TrueTime Network — A Network Simulation Library for Modelica

Philip Reuterswärd[a], Johan Åkesson[a,b], Anton Cervin[a], Karl-Erik Årzén[a]
[a]Department of Automatic Control, Lund University, Sweden
[b]Modelon AB, Sweden

Abstract

We present the TrueTime Network library for Modelica, developed within the ITEA2 project EUROSYSLIB. It allows for simulation of various network protocols and is intended for use within real-time networking. We describe some its features and discuss implementational issues. Since TrueTime Network is programmed in C, special attention is given to the how Modelica's external function interface is used. We also discuss briefly a future native Modelica implementation.

Keywords: Modelica; Network Simulation; TrueTime; Real-time Control

1 Introduction

Networked systems and networked control are increasingly common in many domains, e.g., in automotive systems. Sending signals over networks cause delays that, depending on the network protocol, can be more or less deterministic. Some examples of delays are network interface delays, transmission delays, propagation delays, and back-off times in case of collisions on a shared medium. In order to accurately simulate the consequences that these delays have on the overall system performance it is important to be able to model the network at an appropriate level. A too detailed network model including, e.g., the transmission of individual bits, will make the simulation too slow. Furthermore, this level of detail is in most cases unnecessary. A too coarse model, e.g., to model the network as a constant delay, will in many cases fail to capture the dynamics introduced by the network communication.

Within the ITEA2 project EUROSYSLIB, the Department of Automatic Control, Lund University is developing a Modelica network protocol library. The intended application area is real-time networking. In these networks, the upper layers of the ISO/OSI protocol stack are normally not used. Hence the library only models various wired or wireless data-link layer protocols with focus on the MAC-access related sources of delays. The library is based on the Matlab/Simulink toolbox TrueTime [1] developed in the same group.

The Modelica implementation of the TrueTime Network is based on the existing Simulink implementation of TrueTime, with some modifications. TrueTime Network makes it possible to simulate the sending of reals and arrays of reals over a network using different network protocols. It is implemented in C and used in Modelica through the external function interface.

The organization of the paper is as follows. In Section 2 we describe the TrueTime simulation package for Simulink and how it models networks and network protocols. Section 3 gives an introduction to the TrueTime Network library for Modelica from a user's perspective. In Section 4 we discuss the implementational aspects of the TrueTime Network library. Special attention is given to the usage of the external function interface. Section 5 discusses a future native Modelica implementation of the library.

2 TrueTime

TrueTime [1] is a Matlab/Simulink-based simulation tool that has been developed at Lund University since 1999. It provides models of multitasking real-time kernels and local-area wired and wireless networks that can be used in simulation models for networked embedded control systems. The TrueTime Network library is a Modelica port of TrueTime's network part. It supports six simple models of networks — CSMA/CD (Ethernet), CSMA/AMP (CAN), Round Robin (Token Bus), FDMA, TDMA (TTP) and Switched Ethernet.

DOI: 10.3384/ecp09430058

In addition, the wireless network protocols IEEE 802.11b/g (WLAN) and 802.15.4 (ZigBee) are also supported.

TrueTime models networks as a set of FIFO input queues, a shared communication medium, and a set of FIFO output queues. The queues model the send and receive buffers in the nodes connected to the network. A message that should be transmitted from one node to another is placed in one of the input queues. Messages are moved from the input queues, into the network, and into the output queues in an order that depends on the simulated network protocol. A message moves between a number of different queues on its way over the network in a fashion specified by the protocol. The transmission time of each message depends on the length of the message. Collisions and retransmissions are simulated in the relevant protocols. The wireless network models also take the path-loss of the radio signals into account, and as such uses coordinates to specify the locations of the nodes.

Propagation delays are not modeled, since they are typically very small in a local area network. Only packet-level simulation is supported, we assume that higher level protocols have divided long messages into packets.

Figure 1: The TrueTime Network package

3 Modelica Library

An overview of the library as shown in Dymola [3] is shown in Figure 1. The TrueTime Network library supports block based modeling with several different networks running in the same simulation. To each implemented protocol there is a corresponding block to allow for graphical modeling. The different network settings can be changed in the block masks. The inputs and outputs of the network blocks are signals that are used to trigger the sending and receiving of network packages.

