2nd Workshop on Real-Time Tools, Copenhagen, Denmark, August 2002

# Jitterbug and TrueTime: Analysis Tools for Real-Time Control Systems

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

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

**Abstract**

The paper presents two recently developed, MATLAB-based analysis tools for real-time control systems. The first tool, called JITTERBUG, is used to compute a performance criterion for a control loop under various timing conditions. The tool makes it easy to quickly judge how sensitive a controller is to implementation effects such as slow sampling, delays, jitter, etc. The second tool, called TRUETIME, allows detailed co-simulation of process dynamics, control task execution, and network communication in a distributed real-time control system. Arbitrary scheduling policies may be used, and the tasks may be written in C code, MATLAB code, or implemented as Simulink block diagrams.

## 1. Introduction

Real-time control systems are becoming increasingly complex systems, from both control and computer science perspectives. Today, even a seemingly simple embedded control system often contains a multitasking real-time kernel and supports networking. At the same time, the market demands that the cost of the system is kept at a minimum. To use the available computing resources optimally, the design of the control algorithms and the control software need to be considered at the same time. For this purpose, new, computer-based tools for real-time and control co-design are needed.

Digital control theory normally assumes equidistant sampling intervals and a negligible or constant control delay from sampling to actuation. However, this can seldom be achieved in practice. Within a node, tasks interfere with each other through preemption and blocking when waiting for common resources. The execution times of the tasks themselves may be data-dependent or vary due to hardware features such as caches. On the distributed level, the communication gives rise to delays that can be more or less deterministic depending on the communication protocol. Another source of temporal nondeterminism is the increasing use of commercial off-the-shelf (COTS) hardware and software components in real-time control, e.g. general-purpose operating systems such as Windows and Linux, and general-purpose network protocols such as Ethernet. These components are designed to optimize average-case performance rather than worst-case performance.

Of course, the temporal nondeterminism can be reduced by proper choice of implementation techniques and platforms. For example, time-driven static scheduling improves the determinism. However, at the same time it reduces the flexibility and limits the possibilities for dynamic modifications. Other techniques of similar nature are time-driven architectures and languages such as TTA [Kopetz, 1997] and Giotto [Henzinger *et al.*, 2001], time-division communication protocols such as TTP [Kopetz and Grünsteidl, 1994], and synchronous programming languages such as Esterel, Lustre and Signal [Halbwachs, 1993]. But even with these techniques a certain level of temporal nondeterminism is unavoidable.

The delay and jitter introduced by the computer system can potentially lead to significant performance degradation. To achieve good performance in systems with limited computer resources, the constraints of the implementation platform must be taken into account at design time. To facilitate this, it is necessary to have software tools to analyze and simulate how the timing affects the control performance. In this paper two such tools are presented: JITTERBUG and TRUETIME.

JITTERBUG is a MATLAB-based toolbox which allows computation of a quadratic performance criterion for a linear control system under various timing conditions. The tool helps to quickly assert how sensitive a control system is to delay, jitter, lost samples, etc, without resorting to simulation. The tool is quite general and can also be used to investigate for instance jitter-compensating controllers, aperiodic controllers, and multi-rate controllers. The toolbox is built upon well-known theory (LQG theory and jump linear systems). Its main contribution is to make it easy to apply this type of stochastic timing analysis to a wide range of problems. Much of the inspiration for JITTERBUG comes from the work on real-time control systems with random delays presented in [Nilsson, 1998a]. In conjunction with that work two simple MATLAB toolboxes for control analysis and design were also produced [Nilsson, 1998b].

The use of JITTERBUG assumes knowledge of sampling period and latency distributions. This information can be difficult to obtain without access to measurements from the true target system under implementation. Also, the analysis cannot capture all the details and nonlinearities (especially in the real-time scheduling) of the computer system. A natural approach is then to instead use simulation. However, today's 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. Taking a different approach, the MATLAB/Simulink-based tool TRUETIME makes it possible to simulate the temporal behavior of a multi-tasking real-time kernel executing controller tasks. The tasks are controlling processes that are modeled as ordinary Simulink blocks. TRUETIME also facilitates simulation of simple models of communication networks and their influence on networked control loops. Different scheduling policies may be used, e.g., priority-based preemptive scheduling and earliest-deadline-first (EDF) scheduling.

TRUETIME can also be used as an experimental platform for research on dynamic real-time control systems. It is for instance possible to study compensation schemes that adjusts the control algorithm based on measurements of actual timing variations. It is also easy to experiment with more flexible approaches to real-time scheduling of controllers, such as feedback scheduling. There, the available CPU or network resources are dynamically distributed according to the current situation (CPU load, the performance of the different loops, etc.) in the system.

