Simulation of Distributed Automation Systems in Modelica

Florian Wagner Liu Liu Georg Frey
University of Kaiserslautern
Erwin-Schrödinger-Str.12
D-67663, Kaiserslautern, Germany
{wagner | liuliu | frey}@eit.uni-kl.de

Abstract

The increasing application of network technologies and smart embedded devices in the field of automation and control leads to new distributed system architectures. The analysis of the resulting distributed automation systems requires models that cover physical processes as well as computing and communication devices. Modelica as multi-domain modeling language offers the necessary support to build such models. While, for physical systems, there are many Modelica libraries, only limited support for the modeling of computation and communication is currently available. This gap is filled by the presented network and controller libraries. The network library currently supports switched Ethernet, WLAN, and ZigBee. The controller library offers different types of controllers as well as interface devices. Implementation aspects of the presented libraries are discussed in some detail and their application is illustrated by examples.

1 Introduction

As automation systems are constantly increasing in complexity, new methods for controller design have to be applied. One promising approach is the concept of Distributed Automation Systems (DAS). Though distribution simplifies the design of complex control applications, analysis is more difficult than in traditional monolithic or strictly hierarchical systems.

Distributed systems have a concurrent nature. Hence, coordination and synchronization are needed between the individual control devices. This is usually achieved by means of networks. Because of the inexpensiveness of components, the plug-and-play abilities and the possibility for information access from higher level business units, standard networks like Ethernet currently tend to replace special purpose networks (ASI, ProfiBus, ...) in automation.

To design and analyze an automation system, the engineer relies on tool support. Individual tools for algorithm analysis, network analysis and process simulation are available. However, the isolated analysis of any of these aspects does not meet the engineer's requirement of analyzing the closed-loop behavior of the system where the controller interacts with the controlled process via the network.

A simulation environment covering all aspects of distributed automation systems is Matlab/Simulink with the TrueTime [1] toolbox. Matlab/Simulink is a well-known tool for controller design and process simulation. TrueTime adds models for network and controller hardware. As Matlab/Simulink is used as platform, the analysis benefits from its advantages (widely spread, many process models available) but also inherits its disadvantages. Here, especially the causal procedural modeling approach in Matlab/Simulink complicates the design of complex process models and hinders the reuse of components.

Modelica, as a multi-domain modeling language, with the object oriented modeling paradigm and the non-signal-flow-dependant model causality, which increases the reusability of process models, is another adequate basis for overall system analysis.

The paper presents an approach for simulation of networks and controller hardware in Modelica in combination with process models, also modeled in Modelica. Means to analyze both, the functional and the temporal behavior of the overall system in early development stages are provided. First analysis results have already been presented in [2].

This paper focuses on the description of the developed libraries, including implementation details. In the next chapter, the used modeling objective is presented. Chapter 3 describes a library for simulation of network components. Together with the controller library, rendered in chapter 4, it is possible to model distributed automation systems. Both libraries have been implemented and tested with Dymola 6.1. Application examples of the presented libraries are given in chapter 5. Finally, conclusions are drawn and an outlook on future work on the libraries is given.

2 Modeling Objective

As in all simulation applications the initial step of the designer is to identify components and effects to be considered in the simulation. Based on this identification the next step is to decide about the modeling approach for implementation. In the case of distributed automation systems we have chosen a structure conserving modeling approach, mapping real world components to individual models. Figure 1 shows an archetype of a distributed automation system and the component models provided by the Modelica libraries presented in the following chapters.

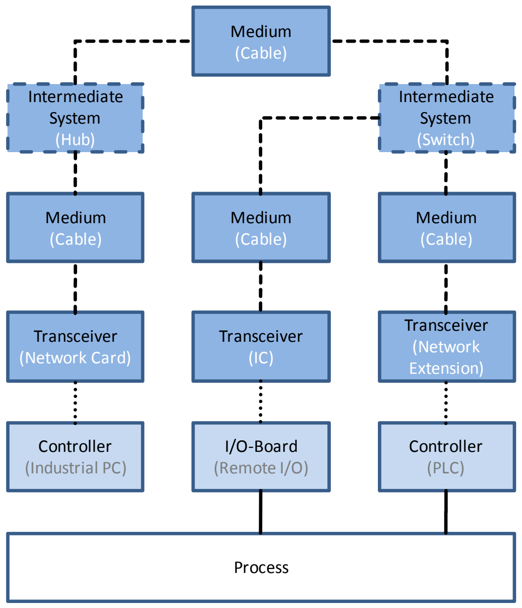

Medium (Cable)
Intermediate System (Hub)
Intermediate System (Switch)
Medium (Cable)
Medium (Cable)
Medium (Cable)
Transceiver (Network Card)
Transceiver (IC)
Transceiver (Network Extension)
Controller (Industrial PC)
I/O-Board (Remote I/O)
Controller (PLC)
Process

Figure 1: Archetype of systems that are to be covered by the presented librarys.

Basically, the components of the archetype can be divided in three domains:

The first domain (dark shading) covers the network. A network consists of one or more communication media *(Medium)*. In the case of Ethernet this is usually a twisted pair cable. If more than one *Medium* is used, coupling devices *(Intermediate System)* have to be used. In Ethernet an *Intermediate System*, could be a hub or a switch. To access the *Medium* a *Transceiver* is used. It manages all physical and protocol issues necessary for proper communication over the network. The network card of a PC is an example of a *Transceiver*.

The second domain (light shading) covers components related to data handling and process interfacing. The components *Controller* and *I/O-Board* can be regarded as embedded devices. The control algorithms are executed in *Controllers* (e.g. industrial PCs or PLCs). In general, a *Controller* has access to the network via a *Transceiver* as well as access to the controlled process via directly connected sensors and actuators. *I/O-Boards* are a simplified version of a *Controller* and allow remote access to sensors and actuators via the network. Usually, *I/O-Boards* have only limited processing abilities and are not used to execute control algorithms.