Additionaly there are blocks for sending and receiving of network packages that are meant to be connected with the network blocks, see Figure 2. Separate blocks exist for the sending and receiving of scalars and arrays. There are also blocks that, given an interval, sample and send a signal periodically over the network. Finally there are blocks representing empty nodes, these should be connected to ground the network in case there are nodes that do not send or receive.

The network protocols have several settings, see e.g. Figure 3, some common to all and some specific to certain protocols. The network ID is an unique identifier for each network. The number of nodes in the networks must be known at the time of compilation and are specified by the user. The frame size and the speed of the network can also be tuned. The loss probability determines the probability that a message is lost in transit. Lost messages still consume bandwidth but never arrive at their destination. It is possible to set the value of the seed for the random number generator used to calculate if a package is lost or not. Setting of the seed makes it possible to conduct Monte-Carlo type simulations.

The wireless protocols have some additional settings. When simulating a wireless network the position of the nodes must be set. This is done either once at initialization, or continuously throughout the simulation in the case of a moving wireless network node. The transmission power and signal threshold parameters determines how the wireless network signals will be intercepted. There are also settings controlling the sending, resending and timing out of network packages.

3.1 Example Usage

The library comes with some examples, showing the intended usage of the TrueTime Network library. The examples deals with control loops that

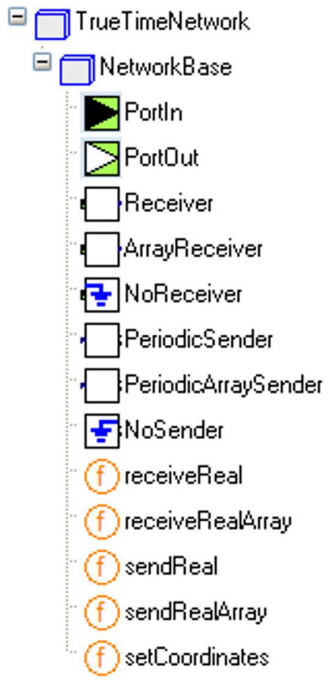

Figure 2: Blocks and external functions for sending and receiving network messages

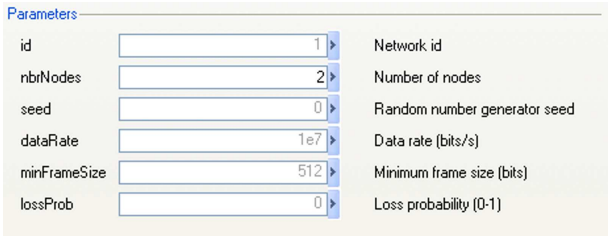

Figure 3: Parameter dialogue of the CSMA/CD network block

are closed over networks. Examples show both how to use a wired network, such as Ethernet, and a wireless network protocol, e.g., WLAN. The latter involves setting the positions of the network nodes during the simulation. It is possible to model both using blocks, see Figure 4, and standard Modelica, see Listing 1.

During the simulation it is possible to log the various networks signal and values sent over the network. The netwoks also log the network schedule, which show when packages were sent and when collisions happpened, see Figure 5.

4 Implementation

The original TrueTime Simulink blocks are implemented as variable-step S-functions written in C++. Internally, each block contains a pointer

```
model NetworkExample
  CSMACD_Network
            network(id=1,nbrNodes=2);
  Receiver rcv1(id=1,address=1)
  Receiver rcv2(id=1,address=2)
  PeriodicSender snd1(id=1,address=1)
  PeriodicSender snd2(id=1,address=2)
  ...
equation
  connect(rcv1.portIn,
          network.portOut[1]);
  connect(rcv2.portIn,
          network.portOut[2]);
  connect(snd1.portOut,
          network.portIn[1]);
  connect(snd2.portOut,
          network.portIn[2]);
  ...
end NetworkExample;
```

Listing 1: The network simulation loop

to a network structure and a discrete-event simulator. Zero-crossing functions are used to force the solver to produce "major hits" at each internal (scheduled) or external (triggered) event. The events include sending and receiving of messages. Events are communicated between blocks using trigger signals that switch value between 0 and 1. At events the network is run and network packages are moved between the FIFO queues that the network comprises.

