TrueTime: Real-time Control System Simulation with MATLAB/Simulink

Dan Henriksson, Anton Cervin, Martin Ohlin, Karl-Erik Årzén

Department of Automatic Control
Lund University
Sweden

Outline

- **Introduction and Background**
- TrueTime Tutorial
- Summary

TrueTime Main Idea

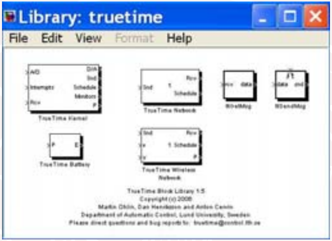
Co-simulation of controller task execution, network transmissions, and continuous plant dynamics.

- Accomplished by providing models of real-time kernels and networks as Simulink blocks
- User code in the form of tasks and interrupt handlers is modeled by MATLAB or C-code

TrueTime Possibilities

- Investigate the true, timely behaviour of time or event-triggered control loops, subject to sampling jitter, input-output latency and jitter, and lost samples, caused by real-time scheduling and networking effects
- Experiment with various scheduling methods, including feedback-based scheduling
- Investigate the performance of different wired or wireless MAC protocols
- Simulate scenarios involving battery-powered mobile robots communicating using wireless ad hoc networks

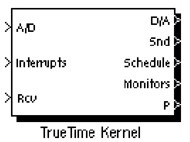
Simulink Blocks



- Offers a *Kernel* block, two *Network* blocks, *Network Interface* blocks and a *Battery* block
  - Simulink S-functions written in C++
  - Event-based implementation using the Simulink built-in zero-crossing detection
  - Portable to other simulation environments

The Kernel Block

- Simulates an event-based real-time kernel
- Executes user-defined tasks and interrupt handlers
- Arbitrary user-defined scheduling policy
- Supports external interrupts and timers
- Supports common real-time primitives (sleepUntil, wait/notify, setPriority, etc.)
- Generates a task activation graph
- More features: context switches, overrun handlers, task synchronization, data logging



TrueTime Code

Three choices:

- C++ code (fast)
- MATLAB code (medium)
- Simulink block diagram (slow)

Kernel Implementation Details

- TrueTime implements a complete real-time kernel with
  - A ready queue for tasks ready to execute
  - A time queue for tasks waiting to be released
  - Waiting queues for monitors and events
- Queues are manipulated by the kernel or by calls to kernel primitives
- The simulated kernel is ideal (no interrupt latency and no execution time associated with real-time primitives)
- Possible to specify a constant context switch overhead
- Event-based simulation obtained using the Simulink zero-crossing function, which ensures that the kernel executes each time an event occurs

The Network Blocks

- Simulates the temporal behaviour of various link-layer MAC protocols
- Medium access and packet transmission
- No built-in support for network and transport layer protocols
  - TCP has been implemented as an example
  - AODV has been implemented as an example

The Network Interface Blocks

- Correspond to the network interface card / bus controller
- Make it possible to use the network blocks stand-alone, without any TrueTime kernels
- Connected to ordinary discrete-time Simulink blocks representing, e.g., controllers

Outline

- Introduction and Background
- **TrueTime Tutorial**
- Summary

Tutorial Outline

- **A Very Simple Example**
- **Tasks**
- **Code**
- **Initialization**
  - **Simple PID Example**

- Real-Time Scheduling
- Data Logging
  - Three Servo Example

- Semaphores, Monitors and Events
- Mailboxes

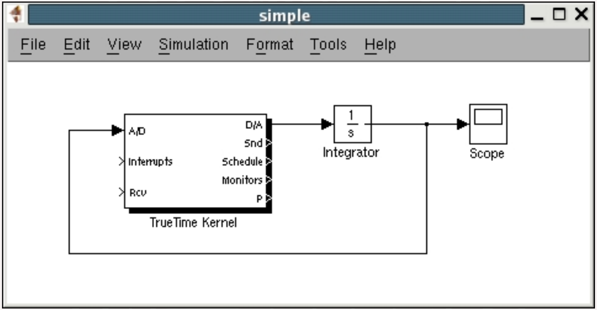
- Interrupt Handlers
- Overrun Handling

- Wired Networks
  - Distributed Example

- Wireless Networks
- Battery Operation
- Local Clocks and Drift
- Network Interface Blocks
- Example
  - Robot soccer

