

# TrueTime: Simulation of Control Loops Under Shared Computer Resources

Henriksson, Dan; Cervin, Anton; Årzén, Karl-Erik

*Published in:*
Proceedings of the 15th IFAC world congress

2002

15th IFAC World Congress on Automatic Control, Barcelona, Spain, July 2002.

# TrueTime: Simulation of Control Loops Under Shared Computer Resources

**Dan Henriksson, Anton Cervin, Karl-Erik Årzén**

Department of Automatic Control
Lund Institute of Technology
Box 118, SE-221 00 Lund, Sweden
{dan,anton,karlerik}@control.lth.se

## Abstract

The paper presents TRUETIME, a MATLAB/Simulink-based simulator for real-time control systems. TRUETIME makes it possible to simulate the temporal behavior of multi-tasking real-time kernels containing controller tasks and to study the effects of CPU and network scheduling on control performance. The simulated real-time kernel is event-driven and can handle external interrupts as well as fine-grained details such as context switches. Arbitrary scheduling policies may be defined, and the control tasks may be implemented using C functions, M functions, or Simulink block diagrams. A number of examples that illustrate the use of TRUETIME are presented.

## 1. Introduction

Most computer control systems are embedded systems where the computer is a component within a larger engineering system. The controllers are often implemented as one or several tasks on a microprocessor using a real-time kernel or a real-time operating system (RTOS). In most cases the microprocessor also contains other tasks for other functions, e.g., communication and user interfaces. The kernel or OS typically uses multiprogramming to multiplex the execution of the different tasks on a single CPU. The CPU time and the communication bandwidth, hence, can be viewed as shared resources which the tasks compete for.

Computer-based control theory normally assumes equidistant sampling intervals and negligible or constant control delays, i.e., the latency between the sampling of the inputs to the controller and the generation of the outputs. However, this can seldom be achieved in practice. Tasks interfere with each other through preemption and blocking due to communication. Execution times may be data-dependent or vary due to, e.g., the uses of caches. The result of this is jitter in sampling periods and latencies. An additional cause of this temporal non-determinism is the increasing use of commercial off-the-shelf (COTS) components in control systems, e.g., general purpose operating systems such as Windows and Linux and general purpose network protocols such as Ethernet. These are designed to optimize average-case performance rather than worst-case performance, and therefore increase the non-determinism.

The effects of this type of temporal non-determinism on control performance are often very hard, if not impossible, to investigate analytically. A natural approach is then to instead use simulation. However, todays simulation tools make it difficult to simulate the true temporal behavior of control loops. What is normally done is to introduce time delays in the control loop representing average-case or worst-case delays.

In this paper the new simulation toolbox TRUETIME is presented. TRUETIME, which is based on MATLAB/Simulink, makes it possible to simulate the temporal behavior of a multi-tasking real-time kernel containing controller tasks. The controller tasks control processes modeled as ordinary Simulink blocks. Different scheduling policies may be used, e.g., priority-driven or deadline-driven scheduling. The execution times of the controller tasks can be modeled as being constant or time-varying, using some suitable probability distribution. The effects of context switching and interrupt handling are taken into account, as well as task synchronization using events and monitors. With TRUETIME it is also possible to simulate the timing behavior of communication networks used in, e.g., networked control loops.

TRUETIME can be used for several purposes: to investigate the true effects of timing non-determinism on control performance, to develop compensation schemes that adjust the controller dynamically based on measurements of actual timing variations, to experiment with new, more flexible approaches to dynamic scheduling, e.g., feedback scheduling [Eker *et al.*, 2000] and Quality-of-Service (QoS) based

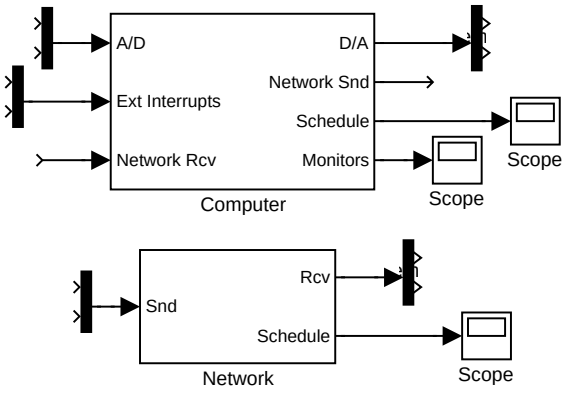


**Figure 1** The interfaces to the Simulink blocks. The Schedule and Monitors ports provide plots of the allocation of common resources (CPU, monitors, network) during the simulation.

scheduling approaches, and to simulate event-based control systems, e.g., combustion engine control systems and distributed controllers.