The C++ implementation of TrueTime was ported to C, so that it could be used with Modelica through the external function interface [2]. External objects are used to represent networks corresponding to different protocols. Since the external objects do not allow for member functions, auxiliary external functions are used to, e.g., run the network and to send and receive network packages. This hides the implementational details from the user.

Modelica currently does not support external states. This means that once we run the network there is no way to roll back to a previous state. Care must be taken when updating the network, so that we do not run the network in the "future". This could happen, depending on the implementation, prior to event detection when the integrator tries to step. The Simulink simulator has richer interface to its integrator, which solves the problem in the Simulink environment. In Modelica we accomplish this by careful use of the `when`-construct.

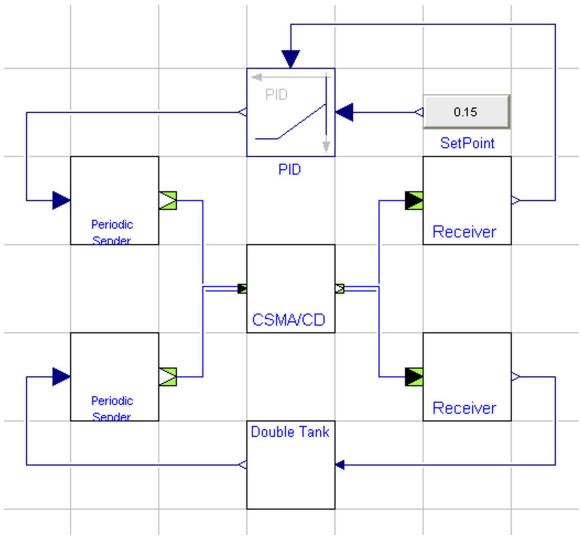

Figure 4: Simulating a PID control loop closed over a network

```
RTnetwork* nw;
int nw_id = 1;

nw = getNetworkPtr(nw_id);
```

Listing 2: Retrieving a network structure pointer

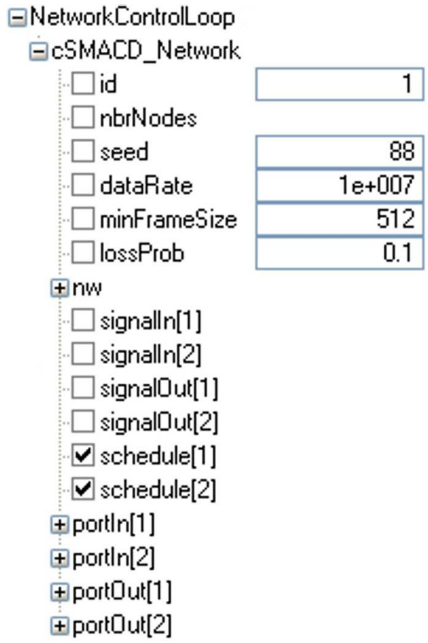

Figure 5: Simulation variables

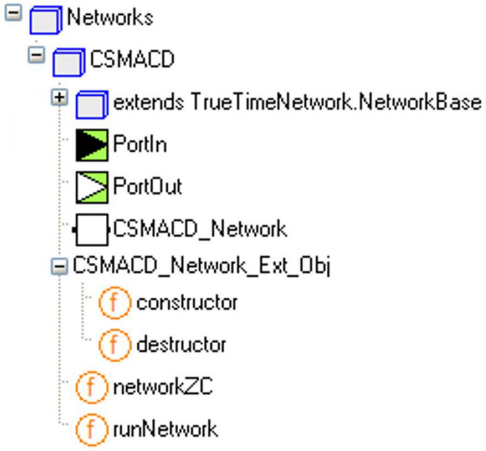

Figure 6: Network protocol implementation

4.1 External Network Objects

Figure 6 shows a network protocol model in Modelica. A network is implemented in dymola using an external object representing each network protocol. It points to the external C implementation of the network model. There is also a network wrapper-class that handles the access to the external object. This is done through the functions `networkZC` and `runNetwork`. Other external functions can also access the external network model by specifying the network ID. Externally, a pointer to the network structure can be obtained by doing a lookup on this number, see Listing 2.

