TrueTime: Simulation of Networked and Embedded Control Systems

Anton Cervin

Department of Automatic Control
Lund University
Sweden

Joint work with Dan Henriksson, Martin Ohlin, Martin Hast, Karl-Erik Årzén
Networked and Embedded Control Systems

- Small, shared CPUs
- Shared networks and buses
- Mix of control technology and software technology
Co-simulation of scheduling algorithms, control algorithms, network protocols, and continuous-time dynamics.

- Accomplished by providing models of real-time kernels and wired/wireless networks as Simulink blocks
- User code (tasks/interrupt handlers) is modeled by MATLAB code or C++ code
Why Co-Simulation?

Networked and embedded control systems can give rise to very complex timing patterns

- event-driven execution and communication
- multiple tasks competing for shared CPU/network resources
- varying computation times, preemption delays, blocking delays, kernel overheads, . . .
- network interface delays, queuing delays, transmission delays, resending delays, lost packets, . . .
TrueTime Possibilities

- Investigate the true, timely behaviour of time-triggered or event-triggered control loops, subject to scheduling and networking effects
- Experiment with various scheduling algorithms and wired/wireless MAC protocols
- Simulate complex scenarios involving mobile robots communicating over wireless ad-hoc networks
  - **Example:** RUNES road tunnel disaster scenario
Scenario within EU/IST/FP7 project RUNES

- Sensor network inside road tunnel
- The networked is partitioned after an accident
- Mobile robots are sent in to recover the connectivity
  - Navigating using ultra-sound trilateration
T-Mote Sky
Ultrasound receiver
AODV (Ad-hoc On-Demand Distance Vector) routing
T-Mote Sky

- Ultrasound transmitter – ATmega16 AVR
- Compute engine – ATmega128 AVR
  - Extended Kalman filter combining ultrasound trilateration with dead reckoning
- 2 wheel controllers – ATmega16 AVR
TrueTime History

- **1999**: First prototype
  - MATLAB implementation and MATLAB API
- **2002**: TrueTime 1.0
  - C++ implementation, MATLAB and C++ APIs
  - Wired Network block (CAN, Ethernet, TDMA, …)
- **2005**: TrueTime 1.3
  - Wireless Network block (802.11b WLAN, 802.15.4 ZigBee)
  - Battery block, Dynamic Voltage Scaling, local clocks
- **2009**: TrueTime 2.0 beta
  - Standalone Send and Receive blocks, Ultrasound Network block
  - New network protocols (FlexRay, PROFINET)
  - Multicore CPU scheduling
- **2016**: TrueTime 2.0
  - New scheduling algorithms (TBS, TimeTriggered, SlotShifting)
  - Stable release
TrueTime offers a **Kernel** block, three **Network** blocks, stand-alone **Send** and **Receive** blocks and a **Battery** block

- Simulink S-functions written in C++
- Event-based simulation using zero-crossing functions
The Kernel Block

- Simulates an event-based real-time kernel
- Executes user-defined tasks and interrupt handlers
- Uses a user-defined scheduling policy
- Supports external interrupts and timers
- Supports all the common real-time primitives
- Generates a task activation graph
- More features: context switches, overrun handlers, task synchronization, data logging
TrueTime User Code

Three choices:

- C++ code (fast)
- MATLAB code (not so fast)
- Simulink block diagram (very slow)
TrueTime implements a “complete” real-time kernel with

- \( n \) ready queues for tasks ready to execute
- a time queue for tasks waiting to be released
- waiting queues for monitors, events, and mailboxes

Queues are manipulated by the kernel or by calls to kernel primitives

The simulated kernel is ideal (no overhead)

However, a constant context switch overhead may be specified
TrueTime Tutorial Outline

- A Very Simple Example
- Tasks
- Code
- Initialization
  - Real-Time Scheduling
  - Data Logging
  - Interrupt Handlers
- The network blocks
  - Wired networks
  - Wireless networks
- Battery Operation
- Local Clocks
Proportional control of an integrator:

- Initialization
- Task code
function simple_init
    ttInitKernel('prioFP')
    ttCreatePeriodicTask('task1', 0, 1.0, 'ctrl_code')

function [executime,data] = ctrl_code(seg,data)
    switch seg,
        case 1,
            y = ttAnalogIn(1);
            data.u = -0.5*y;
            executime = 0.1;
        case 2,
            ttAnalogOut(1,data.u);
            executime = -1;
    end
Tasks

- Tasks are used to model the execution of user code
- 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, jobs may be created manually or in response to external trigger interrupts or network interrupts

```
ttCreatePeriodicTask(taskname, offset, period, codeFcn, data)
ttCreateTask(taskname, deadline, codeFcn, data)
ttCreateJob(taskname)
ttKillJob(taskname)
```
Task Attributes

