An object oriented library for simulating distributed real-time control systems

Luigi Palopoli, Giuseppe Lipari, Gerardo Lamastra,
Fabio Rossi, Luca Abeni, Gabriele Bolognini, Paolo Ancilotti
Scuola Superiore S. Anna, Pisa, Italy
{palopoli, lipari, lamastra, fabior, luca, bolognini, paolo}@sssup.it

5th July 2001

Abstract

This paper presents a set of C++ libraries, called RTSIM, aimed at simulating real-time embedded controllers. The libraries permit a separate specification of the functional behaviour of the controller and of the hardware/software platform to be used for its deployment. In particular, it is possible to provide an accurate modeling of the concurrent architecture of the control tasks and of the run-time support offered by the operating system for the real-time scheduling of the shared resources (CPU, memory buffers, network links). In this way, it is possible to compare different scheduling solutions by evaluating their simulated performance directly in the domain of the control application. Moreover, the tool can be utilized to tune up such design parameters as the activation frequencies of the tasks. The application of the tool is shown on a meaningful case-study.

1 Introduction

Many computer based embedded control applications can intuitively and efficiently be implemented as a set of interacting parallel software *tasks* operated by a multi-task real-time operating system.

The performance of the controller is affected in many (sometimes subtle) ways by design choices related to the behaviour of control tasks with respect to time. In fact, the execution times of the tasks, along with the physical limitations of sensors and actuators, impose upper boundaries on the maximum achievable sampling rates (for the sensors). Moreover, the scheduling of multiple control tasks sharing limited computational resources determines stochastic delays in their response times. Crucial design issues for controlling these factors are the selection of the appropriate task execution rates and the choice of (possibly) the best real-time scheduling policy for the shared resources (CPU, memory, communication links).

Classical approaches to control design often disregard these problems assuming a highly deterministic behaviour for the controller: the time interval between the acquisition of the sensor data and the release of the controller output is assumed negligible or fixed. This choice is motivated by the difficulties encountered in defining an analytical model for stochastic response time variations deriving from data-dependencies and scheduling jitter. For the same reason, real-time computing theory has endeavoured to ensure time determinism on the implementation side. However, the notion of determinism used by real-time theory is the respect of deadlines which is generally difficult to relate to the performance of the control system.

As a consequence, an architectural design strictly complying with the deadline constraints can result into a poorly performing system. When these problems occur, the typical approach is to scale up the power of the hardware platform whereas a better tuned software design could often accomplish the same goals with low cost solutions. For instance in [23] we showed that, if the tasks execution times have a wide variance, the price of occasional overload situations can be outweighed by the "average" improvement of the controller behaviour achieved by using high sampling rates. This technique can pay off up to 50% performance improvement for certain classes of applications (e.g. robotic applications using multimedia sensor data), especially if soft real-time scheduling algorithms are used to control the system's timing behaviour in overload conditions.

In the absence of well-settled analytical approaches, an integrated design of the control law and of the software architecture has to strongly rely on the use of simulation techniques. However, the best known CAD tools are either focused on simulating the plant and the functional architecture of a controller or on performing simulations on the discrete events evolution of the computer system. The present work proposes a collection of C++ libraries, called RTSIM, aimed at bridging this gap. By using a well structured object oriented framework, a separate specification can be provided for the functional model of the controller and for the software architecture. In this way, different architectural solutions can be tested and evaluated using performance indices belonging to the domain of the control application. The best known solutions for real-time CPU scheduling (either on single or on multiprocessor boards) and for bounded delay sharing of resources are available as predefined classes. Moreover, a well defined programming framework guides the user in developing his own scheduling solutions. The functional specification of the system is provided interconnecting a set of reusable components following a data-flow approach. The ensemble of functional and architectural model is directly implementable on a target real-time operating systems (without the need of additional run-time support, such as virtual machines). Consequently the simulated performance turns out to be very close to the performance of the real system. For what concerns the simulation of the plant, RTSIM exploits the functionality of a powerful mathematical library, called OCTAVE [7], embodying the *state of the art* solutions for the integration of differential equations. The tool suite is completed by a set of classes defining stochastic parameters (*job*s durations, packet lengths etc.) with assigned probability distributions. Moreover, it is possible to record the events on a trace file and to collect statistics on the performance indices of interest over multiple runs.

1.1 State of the art

The best known tool suite for simulating control systems is MATLAB. Regarding the simulation of embedded control systems, MATLAB fits very well for modeling and simulating the continuous part of the plant and the functional structure of the controller. Moreover, it permits to automatically generate a prototype for a real-time controller (by the **Real-Time Workshop** tool). However, it is not easy to model generic Hardware/Software entities and there is little or no direct support for modeling and simulation of real-time scheduling algorithms.

In [8], the authors propose a MATLAB toolbox to simulate a real-time scheduler in a SIMULINK block. This allows, to a given degree, the simulation of timing properties and the assessment of the performance of real-time controllers against changes in the timing attributes of the tasks. The most important feature of this tool is the good integration with the MATLAB/Simulink environment. On the other hand, the lack of a clear separation between the functional and the architectural specification hinders the application of the tool to complex systems having event driven and/or time driven activities.

In recent years many interesting tools have been proposed for the analysis and simulation of complex real-time systems, networks and kernels. One of the best know commercial product in this class is TimeWiz, from Timesys corp., which is mostly aimed at the analysis of the timing behaviour of a real-time system with respect to schedulability constraints. The toolset is being integrated with a UML design framework which allows to describe complex systems in a fairly general way. However, to the authors knowledge, the tool does not allow to perform hybrid simulations of a digital controller along with the continuous dynamics of the controlled plant; thus it is not possible to interactively evaluate the performance of control systems against changes in the task architecture and/or in the scheduling policies.

The GIOTTO programming language [33] has been devised to develop hybrid control applications consisting of periodic tasks. The model of computation is primarily aimed at the design and prototyping of time-predictable control system by using the conventional paradigm of separating the functional from the timing behaviour (hard schedulability requirements). Time predictability (schedulability) is obtained by means of restricting the design to a time-triggered architecture [15]. A remarkable advantage of this paradigm is the elimination of input and output jitters in the data-flow between the controller and the plant. However, the introduced delays turns out to be a very pessimistic solution in many cases. Moreover, the time triggered approach does not easily cope with event-driven systems.

An integrated design of real-time control systems encompassing performance and schedulability concerns was first proposed in [24]. In this work an optimization procedure for the activation frequencies of control threads is proposed; the goal is maximizing the controller performance under schedulability constraints. The paper is inspired to the evaluation approach for embedded controllers suggested in [26]. Other noteworthy results on this problem are presented in [14]; the authors first map the classical control design parameters onto the end-to-end requirements of the controller and then apply the method of period calibration [11] to derive the execution parameters of each thread so that the end-to-end requirements are respected. A tool like the one presented in this paper may a very useful aid to validate the assumptions and the result of these methods and of any other co-design procedure.

2 Modeling primitives of RTSIM

A real-time controller can be designed and analyzed considering the system's *functional behaviour*, expressed in terms of the processed data and of the computations performed upon them, and the

HW/SW architecture for the implementation platform.

Obviously, a given functional behaviour can be implemented by a set of possible architectures. For example, one possible choice is to implement the controller as a set of concurrent real-time tasks to be executed on a single computer system; another possible choice is to use a distributed system, with two or more computers connected by a network link.

For this reason, it is useful to define two distinct points of view of the system corresponding to the functional and to the architectural models. In fact, the separation of functional and architectural design permits to easily test different implementations for the same functional specification in order to identify the solution which best fits the performance/cost tradeoffs of the project.

In this section, the most important modeling primitives of RTSIM for defining both the functional and the architectural specification are introduced. A simple example will show how this primitives are applied to a practical case.

2.1 Modeling the functional behaviour

The separation between the functional and architectural specification is aided, in the RTSIM tool, by the use of a well structured data-flow approach for the functional model of the system. Data-flow models are a well-suited tools in the design of real-time software [31, 35], so much so that they are used in a variety of tools including Simulink, Ptolemy [18], Giotto [33]. The functional abstractions of RTSIM are essentially of two types: *computing units* and *storage units*. Computing units are used to perform the computation while storage units are used to exchange data between different computing units or between the controller and the external environment.

A **computing unit** is endowed with a set of input ports and of output ports which must be connected to storage units. Each computing unit can respond to three different external commands. The first command, called *read* is used to acquire external data from the storage units connected with its input ports. The second one, called *execute*, computes an output value, while the third one, called *write*, is used to write the output into the storage units connected with the output ports. A computing unit can have an internal state (i.e. state remaining between two consecutive invocation). Notice that no particular model is required to specify the *execute* method. Thus, a computing unit can be a finite state machine, a digital filter, a proportional integral derivative (PID) controller, or whatever is needed in the controller's structure. A set of well known computing units such as matrix gains, digital filters, discrete time systems are predefined library objects and can be used in constructing a model of the system without any further programming effort.

Storage units are of three types: *input buffers*, *memory buffers* or *output buffers*. Input buffers serve as an interface between the environment and the controller. From the point of view of the environment they can be thought of as sensors performing a measure on a quantity continuously varying in time. From the controller's side, an input port models an I/O card whose content changes when a sampling command is received. We remark that input buffers can only be connected to the input ports of a computing unit. Output buffers can be used to model actuators and can only be connected to the output ports of a computing unit. They model digital to analog converter, i.e. when a computing unit writes a new data into an output buffer, the value is held up to the next writing. Memory buffers can be accessed either for reading or for writing operations and they realize communications among different computing units. In plain words, a communication between a computing unit acting as "writer" and another one acting as "reader" can simply be established by connecting to a memory buffer an output port of the former and an input port of the latter. It is important to observe that when a functional model is constructed no particular assumption is made neither on the hardware implementation of a storage unit nor on the way concurrent access request shall be scheduled.

