Proceedings of the 2006 IEEE
Conference on Computer Aided Control Systems Design
Munich, Germany, October 4-6, 2006

Co-Simulation of Networked Embedded Control Systems, a CSP-like process-oriented approach

Matthijs H. ten Berge, Bojan Orlic, and Jan F. Broenink, *Member, IEEE*

***Abstract*—Complex control software problems can be solved by using structured design methods that take advantage of hardware abstraction and concurrency. In our lab, a toolchain has been developed that facilitates such a design method. This paper presents two extensions to this toolchain. The first, a distributed simulation framework, enables one to simulate a complete distributed control system, prior to the actual implementation. Focus has been on the influence the network communication exerts on the overall behavior of the system. The second extension, a new communication framework, allows for a smooth transition from simulation to a real control system, by hiding all low-level communication details from the control software. This separates the concerns of the control software from distribution and inter-node communication issues, creating freedom in process allocation.**

I. Introduction

THE embedded systems area shows a trend of continuous demands on additional features and performance. For modern systems, producing correct results is no longer sufficient; a growing number of non-functional requirements like reliability, maintainability, security and power consumption, i.e. dependability, must also be met. Consequently, the design and implementation of embedded systems is growing more and more complex. New tools and design methodologies are needed to extend the designer's capabilities to deal with this complexity. The world outside an embedded system is not only complex, but also concurrent. In our lab an approach is developed [1], [2] that structures this complexity and concurrency, by introducing a fine-grained software parallelism. The basis of this approach is derived from CSP process algebra [3].

In our lab, CT (*Communicating Threads*) libraries were created that implement CSP-like constructs in common programming languages: C, C++ and Java [1], [4]. As the CSP and occam languages, our libraries are based on a message-passing process architecture, where concurrent processes communicate exclusively via rendezvous channels.

The target application area of this research is embedded distributed control systems interconnected via fieldbuses. In these systems, the fieldbuses (networks) interconnecting the system parts have a significant influence on system performance. Issues like (varying) communication delays and unreliable communication links become significant. Hence, besides control theory and techniques, communication theory and techniques are necessary to reason about the behavior of the total (control) system. Naturally, one wishes to determine the way in which fieldbuses and their parameters might influence the behavior of a complete networked control system prior to actual realization of the system. Therefore, a way to simulate the behavior of distributed control systems is needed.

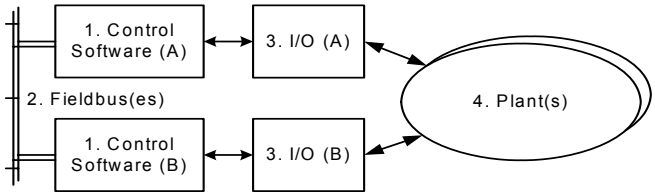

Fig. 1. Components of a networked embedded control system

Besides control software, a control system also contains one or more physical systems, the plant. In our design flow [5], plants are modeled in 20-sim [6]. 20-sim is also used to verify the models by simulation and to design appropriate control laws, which form the basis of the (distributed) control software.

In this paper, we present the design of a simulation framework [7] that features co-simulation of generated distributed control software, network(s) and plant model(s), allowing for design space exploration of e.g. fieldbus parameters. The availability of competent plant models and control laws is assumed. Furthermore, a Remote Communication framework for the CT library has been designed, which eases the use of communication links in distributed control systems. The framework is suitable for usage both in simulations and in real systems, without any differences in the control software.

Section 2 discusses the approach to this work. The simulation framework is covered in sections 3, the communication framework in section 4. Section 5 contains a test case that illustrates some possibilities of our simulation and remote communication frameworks.

II. Approach

A number of simulators and related tools for real-time and control system co-design is already available in research communities. An overview is given in [8]. This project is especially interested in simulation of distributed control sys-

Manuscript received February 10, 2006; revised May 31, 2006. This research is supported by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

The authors are with the University of Twente, Control Engineering, faculty EE-Math-CS, The Netherlands (e-mail: m.h.tenberge@ alumnus.utwente.nl, borlic@ gmail.com, and corresponding author: j.f.broenink@ utwente.nl).

tems, with a focus on the influence of the networks. Therefore, especially appealing was to investigate whether existing network simulators can be reused. Several research projects such as 'ns2', a network simulator from the VINT project [9], 'Real', a network simulator from Cornell University [10] and TrueTime [11] were considered. Designing a new network simulator was also considered a possibility, because of the relative simplicity of the computation methods used.

