SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–32 *Prepared using* speauth.cls *[Version: 2000/03/16 v2.12]*

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

Luigi Palopoli[1], Giuseppe Lipari[1,*], Gerardo Lamastra[2], Luca Abeni[1], Gabriele Bolognini[1], Paolo Ancilotti[1]

[1] *Scuola Superiore Sant'Anna, piazza Martiri della Libertà 33, 56127 Pisa (ITALY)*
[2] *Telecom Italia Lab*

SUMMARY

This paper presents an object oriented software tool, called RTSIM, aimed at simulating real-time embedded controllers. The tool consists of a collection of C++ libraries permitting a separate specification of the functional behaviour of the controller and of the hardware/software architecture 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 and 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.

KEY WORDS: Simulation, Control systems, Real-Time systems, C++ library

INTRODUCTION

During the last years, the application of embedded control systems has become a dominant factor governing the commercial success of several engineering products. The best known example is represented by the automotive industry: from their seminal applications to restricted niches of production vehicles, embedded devices are gradually becoming ubiquitous components of modern cars. In other applications, like avionics and factory automation, the

*Correspondence to: Giuseppe Lipari, Scuola Superiore Sant'Anna, piazza Martiri della Libertà 33, 56127 Pisa (ITALY) - E-mail : lipari@sssup.it

Copyright © 2000 John Wiley & Sons, Ltd.

introduction of embedded control devices began earlier, but newer and newer functionalities, which could not even be conceived only a few years ago, are making inroads.

The integration of flows of data from heterogenous sensors, having different requirements in terms of sampling rates and computation times, induces almost naturally concurrent implementation schemes. The ability of the system designer to specify, manage, and verify the functionality and performance of real-time concurrent processes (tasks) turns out to be a crucial success factor. Moreover, in the design of mass-produced embedded systems, the choice of hardware has a strong influence on the economy of the solution. Therefore, even in front of increasingly complex problems, the push towards minimization of computing hardware cost remains a dominant factor. In this context, an increasing emphasis is put on the effectiveness and on the efficiency of the production process of real-time software. Traditional development cycles tend to separate "rigidly" the work of control engineers from that of software engineers but the final outcome is often far from optimal in terms of performance/cost criteria.

In order to introduce a profound innovation in this field, the availability of co-design tools spanning over diverse engineering disciplines is of utmost importance.

This paper focuses on one of the most familiar problems in real-time control software design, i.e. how the performance of a controller is affected by architectural and implementation choices (e.g. the decomposition of feedback controllers into tasks, the allocation of computation resources to tasks, the scheduling of the shared resources, etc). Realistic and quantitative answers to this question during the early phases of the development are a precious tool for product development.

The concept of performance evaluation for a real-time controller can be developed along different directions. Most of the research in the area of real-time computing has studied the performance of concurrent software systems under the viewpoint of their timing behaviour. Ever since the seminal work of Liu and Layland [25], a fundamental performance metric is considered to be the tasks' schedulability, i.e. the ability for a set of tasks to execute respecting their assigned deadlines. For some classes of real-time applications (qualified as soft real-time), a more useful performance metric is represented by the probability for each task to execute respecting its deadlines [1, 37, 18]. At a higher level of abstraction, the "collective" timing performance of a set of tasks has been evaluated in terms of end-to-end delay, output jitter, and other metrics [11].

The compliance of a controller's timing behaviour with some specified requirements (e.g. schedulability) is not always sufficient to characterize performance at the system level. Classical performance metrics normally used during the control synthesis consider the step response (rise time, overshoot, etc.) or the closed loop transfer function. Quadratic cost functions, or other metrics such as $\mathcal{H}_2/\mathcal{H}_\infty$ norms, are the foundation of popular procedures for analytical control synthesis. However, during the control synthesis phase, effects deriving from the implementation architecture are not usually taken into account. The difficulties in finding tractable analytical models for the stochastic delays deriving from data dependencies and scheduling jitter and the lack of adequate modeling and simulation tools, induce the control designers to synthesize control laws assuming null or fixed delays from the underlying implementation platform. As a consequence, even a software design complying with the deadline constraints can result into a poorly performing system. These problems are detected only during the late phases of the design cycle, and the solution is often sought by cycling

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

through a long series of costly trial-and-error iterations among the different phases of the development cycle.

In this paper, we present a simulation tool, called RTSIM, which alleviates these difficulties, permitting us to efficiently deal with different aspects of the control synthesis. The main goal of RTSIM is to permit the joint simulation of a real-time controller and of the controlled plant, collecting performance measures either on the timing behaviour of the controller or on the quality of the plant dynamics. Specifically, a designer is allowed to specify:

- a set of plants (specified through their differential models) connected to a distributed control system by means of sensors and actuators,
- the functional behaviour of the controller,
- the architectural components of the implementation (real-time tasks, RTOS, shared resources),
- the mapping of functional behaviours onto the architectural components.

By leveraging a complete orthogonalization of the functional and architectural designs, RTSIM enables: 1) an easy comparison of different implementation approaches for the same functionalities, 2) a performance based tuning of such design parameters as the tasks' activation rates/scheduling priorities. The tool is organized as a collection of C++ libraries that include programming facilities for defining stochastic parameters (e.g. for tasks' execution times, network packets dimensions, etc), for collecting performance statistic and for recording events of interest on execution traces.

A very important feature of the tool is that it encompasses the best known solutions for real-time CPU scheduling (either on single or on multiprocessor boards) and for bounded delay sharing of resources, as predefined library classes. The functional specification of the system is provided by interconnecting a set of reusable components, according to a syntax closely related to well-known dataflow paradigms[†] Another important feature of the tool is the presence of a well defined programming framework guiding users in developing their own functional and architectural components. Once the design of the controller has been settled and properly tuned, its implementation on a real-time operating system is straightforward. The fine grained modeling of such software architectural components as real-time tasks, schedulers, synchronization protocols and so on, enables a very accurate simulation of the system's performance.

As far as the simulation of the plant is concerned, RTSIM exploits the functionality of a powerful mathematical library, called OCTAVE [7], embodying *state of the art* solutions for the integration of differential equations.

[†]The term "dataflow" generally denotes a subclass of Kahn processes [13], introduced by Dennis in 1975 [6]. However, since many software environments claim variants of this model even if their semantics bear little resemblance with that proposed by Davis, throughout this paper a loose meaning for this term will be used. Therefore, dataflow will intuitively denote a directed sequence of transformations applied on data flowing from inputs to outputs.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

STATE OF THE ART

The best known tool suite for simulating control systems is MATLAB. The MATLAB/Simulink platform is an excellent choice to model and simulate a plant and a functionally described controller. Moreover, it permits one to automatically generate a prototype on a target real-time operating system (by the use of the *Real-Time Workshop* tool). However, it is not possible to immediately to model generic Hardware/Software architecture and scheduling algorithms. To cope with this shortcoming, a MATLAB tool to simulate a real-time scheduler in a Simulink block is proposed in [8]. 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 functional and architectural specifications hinders the application of the tool to complex systems having event driven and/or time driven activities.