Example. A simple example of functional design is reported in Figure 1. The addressed problem is the control of a simple physical device (an inverted pendulum). The pendulum is mounted on a cart moving on a one-dimensional track. The horizontal position x and the pendulum angle θ are acquired through a couple of sensors and their values are stored into two input buffers (named `In_x` and `In_th` respectively). Data held in the input buffers are processed by the computing units `Filt_x` and `Filt_th` in order to extract the meaningful information and to filter out the sensor noise: the results are stored into the `F_x` and `F_th` memory buffers. Two digital filters, namely `Der_x` and `Der_th`, are derivative blocks and are used to estimate the linear and angular velocities. Finally the four estimated state

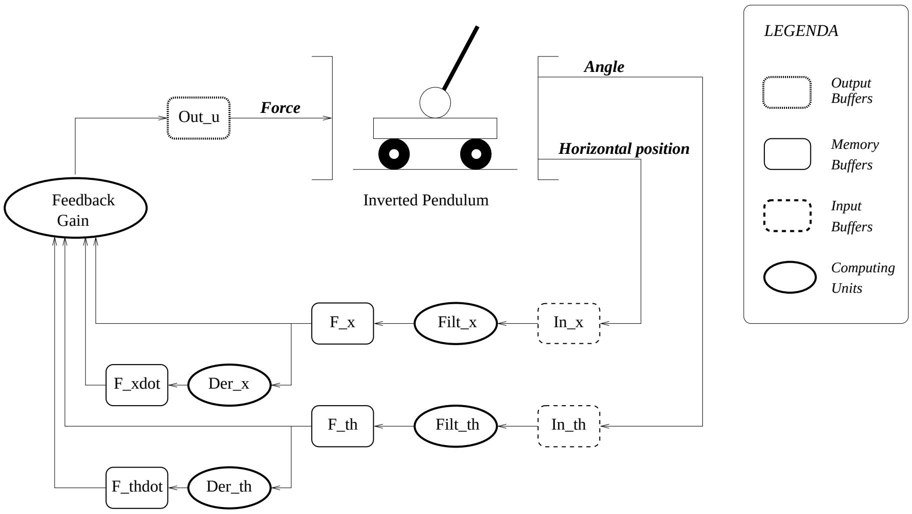


Figure 1: Functional design of a simple controller for an inverted pendulum.

variables are used by a computing unit (`FeedbackGain`) to compute the force to be applied to the cart which is stored into an output buffer (`Out_u`). It is worth observing that the computing units shown in this scheme are instances of library predefined objects (four digital filters and a matrix gain).

2.2 Modeling the architecture of the system

With the help of RTSIM, it is possible to map a given functional specification onto different hardware/software architectures, assessing the performance of a real-time application both in the system domain (i.e. response time of each single software task) and in the application domain (i.e. performance index of the controller). The distribution of tasks to the computational resources, the scheduling algorithm, the synchronization mechanism, the network: all these parameters can greatly influence the system behaviour. RTSIM allows to evaluate different solutions in order to find the best trade-off design between system performance and resource usage.

In our model, a **task** (or process) is a software entity which implements some functional behaviour. Formally, it is a finite or infinite sequence of requests for execution, or *jobs*. Each job is the execution of a piece of code (a sequence of instructions), that is activated at a certain instant of time. When a job is activated, we say that it is *released* or that it *arrives* and the activation time is called *arrival time*. Depending on the pattern of arrival times, tasks can be classified as:

Periodic : if the arrivals are separated by a constant interval of time, called “period”;

Sporadic : if the arrivals are separated by variable intervals of time with a lower bound, called *minimum inter-arrival time*;

Aperiodic : if a lower bound is not known on the inter-arrival times.

In real-time systems, tasks have time constraints, often expressed as **deadlines**: for example, a typical time constraint for a periodic task is that each job must finish before the next activation. Another typical constraint is on the completion jitter (the interval of time between two consecutive job completions).

The **instructions** of a task model its functional and timing behaviour. Basically, an instruction is modeled by a execution time (which can be fixed or variable) and can be associated with the *read*, *write* or *execute* command of a computing unit. In this way, one or more computing units can be easily mapped onto a task.

Tasks are assigned to the computational resources (**nodes**) of the system. Each node consists of one or more processors and a real-time operating system (kernel) endowed with a scheduling policy and a synchronization protocol.

Finally, the system can be comprised of several computational nodes connected by **network links**. Tasks on different nodes can communicate by means of *real-time messages*. A communication resource is modeled by a shared physical link, an access protocol and a real-time message scheduler.

RTSIM allows to model and simulate all these aspects. In fact, the tool was originally designed for experimenting different real-time scheduling solutions. The resource requirements of the tasks can be specified either by deterministic or stochastic parameters. Moreover, the best known state of the art algorithms for scheduling CPU (such as Cyclic executive, Fixed Priority, Rate Monotonic [22], Earliest Deadline First (EDF) [22], Proportional share [32]) are provided as predefined objects, both for single processor and multi-processor systems. The performance of the schedulers can be enhanced by using aperiodic servers (Polling server [20], Sporadic Server [27], Constant Bandwidth Server [1], etc). It is also possible to choose the most appropriate protocol to handle shared resources as Priority Inheritance/Priority Ceiling [25], Stack Resource Policy [3]. Finally, nodes may communicate through network (Ethernet, CAN bus, etc.).

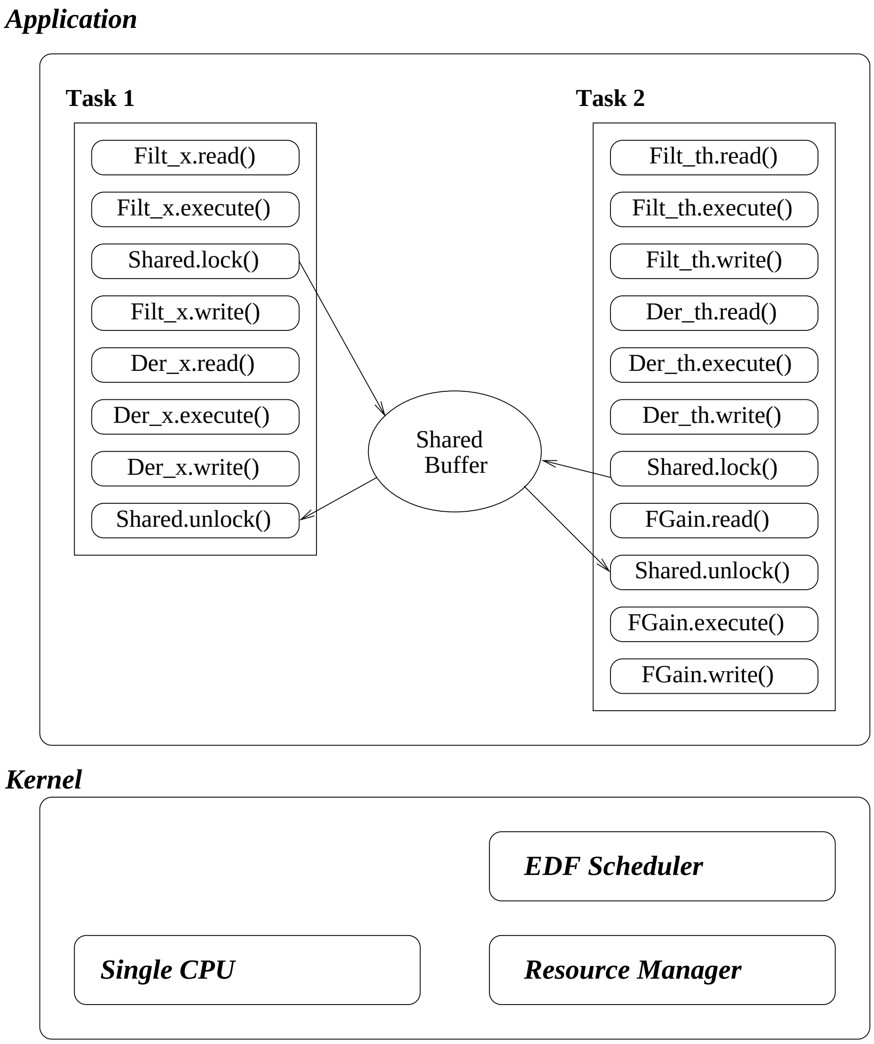


Figure 2: Architectural design for the example shown in Fig. 1.

Example. A better understanding of what is really meant in RTSIM by "architecture of the system" can be achieved getting back to the example shown in figure 1. The main architectural components for this example are depicted in Figure 2.

A situation that may occur in many practical applications is that the data-flows from the sensors have different timing characteristics. Suppose, in the case of the inverted pendulum, that the horizontal position is computed from the images grabbed by a camera, whereas a potentiometer is used to acquire the angle. In this case the computation workload necessary to compute x (associated to computing unit `Filt_x`) is much higher than the workload necessary to compute θ (associated to computing unit `Filt_th`). Thus, a possible architecture for the system can be based on two periodic real-time tasks, `Task 1` and `Task 2`. In particular, `Task 1` triggers the actions on computing unit `Filt_x` and `Der_x` in order to compute x and to estimate the $\dot{x}$ horizontal velocity. `Task 2` triggers the same operations on computing units `Filt_th` and `Der_th`.