A decision was made to design a new simulation framework, in which multiple types of simulation engines can be interconnected to perform co-simulation. The motivation for a new design is that it can provide a smooth transition from simulation to real applications. For the network simulator component, the already existing network simulator part of the TrueTime simulator was reused.

As shown in Fig. 1, a simulation of a networked embedded control system consists of four different parts (numbers refer to the elements in the figure):

1. The control software, which is specific to the application under study. A control law implementation in the form of source code containing difference equations is assumed to be available.
2. The fieldbus, with hardware-specific properties.
3. The I/O hardware and drivers, also specific to the chosen hardware.
4. The plant(s), specific to the application under study. A model of the plant is assumed to be available in the form of differential equations.

The I/O is often neglected in simulations and is therefore not taken into account. The control software and plant components can be simulated with existing simulators. However, the newly designed simulation framework allows us to insert these simulator components as CSP/CT processes. From the control software and plant models, designed in 20-sim, CT processes containing the appropriate simulator code can be generated. The template-based code generation tool of 20-sim is flexible enough to provide this coupling.

The simulation framework is implemented as an extension of the C++ version of our CT library. This choice was made for several reasons:

1) CT forms a hardware abstraction layer

The CT library provides a uniform interface to the I/O hardware on all supported platforms. Differences in the actual hardware are handled by CT. Therefore, the transition from simulation to the real setup does not require any changes to the control software.

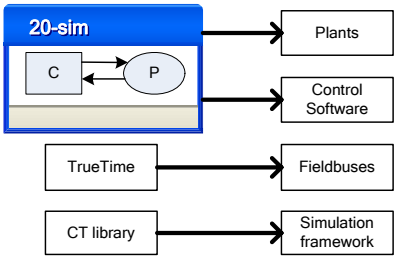

Fig. 2. Origin of the simulation components

2) CT is distributable

The CT library is process-oriented and processes are only allowed to communicate via channels. The implementation of these channels is provided by the CT library and is invisible to the control software. Processes can be located on any processor node; the choice for an implementation of the channels will consequently depend on the location of both channel ends: on the same or on different processor nodes.

III. Distributed Simulation Framework

A. Basic Principles

Simulations of a distributed system are executed on a virtual target architecture, which mimics the behavior of a complete distributed system on a single development PC. In order to have maximal correspondence between target execution and simulation, the control software should stay the same as when executed on the real target. As mentioned in the previous section, the hardware abstraction layer in the CT library takes care of this issue.

Since focus is put on the influence of networks on distributed control systems, the decision was made to first assume that the major overhead in a networked control system is induced by network delays and that computation overhead is negligible compared to any network delays. Due to a significant difference in speed between processing and communication devices, this assumption is a valid first approximation for most real life systems.

The basic idea is that all process-like components of the distributed system are implemented as CT Processes, structured in a concurrency hierarchy (thus 1, 2 and 4 as enumerated in section II). The I/O blocks are not implemented as Processes, their functionality is hidden inside the CT library. As most components are active in parallel in the real setup, the related processes will also be executed in parallel in the simulation. This organization conforms to the nature of the CT libraries and its way of structuring concurrency.

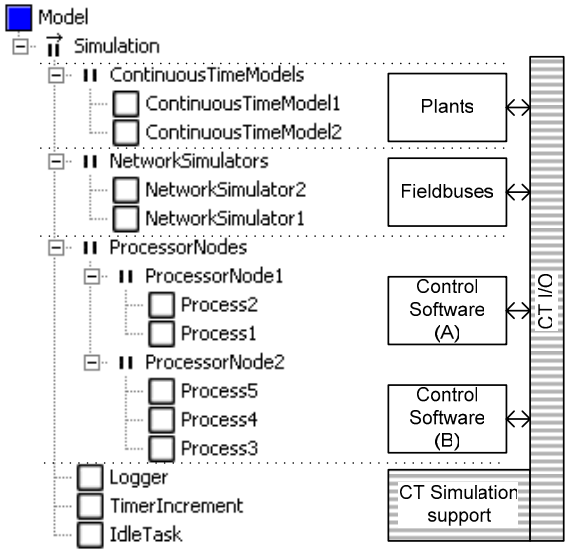

Fig. 3. Simulation hierarchy

B. Simulation system overview

Fig. 3 shows an overview of a complete simulation system. The overall system is represented by the highest-level layer (*Simulation*). This layer defines the system components.

