Proc. 3rd Workshop on European Scientific and Industrial Collaboration WESIC2001, 27-29 June 2001, Enschede, Netherlands, J. v. Amerongen, B. Jonker and P. Regtien (Ed.), pp. 329-338, ISBN: 90-365-16102

BUILDING BLOCKS FOR CONTROL SYSTEM SOFTWARE

Jan F. Broenink and Gerald H. Hilderink

University of Twente, Cornelis J. Drebbel Institute for Systems Engineering and Dept. Electrical Engineering, Control Laboratory, P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: J.F. Broenink@el.utwente.nl

Abstract

Software implementation of control laws for industrial systems seem straightforward, but is not. The computer code stemming from the control laws is mostly not more than 10 to 30% of the total. A building-block approach for embedded control system development is advocated to enable a fast and efficient software design process.

We have developed the CTJ library, Communicating Threads for Java™, resulting in fundamental elements for creating building blocks to implement communication using channels. Due to the simulate-ability, our building block method is suitable for a concurrent engineering design approach. Furthermore, via a stepwise refinement process, using verification by simulation, the implementation trajectory can be done efficiently.

1 Introduction

In general, modern control systems consist of three parts: the *appliance* to be controlled, the *control software* running on the control computer(s), and the specific *interface hardware* (see Figure 1). After developing a control law using sophisticated modern design techniques, the implementation of the controller in software seems straightforward. However, the code stemming from the control law is often not more than 10 to 30% of the total code. The rest deals with command interfaces, safety measures, data integrity checks etc Thus, for the other 70 to 90% of the code, other sophisticated design routes should be followed in order to produce the final software in an efficient way. Note that in *some* cases, like the more or less standard case of (scientific) laboratory experiments, the percentage of non-control law software is less, and is also rather standard.

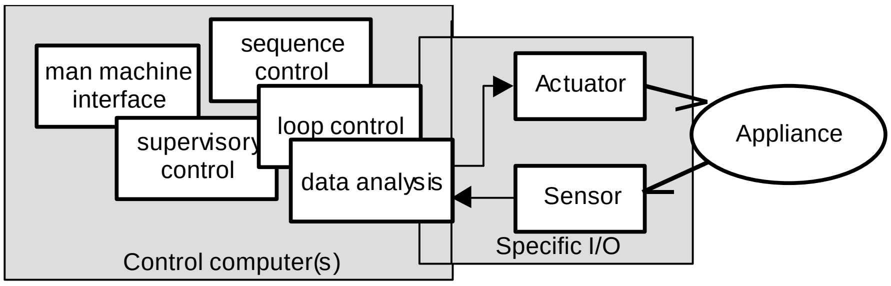

Figure 1: Architecture of modern control systems

Proc. 3rd Workshop on European Scientific and Industrial Collaboration WESIC2001, 27-29 June 2001, Enschede, Netherlands, J. v. Amerongen, B. Jonker and P. Regtien (Ed.), pp. 329-338, ISBN: 90-365-16102

Since the non-control law software part is really diverse, a building-block approach can enable an efficient and fast software design process. Due to the interaction between *all* three parts of modern control systems, a building-block approach can be really efficient, when this approach is used for all parts. Using the real plug-and-play capabilities of the blocks allows for efficient design processes. Furthermore, due the simulate-ability of the designs, concurrent engineering is efficiently supported.

In this paper, we will discuss the building-block approach for control system software, while presenting only briefly the building-block approach of the other two parts. Section two deals with embedded control systems. Section three discusses the building blocks of the three different parts. Section four deals with simulation as verification and discusses a development procedure using stepwise refinement.

2 Embedded Control Systems

At Embedded Control Systems, the dynamic behaviour of the appliance (i.e. the 'machine'-part of the embedded system) is essential for the functionality of the embedded system. The central control loop is *hard real-time*, because missing a dead line means a system failure. Examples are robots or production machines like wafer steppers.

Due to the specific character of modern control systems (also called *Embedded Control Systems*), the properties of the constituting parts are as follows:

- Software
  Principal functions are user interfacing, data processing and appliance control. Especially for appliance control, the software needs to be reliable and safe. Furthermore, its timing needs to be guaranteed. The communication latency and its jitter should be small compared to the sampling time. The triggering of the sampling should hardly contain any jitter.

- Hardware
  Here we mean both the computer hardware and the I/O interfacing. Often, specific processors are used (ASICs, DSPs, MCUs), and also sensors & actuators get integrated with the processor on one chip. There is a trend towards applying more programmable devices such as FPGAs to be more flexible during the design, but also to create possibilities for upgrading.

- Appliance
  This is the machine part of the ECS, e.g. a robot including its actuators (motors) and sensors. Its dynamic behaviour is of crucial importance for the overall behaviour of the ECS. It should therefore be taken into account, when designing the other parts of the ECS.

In fact, the ECS embodies a *closed-loop* control system, where the control loop is spread out over the embedded computer and appliance (Figure 2). The time constants of the appliance dictate the timing constraints of the software.

This specific character of ECS means, that for optimal software, the *complete* ECS needs to be considered. That is why we have applied a building block approach to *all* parts of an ECS.