Components of the third domain (no shading) are related to the physical process.

There are three types of interconnections between components. Network connections (*dashed edges*) describe data wrapped in a protocol frame, dependant on the network type used. Pure data transport is indicated by *dotted edges*. The exchange of physical values is shown as *solid edges*.

Based on the domain classification of components in distributed automation systems, in chapter 3, a Modelica library for network components is presented. The library described in chapter 4 covers the components related to embedded devices for process control.

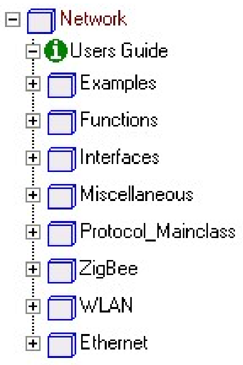
Both libraries make intensive use of the object orientation abilities of the Modelica language. Interface models are used to allow the exchange of components with similar behavior. Wherever possible and appropriate, components implement a predefined interface or are extended from other existing models. Along with the Modelica keyword *replaceable,* this allows e.g. a wide variety of controller models based on a small number of basic components. In the figures of component models replaceable component models can be identified by the gray shaded box around them (e.g. Figure 10 component CPU).

3 Network Library

3.1 Structure of the library

The network library consists of fundamental components which cover the important issues in the area of network transmission, e.g. communication media, intermediate systems, transceiver interfaces, etc. The rule of structure conserving modeling is held. I.e., the network is not modeled as a single class but all the fundamental components are explicitly modeled. The main advantage of this modeling approach is that the network topology, which can have significant influences on the network performance, is visible.

Currently, the library (Figure 2) supports three widespread transmission protocols, namely, fully switched Ethernet [3], WLAN [4] and ZigBee [5]. To increase simulation speed, the protocols are simplified to some extent, thus, only the chosen dominant factors related to the automation system are modeled. Especially noteworthy is the fact that in the network components, only the physical and data link layers of the ISO/OSI model are considered and also here some abstractions have been made. Necessary interfaces for the exchange of events are also included. In the *Functions* library, *external "C"* functions and corresponding wrap functions in Modelica can be found. External functions are used to simplify information exchange, especially for the formatted string communication. In the *Examples* library, application templates are given for each protocol, illustrating how to build a networked system using the models.



Network
Users Guide
Examples
Functions
Interfaces
Miscellaneous
Protocol_Mainclass
ZigBee
WLAN
Ethernet

Figure 2: Tree view of the network library.

3.2 Application template of the network library

Figure 3 shows a typical application example of a ZigBee network. The dark shading blocks on the leftmost side represent primary controller models. They provide the data to be sent and read the data from network messages regardless of the underlying transmission protocols. In the middle there are two transceiver modules connected to the controller and the shared medium. The transceiver module together with the medium defines the transmission protocol. In the example, controller 'A' sends messages to controller 'B' via the network while 'B' does the same to 'A'.

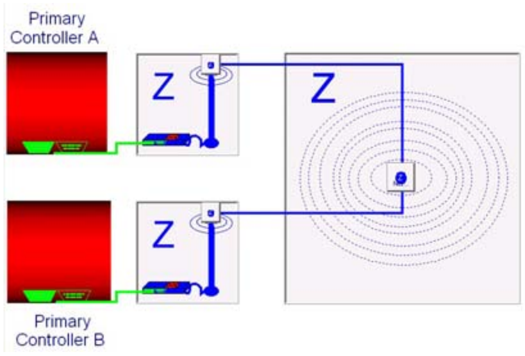


Figure 3: Application template of a ZigBee network.

The source controller renews the data to be sent cyclically and writes the message into the send queue of the transceiver module. Based on the network state and medium access algorithm, the transceiver module decides when to send the network message. After the transmission duration has expired, the network medium writes the message into the receive queue of the destination transceiver. Then, the destination controller determines when to read the data from the network message stored in the receiver queue.

3.3 Implementation of the queuing system

As noted above, the transceiver module serves as an interface between the controller and the network. In addition to the medium access control, it has to accomplish the information exchange. A transceiver interacts with both, the controller and the medium (cf. Figure 4).

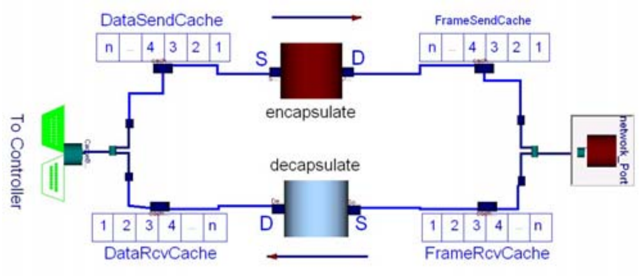


Figure 4: Queuing mechanisms in the Transceiver.

Thus the information flow in a transceiver is bidirectional. Furthermore, the information flow in one direction is split into two segments due to the unsynchronized behaviors of controller, transceiver and network. To manage the information flow, FIFO queues are utilized. Each information flow segment is represented as a FIFO queue. The two nodes on both ends of the same information flow share the access to the same queue. The FIFO queue system is implemented as *external "C"* functions to simplify the Modelica code and reduce the number of events. The interface functions in Modelica are given as:

```
QueueID=CreateQueue(QueueSize);
Enqueue(QueueID,MessageID);
MessageID=Dequeue(QueueID);
Index=ReadQueueIndex(QueueID);
```

CreateQueue(QueueSize) creates a queue with given length and returns a unique ID. Enqueue(QueueID, MessageID) stores the MessageID in the first free place of the queue. Dequeue(QueueID) reads the first message from the queue and shifts the rest of the queue one place towards the beginning position. ReadQueueIndex(QueueID) returns the current position index. The entries of a queue are message identifiers. Each message (string) is indexed with a unique integer ID. This utility is supported by a "C++" library.