While numerous other tools exist that support either simulation of control systems (e.g. MATLAB/Simulink) or simulation of real-time scheduling (e.g. STRESS [Audsley *et al.*, 1994], DRTSS [Storch and Liu, 1996], RTSIM [Casile *et al.*, 1998]), very few tools have been developed that support co-simulation of control systems and real-time scheduling. An early, tick-based prototype of TRUETIME was presented in [Eker and Cervin, 1999]. This version had no support for external interrupts and could not handle fine-grained simulation details. Also, there was no support for simulation of networks. The RTSIM simulator [Casile *et al.*, 1998] has recently been

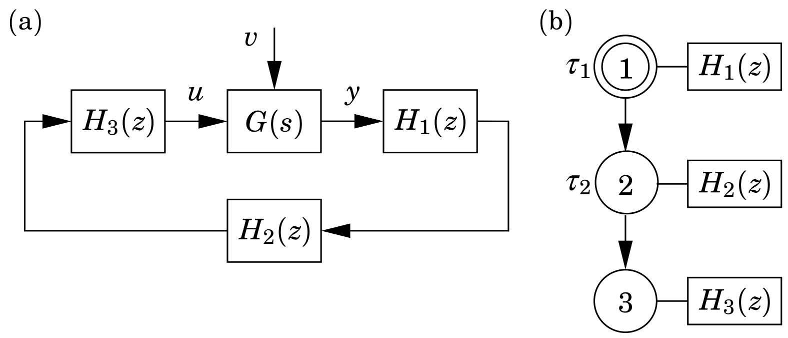

**Figure 1** A simple JITTERBUG model of a computer-controlled system: (a) signal model, and (b) execution model.

extended with a numerical module (based on the Octave library) that supports simulation of continuous dynamics [Palopoli *et al.*, 2000]. Compared to TRUETIME, there is no graphical control systems editor in this tool. At the other end of the usability spectrum, the simulation tool presented in [El-khoury and Törngren, 2001] is entirely based on graphical modeling in Simulink, but is limited to predefined network and CPU scheduling policies.

## 2. Analysis Using Jitterbug

In JITTERBUG, a control system is described by two parallel models: a signal model and an execution model. The signal model is given by a number of connected, linear, continuous-time or discrete-time systems. The execution model consists of a number of execution nodes and describes when the different discrete-time blocks should be updated during the control period.

An example of a JITTERBUG model is shown in Figure 1. Here, a computer-controlled system is modeled by four blocks. The plant is described by the continuous-time system $G$, and the controller is described by the three discrete-time systems $H_1$, $H_2$, and $H_3$. The system $H_1$ could for instance represent a periodic sampler, $H_2$ could represent the computation of the control signal, and $H_3$ could represent the actuator. The associated execution model says that, at the beginning of each period, $H_1$ should first be executed (updated). Then there is a random delay $\tau_1$ until $H_2$ is executed, and another random delay $\tau_2$ until $H_3$ is executed. The delays could model for instance computational delays, scheduling delays, or network transmission delays.

### 2.1 Signal Model

A *continuous-time system* is described by the state-space equations

$$\begin{aligned}\dot{x}_c(t) &= A_c x_c(t) + B_c u(t) + v(t) \\ y(t) &= C_c x_c(t) + D_c u(t)\end{aligned}$$

where $x_c$ is the state vector, $u$ is the input signal vector, $y$ is the output signal vector, $v$ is a continuous-time Gaussian white-noise process with the (incremental) covariance matrix $R_c$, and $A_c$, $B_c$, $C_c$ and $D_c$ are constant matrices describing the dynamics of the system. The *cost* of the system is specified as

$$J_c = \lim_{T\to\infty} \frac{1}{T}\mathbf{E}\left\{\int_0^T x_c^T(t) Q_c x_c(t) dt\right\}$$

where $Q_c$ is a positive semidefinite matrix.

A *discrete-time system* is described by the state-space equations

$$\begin{aligned} x_d(t_{k+1}) &= A_d x_d(t_k) + B_d u(t_k) + e(t_k) \\ y(t_k) &= C_d x_d(t_k) + D_d u(t_k) \end{aligned}$$

where $x_d$ is the state vector, $u$ is the input signal vector, $y$ is the output signal vector, $e$ is a discrete-time Gaussian white noise process with the covariance matrix $R_d$, and $A_d$, $B_d$, $C_d$ and $D_d$ are (possibly time-varying) matrices describing the dynamics of the system. The input signals are sampled when the system is updated, and the outputs signals is held constant between updates. The *cost* of the system is specified as

$$J_d = \lim_{T\to\infty} \frac{1}{T}\mathbf{E}\left\{ \int_0^T x_d^T(t) Q_d x_d(t) dt \right\}$$

where $Q_d$ is a positive semidefinite matrix. Note that the update instants $t_k$ need not be equidistant in time, and that the cost is defined in continuous time.