### 1.1 Related work

While numerous tools exist that support either simulation of control systems (e.g. Simulink) or simulation of real-time scheduling (e.g. STRESS [Audsley *et al.*, 1994] and DRTSS [Storch and Liu, 1996]) very few tools support co-simulation of control systems and real-time scheduling.

An early, tick-based prototype of TRUETIME was presented in [Eker and Cervin, 1999]. Since it was not event-based this early version had very little support for interrupt handling and could not handle fine-grained simulation details. Also, there was no support for simulation of networks.

The RTSIM real-time scheduling simulator (a standalone C++ program) has recently been extended with a numerical module (based on the Octave library) that supports simulation of continuous dynamics, see [Palopoli *et al.*, 2000]. However, it lacks a graphical plant modeling environment, and so far its network capabilities are limited.

### 1.2 Outline of the paper

The simulation environment is described in some detail in Section 2. Three examples are then given to illustrate the use of the simulator. The first example treats scheduling during overload conditions. The subject of the second example is networked control system, whereas the last example evaluates an improved scheduling technique for controller tasks.

## 2. The Simulator

The TRUETIME simulation environment offers two Simulink blocks: a computer block and a network block, the interfaces of which are shown in Fig. 1. The input signals are assumed to be discrete, except the signals connected to the A/D port which may be continuous. All output signals are discrete. The Schedule and Monitors ports provide plots of the allocation of common resources (CPU, monitors, network) during the simulation (c.f. Figs. 4, 7, and 9).

Both blocks are event-driven and execute based on internal and external events. Internal events correspond to clock interrupts caused, e.g., by the release of a task from the time queue or the expiry of a timer. External events correspond to external interrupts which occur when signals connected to the external interrupt port or network ports change value.

The blocks are variable-step, discrete, MATLAB S-functions written in C, the Simulink engine being used only for timing and interfacing with the rest of the model. It should thus be easy to port the blocks to other simulation environments, provided they support event detection (zero-crossing detection).

### 2.1 The computer block

The computer block S-function simulates a computer with a flexible real-time kernel executing user-defined threads and interrupt handlers. Threads may be periodic or aperiodic and are used to simulate controller tasks, communication tasks etc. Interrupt handlers are used to serve internal and external interrupts. The kernel maintains a number of data structures commonly found in real-time kernels, including a ready queue, a time queue, and records for threads, interrupt handlers, events, monitors etc.

The code executed during simulation consists of user-written functions, which have been associated with threads and interrupt handlers. These functions can be written in C (for speed) or as M code (for ease of use).

Execution occurs at three distinct priority levels: interrupt level (highest), kernel level, and thread level (lowest). The execution may be preemptive or non-preemptive. At the interrupt level, interrupt handlers are scheduled according to fixed priorities, whereas at thread level dynamic-priority scheduling may be used. The thread priorities are determined by a user-defined priority function, which is a function of the attributes of a thread. This makes it easy to simulate different scheduling policies. For example, a function returning the absolute deadline of a thread implements deadline-driven scheduling.

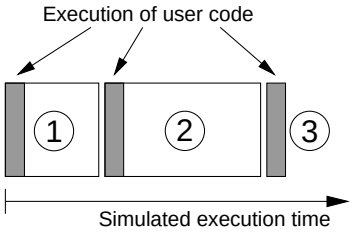


**Figure 2** The execution of the code associated with threads and interrupt handlers is modeled by a number of code segments with different execution times. Execution of user code occurs at the beginning of each code segment.

***Threads*** Each thread is defined by a set of attributes most of which are initialized by the user when the thread is created. These attributes include: a name, a release time, relative and absolute deadlines, an execution time budget, a period (if the thread is periodic), a priority (if fixed-priority scheduling is used), and the user code associated with the thread. Some of the attributes, such as the release time and the execution time budget are constantly updated by the kernel during simulation. The other attributes can be changed by function calls from the user code, but are otherwise kept constant.

An arbitrary data structure may be defined and attached to each thread to represent the local memory of the thread. Other threads may access this data, which can be used for system-level communication between threads to support simulation of, e.g., feedback scheduling. It is further possible to associate three different interrupt handlers with each thread: a code termination handler, a deadline overrun handler, and an execution time overrun handler.