Figure 5 illustrates the information exchange in queues for the example from Section 3.2. New data (string) from the controller is represented by an identifier (ID1). This ID is enqueued in *DataSendCache*. In the next step, ID1 is dequeued and the actual content of ID1 is encapsulated to a frame with protocol header. Thus, a new message (string) is produced and a new identifier (ID2) is enqueued in *FrameSendCache*. In the transmission, medium dequeues the

ID2 and makes a copy named ID3. Later after the transmission, ID3 is enqueued in *FrameRcvCache* on the destination side. Finally, ID3 is decapsulated and a message ID4 is enqueued in *DataRcvCache*. During the whole procedure, each time the Dequeue() function is called, a copy of the dequeued message is made and the original message is deleted. Hence, the message ID4 and ID1 actually have the same content. The information exchange is accomplished.

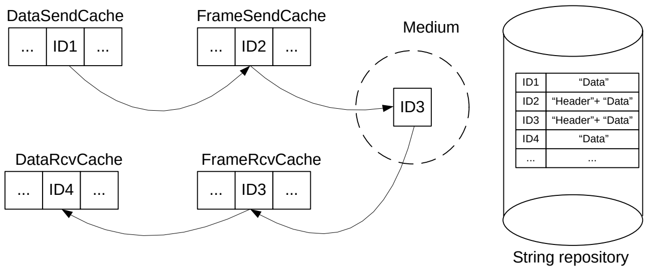


Figure 5: Queue operation in a transmission.

3.4 Implementation of Ethernet

The implemented Ethernet protocol is abstracted from a fully switched, full-duplex Ethernet. The network behavior is illustrated in Figure 6.

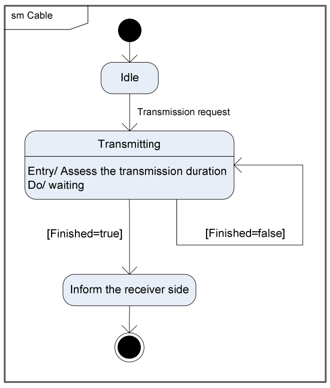


Figure 6: State diagram of Ethernet protocol.

A twisted-pair cable is taken as the modeling pattern for a medium in Ethernet. Since no collisions are considered, the end of a cable can only be connected to one transceiver. In a simulation, the cable model receives request events from connectors on both ends and sends out notifications on connectors after the internal processing.

3.5 Implementation of wireless communication

In principle, the network is modeled as a discrete event system. It reacts on external events with deterministic or non-deterministic delays. The modeling focuses on the standard MAC layer taking into account random access and conflict handling.

The implementation of the protocol is divided into software and hardware parts. The software part covers medium access, frame format etc. It is the algorithm integrated in the transceiver and intermediate system. The hardware part is the communication medium. The communication state, the transmission duration and other relevant variables are decided by electromagnetic characteristics of the medium. From this point of view, there is always a clear relation between the network protocol and the underlying physical medium. In the presented approach, parts of the protocol codes from the transceiver model are moved into the medium model. In other words, the medium is designed with some extent of intelligence.

In wireless communication, it is not possible to listen while sending because of the nature of the channel (frequency band). Hence, the Collision Avoidance (CA) method is used to improve the performance of Carrier Sense Multiple Access (CSMA). In principle, a network node always listens to the channel and sends only if the channel is sensed as idle. The implementation of this protocol is separated into two parts, namely, the medium part and the transceiver part. The interaction between medium and transceiver is realized by *Network_Port* which can be found in *Network.Interfaces.*

There are two main differences between the CSMA/CA algorithms for ZigBee and WLAN on the MAC layer:

1. WLAN has an unlimited number of retries, while ZigBee is strictly limited on retries.
2. ZigBee assesses the network state only at the end of the whole backoff time, while WLAN checks after each single delay unit.

Then again, they do have some important characteristics in common, e.g. listening before sending, random backoff waiting time before sending, incremental backoff time after collision.

Therefore, the modeling attempt is to design a common model for these two algorithms. The differences can be represented by changing model parameters. The implementation is based on the unslotted CSMA/CA scheme, which means the network works without beacon synchronization and all nodes are working in the Ad-hoc mode. Thus the access to the network is random and contention prone. Details about the protocols can be found in [4] and [5].

The common wireless medium model is illustrated in Figure 7. The shared medium is triggered if any network node sends an attempt of trying. Based on the sum of trying nodes, it decides to begin transmitting or to send a collision notification. After successful transmission, it waits for a certain time before resetting the medium state to idle. This time is given by:

$$WaitingTime = SIFS + ACK + DIFS \qquad (1)$$

where SIFS (Short Inter Frame Space), ACK (Acknowledge), DIFS (Distributed Inter Frame Space) are physic dependent parameters defined in the standard.

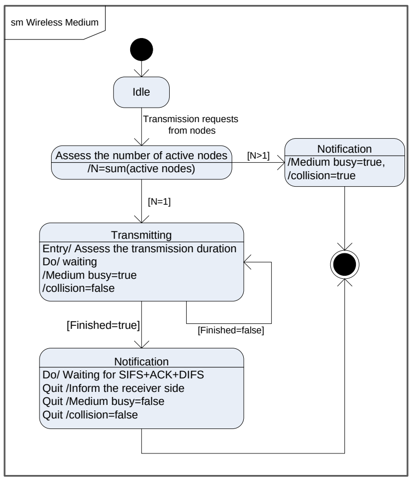

sm Wireless Medium
Idle
Transmission requests from nodes
Assess the number of active nodes /N=sum(active nodes)
[N>1]
Notification /Medium busy=true, /collision=true
[N=1]
Transmitting
Entry/ Assess the transmission duration
Do/ waiting
/Medium busy=true
/collision=false
[Finished=true]
[Finished=false]
Notification
Do/ Waiting for SIFS+ACK+DIFS
Quit /Inform the receiver side
Quit /Medium busy=false
Quit /collision=false

Figure 7: State diagram of wireless medium.