It is worth pointing out that the activation rate for `Task 2` should be higher than the activation rate for `Task 1`, for the computation workload required by `Filt_x` is much greater than the one required by `Filt_th`. In fact, the former computes the cart position by processing digital images, which is a time consuming activity, whereas the latter simply performs some digital filtering on the data sampled from the potentiometer. Therefore, the command values for controlling the system, computed by computing unit `MatrixGain`, can be associated to `Task 2`. If this architectural choice is made, memory buffers `F_th` and `F_thdot` can be privately owned by `Task 2`, whereas `F_x` and `F_xdot` are shared between `Task 1` and `Task 2`.

In this example, we assume that a single CPU board is used. The main architectural components for this example are depicted in Figure 2. The concurrent execution of the two tasks is possible using a scheduler component (named `EDFScheduler`) endowed with the Earliest Deadline scheduling policy [22]. A *resource manager* is used to ensure correctness in the accesses to a shared memory space containing the `F_x` and `F_xdot` memory buffers. Both the task scheduler and the resource manager are components of a software layer modeling a real-time operating system (`Kernel`).

Of course, this is only one of many possible choices for the hardware/software architecture. This particular choice aims at computational efficiency by concentrating in one task all activities which may be performed at the same rate. A potential drawback is the lack of modularity if dynamic reconfigurability of the software is required. For example, replacing `Task 2` by two tasks, the first operating the `F_x` and `F_xdot` computing units, and the second operating (`FeedbackGain`), it could be possible to change "on-line" the way x position is acquired to cope with a potential sensor fault or with a mode change. Another possibility, in case a very high loop rate was needed for stability reasons, is to use two different CPU boards connected by a network link, one performing `Task 1` (which is computationally expensive), and the other one performing `Task 2`. More generally, this simple example shows that the choice of the hardware/software architecture is the solution to a potentially complex problem involving performance issues, cost limitations and physical constraints. This is the reason why decoupling architectural and functional design turns out to be a convenient choice.

Moreover, even with the architecture shown in Figure 2, the developer has some degree of freedom in setting the parameters. The choice of the scheduling algorithm, the resource manager and the task activation rates can influence the delay of the two tasks and this in turn impacts upon the stability of the system and the "quality" of the control. For this reason, it is desirable to know in advance which scheduling strategy and which combination of parameters must be assigned in order to maximize the performance of the control strategy. In Section 5 we will show how it is possible to use RTSIM for simulating this system and find a possibly optimal combination of parameters.

3 Description of the tool

RTSIM is a collection of C++ libraries which can be used inside user defined programs. The modeling technique described in the previous Section suggested to decompose the software into three interacting packages, as shown in Figure 3.

The package denoted as "Numerical Package" is used to simulate the plant. The package called "CTRLIB" is used to describe the functional model of the system. The "RTLIB" package is used to map the functional description onto a deployment architecture. The way the three packages are used and interact one another will be described in detail in the next Section on a practical example. The

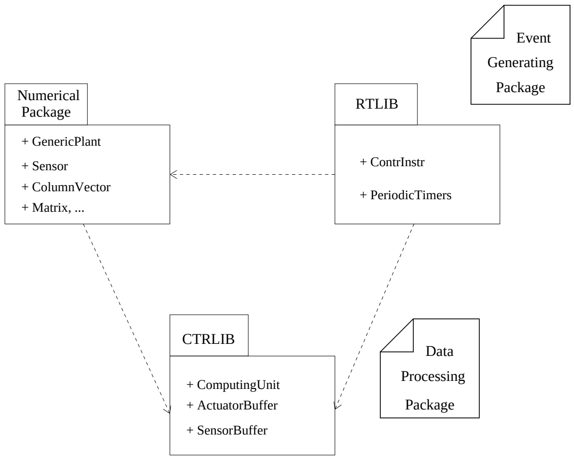


Figure 3: Main components involved in a RTSIM based simulation of a real-time controller

three packages are based on the MetaSim library, a small software layer developed at the Retis Lab of Scuola Superiore S. Anna, which provides the basic classes for writing discrete event simulations. In the following, we will briefly describe the MetaSim library and the three packages. The UML graphical notation [?] is used throughout the next sections, in particular class diagrams and instance diagrams.

3.1 The MetaSim framework

The MetaSim framework has been designed to support the programming of discrete event simulations. It provides the basic simulator engine and several helpful classes for random number generation and data collection. The two basic classes in MetaSim are Entity and Event.

The Entity class represents the abstraction for a component of the system. An entity contains the variables that represent the status of the component and public methods to set/get the status.

All the computations in the simulator are performed at certain particular instants corresponding to Events triggering [5, 16, 17]. Thus, the status of an entity at any time t_2 depends on its status at any previous time t_1 and by the sequence of events between t_1 and t_2. Typically, each entity has a list of events that can be *posted*. When an event is triggered, one of the entity methods is invoked. These methods are called *event handlers* because they respond to the event triggering modifying the entity status.

The main cycle of the simulator is the following:

1. the first event is extracted from the global event queue: the (logical) simulation time is updated to the event's triggering time;

2. the event handler of the related entity is invoked. This event handler updates the status of the system and (possibly) posts some other event in the queue.

3. the statistical probes and the tracing probes are processed.

4. go to step 1.

Metasim provides also several helpful classes for implementing simulations:

- the RandomVar family of classes is used to generate pseudo-random numbers according to certain distributions. Examples of classes that derive from RandomVar are UniformVar, NormalVar, PoissonVar, with the obvious meaning.

- the BaseStat family of classes provides statistical collection of data. The user has to link a statistical object to the opportune event: when the event is triggered, the probe() method of the statistical object is called, and in that method, the user can "observe" the status of any entity in the system. After the simulation has been performed it is possible to get the collected statistics invoking the getMean(), getVariance() and getConfInterval() methods of any statistical object: they return respectively the average value, the variance and the confidence interval of the collected data.

All the main objects described in the following (like tasks, instructions, kernels, etc.) are implemented as entities: more specifically, the related classes derive from Entity.

3.2 The RTLIB Package

RTLIB is a library designed to simulate the timing behaviour of a real-time software system. It models entities like real-time tasks, scheduling algorithms, single and multi processor nodes, network links. The following description is not complete due to space constraints: the interested reader can refer to the library documentation.

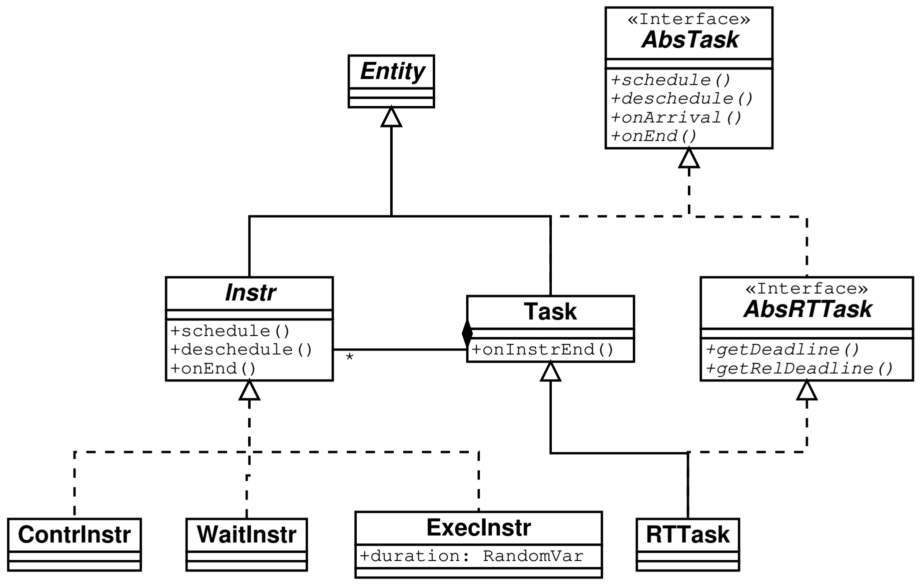


Figure 4: The class diagram representing the Task family of classes.

Tasks. The family of classes for modeling a task is shown in Figure 4. The AbsTask abstract interface defines the basic methods of a task:

- schedule(): signals the task that it has been scheduled to execute;
- deschedule(): signals the task that it is not executing anymore (for example because it has been preempted by another task);
- onArrival(): it is called when a new job is released;
- onEnd(): it is called when the current task's active job completes.

The Task class contains a list[1] of instruction, which are modeled by the Instr class. Examples of instructions are:

- ExecInstr which models a piece of sequential code with a certain execution time; the execution time is described by a RandomVar object: hence it is possible to model a portion of code with an arbitrarily distributed execution time;
- WaitInstr and SignalInstr which model the wait and signal primitives for using semaphores;

[1] We modeled only lists of instructions. We think that this should be sufficient for the scope of this simulator. However, if necessary, the user can model if-then-else branches with special instructions that behave differently depending on the value of some external condition.

- the ControlInstr family of classes which model the execution of computing units.

The definition and the insertion into tasks of pseudoinstructions can be a dull and error prone activity. For this reason, the library provides a programming shortcut based on the use of symbolic strings having an appropriate syntax to represent the pseudoinstructions (see the appendix for some detail).

The relevant events for a task are:

- taskArr: it is triggered when a new job is released; for example, in case of a periodic task, this event is triggered every task's period. This event is handled by the onArrival() method;
- taskEnd: it is triggered when the current active job completes; it is handled by the onEnd() method;
- taskSched: it is triggered when the task is scheduled to execute; it is processed by the schedule() method;
- taskDesched: it is triggered when the task leaves the processor because it has been blocked or preempted by another task; it is processed by the deschedule() method.