An interesting product, mainly targeted to the automotive industry, is Ascet-SD, by Etas engineering tools. The tool includes an easy to use graphical interface that permits modeling the functionalities of a controller in a Simulink like environment. The main focus of Ascet-3D is the generation of high quality real-time code for prototyped or production hardware.

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 first softwares aimed at simulating real-time scheduling was produced by Audsley et al. [3]. The tool permits modeling a system of real-time periodic and aperiodic tasks through a scripting language.

A well-known commercial product in this class is TimeWiz, by 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 one to describe complex systems in a fairly general way. However, the tool does not allow one 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 idea of separating functional and architectural specification is well supported by the VCC tool, produced by Cadence corp. Functional behaviours can be specified using different syntaxes (including the C/C++ language) and the tool permits one to map a given functionality either on hardware components (e.g. Asic) or on software (e.g. concurrent tasks) in order to pursue different performance/cost tradeoffs. The performance assessment in VCC regards mainly the timing behaviour of the components and the simulation of a continuous time plant is not directly supported.

The GIOTTO programming language [36] 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 the usual paradigm of separating the functional from the timing behaviour (hard schedulability requirements). Time predictability (schedulability) is obtained by restricting the design to a time-triggered architecture [15]. A remarkable advantage of this paradigm is the elimination of input and output jitters. However, the introduced delays can be a very pessimistic solution in many cases. Moreover, the time triggered approach does not easily cope with event-driven systems.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

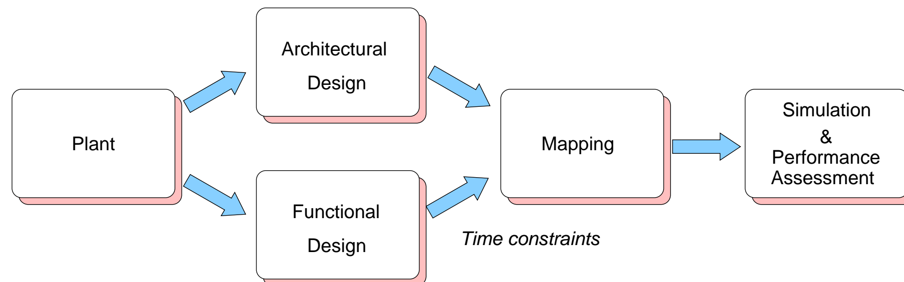


Figure 1. Typical design process for the specification and the simulation of a real-time controller.

An integrated design of real-time control systems encompassing performance and schedulability concerns was first proposed by Seto et al. [27]. In this work an optimization procedure for the activation frequencies of control threads is proposed; the goal is maximizing the controller's performance under schedulability constraints. The paper is inspired to the evaluation approach for embedded controllers suggested by Shin et al. [29]. Other noteworthy results on this problem are presented by Kim et al. [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 RTSIM may be a very useful aid to validate the assumptions and the result of these methods and of any other co-design procedure.

DESIGN PROCESS AND MODELING PRIMITIVES

The construction of a simulation model for RTSIM is carried out considering two orthogonal viewpoints: the *functional behaviour* of the controller and the *HW/SW architecture* of its implementation. In Figure 1, an overview on a typical design process based on RTSIM is depicted.

The functional design, starting from the mathematical model of the plant and of its interactions with the environment, produces a model of the functional behaviour. The functional behaviour specifies a sequence of operations to be performed on data flowing through the controller. Such operations include the computation of the feedback control law, the extraction of meaningful information from sensors and so on. The functional design also produces a set of timing constraints based on the dynamics of the plant and on the physical limitations of sensors and actuators.

The architectural design can be carried out almost independently. This activity leads to the definition of a model consisting of software tasks, schedulers, network protocols and so on.

The functional design is then mapped onto the architectural design, wrapping up the functional components into corresponding architectural entities having specified requirements

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

in terms of execution time, length of messages and so on. In this phase, the timing constraints are translated into real-time constraints on the processes and on the messages on the network.

The separation of the functional and architectural viewpoints permits us to easily test and compare different implementations for the same functional specification in order to identify the solution which best fits the performance/cost tradeoffs of the project.

Finally the system model, composed of its functional and architectural specification, can be simulated obtaining different types of results. A first possibility is to analyze the execution traces (by an appropriate visual tool) to verify if the design meets the desired timing constraints. Moreover, statistics can be collected on the occurrence of events measuring such quantities as the average delay, the jitter and so forth. Most importantly, fundamental information can be derived on the control system's performance by using typical control theoretical metrics (overshoot, rise time, integral cost functions). If the resulting performance is not satisfactory, it is easily possible to return back to any of the previous phases and change the system parameters, the system components (schedulers, communication protocols) and even the entire architecture.

In the rest of 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 these primitives are applied to a practical case.

Modeling the functional behaviour

The separation between the functional and architectural specification is aided, in the RTSIM tool, by the use of a dataflow approach for the functional modeling of the system. Dataflow models are a well-suited tool in the design of real-time software [34, 39] and they are provided, in different flavours, by a variety of tools including Simulink, Ptolemy [20], and GIOTTO [36].

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 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 common use 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 continuous

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

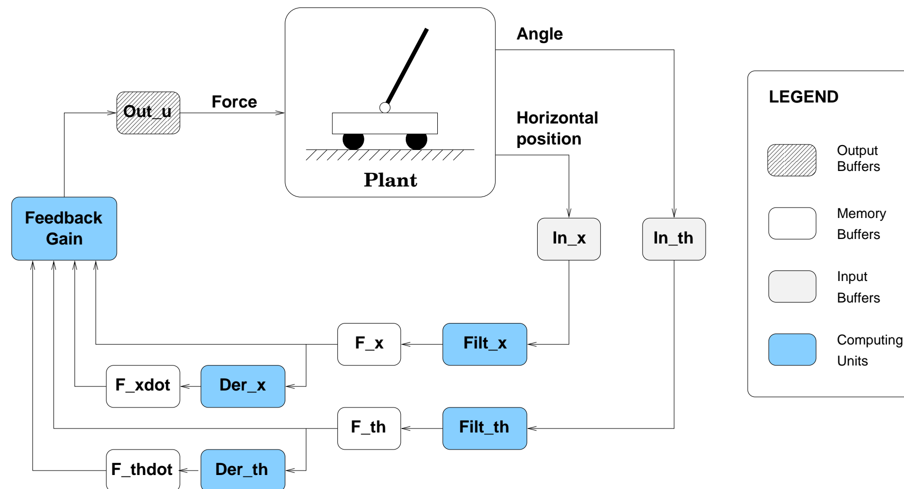


Figure 2. Functional design of a simple controller for an inverted pendulum.

time quantity. RTSIM offers also the possibility of modeling sensors whose measurement are affected by band-limited white noise. From the controller's side, an input buffer models an I/O card whose content changes when a sampling command is received. 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 converters, i.e. when a computing unit writes 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.

It is important to observe that when a functional model is constructed no particular assumption is made either on the hardware implementation of a storage unit, or on the way concurrent access requests should be scheduled.

Example. An example of functional design is reported in Figure 2. 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 variables are used by a computing unit (`FeedbackGain`) to compute the force to be applied to the cart which is stored into an

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using speauth.cls

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).