The *total system* is formed by appropriately connecting the inputs and outputs of a number of continuous-time and discrete-time systems. The *total cost* to be evaluated by the toolbox is summed over all continuous-time and discrete-time systems:

$$J = \sum J_c + \sum J_d \tag{1}$$

### 2.2 Execution Model

The execution model consists of a number of execution nodes. Each node can be associated with zero or more discrete-time systems in the signal model which should be updated when the node becomes active. At time zero, the first node is activated. The first node can be declared to be *periodic* (indicated by an extra circle in the illustrations), which means that the execution will restart at this node every $h$ seconds. This is useful to model periodic controllers and also simplifies the cost calculations a lot.

Each node is associated with a time delay $\tau$ which must elapse before the next node can become active. (If unspecified, the delay is assumed to be zero.) The delay can be used to model computational delay, transmission delay in a network, etc. A delay is described by a discrete-time probability density function

$$P_\tau = [\, P_\tau(0) \quad P_\tau(1) \quad P_\tau(2) \quad \ldots ]$$

where $P_\tau(k)$ represents the probability of a delay of $k\delta$ seconds. The time grain $\delta$ is a constant which is specified for the whole model.

In periodic systems, the execution is preempted if the total delay $\sum \tau$ in the system exceeds the period $h$. Any remaining execution nodes will be skipped. This models a real-time system where hard deadlines (equal to the period) are enforced and the control task is aborted at the deadline.

An aperiodic system can be used to model a real-time system where the task periods are allowed to drift if there are overruns. It could also be used to model e.g. a controller which samples "as fast as possible" instead of waiting for the next period.

***Node- and Time-Dependent Execution*** The same discrete-time system may be updated in several execution nodes. It is possible to specify different update equations (i.e. different $A_d$, $B_d$, $C_d$ and $D_d$ matrices) in the different cases. This can

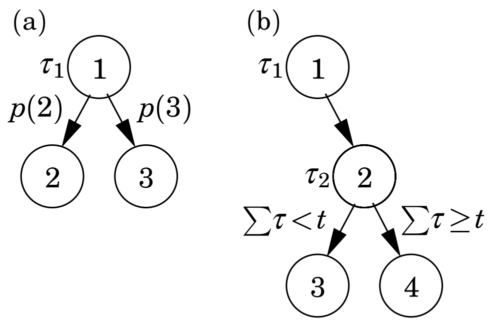

**Figure 2** Alternative execution paths in a JITTERBUG execution model: (a) random choice of path, and (b) choice of path depending on the total delay from the first node.

be used to model e.g. a filter where the update equations look different depending on whether a measurement value is available or not.

It is also possible to make the update equations depend on the time since the first node became active. This can be used to model e.g. jitter-compensating controllers.

***Alternative Execution Paths*** For some systems, it is desirable to specify alternative execution paths (and thereby multiple next nodes). In JITTERBUG, two such cases are possible to model, see Figure 2:

(a) A vector $n$ of next nodes can be specified with a probability vector $p$. After the delay, execution node $n(i)$ will be activated with probability $p(i)$. This can be used to model e.g. a sample being lost with some probability.

(b) A vector $n$ of next nodes can be specified with a time-vector $t$. If the total delay in the system since the node exceeds $t(i)$, node $n(i)$ will be activated next. This can be used to model e.g. time-outs and different compensation schemes.

***Limitations*** Internally, the execution model is translated into a Markov chain, where each state is associated with a set of dynamics in the signal model. To keep the Markov chain finite, there cannot be any timing dependencies past the first node in the execution model. Also, the transitions in the execution model must be independent from the states and the random noise in the signal model. This allows the total system to be treated as a *jump linear system* [Krasovskii and Lidskii, 1961; Ji *et al.*, 1991] whose stationary properties are straight-forward to analyze.

## 2.3 Evaluation of Cost

The evaluation of the total cost (1) is performed in three steps: First, the cost functions, the continuous-time noise, and the continuous-time systems are sampled using the time-grain of the model. Second, the closed-loop system is formulated as a jump linear system, where Markov nodes are used to represent the time-steps in and between the execution nodes. Third, the stationary covariance of all states in the system is calculated. The cost is finally obtained as a linear weighting of the variance. More details on the internal workings of JITTERBUG can be found in [Lincoln and Cervin, ].