The control software can be distributed. This is represented by grouping the software components that run on a single processor node inside a separate process (e.g. *ProcessorNode1, 2*). These processor node processes are all executed inside a Parallel construct (*ProcessorNodes*), as all processors will run in parallel.

The available fieldbus networks are also represented via a group of processes, where each process simulates the behavior of one real network. As the actual networks are concurrent and independent, the processes containing the network simulators (*NetworkSimulator1, 2*) are grouped in parallel. As already mentioned in section II, the control software is unaware of the implementation of its communication channels. In case of simulation, the channel implementation is changed, so that all communication is redirected to the network simulator processes, instead of accessing the real fieldbus hardware.

Besides the processes representing the embedded control system and the network(s), the system also contains processes that represent the physical processes (plant). Those processes contain executable models of the plant, which are recalculated at certain points in simulation time. Currently only fixed-step integration methods are used, but, in order to mimic the continuous nature of those processes, these models are also recalculated whenever other discrete components attempt to access their inputs or outputs.

The described component groups of the simulation setup are organized in a PriParallel construct, which means that the processes are executed in parallel, but ordered by priority. The continuous-time models are on top to have the highest priority, because they need to be recalculated immediately upon any access to their interface. The simulated networks should execute whenever any packet that needs to be processed is available, so the network simulators should also have a high priority.

As an example, Fig. 1 shows a typical networked control system. Such a system is used as a study case for designing software to control a simple mechanical system. Fig. 4 shows the concurrency structure of the simulation environment in the study case. In the example of Fig. 4, the plant is represented by the *LinixPlant* process (a motor driving a relatively heavy load via a flexible belt), the CAN network by the *CANNetworkSimulator* process and the two processor nodes that run in parallel are represented by *ProcessorNode* constructs. In addition, a *TimerIncrement* process that takes care of updating simulation time and an IdleTask process exist.

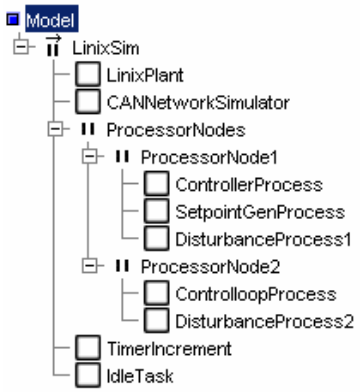

Fig. 4. Concurrency structure of a simulation

C. Advancing simulation time

The key to simulation timing is encapsulated in SimTimer objects (Fig. 5). The simulation utilizes the rendezvous-based communication methods used in the CT library to synchronize the parallel processes. A SimTimer object keeps track of the simulation time. It offers SimTimerChannels as an interface to the other processes: a read on a SimTimerChannel is non-blocking and will return the current simulation time; a write will block the writing process until the time written in the channel is reached. This timer channel mechanism was inspired by occam timer channels.

Simulation is achieved by alternately executing two phases. In the first phase, the model is allowed to progress by executing model calculations or code blocks and by performing communication events. These communication events can occur between peer processes, or between a process and some other simulation component (networks, plants or the SimTimer). Once all processes are blocked on either SimTimerChannels or communication channels, the Timer process (*TimerIncrement* in Fig. 3) is allowed to run, because of all non-blocked processes it has then the highest priority. It will advance the simulation time to the nearest time that was requested by the processes via the timer channels. The processes attached to those channels that correspond to this new simulation time will then unblock, so the simulation can continue its execution.

In addition, processes blocked on SimTimerChannels can be released prior to the requested release time. For instance, waiting continuous-time models representing the plant are released prematurely when there is a need to perform an extra model calculation. The same mechanism is also used in, for instance, the implementation of communication timeouts.