***Interrupt handlers*** When an internal or external interrupt occurs the corresponding interrupt handler is activated and scheduled by the kernel. Similar to threads, interrupt handlers have a set of basic attributes: name, priority, and the associated user code. External interrupts also have a latency, during which they are insensitive to new invocations.

***Code*** The execution of the user code associated with threads and interrupt handlers is divided into segments with different simulated execution times as shown in Fig. 2. The execution times can be constant, random or data-dependent. Execution of user code occurs at the beginning of each code segment. The next segment is not executed until the time associated with the previous segment has elapsed in the simulation. This construction makes it possible to model the timely aspects of the code that are relevant for the interaction with other tasks. This include, e.g., computations, input and output actions, awaiting events, and execution in critical regions using monitors. After execution of the last segment the code termination handler of the thread is activated. For periodic threads this simply updates the release and deadline and puts the thread to sleep until next period. Execution will then again begin in the first segment.

**Table 1** Examples of kernel primitives (pseudo code) that can be called from threads and interrupt handlers.

| | |
|---|---|
| ttAnalogOut(ch,value) | ttAnalogIn(ch) |
| ttWaitUntil(time) | ttCurrentTime() |
| ttSetPriority(prio) | ttSetRelease(time) |
| ttNwSendMsg(msg, node) | ttNwGetMsg() |
| ttEnterMonitor(mon) | ttExitMonitor(mon) |
| ttAwait(event) | ttCause(event) |

## 2.2 The Network Block

The network model is similar to the real-time kernel model, albeit simpler. The network block is event-driven and executes when messages enter or leave the network. A send queue is used to hold all messages currently enqueued in the network (c.f. the ready queue in the real-time kernel). A message contains information about the sending and the receiving computer node, user data (typically measurement signals or control signals), transmission time, and optional real-time attributes such as a priority or a deadline.

A user-defined priority function is used to determine the order in which the enqueued messages should be transmitted. This way, it is easy to model different network policies. When the simulated transmission of a message has completed, it is put in a buffer at the receiving computer node, which is notified by an external interrupt. Transmissions can be preemptive or non-preemptive, the latter being default.

## 2.3 Initialization

Before the start of a simulation, the computer and network blocks must be initialized. This is done in a script for each block. Initialization involves specifying the number of input and output ports, choosing priority functions, defining code functions, creating threads, interrupt handlers, etc.

***Writing a code function*** A code function takes as input argument the segment to be executed, and returns the execution time of this segment. The kernel provides a set of real-time primitives that can be called from the user code, see Table 1 for some examples. A code function for a simple controller is given below

```
function exectime = myController(seg)
switch (seg),
```

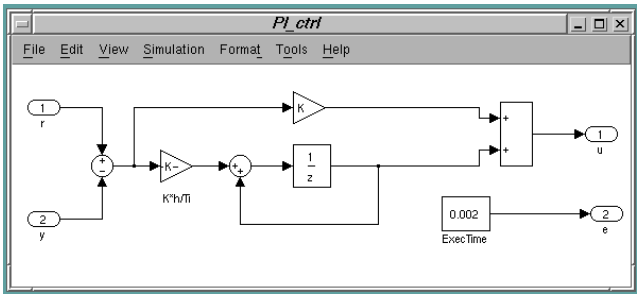


**Figure 3** Controllers represented using ordinary discrete Simulink blocks may be used directly in TRUETIME to evaluate timing performance. The example above shows a PI-controller.

```
  case 1,
    y = ttAnalogIn(1);
    u = calculateOutput(y);
    exectime = 0.002 % execution time
  case 2,
    ttAnalogOut(1,u);
    updateState(y);
    exectime = 0.003 % execution time
  case 3,
    exectime = -1; % code termination
end
```

The input-output latency in the example above is always at least 2 ms, this being the execution time of the first segment. However, preemption from higher-priority threads or interrupts may lead to a longer delay.

***Graphical controller representation*** As an alternative to textual implementation of the controller algorithms, TRUETIME also allows for graphical representations using discrete Simulink blocks. Block systems are called from the code function using the primitive `ttCallBlockSystem`. A block diagram of a PI-controller is shown in Fig 3, and the corresponding use in a code function is given below

```
function exectime = piController(seg)
switch (seg),
  case 1,
    in(1) = ttAnalogIn(1);
    in(2) = ttAnalogIn(2);
    out = ttCallBlockSystem(in,'PI_ctrl');
    exectime = out(2);
  case 2,
    ttAnalogOut(1,out(1));
    exectime = 0.003;
  case 3,
    exectime = -1; % code termination
end
```