The transceiver part algorithm is mainly used to perform the backoff procedure. Here, the separate implementation shows great advantage in the WLAN protocol considering the simulation performance. Since the conflict detection is executed by the medium model, the transceiver models do not have to assess the medium state after each unit delay, but only to wait for the event trigger from medium notification. By doing so, unnecessary events, which slowdown the simulation dramatically, are avoided. For instance, 802.11 standard defines the backoff time in the unit of timeslots:

$$Backofftime = backoff\_counter \times slot\_time \qquad (2)$$

where the initial backoff_counter is a random number in the range [0, 31] and slot_time is 20 µs [6]. If no collision happens, it causes on an average 16 events with a cycle of 20µs per transmission. In the worst case (4 collisions happen successively), there are 1031 events for a transmission. In the separate implementation, the number of events is reduced to 1.

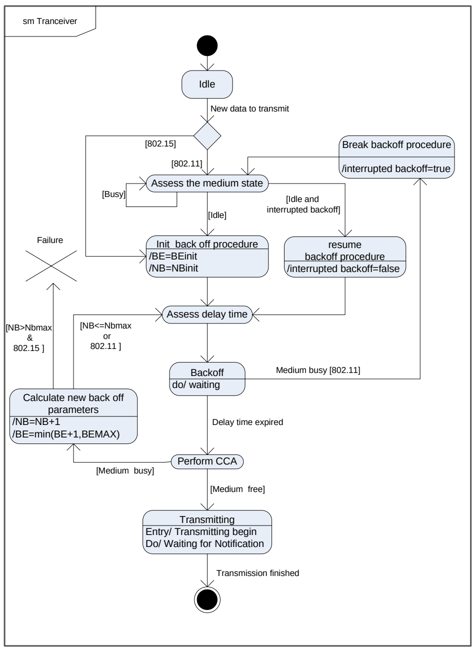


Figure 8: State diagram of wireless transceiver.

The designed common model is illustrated in Figure 8. The exact model behavior is predefined by a model parameter given as "802.11" or "802.15". The common model is a partial model in Modelica, thus in the application, it is instantiated as a replaceable model and can be easily parameterized for different protocols.

There are some important assumptions to be noticed:

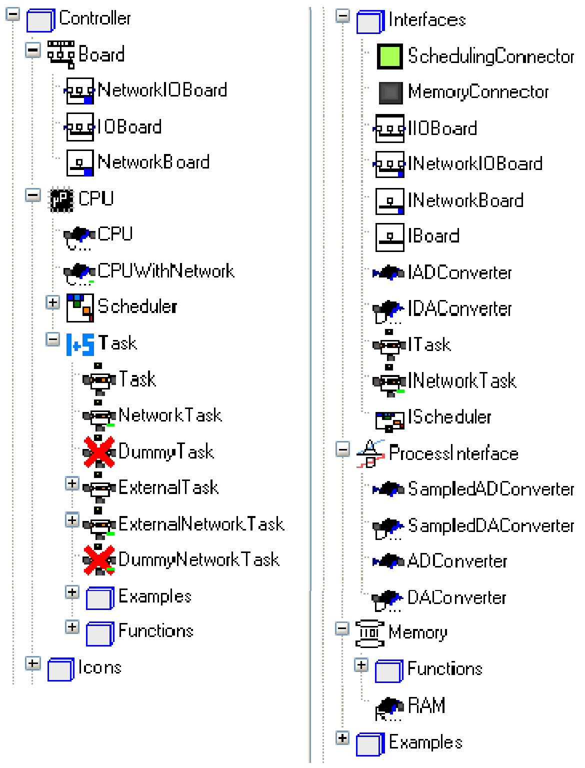
1. No transmission failure is taken into consideration, i.e. no packet is lost in the transmission and no re-transmission is needed.
2. The acknowledge message is not modeled.
3. One shared medium model represents one available channel. All nodes connected to the medium hence operate in the same channel. No dynamic channel switching is considered. As a consequence, the network capacity is restricted.

4 Controller components library

4.1 Overview of the library

The controller components library contains models to describe the behavior of an embedded controller device. In comparison to the simulations of automation systems without detailed controller models, the effects of synchronization, scheduling and queuing are considered in retrieving system behavior which provides more realistic simulation results.

The library is split into sub-libraries that group controller components by their function (cf. Figure 9).



- Controller
  - Board
    - NetworkIOBoard
    - IOBoard
    - NetworkBoard
  - CPU
    - CPU
    - CPUWithNetwork
    - Scheduler
    - Task
      - Task
      - NetworkTask
      - DummyTask
      - ExternalTask
      - ExternalNetworkTask
      - DummyNetworkTask
      - Examples
      - Functions
  - Icons

- Interfaces
  - SchedulingConnector
  - MemoryConnector
  - IIOBoard
  - INetworkIOBoard
  - INetworkBoard
  - IBoard
  - IADConverter
  - IDAConverter
  - ITask
  - INetworkTask
  - IScheduler
- ProcessInterface
  - SampledADConverter
  - SampledDAConverter
  - ADConverter
  - DAConverter
- Memory
  - Functions
  - RAM
- Examples

Figure 9: Screenshot of the controller library.