For systems without a periodic node, the variance must be computed iteratively, until the cost and the variance converge.

For periodic systems, the Markov state always returns to the periodic execution node every $h/\delta$ time steps. The stationary variance can then be obtained by solving a linear system of equations. (Assuming $n$ states in the total system, there will be $n^2$ equations.) In this case the cost calculation is fast and exact.

If the total system is not stable, the toolbox returns an infinite cost ($J = \infty$).

### 2.4 Example: A Networked Control System

As an example, consider a control loop where the sensor, the actuator, and the controller are distributed among different nodes in a network. The sensor node is assumed to be time-driven, while the controller and actuator nodes are assumed to be event-driven. At a fixed period $h$, the sensor samples the process and sends the measurement sample over the network to the controller node. There, the controller computes a control signal and sends it over the network to the actuator node, where it is subsequently actuated.

The JITTERBUG model of the system was shown in Figure 1. The process is assumed to be a DC servo and is described by the continuous-time system

$$G(s) = \frac{1000}{s(s+1)}$$

The process is driven by white Gaussian continuous-time noise.

The process is sampled periodically with the interval $h$. The sampler and the actuator are described by the trivial discrete-time systems

$$H_1(z) = H_3(z) = 1$$

while the controller is assumed to be discrete-time PD (proportional-derivative) regulator on the form

$$H_2(z) = -K\left(1 + \frac{T_d}{h}\frac{z-1}{z}\right)$$

where the controller parameters are chosen as $K = 1.5$ and $T_d = 0.035$.

The delays in the computer system are modeled by the two independent, random variables $\tau_1$ and $\tau_2$. The total delay from sampling to actuation is thus given by $\tau_{tot} = \tau_1 + \tau_2$. It is assumed that the total delay never exceeds the sampling period (otherwise JITTERBUG would skip the remaining updates).

Finally, as a cost function, we choose the sum of the squared process input and the squared process output:

$$J = \lim_{T\to\infty} \frac{1}{T}\mathbf{E}\left\{\int_0^T \left(y^2(t) + u^2(t)\right) dt\right\} \tag{2}$$

An outline of the MATLAB commands needed to specify the model and compute the value of the cost function are given in Figure 3.

***Sampling Period and Delay*** A control system can typically give satisfactory performance over a range of sampling periods. In textbooks in digital control, e.g. [Åström and Wittenmark, 1997], rules of thumb for sampling period selection are often given. One such rule suggests that the sampling interval $h$ should be chosen such that

$$0.2 < \omega_b h < 0.6$$

where $\omega_b$ is the bandwidth of the closed-loop system. In our case, a continuous-time PD controller with the given parameters would give a bandwidth of about $\omega_b = 80$ rad/s. This would imply a sampling period of between 2.5 and 7.5 milliseconds. The effect of computational delay is typically not considered in such rules of thumb, however. Using JITTERBUG, the combined effect of sampling period and computational delay can be easily investigated. In Figure 4, the cost function (2) for the networked control system has been evaluated for different sampling periods in the interval 1 to 10 milliseconds, and for constant total delay ranging from 0 to 100 % of the sampling interval. It is seen that a one-sample delay gives negligible performance degradation when $h = 1$ ms. When $h = 10$ ms, a one-sample delay makes the system unstable (i.e., the cost $J$ goes to infinity).

```
G = 1000/(s*(s+1));                              Define the process
H1 = 1;                                          Define the sampler
H2 = -K*(1+Td/h*(z-1)/z);                     Define the controller
H3 = 1;                                         Define the actuator

Ptau1 = [ ... ];               Define delay probability distribution 1
Ptau2 = [ ... ];               Define delay probability distribution 2

N = initjitterbug(delta,h);                Set time-grain and period
N = addexecnode(N,1,Ptau1,2);               Define execution node 1
N = addexecnode(N,2,Ptau2,3);               Define execution node 2
N = addexecnode(N,3);                       Define execution node 3

N = addcontsys(N,1,G,4,Q,R1,R2);   Add plant, specify cost and noise
N = adddiscsys(N,2,H1,1,1);                   Add sampler to node 1
N = adddiscsys(N,3,H2,2,2);                Add controller to node 2
N = adddiscsys(N,4,H3,3,3);                  Add actuator to node 3

N = calcdynamics(N);                    Calculate internal dynamics
J = calccost(N);                             Calculate the total cost
```

**Figure 3** This MATLAB script shows the commands needed to compute the performance index of the networked control system using JITTERBUG.