A Very Simple Example

Proportional control of an integrator:



- Initialization
- Task code

A Very Simple Example

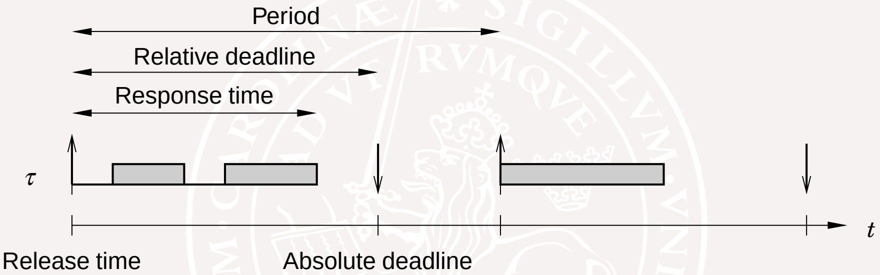
```
function simple_init
ttInitKernel(1, 1, 'prioFP')
ttCreatePeriodicTask('task1', 0, 0.010, 1, 'code', [])


function [exectime,data] = code(seg,data)
switch seg,
 case 1,
  y = ttAnalogIn(1);
  data.u = -0.5*y;
  exectime = 0.005;
 case 2,
  ttAnalogOut(1,data.u);
  exectime = -1;
end
```

Tasks

- Tasks are used to model the execution of user code (mainly control algorithms)
- The release of task instances (*jobs*) may be periodic or aperiodic
- For periodic tasks, the jobs are created by an internal periodic timer
- For aperiodic tasks, the jobs must be created by the user (e.g., in response to interrupts)
- In the case of multiple jobs of the same task, pending jobs are queued

```
ttCreatePeriodicTask(name, offset, period, prio, codeFcn, data)
ttCreateTask(name, deadline, priority, codeFcn, data)
ttCreateJob(taskname)
ttKillJob(taskname)
```

Terminology



- Each job also has an *execution-time budget*

Task Attributes

- Dynamic attributes are updated by the kernel as the simulation progresses
  - Release time, absolute deadline, execution time, ...
- Static attributes are kept constant unless explicitly changed by the user
  - Period, priority, relative deadline, ...

```
ttSetAbsDeadline(taskname, value)
ttSetPeriod(taskname, value)
...
ttGetAbsDeadline(taskname)
ttGetPeriod(taskname)
...
```

Task Code

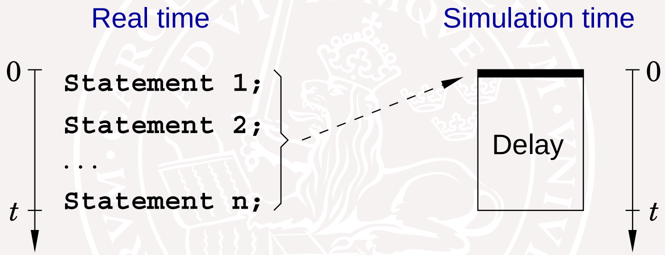
Task code is represented by a *code function* in the format

```
[exectime,data] = function mycode(segment,data)
```

- The `data` input/output argument represents the local memory of the task
- The `segment` input argument represents the program counter
- The `exectime` output argument represents the execution time of the current code segment

Code Segments

A code segment models a number of statements that are executed sequentially



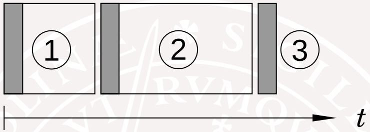
- The execution time t must be supplied by the user
  - Can be constant, random, or data-dependent
  - A return value of -1 means that the job has finished

Code Segments, cont'd

- All statements in a segment are executed sequentially, non-preemptively, and in zero simulation time,
- Only the delay can be preempted by other tasks
- No local variables are saved between segments

(All of this is needed because MATLAB functions cannot be preempted/resumed. . .)