Board

The basic component of a controller device is a board. It hosts devices that are needed to run control algorithms, to interface plants to be controlled and to interchange information between controller devices. The library *Controller.Board* hosts three basic board models which distinct in the interfaces they provide (only process interface (IOBoard), only network interface (NetworkBoard), combination of both(NetworkIOBoard)). The components of the NetworkIOBoard model, shown in Figure 10, will be detailed in the following sections.

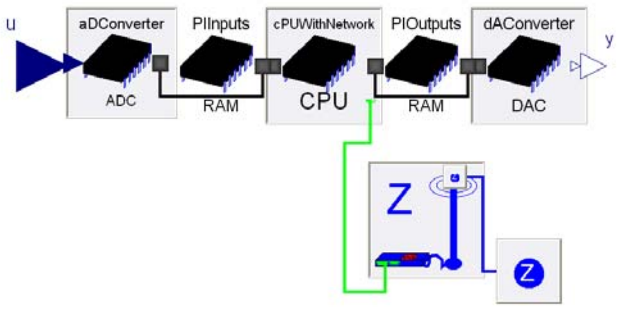


Figure 10: Component model of a board with process and network interfaces.

Process Interfaces and Converters

The process interface consists of an array of continuous input (u) and output (y) signals. As in real controllers input and output signals are not directly connected to the CPU. The input signals are first converted by a hardware AD-Converter, and the DA-conversion of the output signals is done by means of a hardware DA-Converter. The converter models can be found in the *Controller.ProcessInterface* library.

RAM

The results of the AD-conversions are stored in a random access memory (RAM) called process image of inputs (PIInputs), whereas the output signals to be DA-converted are read from the process image of outputs (PIOutputs) by the DA-Converter.

The *RAM* model can be found in *Controller.Memory*. It provides means to exchange information between component models. To attach models to a memory component the *MemoryConnector* is used.

Network Interface

The network interface is provided by the network library. It is regarded as a transceiver IC which performs network operations concurrently. The CPU can transfer messages to be sent via network to the transceiver IC whereas received network messages can be read by the CPU from the transceiver IC.

CPU

The central processing unit (CPU) executes the control algorithms. The CPU models can be found in the library *Controller.CPU*. There are two different CPU-models in the library, one without network access *(CPU)* and one with network access *(CPU-WithNetwork)*. The *CPU* model is instantiated as replaceable in the board models and, thus can be exchanged to other *CPU* models extending *CPU*. The CPU executes the control algorithms wrapped in Tasks as described in the next section. Figure 11 shows the *CPUWithNetwork* model.

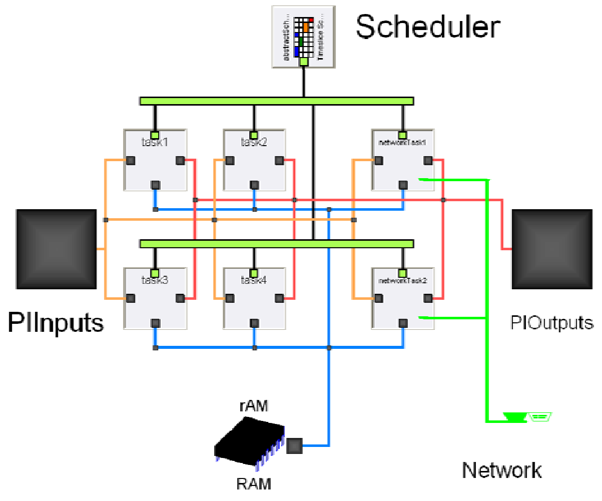


Figure 11: Model of a CPU with network access.

Task and Scheduler

A task (library *Controller.CPU.Task*) is a software process which runs quasi-concurrently to other tasks in a CPU. A scheduler (library *Controller.CPU.Scheduler*) allocates processing time to the tasks according to a certain scheduling policy. The library provides several scheduler models (e.g. Round Robin, FIFO) which are all based on the *Controller.Interfaces.IScheduler* model.

All tasks have access to the process images (PIInputs and PIOutputs) and share a common memory called RAM. The RAM can be used to exchange information between tasks inside a CPU. The connections to the different memory types are drawn in different colors in Figure 11 (PIInputs: orange, PIOutput: red, RAM: blue). The connections between the tasks and the scheduling model are drawn in black. The green horizontal bars are used as a Modelica bus to reduce connections to the scheduler.

The *CPUWithNetwork* model can host up to four tasks without network access (*Task*) and two tasks with network access (*NetworkTask*). *Task* and *NetworkTask* are implemented as partial models and serve as generalized task models. The task models are instantiated as replaceable in the *CPUWithNetwork* model and can be easily changed to specialized ones. The models *DummyTask* and *DummyNetworkTask* can be used to specify that a task is not present. The library user can easily build his own *CPU* models with more Tasks by extending the existing ones.

4.2 Implementation details

RAM

The *Controller.Memory.RAM* model plays an important role in the simulation of an embedded controller. It provides means to interchange information between controller components. On the first look, this does not seem to be a big issue in Modelica, as the *connector* type is especially designed for this purpose. But, in the domain of informational systems a connector is not a convenient tool to exchange information between components. Due to the fact that the amount of data exchanged may vary, not all cases can be covered when designing a general information exchange connector. Instead, it is appropriate to make use of the external function interface of Modelica to implement an information exchange system in a dedicated programming language which is then triggered by Modelica models. In this way the complexity of data handling is hidden from the Modelica models. To do so, a C++ library has been developed which emulates a collection of random access memories that can be accessed via a unique index. Figure 12 shows a coarse overview of the C++ library internals and the interconnection with the Modelica model.