## 3. Scheduling during overload conditions

This example treats scheduling of tasks with long, but rare, worst-case execution times. In these cases

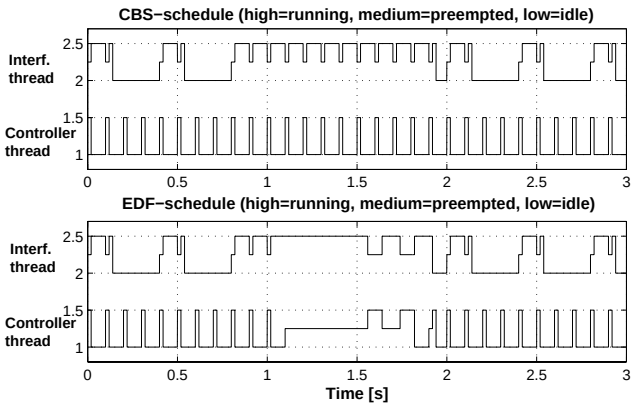


**Figure 4** Comparison of schedules generated by CBS and EDF during an overrun for the interfering thread. Using CBS the controller thread is unaffected by the overrun.

traditional scheduling analysis, based on worst-case execution times, becomes very restrictive. An alternative approach, called the *Constant Bandwidth Server* (CBS) was presented in [Abeni and Buttazzo, 1998]. This server approach is straightforward to simulate using TRUETIME.

In the CBS scheme each task is handled by a dedicated server characterized by a budget, $c_s$, a maximum budget, $Q_s$, and a period, $T_s$. Each server also has a dynamically changing deadline. When the task associated with a server executes longer than its assigned budget, the budget is recharged and a new deadline is generated. The CBS scheme uses deadline-driven scheduling based on the server deadlines. The generation of a new deadline may therefore allow other tasks to run. CBS guarantees that no task consumes more than the bandwidth assigned to its server, $U_s = Q_s/T_s$, i.e., if a task has an overrun it will not affect other tasks. Hence, CBS conceptually divides the CPU into virtual sub-CPU's.

The CBS scheme is simulated in TRUETIME using the execution time budget and execution time overrun handlers associated with threads. When a thread executes longer than its assigned budget an interrupt is generated. The corresponding interrupt handler then recharges the budget and updates the server deadline.

As an example, consider stabilizing control of an inverted pendulum with the state-space realization

$$\dot{x} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 1 \end{pmatrix} u + \begin{pmatrix} 0 \\ 1 \end{pmatrix} w$$
$$y = \begin{pmatrix} 1 & 0 \end{pmatrix} x + v \qquad (1)$$

where $w$ and $v$ are independent zero-mean white noise processes with variances 1 and 0.1 respectively. A LQG-controller with sampling period 100 ms is designed in order to minimize the quadratic cost function

$$J(t) = \int_0^t x^T(s)Q_1x(s) + u(s)Q_2u(s)ds \qquad (2)$$

with $Q_1 = 5I$ and $Q_2 = 0.01$.

The execution time of the controller thread is 20 ms and it is scheduled together with an interfering thread with period 400 ms and a nominal execution time of 100 ms. We further assume that the interfering thread occasionally has a very long execution time of 700 ms occurring with a probability of 5 percent.

In a first simulation the tasks are scheduled using ordinary earliest deadline first scheduling (EDF). Here the task with the shortest remaining time to its deadline will run, with the relative deadlines being equal to the task periods. The effect of an overrun is shown in the lower part of Fig. 4, where it is seen that the controller thread misses five samples due to preemption from the interfering thread. Using the CBS approach we get the desired behavior as seen in the upper part of Fig. 4. The controller thread is unaffected by the overrun of the interfering thread. The loss as measured by the cost function (2) during a simulation of 100 seconds was reduced by 50 percent using the CBS approach.

## 4. A Networked Control System

This example describes simulation of a distributed control system, where a DC servo is to be controlled over a network. The system is shown in Fig. 5 and consists of four nodes. The time-driven sensor node samples the process periodically and sends the samples to the controller node over the network. Upon receiving a sample, the controller computes a control signal which is sent to the actuator node, where it is subsequently actuated. The threads executing in the controller and actuator nodes are both event-driven. There is also a disturbance node generating random interfering traffic over the network.

The network is assumed to be of CAN-type, i.e. transmission of simultaneous messages is decided based on priorities of the packages. The packages generated by the disturbance node have high priority and occupy 50 percent of the network bandwidth. The PD-controller executing in the controller node is designed for the sampling interval 10 ms, which is the sampling interval used in the time-driven sensor node.