Multiple Code Segments



Multiple code segments are needed to simulate

- input-output delays
- self-suspensions (`ttSleep`, `ttSleepUntil`)
- waiting for events or monitors (`ttWait`, `ttEnterMonitor`)
- loops or branches

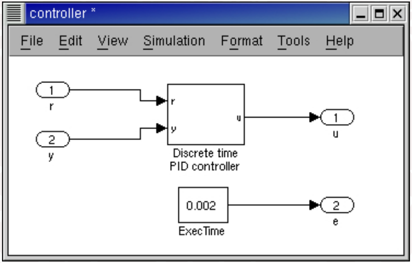
```
ttSetNextSegment(nbr)
```

Example of a Code Function

```
function [exectime, data] = Event_P_Ctrl(segment, data)
switch segment,
  case 1,
     ttWait('event');
     exectime = 0;
  case 2,
     r = ttAnalogIn(1);
     y = ttAnalogIn(2);
     data.u = data.K * (r-y);
     exectime = 0.002 + 0.001*rand;
   case 3,
     ttAnalogOut(1, data.u);
     ttSetNextSegment(1);
     exectime = 0.001;
end
```

Calling Simulink Block Diagrams

- Discrete Simulink blocks may be called from within the code functions to compute control signals
- Block states are stored in the kernel between calls




```
outp = ttCallBlockSystem(nbroutp, inp, blockname)
```

Configuring a Simulation

Each kernel block is initialized in a script (block parameter):

```
nbrInputs = 3;
nbrOutputs = 3;
ttInitKernel(nbrInputs, nbrOutputs, 'prioFP');
periods = [0.01 0.02 0.04];
code = 'myCtrl';
for k = 1:3
   data.u = 0;
   taskname = ['Task ' num2str(k)];
   offset = 0; % Release task at time 0
   period = periods(k);
   prio = k;
   ttCreatePeriodicTask(taskname,offset,period,prio,code,data);
end
```

When to use the C++ API?

- When simulation takes too long time using MATLAB code
- When you want to define your own priority functions
- When you want to define your own kernel hooks

You must use a C++ compiler supported by the MEX facility of the MATLAB version that you are running

- Microsoft C++ Compiler Ver 7 (Visual Studio .NET)
- GNU compiler gcc, g++ on Linux

Example: PID-control of a DC-servo

- Consists of a single controller task implementing a standard PID-controller
- Continuous-time process dynamics

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

- Can evaluate the effect of sampling period and input-output latency on control performance
- Four different ways to implement periodic tasks are shown
- Both C++ function and m-file as well as block diagram implementations will be demonstrated

Tutorial Outline

- A Very Simple Example
- Tasks
- Code
- Initialization
  - Simple PID Example
- **Real-Time Scheduling**
- **Data Logging**
  - **Three Servo Example**
- Semaphores, Monitors and Events
- Mailboxes
- Interrupt Handlers
- Overrun Handling
- Wired Networks
  - Distributed Example
- Wireless Networks
- Battery Operation
- Local Clocks and Drift
- Network Interface Blocks
- Example
  - Robot soccer

Scheduling Policy

- The scheduling policy of the kernel is defined by a priority function, which is a function of task attributes
- Pre-defined priority functions exist for fixed-priority, rate-monotonic, deadline-monotonic, and earliest-deadline-first scheduling
- Example: EDF priority function (C++ API only)

```
double prioEDF(UserTask* t)
   return t->absDeadline;
}

void ttAttachPrioFcn(double (*prioFcn)(UserTask*))
```