As anticipated, it is possible to connect statistical probes to any of these events; hence, they are also "observation points" for the system.

Real-Time Tasks. By definition, a real-time task has timing requirements. The most common constraint is the *deadline*: *absolute deadline* of a job is the instant of time by which the job must finish; the *relative deadline* of a task is the interval of time between the release time and the absolute deadline of each job.

A real-time task is modeled by the abstract interface AbsRTTask which derives from the AbsTask (Figure 4). It comprises the getDeadline() and getRelDline() methods which return respectively the absolute and the relative deadline of a task.

Kernels. The Kernel family of classes models a computational resource, like single processor or multi-processors nodes. The abstract interface AbsKernel defines the methods that are common to all the kernels:

- activate(AbsTask *t): inserts task *t in the ready queue;
- suspend(AbsTask *t): extracts a task from the ready queue;
- dispatch(): compares the current executing task with the first task in the ready queue; in the case that it is not the same task, it performs a context switch. The dispatch() method should be invoked every time a new task is inserted in the queue or a task is suspended. In multi-processor systems, the dispatch() does the same thing for every processor.
- onArrival(AbsTask *t): this method is invoked when a new task instance is released. It inserts the task in the ready queue and calls the dispatch().
- onEnd(AbsTask *t): this method is invoked when a task instance completes. It extracts the task from the ready queue and calls the dispatch().

Notice that, at this point, we have not yet introduced any notion of "task priority". In fact, different scheduling policies compares tasks basing on different parameters. For example, the Rate Monotonic scheduler requires a static priority to be assigned to each task, whereas the Earliest Deadline scheduler uses the absolute deadline of a job as the task priority. Moreover, some scheduling policy (like Proportional Share or Round Robin) does not use any priority at all.

Hence, the ordering of tasks in the ready queue depends on the scheduling policy which is implemented by the Scheduler family of classes. Each one implements a different queue policy: for example, EDFScheduler implements the Earliest Deadline First scheduling algorithm, PrioScheduler implements a generic Fixed Priority scheduling algorithm, and so on. The scheduling parameters are not stored in the task class but instead in the *wrapper* class TaskModel: thus, the task implementation is independent from the scheduling algorithm (see the *Adapter Pattern* in [10]). The TaskModel hierarchy of classes is similar to the Scheduler hierarchy: every scheduler corresponds to a task model. In figure 5 the inheritance relationships between these classes are summarized.

Example. The notification mechanism and the way events are handled in our simulator are better explained with a practical example. The instance diagram shown in Figure 6 captures a snapshot of

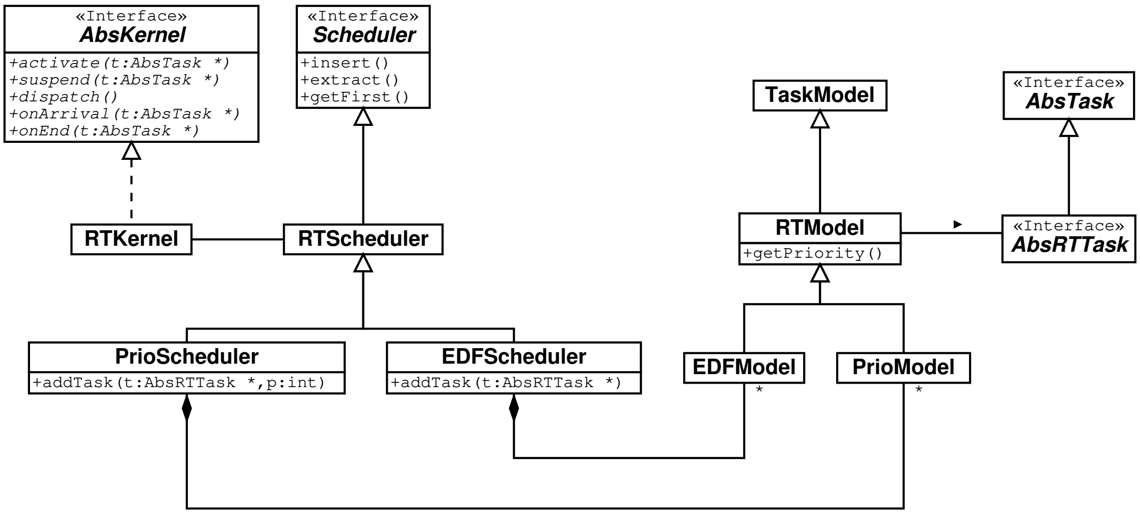


Figure 5: The class diagram representing the Kernel family of classes.

the system when a preemption occurs. At some time, the taskArr event is processed for task t2: then the onArrival() method of t2 is invoked. After updating its internal status (for example recording the arrival time and resetting the current instruction pointer to the first instruction), task t2 calls the onArrival() method of the kernel. It inserts the task in the ready queue (calling the s.insert()), and checks if this task was inserted as the first element in the queue. If so, a preemption must occur: the current executing task, t1, leaves processor, and t2 becomes the current executing task. Hence, t1 must be signaled calling its deschedule() method; in turn, it calls the deschedule() method of its currently executing instruction. Finally, task t2 is signaled calling its schedule() method.

Servers. When soft real-time aperiodic tasks are to be scheduled together with hard real-time periodic tasks, the goal is to improve the response time of the aperiodic tasks without compromising the schedulability of the hard real-time tasks. A popular conceptual framework for modeling the behaviour of such application systems is to associate a *server* with each soft aperiodic task, with each server characterized by certain parameters which specify exactly its performance expectations. Several aperiodic service mechanisms have been proposed under RM [20, 19, 2, 34] and under EDF [28, 12, 30, 29, 1, 21].

The *Server* class models these algorithms. We modeled a Server to have an interface that is the

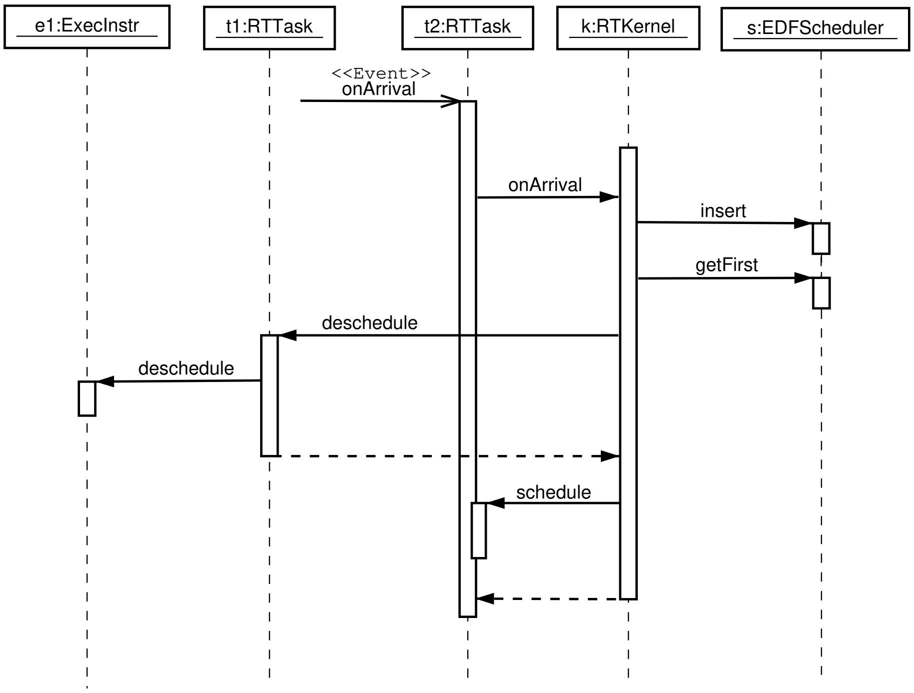


Figure 6: Sequence diagram: task t2 preempts task t1.

union of the AbsKernel and of the AbsRTTask interface. In this way a server can be inserted between a kernel and a task transparently: the kernel sees the server as a normal task; the task sees the server as a kernel.

The server can also have its own events: for example, all the servers based on a capacity must set an event of capacity exhausted, to avoid that the handled task executes too much. This is the case of Servers like the Polling Server, Deferrable Server, Sporadic Server for the rate monotonic class of algorithms, and the Constant Bandwidth Server [1] for the EDF class of algorithms.

Sharing other resources. Sometimes, tasks access mutually exclusive resources (other than processor): for example, tasks can access the same memory block that is protected by a mutex semaphore. In RTSIM, this can be simulated by means of a class *Semaphore* and by a *Resource Manager*, that is the entity that manage the operations on a semaphore, holding the blocked tasks in blocking queues. Tasks can operate on semaphores by means of the WaitInstr and SignalInstr pseudo-instructions.

