oriented techniques and UML-CASE tools. The basic assumption for the usage of this profile is that the modeler will create what is called the MAST_RT_View of the system. This is a new complementary view in the UML description of the system that models its real-time behavior. By using it, the designer is able to build MAST models that can be used to analyze and predict crucial real-time properties of the system 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 [11] that is shown in Fig. 6, the MAST_RT_View may be considered as an added component to the Process View.

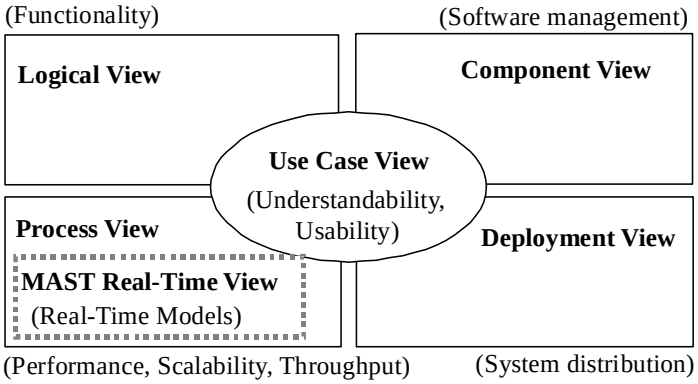

Fig. 6. The MAST_RT_View of the UML_MAST profile in the 4+1 architecture paradigm.

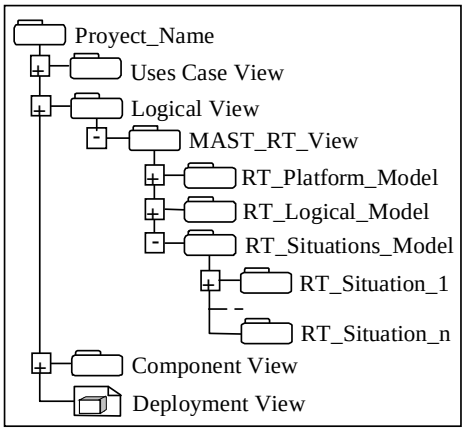

Fig. 7 Package structure of MAST_RT_View

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 profile meta-model [3]. This is a UML-independent conceptual model with a number of abstract components that rely semantically on the MAST model and are also structured in a similar way. They 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 meta-model and the general UML artifacts that are used in the MAST_RT_View. We use three types of UML diagrams and structures for representing this view on a standard UML CASE tool:

- The three MAST_RT_View sections —RT_Platform, RT_Logical and RT_Situations— 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 Fig. 7. This generic structure must be accurately implemented in order to allow the automatic tools to find the components of the model. The MAST_RT_View is incorporated in practice into the Logical View of the UML-CASE tool modeling structure just when the 4+1 paradigm Process View is not supported.
- 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 meta-classes. The meta-class from which a given class is derived is specified 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, even though 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 instance to be described.

The UML_MAST modeling framework is a “pseudo Add-In” that enhances the use of a UML graphical design and development tool by including a 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 meta-model 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 UML-CASE tool model.

As it is shown in Fig. 4, 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.

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

UML is in the process of being extended to support real-time models, as it is proposed in [21][20], 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 1.4) and currently available UML tools. 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.

4.2. ADA_MAST Profile

In this case the modeling components defined in [16] are oriented to enable the modeling of real-time distributed applications, built with basic Ada logical components. The methodology has been designed to facilitate the modeling of systems written for an Ada 95 platform that satisfy both, Annex D and Annex E of the Ada 95 standard.

Ada has been conceived as an object-oriented language to facilitate the design of reusable modules in order to build programs that use previously developed components. To design Ada component-based real-time applications it is necessary to have strategies for modeling the real-time behavior of these components, and also tools for analyzing the schedulability of the entire application, to find out whether its timing requirements will be met or not. With the possibility of distribution in real-time applications development, we also need to address issues like the modeling of the communications, or the assignment of priorities to the tasks in the processors and to the messages in the communication networks. On these premises we have identified some modeling basic Ada components that can be useful in the development of real-time distributed programs and for which the schedulability analysis tools can be applied. Hence, to address these challenges, the semantics of the high-level modeling components defined in this profile and the syntax and naming conventions proposed are particularly suitable and certainly adapted to represent systems conceived and coded in Ada. The main characteristics of the models generated with this profile that can be achieved with this approach are outlined next:

The RT-model has the structure of the Ada application: The <<Ada_Component>> instances model the real-time behavior of packages and tagged types, which are the basic structural elements of an Ada architecture:

- Each Component object describes the real-time model of all the procedures and functions included in a package or Ada class.
- Each Component object declares all other inner Component objects (package, protected object, task, etc.) that are relevant to model its real-time behavior. It also preserves in the model declarations the same visibility and scope of the original Ada structures.

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

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

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

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

4.3. CBSE_MAST Profile

Component-Based Software Engineering is one of the most promising technologies in the path to increment software quality, shorten time to market and manage the ever growing complexity of software. Even though this technology is already available in several fields like multimedia, graphical interfaces, etc., it still presents unsolved problems to become applicable in other areas, in particular in the real-time domain. Difficulties arise not only because of the great diversity and strength in the requirements it demands or the restrictions imposed over these kinds of systems, but mainly by the fact that new infrastructures and tools are necessary to implement it in real-time environments [8][26][12][25]. The design of real-time reusable components presents particular difficulties that are not found in general-purpose component-based software. In the first place, to get real-time responses, collaboration and synchronization mechanisms among components are required at levels certainly lower than those usually offered by the CBSE interfaces that are used as specification and interconnection units. In the second place, real-time systems are frequently embedded, heterogeneous, and far away from the target environments of the standard platforms used in component technologies. Finally, the reusability of real-time components relies in the usage of operating and communication systems or data bases that offer specific services for the management of time, are predictable, and allow for schedulability analysis; these services are not easily specifiable as usual component interfaces.