Without influence from the interfering node, the round-trip delay (the delay from sampling to actuation) will be fairly constant. Assume that the round-trip delay in this case is equal to 3.5 ms, which will lead to satisfactory control performance. Next, consider influence from the disturbance node. Assume further, that an interfering, high-priority activity (period 7 ms, execution time 3 ms) executes in the controller node. The round-trip delay will now be longer on average and time-varying. This causes the control performance to degrade, which can be seen in the simulated step response in Fig. 6. The execution of the threads in the controller node and the transmission of messages over the network can be studied in detail, see Fig. 7.

Different scheduling policies would have yielded different execution and transmission patterns, and hence different control performance. This can easily be studied with TRUETIME, as well as different delay and jitter compensation schemes.

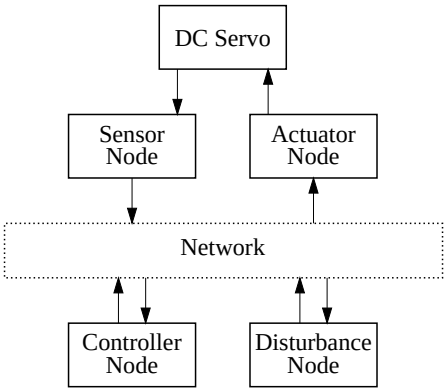


**Figure 5** A distributed control system with time-driven sensor node and event-driven actuator and controller nodes. The disturbance node generates random high-priority traffic over the network.

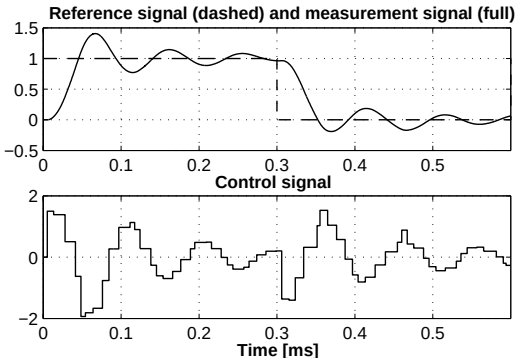


**Figure 6** Step response with interfering network messages and interfering computer task.

## 5. Sub-task Scheduling

To minimize control delay, control algorithms are traditionally divided into two separate parts, calcu-

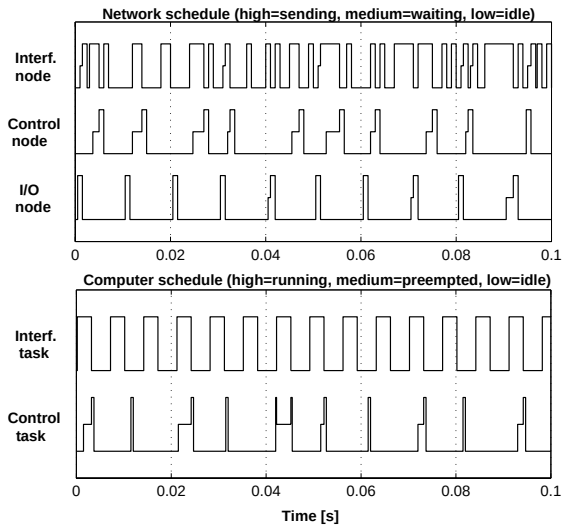


**Figure 7** Close-up of schedules showing the allocation of common resources: network (top) and control computer (bottom).

late output and update state. In the first part the measurement is read, the control signal is computed and sent to the process. In the other part the internal state of the controller is updated. A scheme to improve scheduling of multiple control tasks by scheduling the calculate output and update state parts as separate tasks was presented in [Cervin, 1999]. In this scheme the calculate output parts are given higher priorities than the update state parts. A drawback with the method is an increased number of context switches. This example shows how TRUETIME can be used to evaluate the two different scheduling policies and their affect on control performance taking the effect of context switching into account. The problem considered involves simultaneous control of three inverted pendulums on a single CPU. Each pendulum is described by the continuous-time equations (1).