Modeling the architecture of the system

In our model, a **task** (or process) is a finite or infinite sequence of requests for execution, or *jobs*. Each job executes a piece of code (a sequence of instructions) implementing some functional behaviour. When a job is activated, we say 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 are used to model its timing behaviour. Basically, an instruction is modeled by an execution time (which can be deterministic or stochastic) 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. The state of the art algorithms for CPU scheduling (such as Fixed Priority, Rate Monotonic [25], Earliest Deadline First (EDF) [25], Proportional share [35]) 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 [22], Sporadic Server [30], Constant Bandwidth Server [1], etc). Priority inversion in accessing mutually exclusive resources [28] can be avoided by using appropriate synchronization protocols implemented in the tool, such as the Priority Ceiling Protocol [28] or the Stack Resource Policy [4].

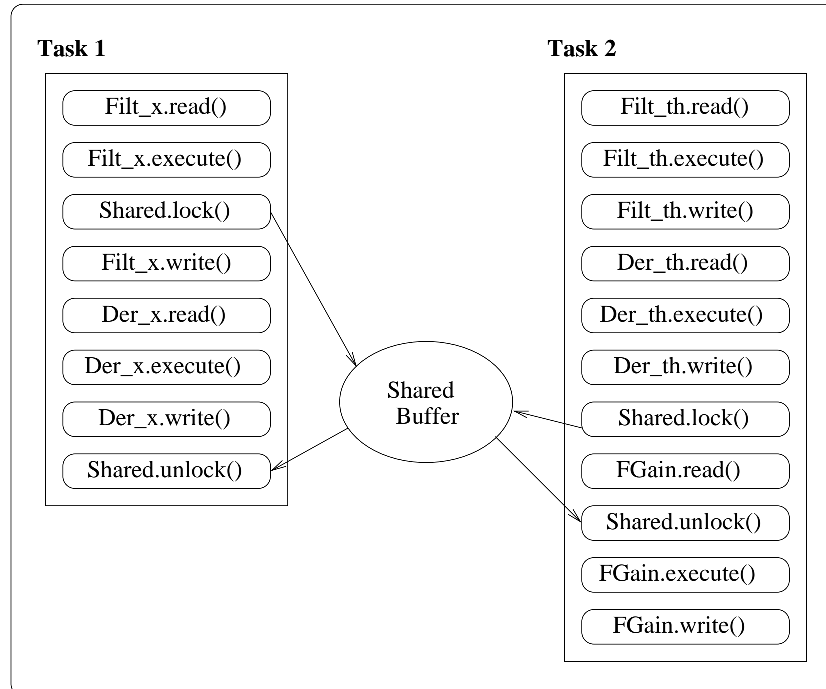
Finally, the system can be comprised of several computational nodes connected by **network links**. Tasks on different nodes can communicate by means of *real-time messages*. A communication resource is modeled by a shared physical link, an access protocol and a real-time message scheduler.

Example. A better understanding of what is really meant in RTSIM by "architecture of the system" can be achieved by getting back to the example shown in Figure 2.

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

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls



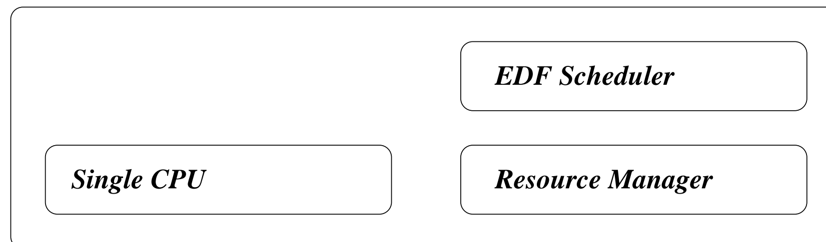


Figure 3. Architectural design for the example shown in Figure 2. The instructions inside each task are executed sequentially at every activation.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

`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`.

The main architectural components for this example are depicted in Figure 3: each task is represented by a box containing the list of instructions executed every period. The two tasks communicate by means of a shared buffer accessed in mutual exclusion (through the `shared.lock()` and `shared.unlock()` instructions). The concurrent execution of the two tasks is possible using a scheduler component (named `EDFScheduler`) endowed with the Earliest Deadline First scheduling policy [25]. A *resource manager* is used to select the access policy: in this example we use a simple blocking policy. 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 that may be performed at the same rate. A potential drawback of this choice is the lack of modularity. For example, `Task 2` could be replaced by two tasks, the first operating the `F_th` and `F_thdot` computing units, and the second operating the gain unit (`FeedbackGain`). In this way, 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 3, 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.

Assessing performance

Once a system has been modeled, a designer is provided with different opportunities to simulate the system and evaluate the quality of the design. A simulation consists of a sequence of events associated with relevant situations in the architectural model of the system (i.e. task arrivals, task terminations, deadline expirations etc.), which may trigger actions in the functional model. Therefore, events are the fundamental element of any simulation and they can be used in a variety of ways to evaluate the system's performance. With this respect, the first possibility a designer is offered, is to record all events of a simulation, or a meaningful subset, into a trace file. The toolset comprises a utility, called RTTracer, which interprets a trace file and visualizes events in a clear form (see Figure 6). In order to facilitate portability RTTracer is

Copyright © 2000 John Wiley & Sons, Ltd.

entirely written in Java. The application of RTTracer is particularly useful for performing a "temporal" debugging of a complex system when simulations reveal a failure in respecting deadlines for some task or network message. The second important possibility is to define statistical probes, which can be attached to objects to measure the occurrence of events. Statistics can be collected over multiple runs when such parameters as computation times are assigned to vary stochastically according to specified distributions. The main use of this feature is to derive such measures of the system's performance as jitter, latency of data, end-to-end delays on pipelines of tasks and so forth. Finally, particular types of input buffers can be used to measure the evolution of some quantities of interest in the plant (very much like in Simulink). Such units can be connected to files in order to record the time evolution of the observed quantities. In a similar way it is possible to define performance probes which can, for instance, integrate over time the squared norm of the measured quantity.

Example. In order to show some of the possibilities offered by RTSIM, we get back to the example of the inverted pendulum introduced in the previous sections. The code for this example is included in the official distribution of RTSIM (it can be downloaded from the web site `http://rtsim.sssup.it`), where the interested reader can find the exact parameters of the simulation.

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 $[0\ ,\ 0,\ 0,\ 0]^T$.

The functional and the architectural model of the controller have been introduced above. In order to provide an experimental validation for the use of the tool, 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 execution times of the tasks were profiled and imported into the simulation model.

A first element of information on the correctness of the system's behaviour can be obtained by visually inspecting the execution traces of the tasks. In Figure 6 the RTTracer output for a simulation is shown. The assumed hard real-time algorithm is the classic Earliest Deadline First. In order for the compliance of the control design with some performance expectation to be verified, it is very important to show the evolution of state variables in time. In Figure 4, the dynamics of x and θ obtained from a simulation run are shown. In order to verify the quality of the simulation we report on the same plot also data obtained from an experimental realization. For both simulation and experimental dynamics convergence to zero takes approximatively four seconds.