The use of SimTimer is not restricted to simulations. The same code is used in real applications. The only difference is that the current time is not updated by a Timer (Increment) process (which is absent now), but from a hardware timer ISR or a timer callback function provided by an underlying OS. The SimTimer can represent time by either double-precision floats or integers. Doubles are required by parts of the simulation framework, which makes it the only choice for the simulation framework. Both time representations are however possible in real setups. Here, integer time represen-

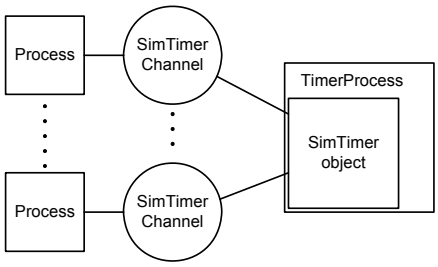

Fig. 5. SimTimer and SimTimerChannels

tation can be advantageous, for instance when all timing is synchronized to fixed-step sampling.

Finally, the *Logger* (refer to Fig. 3) transfers the results from simulation runs or from real target execution runs to an external data processing application. The Logger's priority is higher than the Timer Increment process and lower than that of the rest of the application code. This means that in simulations the Logger is executed exactly once for every time point and will run to completion every time, before the simulation continues. In a real application the Logger is the process with the lowest priority, so it will be preempted whenever some of the application processes are ready to execute.

D. Network simulator

Our simulation framework reuses the core of the network simulator from the TrueTime framework [11]. Reasons to choose for this network simulator are its public availability, the ease of integration in our CT library (a large part was already written in C++ code), its use of real data flows (as opposed to parameterized data flows) and its support for simulation on Data Link Layer level with the most common layer types already supported.

Several changes were made to original network simulator in order to improve its usability in the context of our simulation framework. For instance, network node numbering can now be random, which is needed because in our case nodes are created and associated with networks dynamically and in unknown order. A single node can also be connected to multiple network simulators (e.g. redundant networks). Furthermore, the network simulator core was converted into a CT process, for inclusion in the simulation tree.

Other parts of TrueTime were not reused. Our simulation framework provides some advantages compared to the process simulation part of TrueTime:

1. Communication and synchronization is done in a rendezvous-based way, native to our CT library.
2. The transition from simulation to the real target does not require changes to the control software. Changes in the networks and plant are hidden for the control software by the CT library (see Fig. 3).

IV. Remote Channel Framework

In CT, all communication between Processes is performed via channels. Several types of channels were already available in the CT library, which also include support for external communication (i.e. communication that is not local to a single processor). For several reasons however, a new communication framework for our CT library has been developed. As communication via the network simulators is also external communication, it can benefit from the features of this new communication framework. The main benefit is that switching the hardware type is completely transparent to the user's application. In other words, switching from simulated networks to real communication hardware does not require changes to the user application, thereby smoothening the transition from simulation to the real setup.

If two communicating Processes are allocated on the same processor node, their mutual communication is said to be *local* communication. This type of communication is handled inside the Channel object, including handshaking, synchronization and the actual data transfer.

A. Remote Channel concept

Since the original CT library [12], the hardware dependency of communication is handled via a plug-in system. A pluggable unit, a *Link Driver*, can be inserted into a channel; thereby extending the channel's functionality with read and write methods tailored to the communication hardware.

In [13] a new communication model has been introduced where the communication Link Driver is split into two parts. The *Remote Link Driver (RLD)* plugs into a channel like any other Link Driver, but instead of communicating directly with hardware, it only handles the data processing that is local to every channel instance (for instance rendezvous mechanisms and handshaking). The other driver part, the *Network Device driver (NDD)*, is associated to a specific communication hardware instance, and handles the transfer from data from the RLD to the outside world and back (which contains hardware-specific operations).