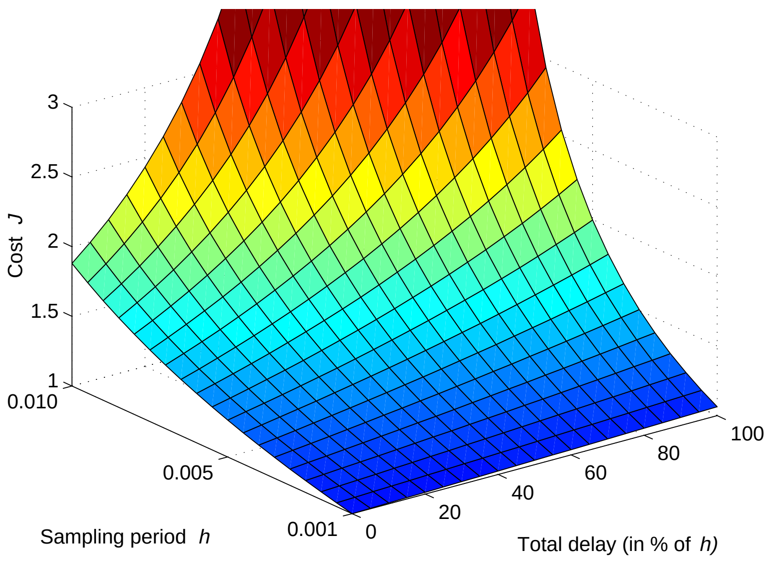

**Figure 4** Example of a cost function computed using JITTERBUG. The plot shows the cost as a function of sampling period and delay in the networked control system example.

## 3. Simulation Using TrueTime

Analysis using JITTERBUG can be used to quickly determine how sensitive a control system is to slow sampling, delay, jitter, etc. For more detailed analysis, and further, system-wide real-time design, the more general simulation tool TRUETIME can be used.

### 3.1 The Simulation Environment

In TRUETIME, which is based on MATLAB/Simulink, computer and network blocks are introduced. The computer block executes user-defined threads and interrupt

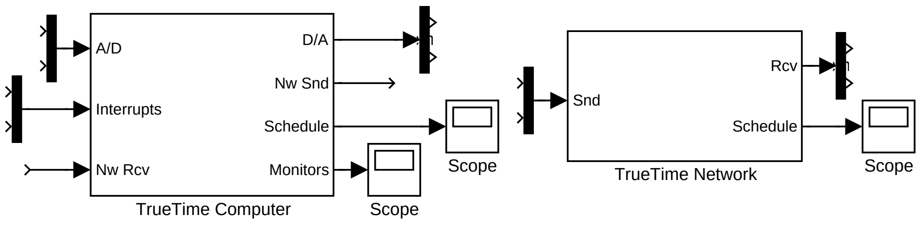

**Figure 5** The interfaces to the TRUETIME blocks. The Schedule and Monitor outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

handlers representing, e.g., I/O tasks, controller tasks, and network interface tasks. The scheduling policy of individual computer blocks is arbitrary and decided by the user. Likewise, in the network block, messages are sent and received according to a chosen network model.

The level of simulation detail is also chosen by the user—it is often not necessary nor desirable to simulate code execution on instruction level or network transmissions on bit level. TRUETIME allows the execution time of threads and the transmission times of messages to be modeled as constant, random, or data-dependent. Furthermore, TRUETIME allows simulation of context switching and task synchronization, using events or monitors.

The interfaces to the computer and network Simulink blocks are shown in Figure 5. Both blocks are *event-driven* with the execution determined by 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. An external interrupt is associated with one of the external interrupt channels of the computer block. The interrupt is triggered when the signal of the corresponding channel changes value. This type of interrupt may be used, e.g., to simulate engine controllers that are sampled against the rotation of the motor or distributed controllers that execute when measurements arrive on the network.

The block inputs are assumed to be discrete-time signals, except the signals connected to the A/D converters of the computer block which may be continuous-time signals. All outputs are discrete-time signals. The Schedule and Monitors outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

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, e.g., the continuous dynamics of controlled plants. It should thus be easy to port the blocks to other simulation environments, provided these environments support event detection (zero-crossing detection).

### 3.2 The Computer Block

The computer block S-function simulates a computer with a simple but flexible real-time kernel, A/D and D/A converters, a network interface, and external interrupt channels.

Internally, the kernel maintains a number of data structures that are commonly found in a real-time kernel: a ready queue, a time queue, and records for threads, interrupt handlers, monitors, timers etc., that have been created for the simulation.