In order to achieve a quantitative assessment of the influence of the scheduling choices on the control performance, it is necessary to introduce a performance index. A possible choice, as proposed by Shin et al. [29], is the use of a quadratic function:

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

where:

- E{.} denotes the expectation value (calculated over stochastically varying parameters),

Copyright © 2000 John Wiley & Sons, Ltd.

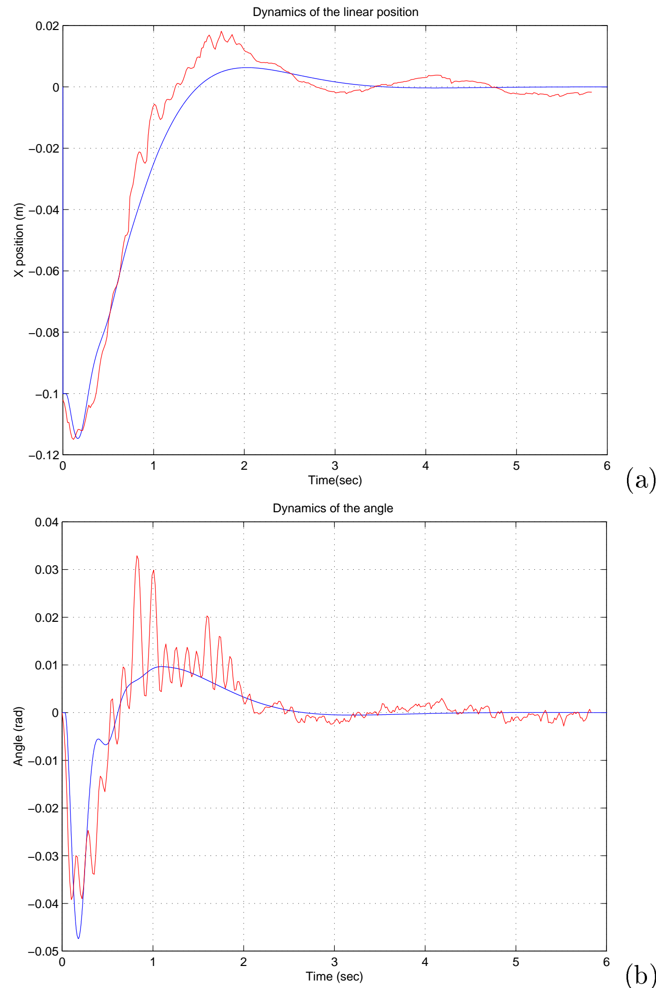


Figure 4. Dynamics of the x (a) and θ (b) variables for a simulation run compared with an experimental realization.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

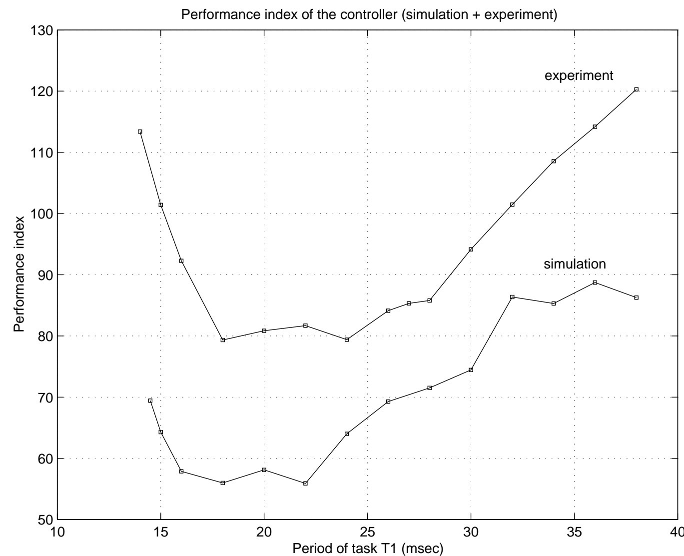


Figure 5. Performance index variations with respect to the activation period of `Task 1`.

- $\vec{x}$ denotes the state vector,
- u denotes the command variable,
- the Q matrix and the R constant are two weighting factors.

As said above, a particular type of input buffer can be attached to the state and to the input variables in order to compute $\int_0^+ \infty(\vec{x}^T Q\vec{x}+Ru^2$ as the simulation takes place. The expectation value can easily be approximated by attaching a statistical probe to the storage unit and by collecting the measures over a sufficient number of runs.

The simulations were aimed at evaluating the impact of the task frequencies. 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 times and $U_l = 1$. Residual computation activities (for data logging and man/machine interfaces) where considered by using a lower utilization bound: $U_l = 0.8$.

The simulated and the experimental plots for the performance index are reported in Figure 5. In the horizontal axis period T_1 is varied while T_2 is chose accordingly to the relation $\frac{C_1}{T_1} + \frac{C_2}{T_2} = 0.8$. The performance index for each point was evaluated averaging the result of twenty execution and simulation runs. As a remark, the evaluation of each point required approximately forty seconds on a PC with an Athlon 1.2 Ghz processor running the Linux operating system.

As it is possible to see, if high values are chosen for T_1, the system tends to instability and the value of the performance index increases. Similarly, if T_1 becomes too small there is a steep

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

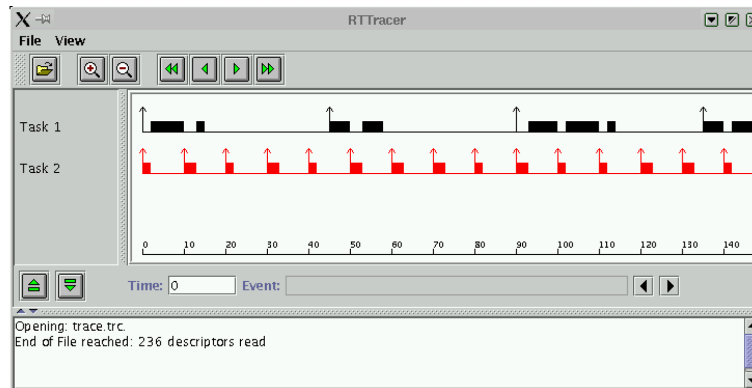


Figure 6. Graphical output of a trace of a RTSIM simulation.

degradation of the performance. The latter phenomenon is due to the corresponding value of T_2, which tends to increase according to the schedulability relation. The best performance is achieved by a trade-off choice for the periods. The behaviour of the cost function is pretty similar in the two plots, except for the higher values of the experimental data. This difference, which is also evident in the plots in Figure 4, is due to the adoption of a simplified model for the plant. As a matter of fact, such aspects as the transfer function of the motor, the sensors and process noise and the nonlinearities on the actuators were neglected in the construction of the plant model, since the accuracy level obtained with the simplified model was deemed satisfactory for the purposes of this work.

DESCRIPTION OF THE TOOL

Summarizing the illustration above, RTSIM consists of a collection of C++ libraries containing three types of objects:

- continuous time plants,
- functional components of control software, and
- architectural components of control software.

The distinction of these conceptual domains dictated a decomposition of the software into three interacting packages, as shown in Figure 7.