This concept forms the basis of the *Remote Channel*

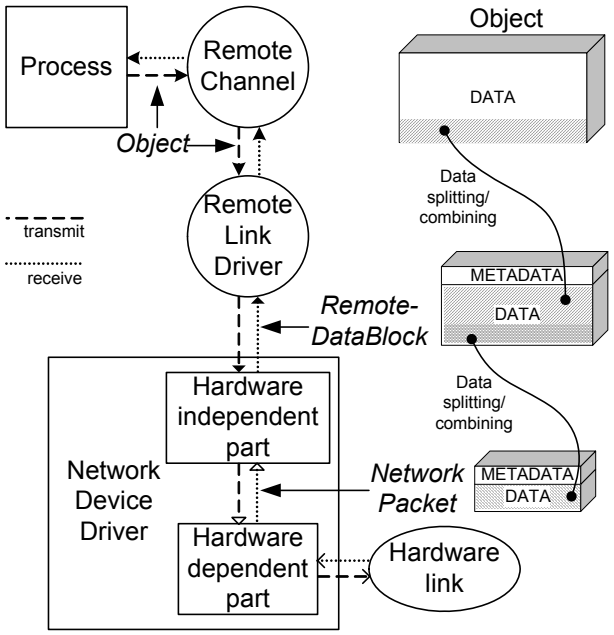

Fig. 6. Structure of a Remote Channel, and the data flow through it

framework presented here. The Remote Channel concept however takes hardware abstraction a step further, by also splitting the NDD into two parts, a hardware-dependent and a hardware-independent part (Fig. 6).

The hardware-dependent part of the NDD contains support for addressing, formatting network messages (message headers, checksums) and actual hardware access. The RLD includes support for rendezvous mechanisms, handshaking, retransmission and timeout handling. By utilizing inheritance, it is relatively easy to add support for new types of hardware; only the low-level hardware-dependent functions need to be re-implemented.

Fig. 7 shows how a Remote Channel can easily be modified for use in the simulation framework, where real networks are replaced by network simulators. This is done by replacing the hardware-dependent part of the NDD with a driver that interfaces with the appropriate network simulator.

This new architecture encapsulates support for shared access to the NDD instance in its hardware independent part (i.e. communication hardware can be shared among multiple channels, independent of the type of hardware, by connecting multiple RLDs to the same NDD).

Each channel endpoint can be uniquely addressed by the combination of a Node ID and Link ID. The Node ID must be unique across the network and is assigned to every instance of the Network Device Driver. A Link ID must be unique for every Remote Link Driver on the same node.

B. Data flow through a Remote Channel

Fig. 6 depicts how data flows through a Remote Channel. One unit of data that a process wants to transfer is called an Object. It can be of any type or size. On the interface between Remote Link Driver and Network Device Driver only one data type can be exchanged – a Remote Data Block (RDB). The RDB contains a data block and a header containing source and destination identifiers. The size of the RDB is set at compile-time. The fixed size allows any RLD to be connected to any NDD. For instance, it is possible to implement a mechanism that, in case a network link is broken, will dynamically switch to one of the alternative NDDs capable to deliver RDBs to the same destination node. This adds fault-tolerance possibilities to the communication channels.

Objects larger then one RDB will be split in multiple RDB blocks by the RLD. All Network Device Drivers must be capable of handling RDBs of arbitrary configured size. However, most network hardware imposes a maximum allowed message size, so the NDD splits one RDB into multiple Network Packets. This part is performed in the hardware-independent part of the NDD, so it does not need to be reimplemented for every hardware type. It uses only one hardware-dependent parameter, the maximum packet size.

Several transmission protocols are already implemented in the RLD, e.g. asynchronous or synchronous (rendezvous) transfers. Acknowledgments can be sent for each RDB block or only for the whole object. In the first case, when a packet is lost, only the corresponding data block needs to be retransmitted. The price is paid in somewhat larger protocol overhead, making this handshaking scheme more convenient for lossy networks and large object sizes. Object-based acknowledgments on the other hand are more convenient for smaller objects on reliable networks, as it reduces the overhead. The retransmission process can be further customized with configuration parameters as retransmit timeout and maximal retransmit count. Another parameter is priority,

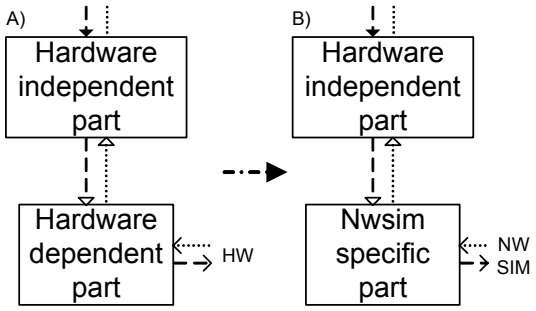

Fig. 7. Using a Remote Channel in the simulation framework

which can be specified for every channel link. If the underlying network supports it, this priority is used to indicate the priority of the network packets.