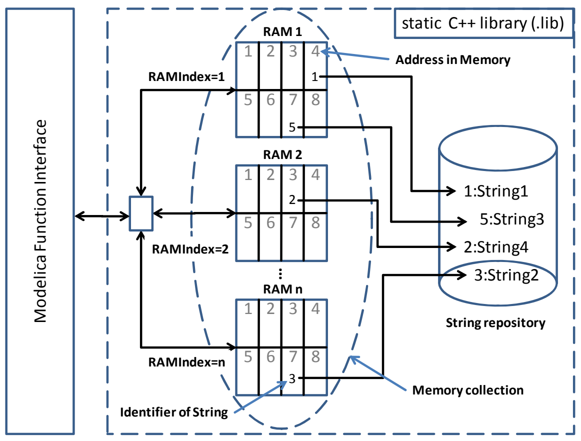


Figure 12: Coarse overview of the C++ library.

Interface functions to use the C++ memory collection are provided in the sub-libraries *Controller.Memory.Functions.WrapperFunctions*:

```
RAMIndex:=createRAM();
value:=readRAM(RAMIndex,address);
writeRAM(RAMIndex,address,value);
```

The function *createRAM* creates a random access memory in the collection and returns the unique identifier of the memory (*RAMIndex*) as an Integer. When reading or writing from or to a memory, the *RAMIndex* has to be passed to identify the memory.

readRAM is used to read values (as String) from the memory *RAMIndex* from the given position *address* (as Integer) in the memory, whereas *writeRAM* is used to store a value in the given memory *RAMIndex* at position *address*.

In the C++ library, a RAM is organized as a collection of references to strings (character arrays) stored in a global string repository. Library internal functions provide the mechanism to read and write strings to the string repository, identified by the *RAMIndex*, using the given *address* which is a local identifier inside a RAM.

A slightly varied functionality is used to implement the queuing mechanism in the same library. Instead of giving random access to values in a *queue*, the interface functions only allow reading (dequeueing) from the first address in the queue and writing (enqueuing) at the end of the queue. The string management is done in the same way as in the case of RAM, using the string repository.

For maximum flexibility, the values passed to or from the memory are of type String. This way arbitrary information can be used in the simulation. When necessary, numerical information contained in the strings can be parsed by means of functions provided in the *Modelica.Utilities.Strings* library. In the case of real values, the library provides the functions *readReal* and *writeReal* in the sub-library *Controller.Memory.Functions*, where parsing is done automatically.

The *RAM* model itself is just a placeholder for one of the memories managed in the C++ library. Its only dynamic behavior is a function call to create a memory in the memory collection in the initial simulation step. The unique index of this memory is then stored in the RAM model and published via the *MemoryConnector*, which only consists of the unique memory index.

To improve debug capabilities, the interface functions provide the possibility to trace read and write accesses to the RAM model. For this purpose, each function call, including function parameters, of readRAM and writeRAM can be stored in a textfile or a database table (cf. Table 1).

Table 1: Dump of a database trace of operations on RAM models.

id	operation	RAMindex	address	value
		//		
28	W	1	2	"5"
29	W	2	5	"-3"
30	R	1	2	"5"
31	W	2	3	"3"
32	R	2	2	"1"
33	R	1	2	"5"
		//		

Tasks

As described above, the control algorithms are organized in tasks to allow multiple algorithms to run quasi-concurrently on a single CPU. Hence, if multiple tasks are active at once, they are competing for processing time. The scheduling instance manages that the processing time is spread among all tasks.

The task models (*Controller.Task.**) can be seen as wrapping units that provide interfaces to access CPU-internal (e.g. Memory, Timer) and CPU-external components (e.g. process images, network port). Therefore, all tasks extend the *Controller.Intefaces.ITask* interface. The behavior is described in Figure 13.

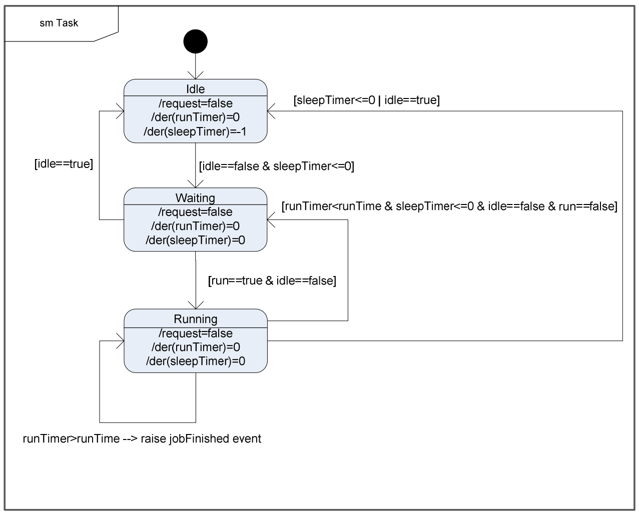

sm Task
Idle
/request=false
/der(runTimer)=0
/der(sleepTimer)=-1
[sleepTimer<=0 | idle==true]
[idle==true]
[idle==false & sleepTimer<=0]
Waiting
/request=false
/der(runTimer)=0
/der(sleepTimer)=0
[runTimer<runTime & sleepTimer<=0 & idle==false & run==false]
[run==true & idle==false]
Running
/request=false
/der(runTimer)=0
/der(sleepTimer)=0
runTimer>runTime --> raise jobFinished event

Figure 13: State diagram of a task.

A task can be in one of three states (*Idle*, *Waiting*, *Running*), and is controlled by four variables: *run*, *idle*, *runTime* and *sleepTime* which enforce state switches. Depending on the state, the timers *runTimer* and *sleepTimer* are active or inactive and processing time is requested or not (variable *request*).

The external variable *run* is provided by the scheduler and determines if the task is currently the active task in the CPU. The internal variable *idle* enforces the task to go to *Idle* state if its value is true. *runTime* determines the processing time that is needed to finish the current job of the task, while *sleepTime* is used to assign the period to send the task to *Idle* state.