The package denoted as “Numerical Package” is used to model and simulate plants. Objects living in this package evolve in continuous time and they are described by means of differential equations. The package called “CTRLIB” is used to construct the functional model of the system. Objects belonging to this package do not posses an intrinsic concept of time evolution: their actions are triggered by objects belonging to other packages (in particular to RTLIB).

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

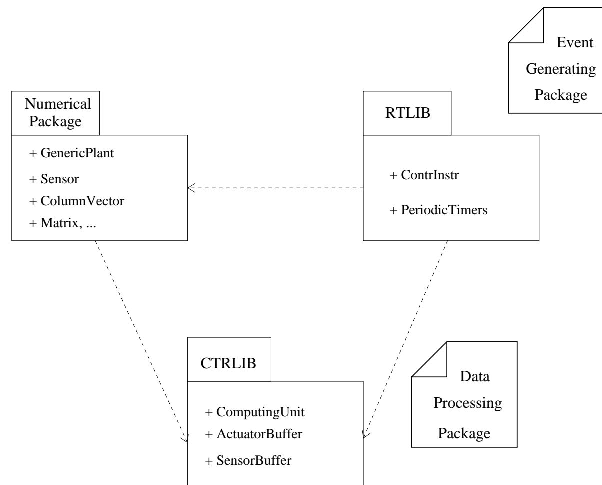


Figure 7. Main components involved in a RTSIM based simulation of a real-time controller

The "RTLIB" package is used to describe the architectural components a functional model is mapped onto. Objects evolve according to a discrete event model of computation [19]: they react to events and are able to generate other events in their turn.

When designing the class hierarchies for the packages, we wanted to achieve a high degree of decoupling so as to facilitate an autonomous evolution of the tool along the three different dimensions. For instance, in our intentions, a developer should be able to extend the library of computing units with new algorithms without caring too much for the structure of kernels or scheduling algorithms and vice versa. In order to achieve this goal, structural relations between components and their interactions had to be captured through a set of clear interfaces. Particularly, for what concerns the interaction between the three packages, we could leverage an important property of the addressed systems: meaningful interactions between plants and controllers take place only on the occurrence of a specific set of events generated by RTLIB. On one hand, in the time interval separating two writings on the output buffer, the differential equations of a 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 with sampling is generated. Hence, a substantial role in the RTSIM simulation environment is played by the generation of discrete events for RTLIB. This is achieved by using the Metasim library, which is a small software layer developed at the Retis

[‡]More sophisticated actuator schemes such as first order hold or analog loops can easily be modeled in the plant description.

Copyright © 2000 John Wiley & Sons, Ltd.

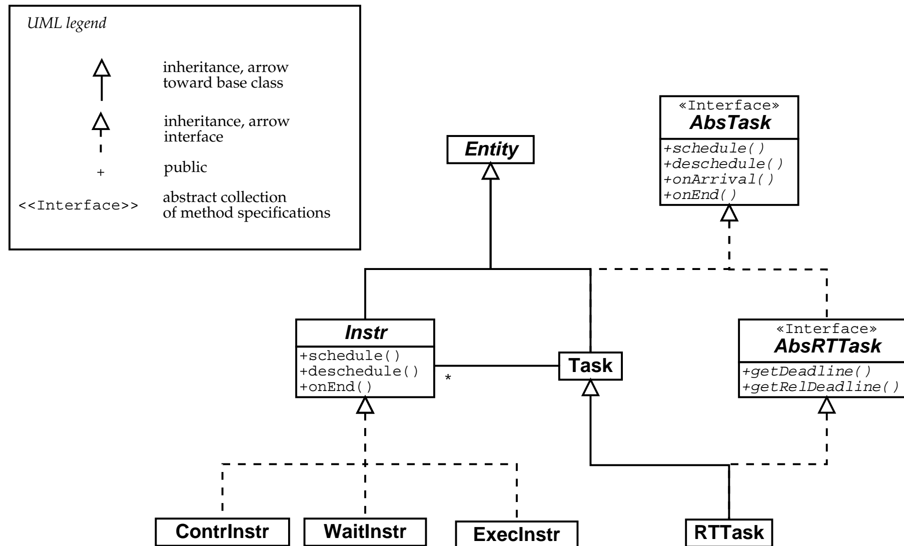


Figure 8. Class diagram representing the Task family of classes.

Lab of Scuola Superiore S. Anna. Metasim provides the basic classes for writing generic discrete event simulations [5, 16, 17] and a clear framework to use them.

The remainder of this section is devoted to a short description of the three packages (both structural and behavioural) and of their most important interactions. For obvious space constraints, the description is far from complete. The interested reader is referred to the technical documentation of the tool [23]. The components of the libraries and their behaviour are described by the UML graphical notation [26].

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, and network links.

Tasks. One of the most important entities needed to specify a software architecture is the task. The family of classes for modeling tasks is shown in Figure 8 as a UML class diagram. In order to de-couple the interface of a task from its internal implementation, we decided to provide an abstract interface `AbsTask` that exposes the basic methods to handle a task (`schedule`, `deschedule`, `onArrival`, `onEnd`). This same interface is used by all entities that can be scheduled: for example, an aperiodic server will implement the `AbsTask` interface (see the server section below).

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

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

- ExecInstr that 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 random execution time;
- WaitInstr and SignalInstr that model the wait and signal system calls for concurrent access to shared resources using semaphores; and
- the ControlInstr family of classes that model the execution of computing units.

A programmer inserts instructions into tasks, just as she/he would write a real implementation. Instructions are executed sequentially [§] and have a duration, which can either be deterministic or specified as a random variable.

In the types of applications we want to model, tasks have timing requirements. The most common constraint is the *deadline*: the *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 arrival 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 8). It comprises the getDeadline() and getRelDeadline() 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-processor nodes. As in the case of tasks, we found it useful to introduce an abstract interface, AbsKernel, capturing the minimum set of services required to any type of kernel. In particular we identified the following services:

- task insertion into a ready queue (method activate),
- task extraction from the ready queue (method suspend),
- task dispatch (method dispatch): the currently executing task is revoked use of the CPU, which is assigned to the first task in the ready queue. In multiprocessor systems the kernel performs this operation on each processor under its control.

The kernel interface also includes methods to handle the most important events a kernel can receive: the arrival of a new task (method onArrival) and the termination of a task's job (method onEnd).