To simulate network transmissions TrueTime Network relies on two functions, the zero-crossing function `networkZC` and `runNetwork`, see Listing 3. When a package is sent over the network, the network does not receive the package itself. Instead it reads a boolean signal and triggers on the flanks of it. When an incoming signal is received, signaling the arrival of a new network package, the network is run by calling `runNetwork`. By polling the network using the `networkZC`, we know when the network should be run the next time. If it returns zero, a `when`-clause triggers and the triggering package is either delivered to its destination or moved towards it through the FIFO queues that make up the network. Dropping a package simply means removing it from the network.

4.2 Sending and Receiving

Before triggering a signal on the send port of the network the sending node must create a network package and enqueue it in the external network data structure. When a message is sent, by calling the external function `sendReal`, see Listing 4, a message structure is created and is inserted into a FIFO queue. The network is accessed by doing a lookup using the network id number. All this takes

```
algorithm
  when change(signalIn) then
    (signalOut,schedule) :=
        runNetwork(nw,nbrNodes,time);
  end when;

  nextHit := networkZC(nw, time);
  when (nextHit <= 0.0) then
    (signalOut,schedule) :=
        runNetwork(nw,nbrNodes,time);
  end when;
```

Listing 3: The network simulation loop

```
function sendReal
  input Integer id "Network id";
  input Integer sender;
  input Integer receiver;
  input Real u "data";
  ...
  output Real y;
external "C" y = ...
  annotation(Include = ... );
end sendReal;
```

Listing 4: The `sendReal` external function

place in the external C code. At the same time, on the Modelica side, a boolean variable representing the input is toggled. When this happens the network is run, by calling the `runNetwork` function.

When making its way over the network, a message is transferred between a number of queues. The sending and receiving of messages is event based. How and when it is moved is determined by the protocol model of the network that is to be simulated. To determine when to run the network a zero-crossing function is used. A call to `runNetwork` placed within a `when` construct achieves this. When the network is run, it checks to see if any messages are to be transferred between the FIFO queues. The network also calculates the time of the next hit. This updates the value reported by the zero-crossing function.

When a message is ready to be received, a boolean variable is toggled. This triggers a call to the `receiveReal` function, which retrieves the message from the network. When sending and receiving arrays of data, the user specify at compile time the length of arrays, see Listing 5.

```
function receiveRealArray
  input Integer id "Network id";
  input Integer receiver;
  input Integer length;
  output Real[length] y;
external "C" ...
  annotation(Include = ... );
end receiveRealArray;
```

Listing 5: The `receiveRealArray` external function

5 A Native Implementation

In order to increase the transparency of the protocol implementations the network simulation engine may be implemented in native Modelica, rather than in C. Initial work following this approach was done.

The implementation was largely based on the design of TrueTime. The basic building block for this implementation is the `RingBuffer` class, which emulates a buffer of limited size containing network messages. The network messages in turn are represented by a record class `NWMessage` and subclasses thereof for each individual protocol. The implementation also contains connectors for connecting nodes to the network block, similar to TrueTime. The connectors then contain variables corresponding to the addresses of the sending and the receiving nodes, the actual data. The connectors also carry a boolean variable which is used to signal transmission. When this variable is toggled, the receiving side takes appropriate actions, for example reads the message and store it in an internal buffer. The project is still ongoing.

A particularly interesting extension of this work would be to use ModeGraph to model the state machines of sending and receiving nodes as well as the protocols. In particular since network protocols are often specified in terms of graphical state machine descriptions. Indeed, this approach would further improve the clarity and transparency of the network protocol implementations.

6 Summary

We have presented the TrueTime Network library, developed within the ITEA2 project EUROSYS-LIB. The library is implemented using external objects and we have showed key aspects of the im-

plementation related to the external function interface. We have also talked about a future native Modelica implementation, which is being worked on at the time of writing.

References

[1] Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker and Karl-Erik Årzén. *How Does Control Timing Affect Performance? Analysis and Simulation of Timing Using Jitterbug and TrueTime* IEEE Control Systems Magazine 23, 16–30, 2003.

[2] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling — Language Specification Version 3.0*, 2007.

[3] Dynasim AB. Dymola - Dynamic Modeling Laboratory. `http://www.dynasim.se`