Scheduling Hooks

- Code that is executed at different stages during the execution of a task
  - Arrival hook – executed when a job is created
  - Release hook – executed when the job is first inserted in the ready queue
  - Start hook – executed when the job executes its first segment
  - Suspend hook – executed when the job is preempted, blocked or voluntarily goes to sleep
  - Resume hook – executed when the job resumes execution
  - Finish hook – executed after the last code segment
- Facilitates implementation of arbitrary scheduling policies, such as server-based scheduling

```
ttAttachHook(char* taskname, int ID, void (*hook)(UserTask*))
```

Data Logging

- A number of variables may be logged by the kernel as the simulation progresses
- Written to MATLAB workspace when the simulation terminates
- Automatic logging provided for
  - Response time
  - Release latency
  - Sampling latency
  - Task execution time
  - Context switch instances

```
ttCreateLog(taskname, type, variable, size)
ttLogNow(logID)
ttLogStart(logID)
ttLogStop(logID)
```

Example: Three Controllers on one CPU

- Three controller tasks controlling three different DC-servo processes
- Sampling periods $h_i = [0.006 \;\; 0.005 \;\; 0.004]$ sec.
- Execution time of 0.002 sec. for all three tasks for a total utilization of $U = 1.23$
- Possible to evaluate the effect of the scheduling policy on the control performance
- Can use the logging functionality to monitor the response times and sampling latency under the different scheduling schemes

Tutorial Outline

- A Very Simple Example
- Tasks
- Code
- Initialization
  - Simple PID Example
- Real-Time Scheduling
- Data Logging
  - Three Servo Example
- **Semaphores, Monitors and Events**
- **Mailboxes**
- Interrupt Handlers
- Overrun Handling
- Wired Networks
  - Distributed Example
- Wireless Networks
- Battery Operation
- Local Clocks and Drift
- Network Interface Blocks
- Example
  - Robot soccer

Semaphores

- Simple counting and binary semaphores
- No priority inheritance mechanisms
- Only for simple types of synchronization

```
ttCreateSemaphore(semname, initval)
ttTake(semname)
ttGive(simname)
```

Monitors

- Monitors are used to *model* mutual exclusion between tasks that share common data
- Tasks waiting for monitor access are arranged according to their respective priorities (static or dynamic)
- The implementation supports standard priority inheritance to avoid priority inversion
  - Priority ceiling protocols can be implemented
- The simulation generates a graph that shows when different tasks have been holding the various monitors

```
ttCreateMonitor(monitorname, display)
ttEnterMonitor(monitorname)
ttExitMonitor(monitorname)
```

Events

- Events are used for task synchronization and may be free or associated with a monitor (condition variables)
- `ttNotifyAll` will move all waiting tasks to the monitor waiting queue or the ready queue (if it is a free event)
- Events may, e.g., be used to trigger event-based controllers

```
ttCreateEvent(eventname, monitorname)
ttWait(eventname)
ttNotifyAll(eventname)
```

Mailboxes

- Communication between tasks is supported by mailboxes
- Implements asynchronous message passing with indirect naming
- A finite ring buffer is used to store incoming messages
- Both blocking and non-blocking versions of `Fetch` and `Post`

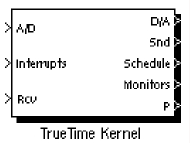
```
ttCreateMailbox(mailboxname, maxsize)
msg = ttTryFetch(mailboxname)
ttTryPost(mailboxname, msg)
```

Tutorial Outline

- A Very Simple Example
- Tasks
- Code
- Initialization
  - Simple PID Example
- Real-Time Scheduling
- Data Logging
  - Three Servo Example
- Semaphores, Monitors and Events
- Mailboxes
  - Ball and Beam Example
- **Interrupt Handlers**
- **Overrun Handling**
- Wired Networks
  - Distributed Example
- Wireless Networks
- Battery Operation
- Local Clocks and Drift
- Network Interface Blocks
- Example
  - Robot Soccer