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

```
ttSetDeadline(value, taskname) ttGetDeadline(value, taskname)
ttSetAbsDeadline(value, taskname) ttGetAbsDeadline(value, taskname)
ttSetPriority(value, taskname) ttGetPriority(value, taskname)
ttSetPeriod(value, taskname) ttGetPeriod(value, taskname)
ttSetWCET(value, taskname) ttGetWCET(value, taskname)
ttSetBudget(value, taskname) ttGetBudget(value, taskname)
```
Task code is represented by a code function in the format

\[
[\text{exectime}, \text{data}] = \text{function mycode(segment, data)}
\]

- The \text{data} input/output argument represents the local memory of the task.
- The \text{segment} input argument represents the program counter.
- The \text{exectime} output argument represents the execution time of the current code segment.
A code segment models a number of statements that are executed sequentially.

```c
Statement 1;
Statement 2;
... 
Statement n;
```

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.
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 are needed to simulate

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

\[ \texttt{ttSetNextSegment(nbr)} \]
An event-based P-controller:

```matlab
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
```
Each kernel block is initialized in a function (block parameter):

```matlab
function node_init(arg)
nbrInputs = 3;
nbrOutputs = 3;
ttInitKernel('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);
    ttCreatePeriodicTask(taskname, offset, period, code, data);
    ttSetPriority(k, taskname);
end
```
When to use the C++ API?

- When simulation takes too long time using MATLAB code
  - C++ can give a speedup factor of $\approx 2–20$
- If you want to define your own scheduling policy using a custom priority function and scheduling hooks

You must use a C++ compiler that is supported by the combination of MATLAB version and OS that you are using

- See http://www.mathworks.com/support/compiler
TrueTime Tutorial Outline

- A Very Simple Example
  - Tasks
  - Code
  - Initialization
- Real-Time Scheduling
  - Data Logging
  - Interrupt Handlers
- The network blocks
  - Wired networks
  - Wireless networks
- Battery Operation
- Local Clocks
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 (\texttt{prioFP}), deadline-monotonic (\texttt{prioDM}), and earliest-deadline-first scheduling (\texttt{prioEDF}).

Example: EDF priority function (C++ API)

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

```c
void ttInitKernel(double (*prioFcn)(UserTask*))
```
Example: Constant Bandwidth Servers (CBS)

- Scheduling algorithm proposed by [Abeni and Buttazzo, 1998]
- Assumes an EDF kernel (prioEDF must be selected)
- A CBS is characterized by
  - a period $T_s$
  - a budget $Q_s$
- A task associated with a CBS cannot execute more than the server budget period each server period (“sandboxing”)
- Implemented using scheduling hooks

```c
#include "true_time.h"

// Create a CBS
void ttCreateCBS(char *name, long Qs, long Ts, char type) {
    // Implementation
}

// Attach a task to a CBS
void ttAttachCBS(char *taskname, char *CBSname) {
    // Implementation
}

// Set CBS parameters
void ttSetCBSParameters(char *name, long Qs, long Ts) {
    // Implementation
}
```
Multicore Scheduling

- Since version 2.0 TrueTime supports partitioned multicore scheduling
  - One ready queue per core
  - The same local scheduling policy in each core
- Tasks can be migrated between cores during runtime

```c
ttSetNumberOfCPUs(nbr)
```

```c
ttSetCPUAffinity(taskname, CPUUnbr)
```
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 (time between arrival and release)
  - Start latency (time between release and start)
  - Execution time

```
ttCreateLog(logname, variable, size)
ttCreateLog(taskname, logtype, variable, size)
ttLogStart(logname)
ttLogStop(logname)
ttLogNow(logname)
ttLogValue(logname, value)
```
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 interrupts (triggers)
  - Network interrupts
  - Task overruns

```
- ttCreateHandler(hdlname, priority, codeFcn, data, queueLength)
- ttCreateTimer(timername, time, hdlname)
- ttCreatePeriodicTimer(timername, starttime, period, hdlname)
- ttAttachTriggerHandler(triggerNbr, taskname)
- ttAttachNetworkHandler(networkNbr, taskname)
- ttAttachDLHandler(taskname, hdlname)
- ttAttachWCETHandler(taskname, hdlname)
```
TrueTime Tutorial Outline

- A Very Simple Example
- Tasks
- Code
- Initialization
- Real-Time Scheduling
- Data Logging
- Interrupt Handlers

- The network blocks
  - Wired networks
  - Wireless networks
- Battery Operation
- Local Clocks
The Network Blocks