If a task has no job to do, it is in *Idle* state. In state *Waiting*, the task is requesting processing time, but currently does not get processing time by the scheduler. In state *Running*, the task gets processing time and is thus running. When the *runTimer* value exceeds *runTime* the *jobFinished* event is raised which indicates that the current job of the task is completed. This event is used to embed the actual control algorithm in the *Task* model, just by extending one of the given template tasks (*Task* or *NetworkTask*). The behavior is given by the extended *Task* model. This way the library user can focus on the implementation of the algorithms itself.

Integration of Algorithms

The Controller library provides basically two different ways to define the algorithms executed in a *Task* model:

1. algorithm definition in Modelica language
2. algorithm definition in external libraries using the Modelica external function interface

To define and implement an algorithm in Modelica, one of the basic Task models (*Task* or *NetworkTask*) must be extended. As intelligence for scheduling is predefined in the basic Task models, only the values of the task control variables *idle*, *sleepTime* and *runTime* as well as the reaction on the *jobFinished* event must be specified.

The basic structure of tasks using Modelica algorithms is defined as shown below.

```
model ExampleTask
 extends Controller.CPU.Task.Task;
initial algorithm
 sleepTime:=0;
 runTime:=100/CPUFreq;
algorithm
 when (pre(jobFinished)) then
  //here, the algorithm semantics are specified
 end when;
end ExampleTask;
```

The *initial algorithm* section assigns start values for the control variables *sleepTime* and *runTime*. In the algorithm section, the processing of the *jobFinished* event is defined by means of a *when* block. The condition *jobFinished* must be wrapped by a *pre* to cut the algorithmic loop involving the task and the scheduler component.

The semantics of the algorithm is then specified in the body of the when block, as shown in the following example of a single input single output P controller:

```
…
import Functions=Controller.Memory.Functions;
…
when (pre(jobFinished)) then
 //when a job is finished, runTimer is reset
 reinit(runTimer,0);
 //state==0: read input value
 if (state==0) then
  //read sensor value from process image
  y:=Functions.readReal(PIInputs, 1);
  //next state is executing control law
  state:=1;
  //executing control law takes 5000 cycles
  runTime:=5000/CPUFreq;
  //state==1: execute control law
 elseif (state==1) then
  //calculate control error and new set value
  e:=w - y;
  u:=k*e;
  //next state is writing new set value
  state:=2;
  //execution for writing takes 500 cycles
  runTime:=500/CPUFreq;
  //state==2: set output value
 elseif (state==2) then
  //write output value to process image
  Functions.writeReal(PIOutputs,1,u);
  //next state is reading input values
  state:=0;
  //reading input signal takes 500 cycles
  runTime:=500/CPUFreq;
 end if;
end when;
…
```

The task is divided into three jobs which are executed sequentially. The job currently processed is indicated by the Integer variable state (0: read inputs, 1: calculate output, 2: write output). The execution time can vary among the jobs and is expressed in terms of processor cycles divided by the processor frequency (*CPUFreq* in Hz). When the *jobFinished* event arises, the semantics of the job is being performed and the settings for the next job are carried out. This means that the *runTimer* variable is reset to zero and that the *runTime* for the next job is assigned. The task should run without breaks and thus is never send to *Idle* state (`sleepTime:=0, idle:=false`).

As described earlier, the tasks do not have direct access to the environment (input and output signals). Instead, the algorithms work on images of the input and output signals by reading from the process image of inputs (*PIInputs*) or writing to the process image of outputs (*PIOutputs*) using the provided interface functions.

For complicated algorithms, the Modelica design language is not the mean of choice. For this case, the *Controller.CPU.Tasks* library provides the *ExternalTask* and *ExternalNetworkTask* models, which allow the definition of algorithms in other programming languages (e.g. C or Java). Using the construct of *replaceable function,* the user of the library can easily access external algorithms by providing Modelica interface functions extending *Controller.CPU.Tasks.externalAlgorithm*.

5 Application of the Libraries

Many analysis problems in DAS can be characterized as runtime or delay investigations. A typical real world problem is the determination of the response time distribution on events in the process to be controlled (e.g. emergency stop). Results, elaborated with an earlier version of the presented library, have been published in [7].

Including the plant under control in the simulation, quality analysis can be performed. Reaction delays, due to the distributed nature of DASs, cause fluctuations in the control quality. As shown in [2], the libraries can be used to analyze the variations of process values based on a collection of simulation runs.

Another application of overall system simulation is feasibility analysis. It can be used to perform proof-of-concept tests in early development stages of DASs. The classical example for feasibility analysis in continuous control is the stabilization of an inverted pendulum in the instable (upper) rest position.

Figure 14 shows the setup of a DAS with an inverted pendulum using a wireless network. The experimental setup consists of the inverted pendulum with 0.5 kg mass for the cart as well as for the pendulum arm with a length of 1 m. In the initial, state pendulum arm and cart are not moving, but the pendulum arm is rotated by $\varphi = 0.25$ rad.

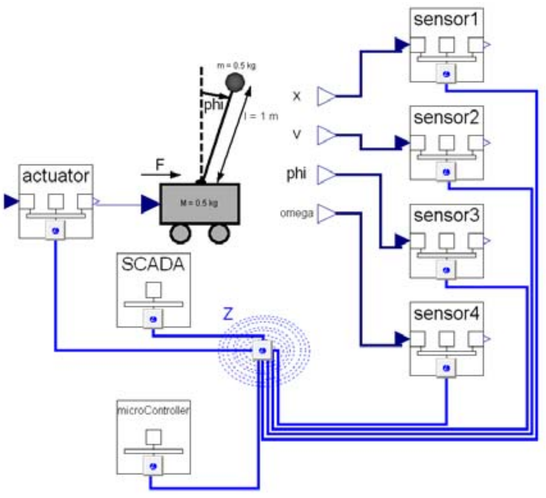


Figure 14: Inverted pendulum experiment.