Notice that, at this point, we have not yet introduced any notion of "task priority". In fact, different scheduling policies compare tasks based on different parameters. For example, the Rate Monotonic scheduler requires a static priority to be assigned to each task, whereas the Earliest Deadline First scheduler uses the absolute deadline of a job to determine the task

[§] Thus far, this model has proven sufficiently expressive, since we restricted the application of the tool to modeling classical "data-flow" oriented real-time control applications. In the future, we plan to model also multimodal applications for which a direct support for branches will be necessary. The addition of this feature requires slight modification to the structure of RTLIB and it is planned for future revisions.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

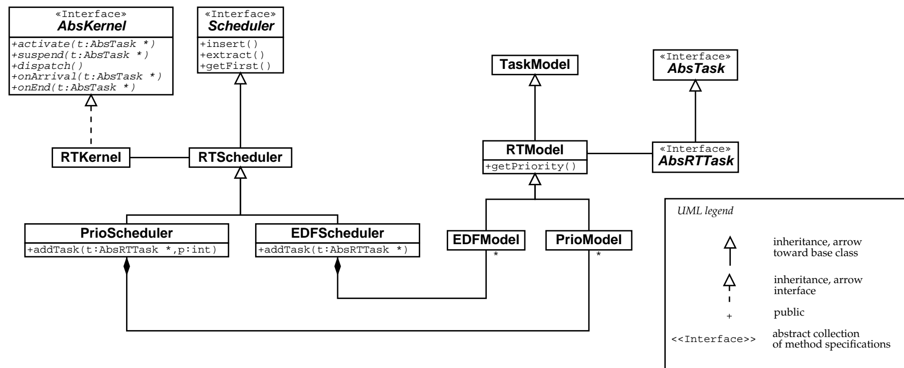


Figure 9. Class diagram representing the Kernel family of classes.

priority. Moreover, some scheduling policies (like Proportional Share or Round Robin) do 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 queuing 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 in the *wrapper* class `TaskModel`: thus, the task implementation is independent from the scheduling algorithm (as in the *Adapter Pattern* [10]). The `TaskModel` hierarchy of classes is similar to the Scheduler hierarchy: every scheduler corresponds to a task model. In Figure 9 the inheritance relationships between these classes are summarized.

The current distribution of RTLIB provides single processor and multi-processor kernels as predefined components, with any of the following scheduling policies: FIFO, EDF, fixed priority (FP) and rate monotonic, and EEVDF [35]. For the multi-processor versions of EDF and FP, it is possible to allow/disallow migration: in the latter case, tasks must be statically allocated to processors.

Example. The notification mechanism and the way events are handled in RTLIB are better explained with a practical example. The sequence diagram shown in Figure 10 captures a snapshot of the system described in Figure 3 when a preemption occurs: while `Task 1` is executing, `Task 2` is activated (arrives) and, having a higher priority, preempts `Task 1`.

When `Task 2` is activated, its arrival event is processed: as a consequence, the `onArrival()` method of `Task 2` is invoked. After updating its internal status (for example recording the arrival time and resetting the current instruction pointer to the first instruction), `Task 2` calls the `onArrival()` method of the kernel. The kernel, in turn, inserts the task in the ready queue (calling `s.insert()`), and checks if this task is now the first element in the queue. If

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

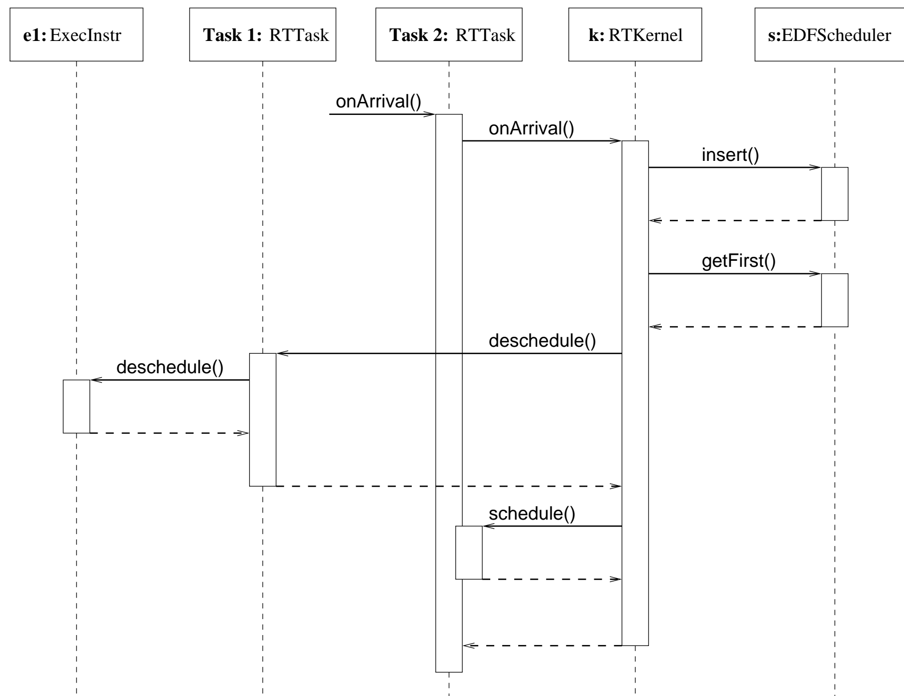


Figure 10. Sequence diagram: `Task 2` preempts `Task 1`.

so, a preemption must occur: the current executing `Task 1` yields the processor and `Task 2` becomes the current executing task.

Hence, `Task 1` must be signaled calling its `deschedule()` method; in turn, it calls the `deschedule()` method of its currently executing instruction. Finally, `Task 2` 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 systems is to associate a *server* to the soft aperiodic tasks. A server is characterized by certain parameters specifying exactly its performance expectations. Several aperiodic service mechanisms have been proposed under RM [22, 21, 2, 38] and under EDF [31, 12, 33, 32, 1, 24] scheduling.

The `Server` class models these algorithms.

We noticed that in almost all the aperiodic server mechanisms, a server is treated as a particular kind of task and is inserted in the ready queue together with the other regular

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

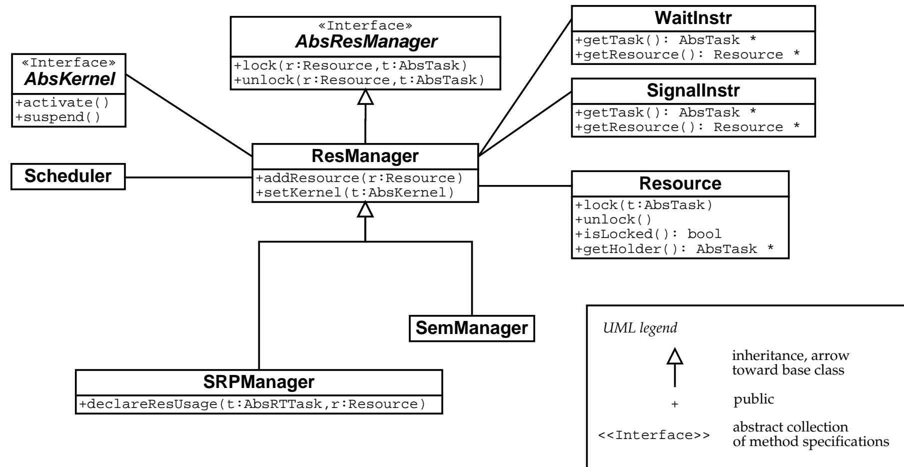


Figure 11. Class diagram representing the Resource Manager family of classes.

tasks. For this reason, we decided to derive the server class from the `AbsTask` interface, so that the scheduler does not need to distinguish a regular task from a server. The main advantage is that, when implementing the server algorithm, the scheduler module can be reused without any modification. On the other side, a server handles aperiodic tasks just as a kernel does: when several aperiodic requests are pending, the server must choose which one must be serviced next. For this reason, the server class also derives from the `AbsKernel` interface. In this way, a task has not to distinguish whether it is served by a server or by a regular kernel, and we can re-use the same code for the task class. In the current RTLIB distribution, the polling server, deferrable server (DS), sporadic server (SS), total bandwidth server (TBS), and constant bandwidth server (CBS) are provided as predefined components.