Designing real-time components to be executed on different HW/SW platforms is a very complex issue. Components running on different platforms present different temporal behavior and, hence, they need to be verified or reconfigured for each of them. Even in the case of having a catalog of well individually tested real-time components, it would be necessary to test compatibility and verify that the communication and synchronization mechanisms retain the expected timing responses once they are deployed on the final platform.

This profile is being designed to overcome this challenges. The basic idea behind the usage of this modeling profile is to increment the formal and functional description of the components in the modeling environment of the methodology used to specify them, in such a way that each component holds its complete timing behavior model and the model has the same connectivity properties of the modeled component. This approach lets us build complete analysis models for the component-based applications, provided the platform used is already modelled and all the necessary components hold their own models. This approach is not a proposal for a new design strategy; it is just an extension of a model-centric design approach that is based on the strong modeling and analysis capabilities of the MAST environment and the high expressive power that can be reached by using the UML extensibility mechanisms.

Some characteristics of the CBSE_MAST modeling approach that make it specially suitable for modeling component-based systems are:

- It builds independent models for the platforms, the logical components used and the real-time situations or final applications in which they are composed.
- The model is composable at different levels, and then can easily adopt the same structure of the software modular decomposition.
- It allows the specification of generic independent models for each high-level logical component. These models are instantiated in an analyzable model at the time of assembling it with its concrete used components and the values assigned to the parameters that it can have defined.

It has implicit support for distributed components. The communications model between components is incorporated automatically, whenever a component is allocated in a node different than the one requiring its services.

5. UML-MAST AND THE OMG REAL-TIME UML PROFILE

A group of OMG member companies have elaborated a proposal for a “UML Profile for Schedulability, Performance and Time” [21][20]. 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 inter operability

among different analysis and design tools. At present, the UML_MAST methodology does not follow the nomenclature and the UML representation rules suggested in that profile, in part 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 [21]. 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 multi threaded 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 [17][18].
- 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 [7].
- 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 still in the standardization process and therefore, we think that based upon the experience with UML_MAST, it is not too late to submit issues or proposals to the Finalization Task Force for relaxing some of its restrictions, making it possible for both approaches to have the same modeling power.

6. CONCLUSIONS

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.

The MAST suite is open-source software and it is distributed under the GNU general public license. It can be found at:
`http://mast.unican.es/`

7. REFERENCES

[1] 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.

[2] http://www.ics.uci.edu/~arcadia/Aflex-Ayacc/aflex-ayacc.html
[3] J.M. Drake and J.L. Medina: “UML-MAST Metamodel, specification, example of application and source code”. http://mast.unican.es/umlmast
[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 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, IEEE Computer Society Press, pp. 125-134, June 2001
[5] M. González Harbour, M.H. Klein, and J.P. Lehoczky. “Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority”. Proceedings of the IEEE Real-Time Systems Symposium, December 1991, pp. 116-128
[6] 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.
[7] 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.
[8] Isovic D. and Norström C.: “Components in Real-Time Systems”. The 8th Int. Conf. On Real-Time Computing Systems and Applications (RTCSA'2002), Tokyo (Japan), 2002.
[9] M. Joseph and P. Pandya, “Finding Response Times in a Real-Time System,” *BCS Computer Journal*, Vol. 29, no. 5, October 1986, pp 390-395.
[10] 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.
[11] P. Kruchten: “The 4+1 View Model of Architecture”. IEEE Software . Nov. 1995.
[12] Lehman M.M. and Ramil J.F.: “EPiCS: Evolution Phenomenology in Component-intensive Software” Proposal draft, Dept. of Computing of Imperial College, London, Aug. 2001.
[13] J.P. Lehoczky. “Fixed Priority Scheduling of Periodic Tasks Sets with Arbitrary Deadlines”. *IEEE Real-Time Systems Symposium*, 1990.
[14] 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.
[15] J.L. Medina, M. González Harbour, and J.M. Drake: “MAST Real-Time View: A Graphic UML Tool for Modeling Object-Oriented Real-Time Systems” RTSS'01, London, December, 2001.
[16] J.L. Medina, J.J. Gutiérrez, M. González Harbour and J.M. Drake: “Modeling and Schedulability Analysis of Hard Real-time Distributed Systems Based on Ada Components”. In *Reliable Software Technologies - Ada-Europe 2002, Proceedings*, Lecture Notes in Computer Science, vol. 2361, Springer-Verlag, 2002, 7th Ada-Europe International Conference on Reliable Software Technologies, Vienna, Austria, June 17-21, 2002.
[17] 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.
[18] 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.
[19] T. Quatrany: “Visual Modeling with Rational ROSE 2000 and UML” Addison Wesley Longman, Reading, Mass., 2000.
[20] B. Selic: “A Generic Framework for Modeling Resources with UML”. IEEE Computer, Vol. 33, N. 6, pp. 64-69. June, 2000.
[21] Bran Selic, Alan Moore, Murray Woodside, Ben Watson, Morgan Bjorkander, Mark Gerhardt and Dorina Petriu: “UML Profile for Schedulability, Performance and Time”. OMG document ptc/2002-01-20, January 2002
[22] 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.
[23] 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.
[24] 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.
[25] Welling A. and Cornwell, P.: “Transaction Integration for reusable hard Real-time Components” Proc. of Ada in Europe, pp. 365 - 378, Springer-Verlag April, 1996.
[26] Yau S.S. and Xia B.: “An approach to Distributed Component-based Real-time Application Software Development” Proc. IEEE Int´l Symp.Object-oriented Real-Time Distributed Computing (ISORT'98), April, 1998, pp. 275-283.