Attached to the pendulum there are four *Controller.Board.IOBoard*s (sensor1-4) providing information about position and velocity of the cart as well as angle and angular velocity of the pendulum arm. The *Controller.Board.IOBoard* actuator drives the cart with the translation force calculated by the *Controller.Board.NetworkIOBoard* microController. The *Controller.Board.NetworkIOBoard* SCADA (Supervisory Control And Data Acquisition) is allotted to collect the overall system status periodically.

The control algorithm on the microcontroller is split up into three concurrent tasks. The first task cyclically requests sensor values from the sensor boards and sends the calculated force value to the actuator board. The second task handles incoming network messages, and is only active when network messages are available. The third task is the control algorithm itself working in three sequential steps:

1. reading input signals from the RAM, provided by the remote sensors,
2. calculating the force value (control law),
3. writing the force value to the RAM.

The control law is a state controller, with gains acquired from a linear continuous time state space model.

The requesting task runs 4 ms and then falls asleep for 20 ms. The message handling task needs 1 ms to process each incoming message. The control algorithm task needs 1 ms to read the input signal values as well as 1 ms to write the force value. To execute the control law, 11 ms are needed.

Processing of network messages in the sensor and actuator boards takes 2 ms. As there are no concurrent tasks in these boards execution starts as soon as a network message is received.

The example setup has been simulated with two different networks, ZigBee with 250 kbps transmission rate and WLAN with 11 Mbps transmission rate. Additionally, the inverted pendulum has been simulated using the same control law emulating a continuous controller neglecting all delays imposed by the automation system.

As shown in Figure 15, the pendulum angle varies a lot among the three simulated scenarios. As expected, the scenario neglecting delays shows best performance and the WLAN scenario is superior to the ZigBee scenario.

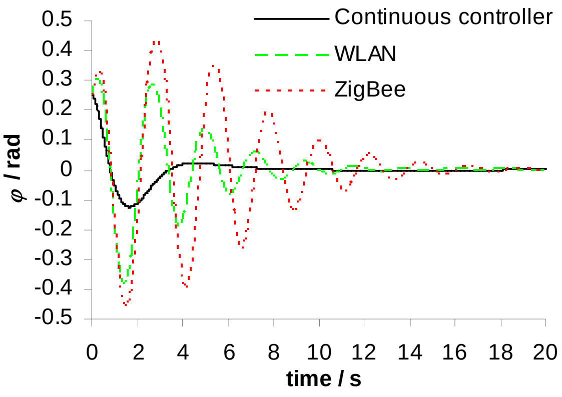


Figure 15: Pendulum arm angle sequence plot.

The reason for the differences among the scenarios is the delayed application of the actuator force caused by the automation system delays. It can be seen from Figure 16 that the first update of the actuator value occurs after approx. 80 ms using WLAN. In the scenario with ZigBee this delay increases to 100 ms.

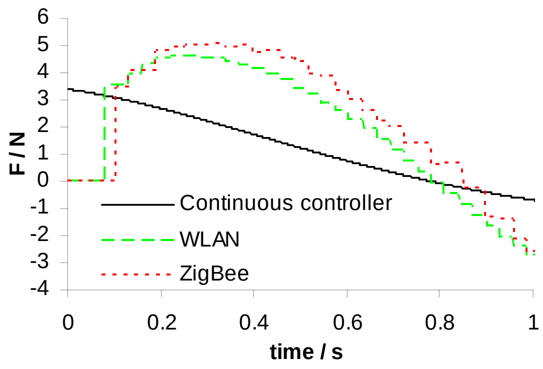


Figure 16: Zoomed applied translational force sequence plot.

A 20 s lasting simulation (carried out in Dymola 6.1) takes 94.2 s CPU time on a PC with 2.8 GHz Pentium IV HT and 2 GB RAM. Best performance has been achieved using the *Lsodar* integration algorithm with 1e-6 tolerance.

Using the common setup for storing variables consumes too much memory and disk space, even for short simulations (less than 10 seconds). Thus, interesting simulation values have to be stored using proprietary mechanism, e.g. sampled saving of data to a file with using *Modelica.Utilities.Streams.print*.

6 Conclusions and Outlook

Libraries for simulation of Distributed Automation Systems using the Modelica language have been presented. The libraries allow delay time determination, quality of control analysis and feasibility analysis in closed-loop applications. The application of the libraries has been illustrated by an example using wireless communication.

Future work will focus on improvement of the network models regarding failure behavior (e.g. packet losses) and the integration of other networks especially in the field of automotive applications (CAN, LIN, …).

The presented libraries can be downloaded from http://www.eit.uni-kl.de/frey.

References

[1] Cervin, A.; Ohlin, M.; Henriksson, D. *Simulation of networked control system using TrueTime*. 3rd International Workshop on Networked Control Systems: Tolerant to Faults, Nancy, France, June 2007

[2] Liu, L.; Frey, G. *Simulation Approach for Evaluating Response Times in Networked Automation Systems*. IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA), Patras, pp. 1061-1068, Sept. 2007.

[3] IEEE Std 802.3, Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, 1999.

[4] IEEE Std 802.11, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 1999.

[5] IEEE Std 802.15.4, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs) 2003.

[6] Ferre, P.; Doufexi, A.; Nix, A.; Bull, D. *Throughput analysis of IEEE 802.11 and IEEE 802.11e MAC*. Wireless Communications and Networking Conference (WCNC), Vol. 2, pp. 783 - 788 March 2004

[7] Greifeneder, J.; Liu, L.; Frey, G. *Methoden zur Antwortzeitanalyse in vernetzten Automatisierungssystemen*. SPS/IPC/DRIVES, Nürnberg, Germany, pp. 517 - 525, Nov. 2007.