Interrupt Handlers

- Code executed in response to interrupts
- Scheduled on a higher priority level than tasks
- Available interrupt types
  - Timers (periodic or one-shot)
  - External (hardware) interrupts
  - Task overruns
  - Network interface




```
ttCreateInterruptHandler(hdlname, priority, codeFcn, data)
ttCreateTimer(timername, time, hdlname)
ttCreatePeriodicTimer(timername, start, period, hdlname)
ttCreateExternalTrigger(hdlname, latency)
```

Overrun Handlers

- Two special interrupt handlers may be associated with each task (similar to Real-time Java)
  - A deadline overrun handler
  - An execution time overrun handler
- Can be used to dynamically handle prolonged computations and missed deadlines
- Implemented by internal timers and scheduling hooks

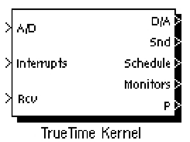
```
ttAttachDLHandler(taskname, hdlname)
ttAttachWCETHandler(taskname, hdlname)
```

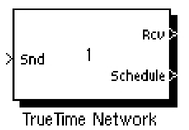
Tutorial Outline

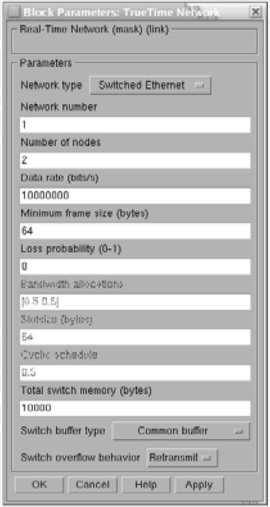
- A Very Simple Example
- Tasks
- Code
- Initialization
  - Simple PID Example
- Real-Time Scheduling
- Data Logging
  - Three Servo Example
- Semaphores, Monitors and Events
- Mailboxes
- Interrupt Handlers
- Overrun Handling
- **Wired Networks**
  - **Distributed Example**
- Wireless Networks
- Battery Operation
- Local Clocks and Drift
- Network Interface Blocks
- Example
  - Robot Soccer

The Network Block

- Supports six common MAC layer policies:
  - CSMA/CD (Ethernet)
  - CSMA/AMP (CAN)
  - Token-based
  - FDMA
  - TDMA
  - Switched Ethernet
- Policy-dependent network parameters
- Generates a transmission schedule





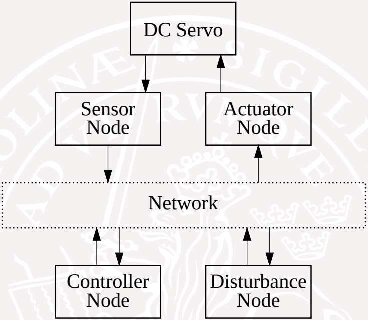


Network Communication

- Each node (kernel block) may be connected to several network blocks
- Dedicated interrupt handler associated with each network receive channel
  - Triggered as a packet arrives
  - Similar to external interrupts
- The actual message data can be an arbitrary MATLAB variable (struct, cell array, etc)
- Broadcast of messages by specifying receiver number 0

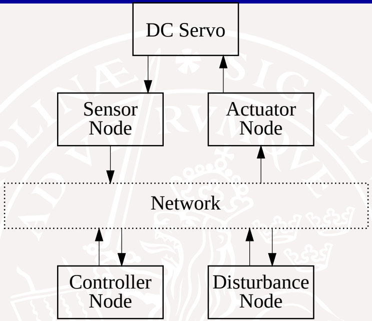
```
ttInitNetwork(network, nodenumber, hdlname)
ttSendMsg([network receiver], data, length, priority)
ttGetMsg(network)
```

Example: Networked Control System



- Time-driven sensor node
- Event-driven controller node
- Event-driven actuator node
- Disturbance node generating high-priority traffic

Example: Networked Control System



- Will try changing the bandwidth occupied by the disturbance node
- Possible to experiment with different network protocols and network parameters
- Can also add a high-priority task to the controller node

Tutorial Outline

- A Very Simple Example
- Tasks
- Code
- Initialization
  - Simple PID Example
- Real-Time Scheduling
- Data Logging
  - Three Servo Example
- Semaphores, Monitors and Events
- Mailboxes
- Interrupt Handlers
- Overrun Handling
- Wired Networks
  - Distributed Example
- **Wireless Networks**
- **Battery Operation**
- **Local Clocks and Drift**
- **Network Interface Blocks**
- **Example**
  - **Robot soccer**

Wireless Networks

Wireless networks are very different from wired ones.