V. CASE STUDY

Throughout the development of the simulation framework, its functionality has been tested using a study case [7], based on an already available setup [14]. The plant is controlled by two processors connected by a CAN bus. The plant and the control software (without the fieldbus) are modeled and simulated in 20-sim.

Simulation code is generated from 20-sim for both control software and plant (according to Fig. 2). Using these components, a CT-based simulation is constructed and executed.

Testing of the simulation framework was performed by constructing a simulation setup where all control software is located on a single processor node. There are no networks in the loop, so the step response results of the CT-based simulation can directly be compared to the results of the 20-sim simulations (curves B and X in Fig. 8).

As a next step, the functionality of the simulation framework was tested. The control software is distributed over two processor nodes, with a simulated network in between. In this example (Fig. 4), the complete control law is placed on the first processor; the second processor only passes the calculated value to the motor steering. Both processor nodes also contain extra intercommunicating processes, in order to generate unrelated network traffic that might disturb the control system. The communication priority of this disturbance traffic is configurable, and can be higher or lower than the priority of the control loop communication.

Several simulation runs have been performed for different system parameters. In the sample results shown in Fig. 8,

three parameters were varied: the network bitrate, the presence of disturbance traffic and its relative priority. In the figure, the behavior of the control system for different parameter values (curves C-G) can be compared to its behavior when no network is present (curves B and X).

The simulation results indicate that when the network is part of the control loop, the behavior of the system changes. For the parameter values chosen in this case, it can be concluded that the plant has a relatively good step response for network speeds of 100 kBps or higher (curves C and D). In case of lower network speeds (curve E), the example control system behaves inadequate.

The presence of disturbance traffic on the network is noticeable, but has a rather limited influence when the priority of the control traffic is higher than that of the disturbance traffic (curve F). However, the behavior of the control system changes drastically when the priorities are inverted (curve G), resulting in an unacceptable step response.

VI. Conclusions and Recommendations

The presented simulation framework is based on a CSP architecture, and performs co-simulation of different simulation engines and as such organizes simulation of complete distributed systems. Using this simulation framework, facilities are available to evaluate the influence of various fieldbus parameters on the behavior of the overall system. This way the influence of the fieldbus on the total system behavior can thoroughly be checked in a rather early design phase. Design methodologies for networked embedded control systems can now be updated to incorporate the presented simulation framework [15], [16]. The framework facilitates an easy transition from simulation to reality.

The network simulator must be validated to get sophisticated prediction of the influence of networks on the behavior of the embedded control system to be designed.

A further extension is to include the execution times of the control software (code blocks) to relax the assumption on negligible processing time of the control algorithms.

Related to our research, we plan to enrich the network drivers with dynamic reconfiguration and rerouting options, creating fault tolerance against network failures [14].