The execution of threads and interrupt handlers is defined by user-written code functions. These functions can be written either in C (for speed) or as MATLAB-code (for ease of use). Control algorithms may also be defined graphically using ordinary Simulink blocks.

***Threads*** Each thread is defined by a set of attributes and a code function. The attributes include a name, a release time, a worst-case execution time, an execution time budget, relative and absolute deadlines, a priority (if fixed-priority scheduling is used), and a period (if the thread is periodic). Some of the attributes, such as the release time and the deadline, are constantly updated by the kernel during simulation. Other attributes, such as period and priority, are normally kept constant but can be changed by calls to kernel primitives when the thread is executing.

Furthermore, it is possible to associate three different interrupt handlers with each thread. A thread termination handler will be triggered when the code function of the thread has executed its last segment, see below. Similar to [Bollella *et al.*, 2000] a deadline overrun handler and an execution time overrun handler may also be attached to each thread.

***Interrupts and Interrupt Handlers*** When an external or internal interrupt occurs, a user-defined interrupt handler is scheduled to serve the interrupt. An interrupt handler works much the same way as a thread, but is scheduled on a higher priority level. An interrupt handler is defined by a name, a priority and a code function. External interrupts also have a latency, during which they are insensitive to new invocations.

***Priorities and Scheduling*** Simulated execution occurs at three distinct priority levels: the interrupt level (highest priority), the kernel level, and the thread level (lowest priority). The execution may be preemptive or non-preemptive; this can be specified individually for each thread and interrupt handler.

At the interrupt level, interrupt handlers are scheduled according to fixed priorities. At the thread level, dynamic-priority scheduling may be used. At each scheduling point, the priority of a thread is given by a user-defined priority function, which is a function of the thread attributes. This way it is easy to simulate different scheduling policies. Predefined priority functions exist for most of the commonly used scheduling schemes, e.g., rate-monotonic scheduling and earliest-deadline-first scheduling.

***Code*** The code associated with threads and interrupt handlers is scheduled and executed by the kernel as the simulation progresses. The code may be divided into several code segments, as shown in Figure 6. The code can interact with other tasks and with the environment at the beginning of each code segment. This construct makes it possible to accurately model input-output delays, blocking when accessing shared resources, etc. The simulated execution time of each segment is returned by the code function, and can be modeled as being constant, random, or even data-dependent. The kernel keeps track of the current segment and calls the code functions with the proper argument during the simulation. Execution resumes in next segment when the task has been running for the time associated with the previous segment. This means that preemption from higher-priority activities and interrupts may cause the actual delay between the segments to be longer than the actual execution time.

Figure 7 shows an example of a code function corresponding to the time line in Figure 6. The function implements a simple controller. In the first segment, the plant is sampled and the control signal is computed. In the second segment, the control signal is actuated and the controller states are updated. The third segment indicates the end of execution, which will trigger execution of the termination handler of the thread.

The functions `calculateOutput` and `updateState` are assumed to represent the implementation of an arbitrary controller. A/D and D/A conversion is performed

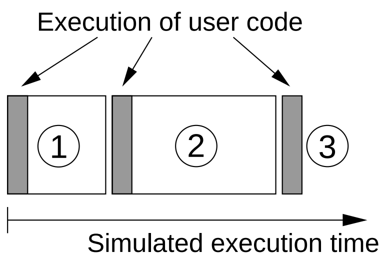

**Figure 6** The execution of the code associated with threads and interrupt handlers is modeled by a number of code segments with different execution times. Execution of user code occurs at the beginning of each code segment.

```
function exectime = myController(segment)
 switch (segment),
    case 1,
        y = ttAnalogIn(1);
        u = calculateOutput(y);
        exectime = 0.002;
    case 2,
        ttAnalogOut(1, u);
        updateState(y);
        exectime = 0.003;
    case 3,
        exectime = -1; % finished
 end
```

**Figure 7** Example of a simple code function in TRUETIME.

using the kernel primitives `ttAnalogIn` and `ttAnalogOut`.

Besides A/D and D/A conversion many other kernel primitives exist that can be called from the code functions. These include functions to send and receive messages over the network, create and remove timers, perform monitor operations, change thread attributes etc.

***Graphical Controller Representation*** As an alternative to textual implementation of the controller algorithms, TRUETIME also allows for graphical representation of the controllers. Controllers represented as discrete Simulink blocks, can be used directly in TRUETIME to evaluate timing performance. The block system is called from the code function using the primitive `ttCallBlockSystem`. A block diagram of a PI-controller is shown in Figure 8. The block system has two inputs, the reference signal and the process output, and two outputs, the control signal and the execution time.