- Wireless devices can often not send and receive at the same time
- The path loss or attenuation of radio signals must be taken into account
- Interference from other terminals (shared medium)
- Hidden terminals
- Multi-path propagation
- Shadowing and reflection

The Wireless Network Model

- Ad-hoc wireless networks
- Isotropic antenna
- Interference from other terminals (shared medium)
- Path-loss default model:

$$\frac{1}{d^a}$$

  where:

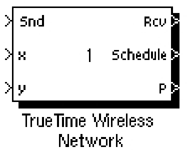
  - d is distance and
  - a is a suitably chosen parameter to model the environment, e.g., 2-4
- User-defined path-loss function:
  - To model fading, multi-path propagation, etc

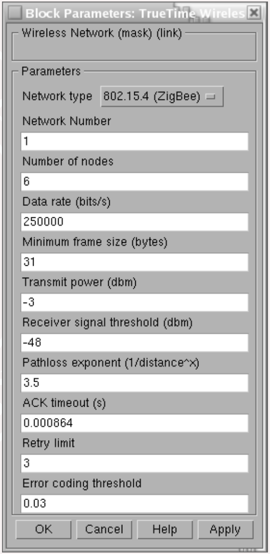
Package Loss

- The signal level in the receiver is calculated according to the path loss formula, $\frac{1}{d^a}$ (or user-defined)
- The signal can be detected if the signal level exceeds a certain configurable threshold
- The SIR is calculated and a probabilistic measure is used to determine the number of bit errors in the message
- A configurable error coding threshold is used to determine if the package can be reconstructed or is lost

The Wireless Network Block

- Used in the same way as the wired network block
- Supports two common MAC layer policies:
  - 802.11b/g (WLAN)
  - 802.15.4 (ZigBee)
- Variable network parameters
- x and y inputs for node locations
- Generates a transmission schedule





Contention in 802.11b/g (WLAN)

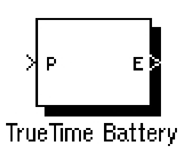
- A packet is marked as collided if another ongoing transmission has a larger signal level at the receiver
- From the sender perspective, package loss and collisions are the same (no ACK received)
- Random back-off time within a contention window
- A configurable number of re-transmission are made before the sender gives up
- More advanced schemes are specified in the standard (using RTS and CTS frames to solve the hidden node problem) but not part of the TrueTime implementation

The Wireless Network Parameters

- Data rate (bits/s)
- Transmit power (dBm)
    - configurable on a per node basis
- Receiver sensitivity (dBm)
- Path-loss exponent
- ACK timeout (s)
- Maximum number of retransmissions
- Error coding threshold

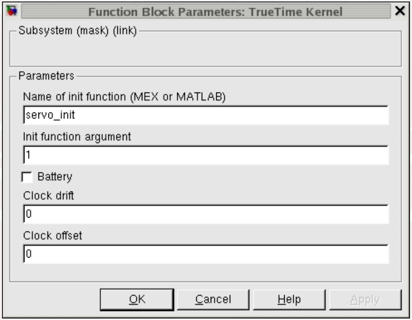
The Battery Block

- Simulation of battery-powered devices
- Simple integrator model
  - discharged or charged (energy scaffolding)
- Energy sinks:
  - computations, radio transmissions, usage of sensors and actuators, . . .
- Dynamic Voltage Scaling
  - change kernel CPU speed to consume less power