Sharing other resources. Sometimes, tasks access mutually exclusive resources: for example, tasks can access the same memory block that is protected by a mutex semaphore. 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 of a `Resource Manager`, which is the entity that manages the operations on a semaphore, holding the blocked tasks in queues. Tasks can operate on semaphores by means of `WaitInstr` and `SignalInstr` instructions.

In Figure 11 the relationship among the classes is shown while in Figure 12 we show a possible scenario of execution.

When a task executes a `WaitInstr` instruction, the `Resource Manager` checks if the semaphore is free by invoking `lock(Semaphore *s)`. In the considered scenario, the semaphore

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

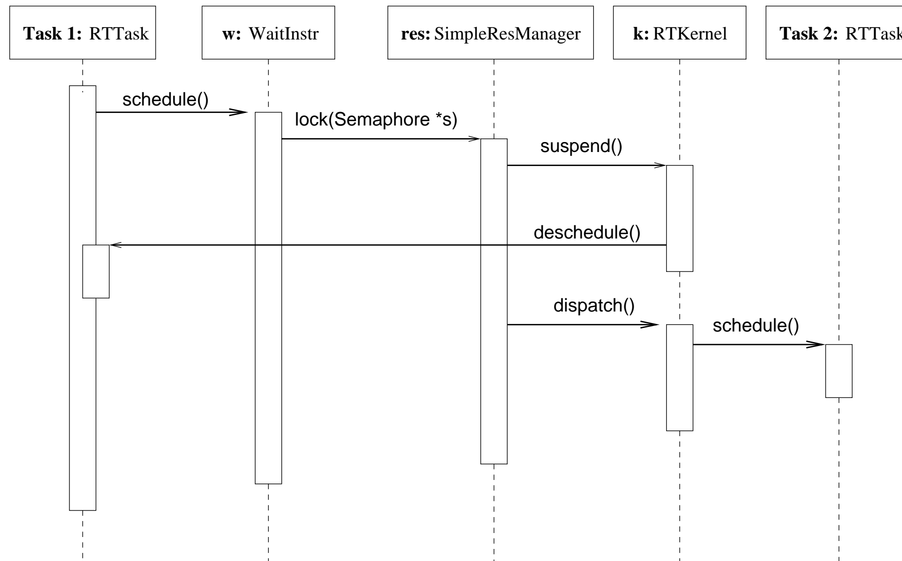


Figure 12. Sequence diagram showing a locking operation on a semaphore.

is locked, thus the task must be blocked: the resource manager invokes the `Kernel::suspend()` method to block the task and `Kernel::dispatch()` methods, in order to schedule another task.

In the current implementation of RTLIB, a simple locking policy, the Priority Inheritance protocol (PIP), the Priority Ceiling protocol (PCP), and the Stack Resource Policy (SRP) are provided as predefined components. In the case where one of these protocols is used, the corresponding resource manager has to interact with Scheduler component to change the task priority according to the protocol. This justifies the relation between the Resource Manager and the Scheduler component in Figure 11.

Networks. Every kernel may have one or 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.

A task can send a message, modeled by the `Message` class, 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 realizes the medium access protocol, such as the Ethernet or CAN bus protocol. In particular, the `CANInterface`

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

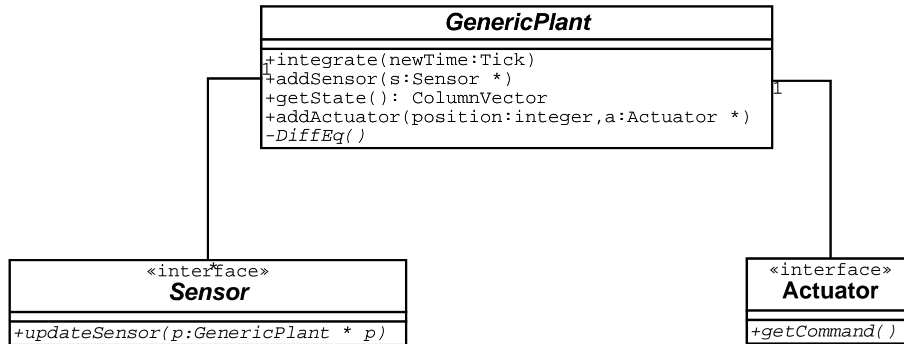


Figure 13. Class diagram representing the components of the numerical package to be used for modeling plants.

has a pointer to a function that transforms the message priority (or deadline) in a CAN priority¶.

Two additional 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.

In the current distribution of RTLIB, the Ethernet network and the CAN bus are provided as predefined components.

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 a structural viewpoint, the numerical package is a software layer built on the top of a library which provides some services, such as differential equation integration and linear algebra operations. 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 us to replace OCTAVE with any other similar solution without affecting the structure of the simulator. As well as permitting the definition of a plant, the numerical package also exports a set of useful classes for linear algebra, such as `Matrix`, `ColumnVector` and so on.

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

Copyright © 2000 John Wiley & Sons, Ltd.

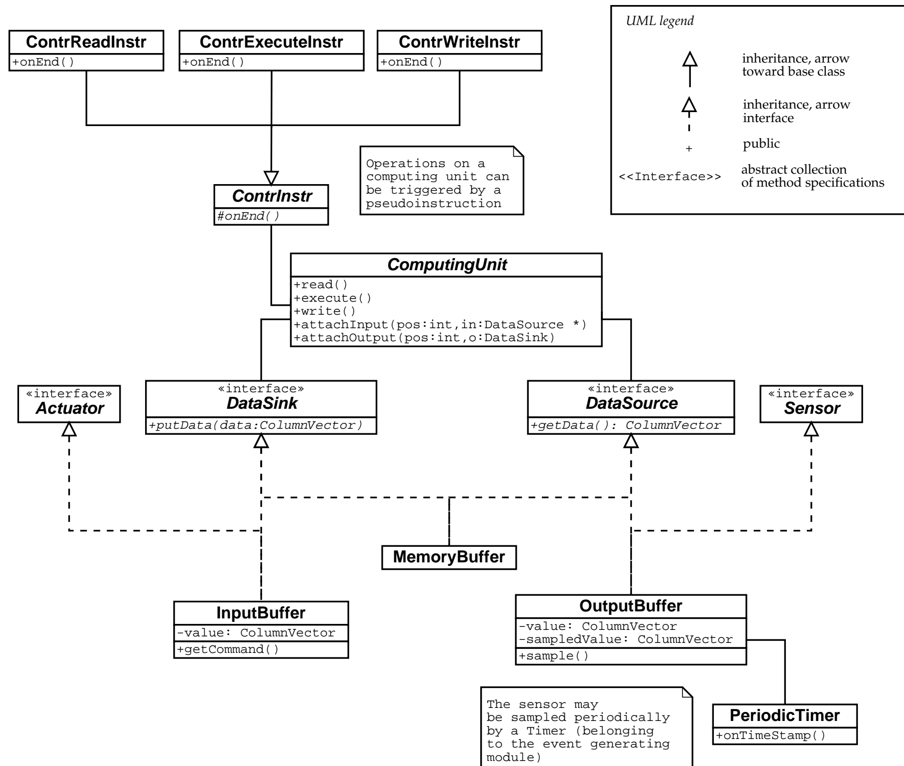


Figure 14. The most important classes used to model the functional behaviour of a controller.