***Output Graphs*** Depending on the simulation a number of different output graphs are generated by the TRUETIME blocks. Each computer block will produce two graphs; a computer schedule and a monitor graph, and the network block will produce a network schedule. The computer schedule will display the execution trace of each thread and interrupt handler during the course of the simulation. If context switching is simulated, the graph will also display the execution of the kernel. In an analogous way the network schedule shows the transmission of messages over the network.

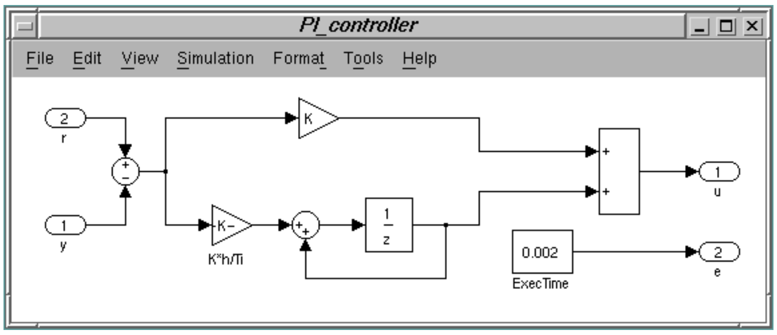

**Figure 8** Controllers represented using ordinary discrete Simulink blocks may be used directly in TRUETIME to evaluate timing performance. The example above shows a PI (proportional-integral) controller.

### 3.3 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 message contains information about the sending and the receiving computer node, user data (typically measurement signals or control signals), the length of the message, and optional real-time attributes such as a priority or a deadline.

In the network block, it is possible to specify the speed of the network, the medium access control protocol (CSMA/CD, CSMA/CA, round robin, FDMA, or TDMA) and a number of other parameters. When the simulated transmission of a message has completed, it is put in a buffer at the receiving computer node, which is notified by a hardware interrupt.

### 3.4 Example: The Networked Control System Revisited

Using TRUETIME it is possible to do a more general simulation of the distributed control system, where the effects of scheduling in the CPU's and simultaneous transmission of messages over the network can be studied in detail. TRUETIME can then be used to simulate different scheduling policies of CPU and network and to experiment with different compensation schemes to cope with the induced delays.

A more realistic model of the delays is obtained by the introduction of a disturbance node, which generates random high-priority traffic over the network. Also, in the controller node, an interfering high-priority thread is executing. The TRUETIME simulation model of the system contains one computer block for each node and a network block, see Figure 9. The oscillatory control performance seen in the figure is a result of the delays induced by the disturbance node and the interfering thread in the controller node.

## 4. Conclusion

Designing a real-time control system is essentially a co-design problem. Choices made in the real-time design will affect the control design and vice versa. For instance, deciding on a particular network protocol will give rise to certain delay distributions which must be taken into account in the controller design. On the other hand, bandwidth requirements in the control loops will influence the choice of CPU and network speed. Using an analysis tool such as JITTERBUG, it is possible to quickly assert how sensitive the control loop is to slow sampling rates, delay, jitter, and other timing problems. Aided by this information, it is then possible