In Figure 7 the relationship among the classes is shown. When a tasks executes a Wait pseudo-instruction, the *Resource Manager* checks if the semaphore is free: in this case, the semaphore is

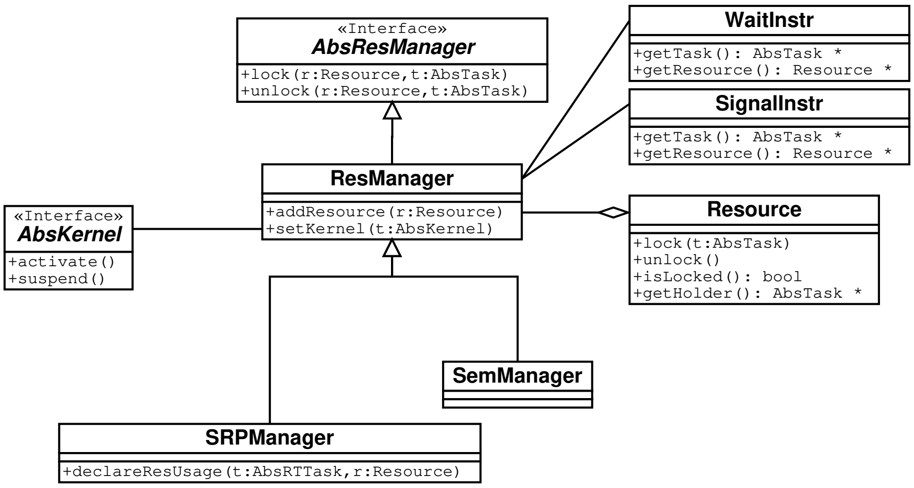


Figure 7: The class diagram representing the Resource Manager family of classes.

locked and the task continues executing the next pseudo-instruction. Otherwise, the task is blocked and the Kernel::suspend() and the Kernel::dispatch() are called to schedule another task. When a tasks execute a Signal pseudo-instruction, if the semaphore is locked, the first blocked task is activated, and the Kernel::activate() and Kernel::dispatch() are called.

Using a simple locking policy, a particular phenomenon called *priority inversion* can arise in hard real-time systems. Priority inversion was first discovered by Sha [25]: if a low priority task blocks an higher priority task, and a medium priority task preempts the low priority task while in the critical section, the high priority task experience an unbounded delay. To overcome this problem, several concurrency control algorithms have been devised for hard real-time systems [25, 6, 4, 13].

These algorithms are implemented in our simulator in the *Resource Manager class*. Thus the Priority Ceiling algorithm [25] is implemented by the PCManager class, and the Stack Resource Policy [4] is implemented by the SRPManager class. They interact also with the kernel and with the scheduler policy, because they require to dynamically modify the priority of the tasks.

Networks. Every kernel may have one ore more network interfaces, modeled by the NetInterface family of classes, each one connected to a network link, modeled by the NetLink family of classes. For each network link class, there is a corresponding network interface class. At this moment, the Ethernet network and the CAN bus have been implemented.

A task can send a message, modeled by the Message family of classes, to another task passing it to the appropriate network interface of its kernel. The Message class implements the AbsTask interface: in this way, it can be handled by a Scheduler. A network interface has a pointer to a Scheduler object for implementing the message en-queuing policy. It implements the medium access protocol, such has the Ethernet or CAN bus protocol. In particular, the *CANInterface* has a pointer to a function that transforms the message priority (or deadline) in a CAN priority.

Two additional pseudo-instructions have been defined:

- SendInstr instruction: takes as parameters the name of the destination task and a function object for building new messages.
- ReceiveInstr instruction: if a message has already arrived for the task, it gets the message, otherwise it blocks the task waiting for a message from the network interface.

High level protocols (like TCP/IP) have not been implemented for they are well beyond the scope of this work.

3.3 Graphical representation of the output

Summarizing, once a structure of tasks has been specified along with the underlying hardware/software architecture, RTLIB permits to construct a timed sequence of events associated with the jobs activation and termination instants, detailing, if necessary, with the beginning and the termination of pseudo-instructions. The occurrence of an event can be recorded into a simulation trace or accounted for to collect statistics. In this way it is possible to gather a great deal of information on the system timing behaviour, including the maximum response time of a task, the number of preemptions it undergoes, the end-to-end delay of a pipeline of tasks and so on. Figure 8 shows an execution trace reconstructed using the RTSIM simulator. The trace visualization tool is part of RTSIM suite and is entirely written in Java to enhance portability.

3.4 The Numerical Package

The main purpose of the numerical package is to provide programming models for continuous time plants. A plant is described by means of its state variables, differential equations and so on. From

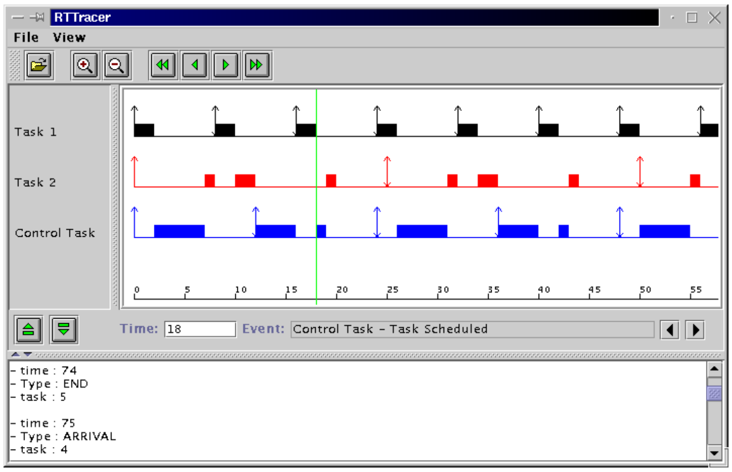


Figure 8: Graphical representation of the output of a RTSIM simulation.

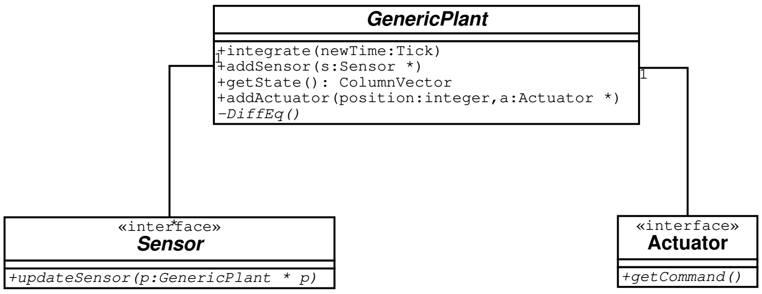


Figure 9: Class diagram representing the components of the numerical package to be used for modeling plants.

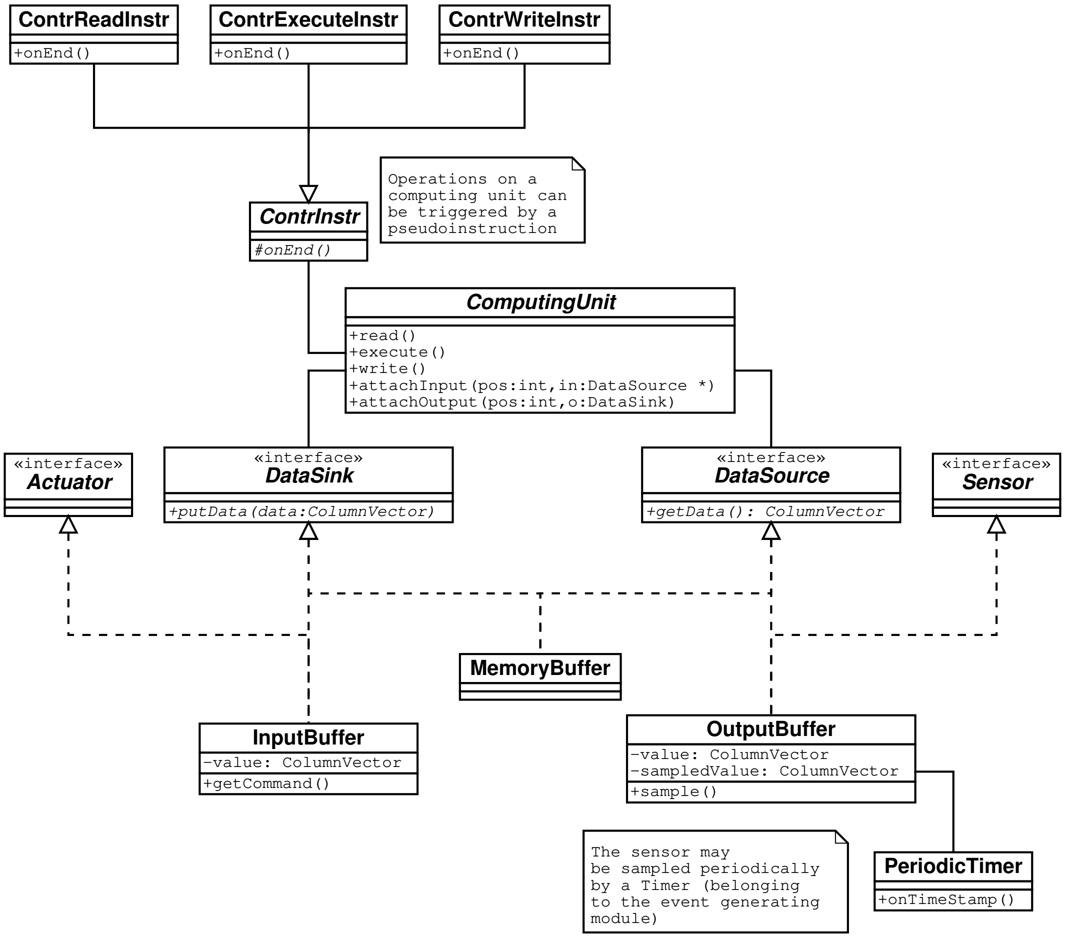


Figure 10: The most important classes used to model the functional behaviour of a controller.

a structural viewpoint, the numerical package is a software layer built on the top of a library which provides such services as differential equation integration and linear algebra. The current implementation is based on the OCTAVE library, which is a freely available tool encompassing the best known algorithms for numerical computation. The presence of a software abstraction layer allows to replace OCTAVE with any other similar solution without affecting the structure of the simulator. As well as permitting the definition of plant, the numerical package also exports a set of useful classes for linear algebra, such as `Matrix`, `ColumnVector` and so on.