Three discrete controllers with different sampling periods are designed based on the desired bandwidths 3, 5, and 7 rad/s, respectively, with corresponding sampling periods of 167, 100, and 71 ms. The controllers are based on state feedback with observers and implemented on discrete state-space form, see e.g. [Åström and Wittenmark, 1997]. In the simulation of the improved scheduling the controllers contain four code segments. The first segment performs the calculate output part. In the second segment the plant is actuated followed by a call to the kernel to lower the priority for the update state part. The third segment updates the state and the final segment resets the priority for the calculate output part. The execution times for segment one and three are 10 and 18 ms, respectively. Segments two

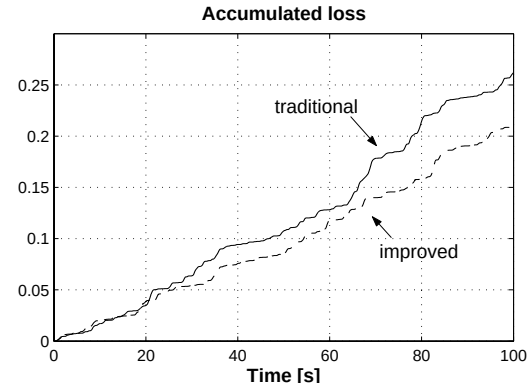


**Figure 8** The accumulated loss $J_1$ for the slow (low-priority) pendulum using traditional and improved scheduling. The loss is reduced considerably for the improved scheduling in spite of an increased number of context switches.

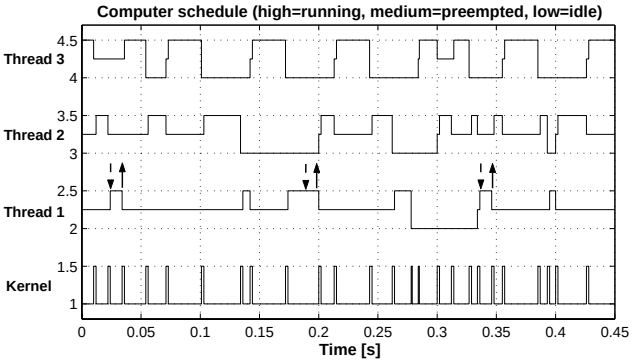


**Figure 9** The computer schedule when using the improved scheduling scheme. The control delay for the low-priority thread is about the same as for the other threads. The kernel graph shows the time to perform the context switches.

and four are modeled as having zero execution time. The time for a full context switch is set to 2 ms.

The two scheduling schemes are simulated for 100 seconds and the quadratic loss functions

$$J_i(t) = \int_0^t \theta_i^2(s)ds \quad i = 1,2,3 \tag{3}$$

are recorded, $\theta_i$ being the angle of the $i$-th pendulum.

The accumulated loss $J_1$ for the slow (low-priority) pendulum using traditional and improved scheduling is shown in Fig. 8. The loss is reduced considerably for the improved scheduling in spite of an increased number of context switches. It can be seen in the computer schedule in Fig. 9 that the control delay for the low-priority thread is about the same as for the two other threads, i.e. 10 ms.

## 6. Conclusions

This paper presented TRUETIME, an event-based simulator for control and real-time systems co-design.

The simulations capture the true, timely behavior of real-time controller tasks, and dynamic control and scheduling strategies can be evaluated from a control performance perspective.

### 6.1 Acknowledgments

The work has been supported by LUCAS—the VINNOVA-funded Center for Applied Software Research, and by ARTES—the Swedish Network for Real-time Research and Education.

## 7. References

Abeni, L. and G. Buttazzo (1998): "Integrating multimedia applications in hard real-time systems." In *Proceedings of the 19th IEEE Real-Time Systems Symposium*. Madrid, Spain.

Åström, K. J. and B. Wittenmark (1997): *Computer-Controlled Systems*, third edition. Prentice Hall.

Audsley, N., A. Burns, M. Richardson, and A. Wellings (1994): "STRESS—A simulator for hard real-time systems." *Software—Practice and Experience*, **24:6**, pp. 543–564.

Cervin, A. (1999): "Improved scheduling of control tasks." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 4–10. York, UK.

Eker, J. and A. Cervin (1999): "A Matlab toolbox for real-time and control systems co-design." In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pp. 320–327. Hong Kong, P.R. China.

Eker, J., P. Hagander, and K. Erik Årzén (2000): "A feedback scheduler for real–time control tasks." *Control Engineering Practice*, **8:12**, pp. 1369–1378.

Palopoli, L., L. Abeni, and G. Buttazzo (2000): "Real-time control system analysis: An integrated approach." In *Proceedings of the 21st IEEE Real-Time Systems Symposium*. Orlando, Florida.

Storch, M. F. and J. W.-S. Liu (1996): "DRTSS: A simulation framework for complex real-time systems." In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pp. 160–169.