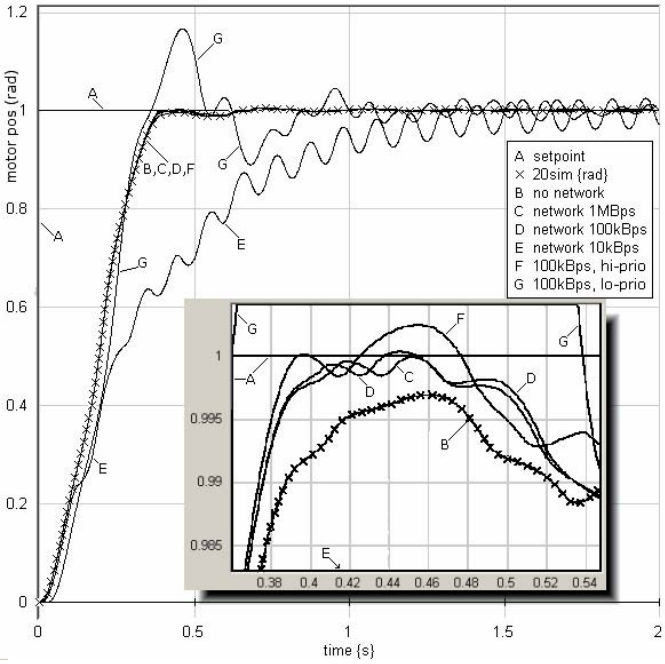

Fig. 8. Several CT simulations of the system's step response with different fieldbuses settings. Inset: zoomed portion of the curve (see scales)

References

[1] G. H. Hilderink, *Managing Complexity of Control Software through Concurrency,* PhD thesis, University of Twente, Netherlands, 2005, ISBN: 90-365-2204-8.

[2] D. S. Jovanovic, *Designing dependable process-oriented software, a CSP approach,* PhD thesis, Control Engineering, University of Twente, Enschede, NL, Enschede, 2006, ISBN: 90-365-2334-6.

[3] C. A. R. Hoare, *Communicating Sequential Processes*: Prentice Hall, 1985.

[4] B. Orlic and J. F. Broenink, Redesign of the C++ Communicating Threads Library for Embedded Control Systems, in *5th PROGRESS Symposium on Embedded Systems*, F. Karelse, Ed. Nieuwegein, NL: STW, 2004, pp. 141-156, ISBN: 90-73461-41-3.

[5] J. F. Broenink and G. H. Hilderink, A structured approach to embedded control systems implementation, in *2001 IEEE International Conference on Control Applications*, M. W. Spong, D. Repperger, and J. M. I. Zannatha, Eds. México City, México: IEEE, 2001, pp. 761-766, ISBN: 0-7803-6735-9.

[6] CLP, Controllab Products B.V. http://www.20sim.com, 2005.

[7] M. H. ten Berge, Design Space Exploration for Fieldbus-based Distributed Control Systems, Control Engineering, University of Twente, Enschede, MSc Report 029CE2005, August 2005.

[8] D. Henriksson, O. Redell, J. El-Khoury, M. Törngren, and K.-E. Årzén, Tools for Real-Time Control Systems Co-Design - A Survey, Department of Automatic Control, Lund Institute of Technology, Lund, Internal Report ISSN 0280-5316, April 2005.

[9] VINT, The Network Simulator - ns-2 http://www.isi.edu/nsnam/ns/, 2005.

[10] S. Keshav, REAL 5.0 Overview http://www.cs.cornell.edu/skeshav/real/overview.html: Cornell University, 1997.

[11] D. Henriksson and A. Cervin, TrueTime 1.13—Reference Manual, Department of Automatic Control, Lund Institute of Technology, Lund, Technical report ISRN LUTFD2/TFRT--7605--SE, October 2003 2003.

[12] G. H. Hilderink, A. W. P. Bakkers, and J. F. Broenink, A Distributed Real-Time Java System Based on CSP, in *The third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC 2000*. Newport Beach, CA: IEEE, 2000, pp. 400-407.

[13] B. Orlic and J. F. Broenink, Real-time and fault tolerance in distributed control software, in *Communicating Process Architectures 2003*, J. F. Broenink and G. H. Hilderink, Eds. Enschede, Netherlands: IOS Press, 2003, pp. 235-250, ISBN: 1 58603 381 6.

[14] H. Ferdinando, Fault Tolerance in Real-Time Distributed Systems Using the CT Library, Control Laboratory, University of Twente, Enschede, MSc Thesis 002CE2004, 2004.

[15] M. A. Groothuis and J. F. Broenink, Multi-View Methodology for the Design of Embedded Mechatronic Control Systems, in *Proc. IEEE Int'l Symposium on Computer Aided Control Systems Conference, CACSD 2006*. Munich: IEEE, 2006.

[16] P. M. Visser and J. F. Broenink, Controller System Design Trajectory, in *Proc. IEEE Int'l Symposium on Computer Aided Control Systems Conference, CACSD 2006*. Munich: IEEE, 2006.