- Simulate the temporal behaviour of various link-layer MAC protocols
- Only medium access and packet transmission
- No built-in support for higher layers
  - TCP and AODV have been implemented as application examples
The Wired 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
Each network is identified by a number
Each kernel block may be connected to several network blocks
The message data can be an arbitrary MATLAB variable (struct, cell array, etc)
Direct addressing or broadcast

- ttSendMsg([network receiver], data, length, priority)
- ttGetMsg(network)
- ttDiscardUnsentMessages(network)
- ttAttachNetworkHandler(networkNbr, taskname)
The Network Parameters

- Network number
- Data rate (bits/s)
- Minimum frame size (bits)
- Loss probability (0–1)
- Network interface delay (s)
- FDMA: Bandwidth allocations
- TDMA: Slotsize, schedule
- Switched Ethernet: Switch memory, buffer type
Example: Networked Control System

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

Anton Cervin
TrueTime
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 nodes (shared medium).
- Hidden nodes.
- Multi-path propagation, shadowing and reflection.
The TrueTime Wireless Network Model

- Isotropic antennas
- Interference from other nodes (shared medium)
- Default path-loss modeled as

\[ \frac{1}{d^\alpha} \]

where:

- \( d \) is the current distance between the nodes and
- \( \alpha \) is a suitably chosen parameter to model the environment, e.g., 2–4

User-defined path-loss can be defined by a function

\[ P_2 = f(P_1, n_1, x_1, y_1, n_2, x_2, y_2, t) \]
The signal level in the receiver is calculated according to the path loss formula.

The signal can be detected if the signal level exceeds a certain threshold (configurable).

The signal-to-interference ratio is calculated and a probabilistic measure is used to determine the number of bit errors in the message.

An error coding threshold (configurable) is used to determine whether the package can be reconstructed or is lost.
The Wireless Network Block

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

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

```c
ttSetNetworkParameter(parameter, value)
```
The Battery Block and Dynamic Voltage Scaling

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

```
ttSetKernelParameter(parameter, value)
```
To simulate distributed systems with local time
- Sensor networks are based on cheap hardware:
  - low manufacturing accuracy ⇒ large clock drift
- Simulate clock synchronization protocols

Code:
```plaintext
time = ttCurrentTime
time = ttCurrentTime(newTime)
```
Example: Mote Soccer
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."
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
Co-Simulation of:
- computations inside the nodes
- tasks, interrupt handlers, scheduling hooks
- wired or wireless communication between nodes
- sensor and actuator dynamics
- mobile robot dynamics
- dynamics of the environment
- dynamics of the physical plant under control
- the batteries in the nodes

Control performance assessment
- time domain
- cost function evaluation (via Monte Carlo simulations)
Some Limitations

- Developed as a research tool rather than as a tool for system developers
- Cannot express tasks and interrupt handlers directly using production code
  - Code must be divided into code segments and execution times must be assigned
- The zero-crossing functions generate quite a few events $\Rightarrow$ large models tend to be slow
- No built-in support for, e.g.,
  - higher-level network protocols
  - task migration between kernel blocks
  - …
Download and documentation:

www.control.lth.se/truetime

Contact:

truetime@control.lth.se
Three main options:

1. Setup, explore the basic demos, especially simple, servo, threeservos, networked
2. Develop your own simple application
3. Do a “mini-project” on distributed consensus
“Mini-project”

- Simulation of a distributed consensus algorithm
- 6 mobile nodes should agree upon a location in the plane to meet
- Communication over a wireless network
- Each node is modeled by
  - A Kernel block
  - Two integrator blocks (for the x and y positions)
Distributed Consensus

A simple algorithm to reach consensus regarding the state of \( n \) integrator agents with dynamics \( \dot{z}_i = u_i \) can be expressed as

\[
u_i(t) = K \sum_{j \in \mathcal{N}_i} (z_j(t) - z_i(t)) + b_i(t)\]

where \( K \) is a gain parameter and \( \mathcal{N}_i \) are the neighbours of agent \( i \) (i.e., the nodes within communication range).

The bias term \( b_i(t) \) should be zero if the nodes are to meet at a common location.
A model with six integrator agents connected to a wireless network is provided (consensus.mdl)

Each kernel block has to be configured – use the same initialization function and code function(s) for all blocks

The consensus algorithm can be implemented as a simple periodic task (ttCreatePeriodicTask):
  - Collect all $x$ and $y$ values sent from your neighbours during the last period (repeated calls to ttGetMsg)
  - Read your own $x$ and $y$ coordinates (ttAnalogIn)
  - Compute the control signals in the $x$ and $y$ directions according to the formula
  - Output the control signals (ttAnalogOut)
  - Broadcast your own $x$ and $y$ position to your neighbours (ttSendMsg with receiver 0)
Add bias terms to the consensus algorithm to simulate “formation flight”

Let one node lead (by not running the consensus algorithm) and let the other ones follow