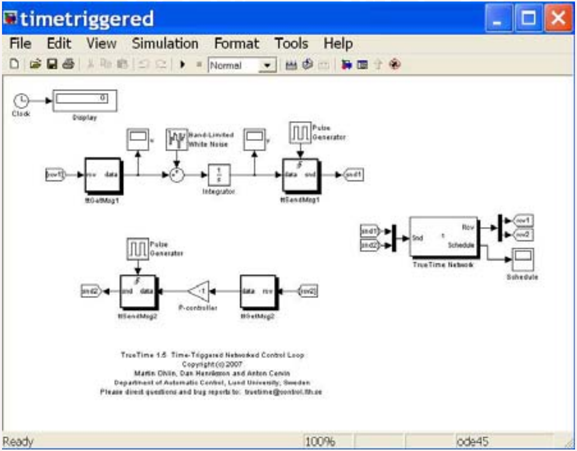
Local Clocks with Offset and Drift

- To simulate distributed systems with local time
- Sensor networks are based on cheap hardware:
  - low manufacturing accuracy ⇒ large clock drift
- Simulate clock synchronization protocols



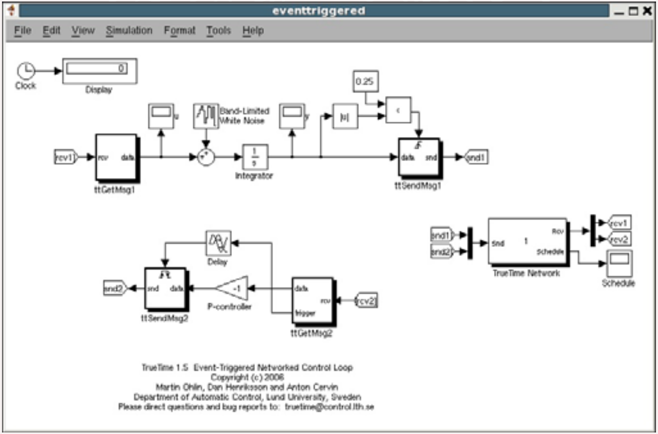
Network Interface Blocks

- Time-triggered networked control loop without kernel blocks



Network Interface Blocks

- Event-triggered networked control loop without kernel blocks



Robot Soccer Example

Outline

- Introduction and Background – Karl-Erik Årzén
- TrueTime Tutorial – Dan Henriksson and Anton Cervin
- **Summary** – Karl-Erik Årzén

TrueTime Possibilities

Co-simulation of

- computations inside computer nodes
  - tasks, interrupt handlers
- wired and wireless communication between nodes
- the dynamics of the physical plant under control
- sensor and actuator dynamics
- the dynamics of mobile robots/nodes
- the dynamics of the environment
- energy consumption in the nodes

A Real-World Application

- Multiple processors and networks
- Based on VxWorks and IBM Rational Rose RT
- Using TrueTime to describe timing behavior
- Has ported TrueTime to a mechatronics simulation environment

”We found TrueTime to be a great tool for describing the timing behavior in a straightforward way.”

More Real-World Applications

Bosch AG

- Extended the network block with support for TTCAN and Flexray
- Used in a simulation environment for investigating the impacts of time-triggered communication on a distributed vehicle dynamics control system

Haldex AB

- Simulation of CAN-based distributed control systems

TrueTime Limitations

- Developed as a research tool rather than as a tool for system developers
- Cannot express tasks and interrupt handler directly using production code
  - code is modeled using TrueTime MATLAB code or TrueTime C code
  - no automatic translation
- Execution times or distributions assumed to be known
- How to support automatic code generation from TrueTime models?
  - Generate POSIX-thread compatible code?
  - Generate monolithic code (TVM = TrueTime Virtual Machine)
- Based on MATLAB/Simulink

More Material

- The toolbox (TrueTime 1.5) together with a complete reference manual can be downloaded at:

    `http://www.control.lth.se/user/truetime/`