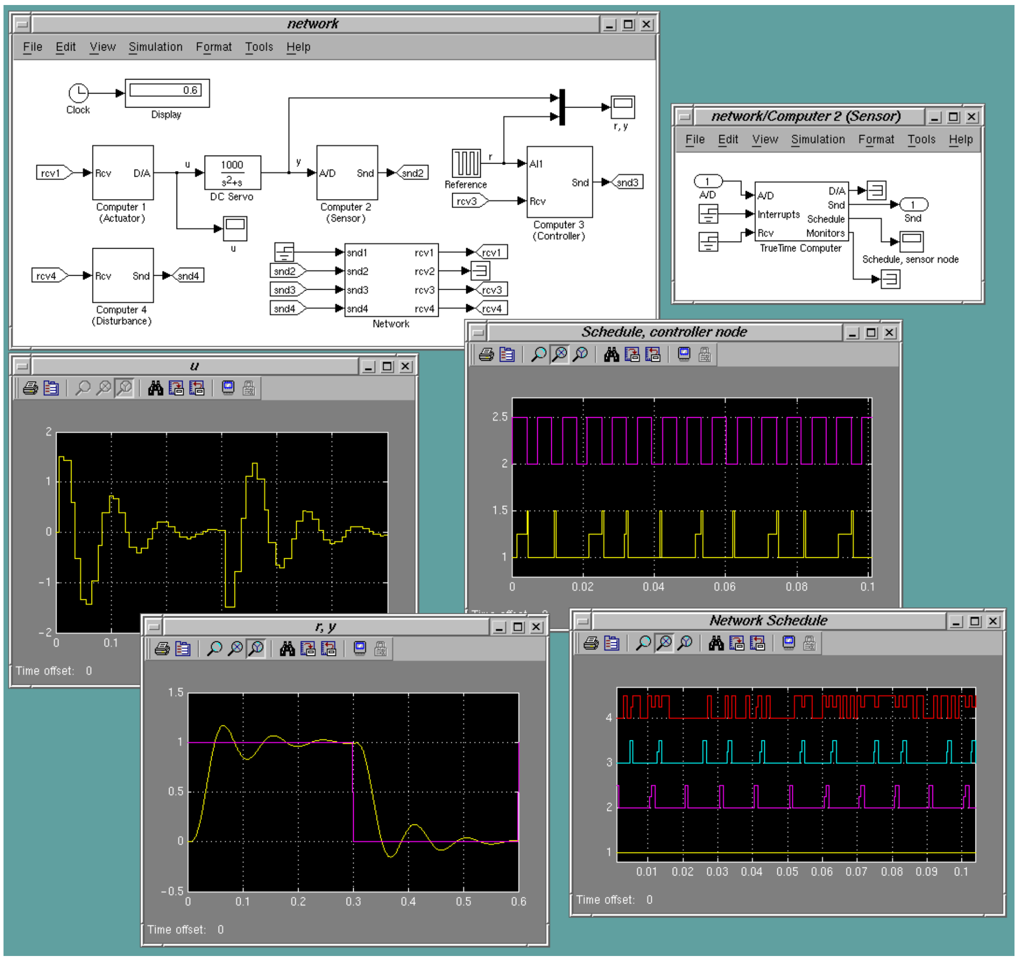

**Figure 9** TRUETIME simulation of a distributed control system. The poor control performance is a result of delays caused by colliding network transmissions and preemption in the controller node.

to proceed with more detailed, system-wide real-time and control design using a simulation tool such as TRUETIME.

JITTERBUG allows the user to compute a quadratic performance criterion for a linear control system under various timing conditions. The control system is described using a number of continuous-time and discrete-time linear systems. A stochastic timing model with random delays is used to describe the execution of the system. The tool can also be used to investigate for instance aperiodic controllers, multirate controllers, and jitter-compensating controllers.

TRUETIME facilitates event-based co-simulation of a multi-tasking real-time kernel containing controller tasks and the continuous dynamics of controlled plants. The simulations capture the true, timely behavior of real-time controller tasks and communication networks, and dynamic control and scheduling strategies can be evaluated from a control performance perspective.

## References

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.

Bollella, G., B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull (2000): *The Real-Time Specification for Java*. Addison-Wesley.

Casile, A., G. Buttazzo, G. Lamastra, and G. Lipari (1998): "Simulation and tracing of hybrid task sets on distributed systems." In *Proc. 5th International Conference on Real-Time Computing Systems and Applications*.

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.

El-khoury, J. and M. Törngren (2001): "Towards a toolset for architecural design of distributed real-time control systems." In *Proc. Real-Time Systems Symposium*.

Halbwachs, N. (1993): *Synchronous programming of reactive systems*. Kluwer Academic Pub.

Henzinger, T. A., B. Horowitz, and C. M. Kirsch (2001): "Giotto: A time-triggered language for embedded programming." In *Proceedings of EMSOFT*.

Ji, Y., H. Chizeck, X. Feng, and K. Loparo (1991): "Stability and control of discrete-time jump linear systems." *Control-Theory and Advanced Applications*, **7:2**, pp. 447–270.

Kopetz, H. (1997): *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer.

Kopetz, H. and G. Grünsteidl (1994): "TTP—A protocol for fault-tolerant real-time systems." *IEEE Computer*, January, pp. 14–23.

Krasovskii, N. and E. Lidskii (1961): "Analytic design of controllers in systems with random attributes, I, II, III." *Automation and Remote Control*, **22:9–11**, pp. 1021–1025, 1141–1146, 1289–1294.

Lincoln, B. and A. Cervin "Jitterbug: A tool for analysis of real-time control performance." Submitted to the 2002 Conference on Decision and Control.

Nilsson, J. (1998a): *Real-Time Control Systems with Delays*. PhD thesis ISRN LUTFD2/TFRT--1049--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Nilsson, J. (1998b): "Two toolboxes for systems with random delays." Report ISRN LUTFD2/TFRT--7572--SE. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

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

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