User defined plants are derived class of an abstract class named `GenericPlant` (see Figure 9). The inheritance mechanism permits to add plant specific information by inserting new data members in the derived class. The differential equations are specified by providing a definition to the abstract method `DiffEq`.

The plant evolution can be observed by a set of objects implementing the `Sensor` interface. Formally speaking, if the state of the plant is represented by the column vector $\mathbf{x}$, a `Sensor` realizes an output function $\mathbf{y} = h(\mathbf{x}, t)$. The programmer is required to implement function h by writing a virtual method, called **updateSensor**, which can read the plant state by issuing a call to the `getState` method of the plant. The mechanism used to updated the value of the sensor is the *observer pattern*[10], which will be described below.

The evolution of a plant can be influenced by a set of actuators. An actuator is an object implementing the `Actuator` interface. Each actuator is registered into a position, denoted by an integer number. This convention is to simplify the writing of differential equations.

The integration of the plant differential equations is performed by issuing a call to the `integrate` method exported by the plant.

3.5 CTRLIB

The functional model of the system is expressed using the classes of the CTRLIB package.

According to the adopted data-flow model, CTRLIB offers two types of components: computing units and storage units. Both of these components are framed within a hierarchy of classes. In order to specify a computing unit, the programmer has to derive it from the abstract class `ComputingUnit` and has to provide an implementation for three pure virtual methods: `read`, `execute` and `write`. Once the class is defined, the programmer can instantiate objects from it to be used in different contexts. For example, a class implementing a PID controller is likely to be a reusable component.

The structure of the basic classes of CTRLIB is shown in Figure 10. A `ComputingUnit` is connected to a set of inputs, which are objects implementing the `DataSource` interface, and to a set of outputs which implement the `DataSink` interface. Each computing unit can be associated a special pseudoinstructions triggering the execution of the `read`, `execute` and `write` operation. Such pseudoinstructions derive from the `ContrInstr` class.

Input buffers are realized as classes implementing both the `Sensor` and `DataSource` interfaces. A predefined method, called **sample**, is used to sample the value of the sensor upon the occurrence of certain events. A particular choice can be the use of a RTLIB object implementing a periodic timer. Another possibility is to have the `sample` method called by a pseudoinstruction of a task. The sampled value can be read by a computing unit calling the **getValue** method.

Output buffers are objects implementing both the `Actuator` and the `DataSink` interfaces. Thus they export the **putValue** method to the computing units and the **getCommand** method to the plant. Memory buffers implements both the `DataSource` and `DataSink` interfaces and are used to exchange information between the different computing units. Output and memory buffers can be used with no other efforts than defining the width of the data vector when an object is instantiated. In order to simplify the simulation code, the creation of memory buffers connecting different computing units can be made in a semi-automatic fashion by appropriate programming facilities (see the example in the appendix for further details).

4 Some insight into the hybrid simulation

This section is devoted to showing the main interactions between the different components of the RTSIM tool suite when the libraries are employed to perform a hybrid simulation between a continuous time plant and a digital controller, whose timing evolution is simulated by a RTLIB discrete event model.

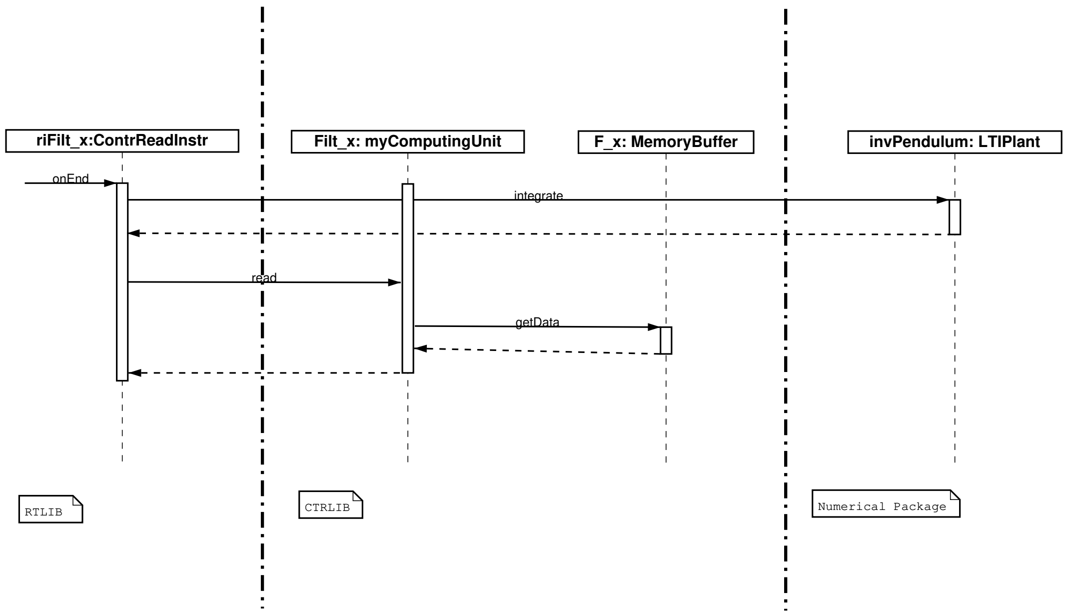


Figure 11: Sequence diagram showing the interactions which take place when an end event for a pseudoinstruction is handled.

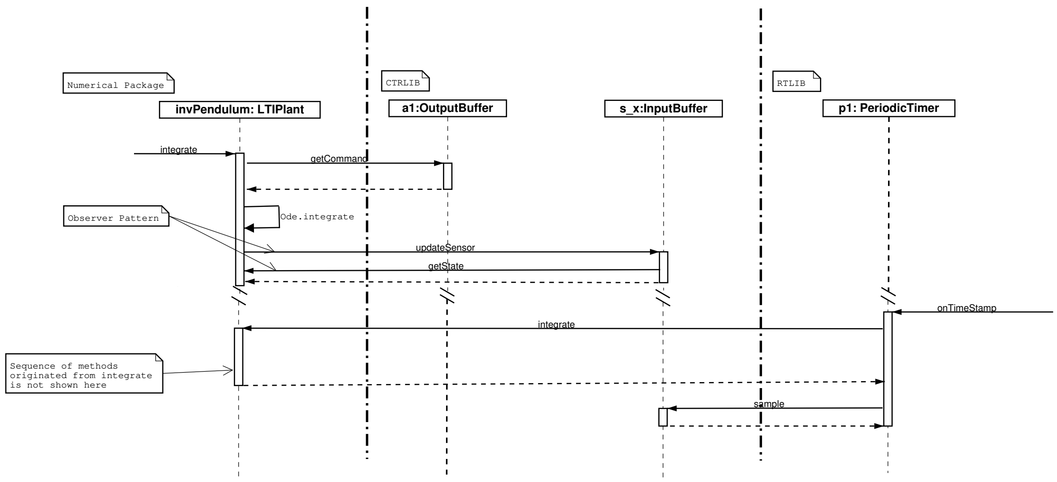


Figure 12: Sequence diagram showing: 1) how the integration is performed, 2) how the sensor values are sampled.

In order to perform this type of simulations, it is possible to exploit an important property of the class of control systems analyzed in this work. This property consists in the fact that the interactions between the plant and the controller take place only in correspondence with specific events. On the one hand, in the time interval separating two writing on the output buffer, the differential equations of the plant can be integrated assuming constant values in the actuators. On the other hand the plant state can be observed through the objects simulating the sensors only when an event associated to the sampling is generated. Therefore, the simulation of the plant and of the controller can be carried out almost independently.

In order to highlight the interactions between different components of RTSIM which take place upon the occurrence of some meaningful events, consider the sequence diagram in Figure 11. The boxes represent RTSIM objects involved in a simulation. The diagram is partitioned according to the three different packages objects belong to. The diagram shows a sequence of method calls which follow up the termination event of the **Read** pseudoinstruction. According to the programming framework of RTLIB, the event has a target entity, in this case the **Read** object, which is responsible of handling it. The first action performed by **Read** is calling a method on the **Plant** object. This method call determines the integration of the differential equation up to the instant related to the event. After the plant state has been updated, the two simulations regarding the plant and the controller have been synchronized. The second action is a call on the "read" method of the computing unit associated with the pseudoinstruction, which, in its turn, reads the data from the buffer.

It is also very interesting to observe how the integration is performed (diagram in Fig. 12). At the beginning of the integration the value of the command variables, contained in the output buffer, is acquired. Then the integration can be performed (by calling the appropriate functions of the OCTAVE library) assuming constant values for the input throughout the integration interval. At the end of the integration, the values contained in the input buffers, which model the sensors, are updated. We observe that keeping the sensor value up to date while the integration is performed would be a useless effort for, as stated above, the interactions between the plant and the controller may take place only at the end of each integration interval (corresponding to the RTLIB generated events). In the same figure, after a while, a clock stamp event is triggered for the timer object. The event handling method calls the `integrate` method on the plant (the sequence of messages originated by this call are not reported for brevity) and then issues a call to a method of input buffer $\mathbf{s}_x$ determining the sampled value to be updated.