User-defined plants are derived from an abstract class named `GenericPlant` (see Figure 13). The inheritance mechanism permits us 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 update the value of the sensor is based on the *observer pattern* [10].

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

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

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

CTRLIB

The functional model of the system is expressed using the classes of the CTRLIB package. CTRLIB offers two types of components: computing units and storage units. Both of these components are framed within a hierarchy of classes. The structure of the basic classes of CTRLIB is shown in Figure 14.

In order to specify a new type of 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.

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 with special instructions triggering the execution of the `read()`, `execute()` and `write()` operation. Such instructions 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 an instruction 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 implement 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.

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.

In order to highlight the interactions between different components of RTSIM that take place upon the occurrence of some meaningful events, consider the sequence diagram in Figure 15. 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

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

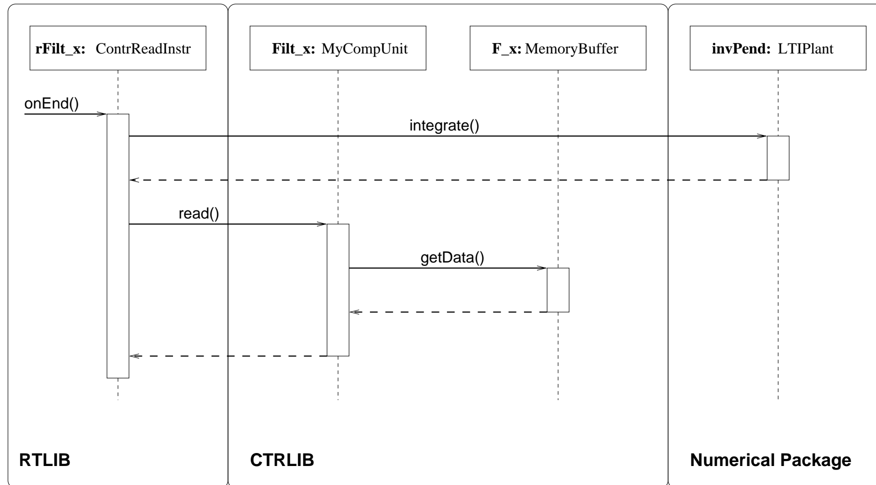


Figure 15. Sequence diagram showing the interactions which take place when an end event for a instruction is handled.

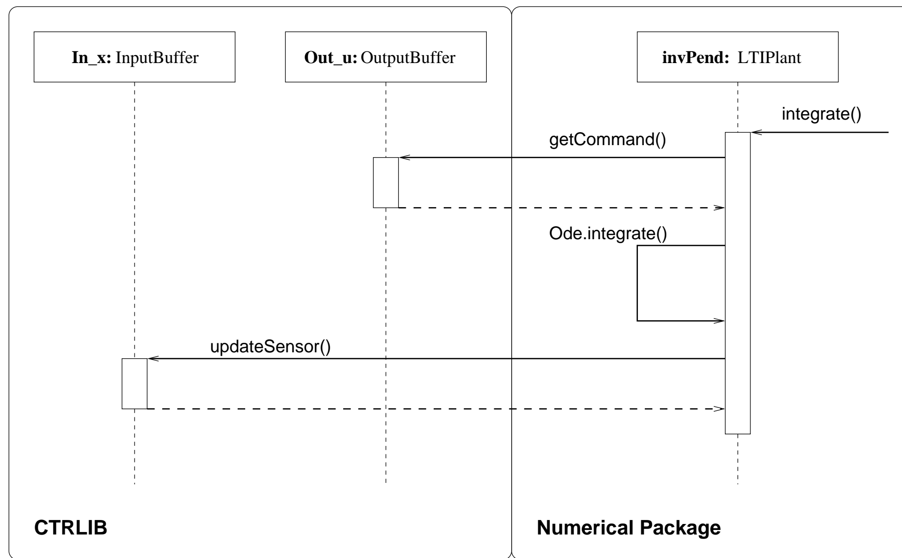


Figure 16. Sequence diagram showing how the integration is performed.

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using speauth.cls

of method calls that follows the termination event of the **rFilt_x** instruction. This event is handled by the `onEnd()` method of the **rFilt_x** object. The first action performed by **rFilt_x** calls the `integrate()` method on the **invPendulum** object, which determines the integration of the differential equation up to the current instant of time. The second action is a call on the `read()` method of the computing unit associated with the instruction, which, in its turn, reads the data from the buffer.

It is also interesting to observe how the integration is performed by detailing the sequence of operations performed by calling the `integrate()` method (diagram in Figure 16). At the beginning of the integration the value of the command variables, contained in the output buffer, are acquired through the `getCommand()` method. Then, the integration can be performed (by calling the `Ode.integrate()` function 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.

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 supports realistic modeling for many implementation related issues, which are not usually accounted for during 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 evaluation of 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.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their precious suggestions, which helped to improve the presentation of the material.

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. N.C. Audsley, A. Burns, M.F. Richardson, K. Tindell, and A.J. Wellings. Stress: A simulator for hard real-time systems. *Software: Practice and Experience*, 6(24), 1994.
4. T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

Copyright © 2000 John Wiley & Sons, Ltd.

5. G. Booch. *Object oriented design with applications.* Benjamin/Cummings Publishing Company, Inc., 1991.
6. J.B. Dennis. First version dataflow procedure language. Technical report, Massachusetts In. of Tecnology, Lab. Comp. Sc., 1975.
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. G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, Amstrdam, 1974.
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. Chen Lee, Raj Rajkumar, John Lehoczky, and Dan Siewiorek. Pratical solutions for qos-based resource allocation. In *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.
19. E. Lee and A. Sangiovanni-Vincentelli. A unified framework for comparing models of computation. *Transaction on Computer aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.
20. Edward A. Lee. Computing for embedded systems. In *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 2001.
21. 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.
22. 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.
23. G. Lipari and L. Palopoli. A framework for simulationg distributed embedded real-time controllers. Technical report, RETIS-LAB, Scuola Superiore S.Anna, 2002.
24. Giuseppe Lipari and Giorgio Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of Systems Architecture*, 46:327–338, 2000.
25. 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.
26. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.
27. 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.
28. 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.
29. 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.
30. B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1, July 1989.
31. M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1994.
32. M. Spuri and G.C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-Time Systems*, 10(2), 1996.
33. 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.
34. 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.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using speauth.cls

35. 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.
36. 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.
37. T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Real-Time Technology and Applications Symposium*, pages 164–173, Chicago,Illinois, January 1995.
38. 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.
39. M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *J. of Real-time systems*, 14:219–250, 1998.

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using speauth.cls