5 An application example

In this section we will show a practical application of the proposed tool. The considered system is the inverted pendulum shown in Section 2. Despite its simplicity, this example permits to highlight the potential use of the tool. The code for this example is included in the official distribution of CTRLIB (downloadable from the web site `http://rtsim.sssup.it`), where the interested reader can find the exact parameters of the simulation, and its most important parts are shown in appendix A.

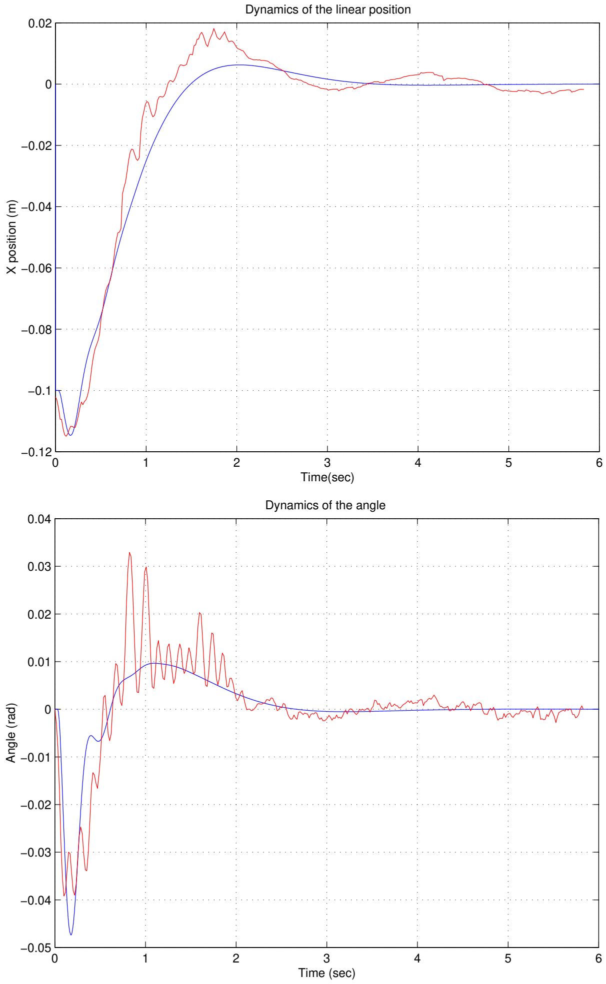
The goal of the control application is to *stabilize* the system from a generic position in the state space into a desired state. The state space of the pendulum is composed of four variables: $[x,\ \dot{x},\ \theta,\ \dot{theta}]^T$, where x is the linear position, $\dot{x}$ is the linear velocity, θ is the pendulum angle and $\dot{\theta}$ is the angular velocity. In the simulations presented in this section, the pendulum starts from the state $[-0.1,\ 0,\ 0,\ 0]^T$ and has to be stabilized into the origin of the state space $[0\ ,\ 0,\ 0,\ 0]^T$. We point out that the results presented below are pretty similar to those obtained with other movements of the system.

The assessment of the influence of the scheduling choices on the performance of the control has to be based on well-defined performance metrics. A possible choice, as proposed in [26], is the use of a quadratic cost function:

$$J = E\{\int_0^+ \infty(\vec{x}^T Q\vec{x} + Ru^2\} \tag{1}$$

where:

- E{.} denotes the expectation value,
- $\vec{x}$ denotes the state vector,
- u denotes the command variable,



(a)
(b)

Figure 13: Dynamics of the x (a) and θ (b) variables for a simulation run compared with an experimental realization.

- the Q matrix and the R constant are two weighting factors.

As shown in Section 2 the system architecture is based on two tasks, `Task1`, `Task2` which manage the data-flow from sensors and compute the feedback control values. We realized a physical implementation of the system based on the SHARK [9] kernel (for details see the WEB site `http://shark.sssup.it`). The probability distribution of the execution tasks was profiled and imported into the simulation model using a suitable class for random variables' generation included in METASIM. In Figure 15 the histograms recorded for `Task1` are reported (this task is the slowest since it operates on a multimedia data-flow).

The simulations were aimed at evaluating the impact of the task frequencies when a classical hard

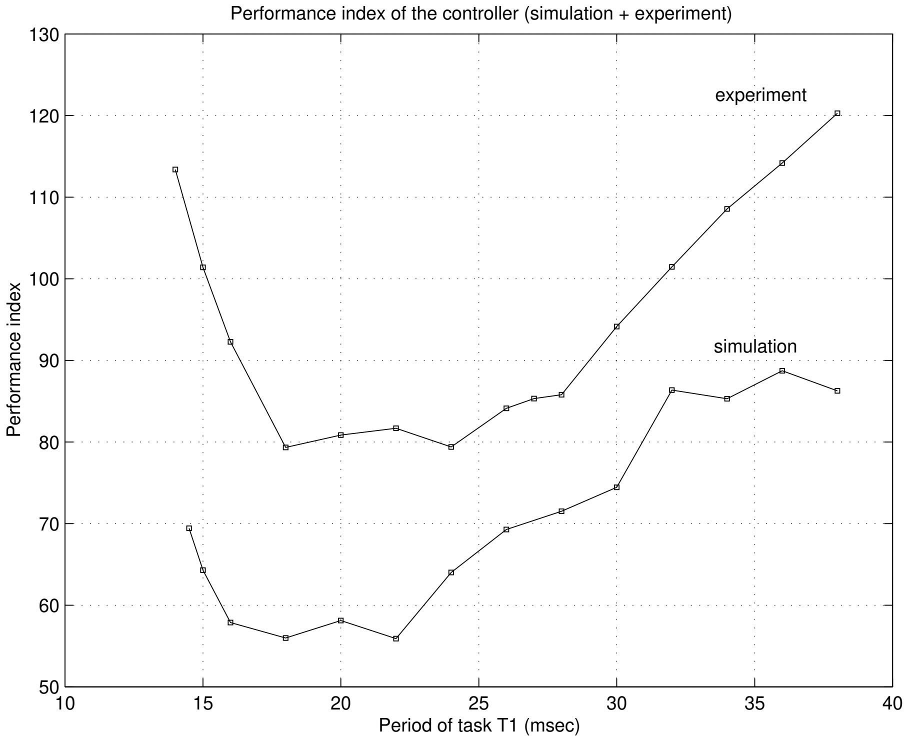


Figure 14: Cost function variations with respect to the activation period of `Task 1`.

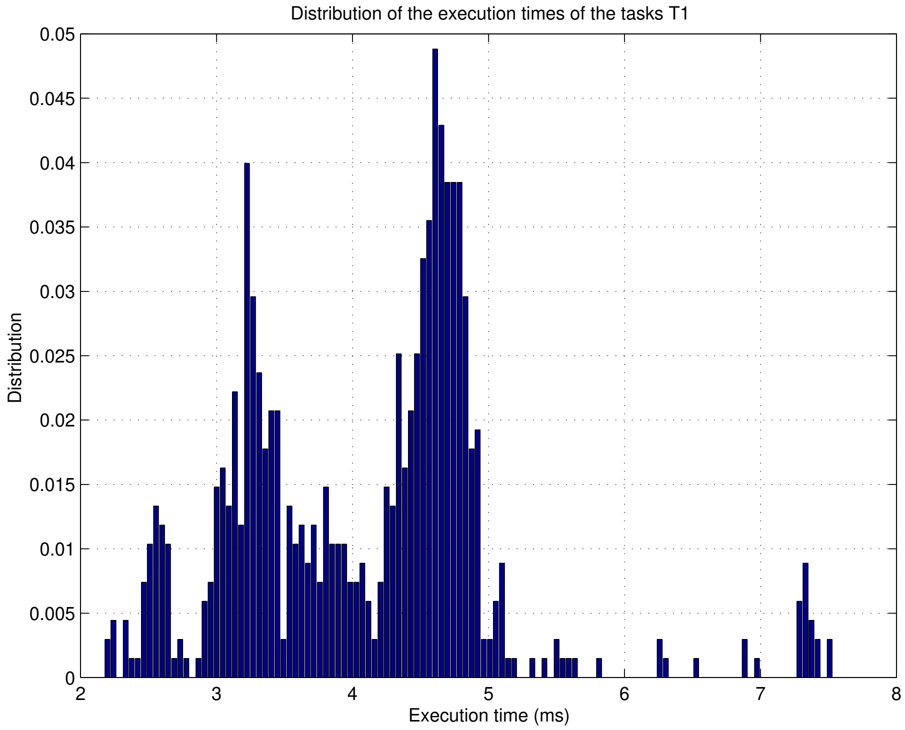


Figure 15: Histograms of the execution time of `Task 1`.

real-time scheduling algorithm (Earliest Deadline First) is used. The schedulability of tasks for this algorithm is ensured, provided that $\frac{C_1}{T_1} + \frac{C_2}{T_2} \leq U_l$, where T_1 and T_2 are the activation periods of the tasks, C_1, C_2 are the worst case execution time and $U_l = 1$. In order to consider residual computation activities (for data logging and man/machine interfaces) we used a lower utilization bound: $U_l = 0.8$. The solid line plot in Figure 14 shows the variation of the cost function with respect to T_1, where T_2 is chosen according to the relation $\frac{C_1}{T_1} + \frac{C_2}{T_2} = 0.8$. The cost function for each point was evaluated averaging the result of twenty execution runs, thus limiting the 90% confidence interval to less than 3% of the expectation value. As it is possible to see, if high values are chosen for T_1, the system tends to instability and the value of the cost function increases. Similarly, if T_1 becomes too small there is a steep degradation of the performance. The latter phenomenon is due to the corresponding value of T_2 which tend to increase according to the schedulability relation. The best performance is achieved by a trade-off choice for the periods. In order to validate the approach, the same experiments were carried out with the physical system. The resulting cost function is shown in dotted line in Figure 14. The behaviour of the cost function is pretty similar in the two plots, apart from the higher values of the experimental data, due to the presence of unmodeled dynamics and noise. This consideration can be visually inferred from Figure 13, where the dynamics of x and θ obtained from a simulation run and an experimental realization are compared.

6 Conclusion and future work.

In this paper a tool for the joint simulation of a plant and of a real-time embedded controller has been presented. By using hybrid techniques the tool permits to provide a realistic modeling for many implementation related issues which are not usually accounted for during the controller design. The tool consists of a complete set of C++ libraries for modeling, simulating and gathering statistical profiles of performance metrics. The application of the tool is particularly useful whenever a given control design is based on heterogeneous dataflows from the environment inducing the use of a complex Hardware/Software implementation. In these cases the tool provides important guidelines in the choice of such parameters as the sampling rates of sensors and, more generally, permits to evaluate different architectural alternatives. The future activities of the RTSIM team will be concentrated on the integration of the tool in more complex design environments, including visual modeling tools and automatic code generation for real-time execution environments.

A Constructing the simulation of the inverted pendulum.

A simulation is essentially a C++ program linked to the different RTSIM libraries. A possible template code for the main function is composed of the following sections:

```
void main() {
        <Definition of the plant>
        <Functional structure of the controller>
        <Architectural structure of the controller>
        <Simulation Run>
}.
```

In our example, assuming a linearized model for the inverted pendulum, it is possible to use the library class `LTIPlant`, and the required instructions are the following:

```
...
Matrix A(4,4);
Matrix B(4,1);
...
//Definition of the plant
LTIPlant InvPend(x0, TICK_PER_SECOND, ode_opts, A, B, "Inverted_Pendulum_1");
ldots
```

The constructor of the plant requires such data as the state transition matrix `A`, which can be composed using the numerical library data structure: `Matrix`, `ColumnVector` and so forth. It is worth observing that all entities, upon their construction, can be assigned a symbolic name identifying them uniquely. This feature can be used for interconnections of objects or any other kind of mutual reference.

As far as the definition of the functional model is concerned, consider the entities depicted in Figure 1. The definition of input and output buffers and their connection to the plant is realized by the following instructions:

```
LinearInputBuffer In_x(c,"Measured_x");
LinearInputBuffer In_th(c1,"Measured_Theta");
OutputBuffer a1(1, "actuator");
plant.attachSensor(&In_x);
plant.attachSensor(&In_th);
plant.attachActuator(&a1);
```

where `LinearInputBuffer` is a predefined class for measuring a linear combination of the state variables. The definition of the computing units is aided by a useful set of predefined objects:

```
MatrixGain FeedbackGain(K, "LQR_Gain");
DigitalFilter Filt_x(lp1_den, lp1_num,"Filt_x");
DigitalFilter Filt_th(lp2_den, lp2_num,"Filt_th");
DigitalFilter Der_x(der1_den, der1_num,"Der_x");
DigitalFilter Der_th(der2_den, der2_num,"Der_th");
Filt_x.attachInput(0,&In_x);
Filt_th.attachInput(0,&In_th);
FeedbackGain.attachOutput(0,&a_1);
```

The digital filters are defined by providing the numerator and the denominator of their transfer functions, which are instances of the `RowVector` class, whereas a `Matrix` object has to be provided when defining a `MatrixGain` object. The last three instructions of this group are used to attach the 0^{th} inputs of the `Filt_x` and `Filt_th` computing units to their input buffers and the output of `FeedbackGain` to the output buffer.

Communications between the computing units is realized through the definition of `MemoryBuffer` objects. This task can be boring and error prone; for this reason it can be aided by a programming facility, based on an instance of the *Abstract Factory* pattern, which is implemented as a function `connect` included into a static object. Each connection is specified by passing to this function a string having a straightforward syntax:

```
SenderSymbolicName.out(port_num)-> ReceiverSymbolicName.in(port_num).
```

The `join/split` keyword can be used to make an ordered grouping/ungrouping of a set of ports. The set of connections in our example is realized by:

```
CUConnector::connect("Filt_x.out(0) -> Der_x.in(0)");
CUConnector::connect("Filt_th.out(0) -> Der_th.in(0)");
CUConnector::connect("join(Filt_x.out(0),Der_x.out(0),Filt_th.out(0), \
              Der_th.out(0)) -> FeedbackGain.in(0)");
```

As far as the archictural design is concerned, consider Figure2. At low level, the kernel and its components (consisting of a scheduler and of a resource manager, can be modeled by the following declarations:

```
//Architectural description of the controller
EDFScheduler sched;
RTKernel edfkern(&sched);
FCFSResManager resman(&sched);
edfkern.setResManager(&resman);
```

The scheduling policy of the tasks is supposed to be enforced by an EDF scheduler and the access protocol to the shared buffer, is, in this case, ruled by a simple First Come First Serve approach. If the interacting tasks were more that two, other choices such as Priority Inheritance, could be made. At a higher logical level we can write the code for tasks and shared buffers and plug them into the kernel facilities (scheduler and resource manager):

```
resman.addResource("R0");
RTTask t1(new DeltaVar(T1), T1, 0, "TaskSlow");
RTTask t2(new DeltaVar(T2), T2, 0, "TaskFast");
t1.setKernel(&edfkern);
t2.setKernel(&edfkern);
```

Finally we have to fill in the tasks with their pseudoinstructions. We could declare each pseudoinstruction and add it to its task or use a shortcut representation based on conventional strings. The second method has been provided to make life easier for the programmer: the following instructions show the code assigned to the first task.

```
t1.addInstructions("read(Filt_x); \
              execute(Filt_x); \
              delay(NormalVar(0,10)); \
              lock(R0); \
              write(Filt_x); \
              read(Der_x); \
              execute(Der_x); \
              delay(NormalVar(0,5)); \
              write(Der_x); \
              unlock(R0);");
```

Each entry of the string corresponds to the creation and to the insertion into the task of a pseudoinstructions. It is also possible to specify parameters a pseudoinstruction should be created with, e.g. the computing unit a `ContrInstr` acts upon, the mutex region for a `WaitInstr` and so on and so forth.

The simulation is simply run by the following statement:

```
Simulation::run(SIM_DURATION*TICK_PER_SECOND, NUM_RUNS);
```

The possibility of running multiple simulation runs is particularly useful when collecting performance statistics over stochastically varying parameters.

References

[1] L. Abeni and G.Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, december 1998. IEEE.

[2] N.C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(8):284–292, Sep 1993.

[3] T. P. Baker. A stack-based allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, december 1990.

[4] T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

[5] G. Booch. *Object oriented design with applications*. Benjamin/Cummings Publishing Company, Inc., 1991.

[6] M. Chen and K. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2, 1990.

[7] John Eaton et al. http://bevo.che.wisc.edu/octave.

[8] J. Eker and A. Cervin. A matlab toolbox for real-time and control systems co-design. In *Proc. of The Real-Time Computiong Systems and Applications*, Hong Kong, China, December 1999.

[9] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1997.

[11] R. Gerber, S. Hong, and M. Saksena. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Transaction on Software Engineering*, 21(27), 1995.

[12] T.M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Journal of Real-Time System*, 9, 1995.

[13] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 89–99, Phoenix, December 1992.

[14] N. Kim, M. Ryu, S. Hong, M. Saksena, C. Choi, and H. Shin. Visual asessment of a real-time system design: a case study on a cnc controller. In *Proceedings of the IEEE Real-time Systems Symposium*, 1996.

[15] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabla, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The mars approach. *IEEE Micro*, 9(1), February 1989.

[16] W. Kreutzer. *Systems Simulation - Programming Styles and Languages*. Addison-Wesley, 1986.

[17] A.M. Law and W.D. Kelton. *Simulation modeling and analysis*. McGraw-Hill Book Company., 1991.

[18] Edward A. Lee. Computing for embedded systems. In *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 2001.

[19] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1992.

[20] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.

[21] Giuseppe Lipari and Giorgio Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of Systems Architecture*, 46:327–338, 2000.

[22] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[23] Luigi Palopoli, Luca Abeni, Fabio Conticelli, Marco Di Natale, and Giorgio Buttazzo. Real-time control system analysis: An integrated approach. In *Proc. of the Real-Time Systems Symposium*, Orlando, Florida, November 2000.

[24] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin. On task schedulability in real-time control systems. In *IEEE Real Time System Symposium*, December 1996.

[25] Lui Sha, Ragunathan Rajkumar, and john P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), September 1990.

[26] K.G. Shin, C.M. Krishna, and Y. Lee. A unified method for evaluationg real-time computer controllers and its application. *IEEE Transactions on Automatic Control*, AC30(4):357–366, April 1985.

[27] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1, July 1989.

[28] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1994.

[29] M. Spuri and G.C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-Time Systems*, 10(2), 1996.

[30] M. Spuri, G.C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1995.

[31] D.B. Stewart, R.A. Volpe, and P.K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE trans. on Software Engineering*, 23(12), 1997.

[32] Ian Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE Real Time System Symposium*, 1996.

[33] C.M. Kirsch T. Henzinger, B. Horowitzm. Embedded control systems development with giotto. In *Proc. of ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'2001)*, June 2001.

[34] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real Time Systems*, 6(2):133–151, Mar 1994.

[35] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *J. of Real-time systems*, 14:219–250, 1998.