Proc. 3rd Workshop on European Scientific and Industrial Collaboration WESIC2001, 27-29 June 2001, Enschede, Netherlands, J. v. Amerongen, B. Jonker and P. Regtien (Ed.), pp. 329-338, ISBN: 90-365-16102

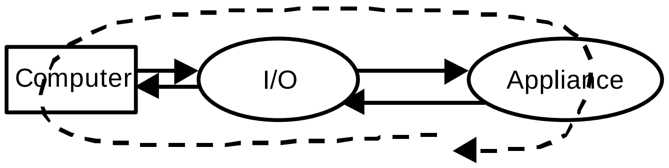

Figure 2: Control loop in an ECS

Note that at the other class of embedded systems, *Embedded Data Systems*, the behaviour of the appliance can competently be described by waiting times to take into account the actions of the appliance invoked by commands of the software. Computational latency and jitter should be small compared to the reaction time of a user. Missing deadlines decrease the quality of service, but are not fatal. This is called *soft real-time*. Examples are (cellular) phones, and other telecom systems.

3 Building blocks

Building blocks for complex systems need to comply with the following demands:

- *Overview* of the system description consisting of building blocks must be guaranteed. By allowing hierarchy, and indicating that as such, overview can be maintained.

- *Reusability* of the blocks need to be sufficiently high, to allow for competitive fast development. This requires well-designed interfaces, and the connection of blocks may not influence the description of the blocks itself.

- *Simulate-ability* of the total description, to allow for checking alternative solution proposals during design. In such a way, a real system's approach is possible, such that the system can be designed to function optimal in a global sense. Furthermore, a concurrent engineering attitude is possible.

In order to accomplish these demands, we use an object-oriented approach for *all* three parts of the ECS. This is possible since object-orientation allows for hierarchy and encapsulation. However, the description methods specific for the three ECS parts need to support this, and it has to be possible to combine those descriptions in order to reason about the complete ECS.

The object-oriented approaches for modelling *all three* parts of embedded control systems are:

- *Compositional Programming Techniques* for the embedded software parts, using *CSP-based channels* for information exchange between processes (Hilderink et al., 2000).

- *VHDL* for the specific I/O hardware parts, which remain configurable when using FPGA's

- *Bond Graphs* (directed graphs describing both the dynamic structure and dynamic behaviour of the device) for the appliance to be controlled (Breedveld, 1985).

It appeared that these three description methods *do* support hierarchy and encapsulation. Furthermore, combination of the methods is possible. In the following subsections, we will discuss the three methods. Note that our research currently focuses on the software description part. The appliance description part has been worked on (Broenink, 1999a; Broenink, 1999b), while we just use VHDL descriptions for the computer hardware.

3.1 Software building blocks

To describe the software, we use *Data Flow Diagrams*, and draw them as directed graphs. The vertices denote the processes, and the edges denote the communication of data. Note that this communication also performs the synchronisation between the processes. Such a data flow diagram shows the structure of the software, and allows for hierarchy, i.e. different levels of nesting can be used.

In Figure 3, the *Real-Time Yourdon* syntax (Yourdon, 1989) is used to denote the data flow diagram of a safety controller of a robot, which is located between the 'normal' loop controller and the robot itself. This safety controller fits into the *data analysis* block of Figure 1. Solid lines denote the flow of data, and the dashed lines denote the flow of commands (control flow), like starting and stopping the processes.

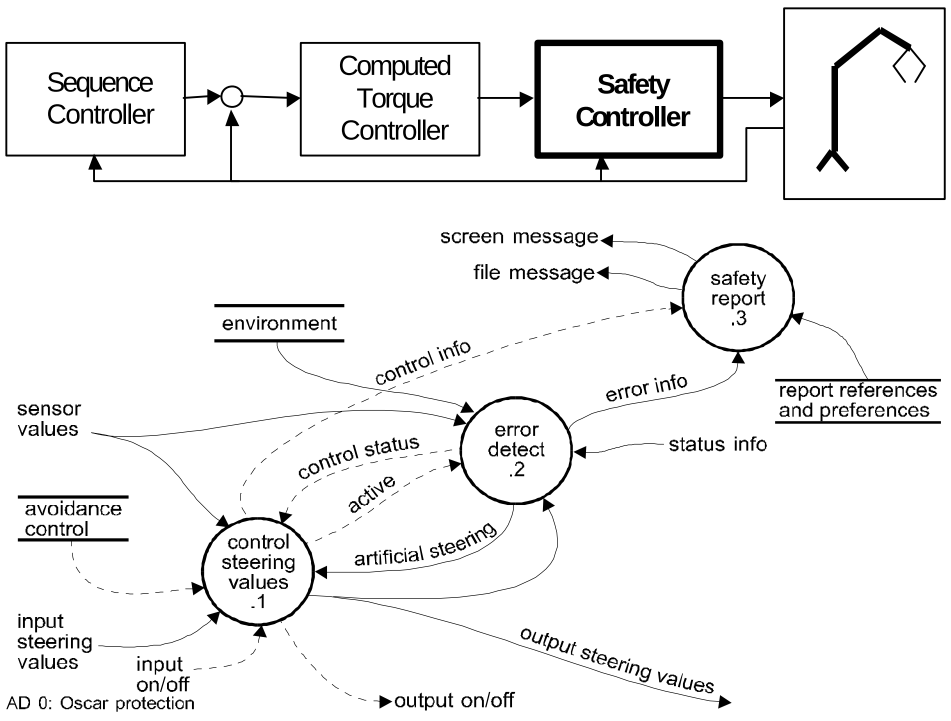

Figure 3: Block diagram of the Robot Control System (above) and the data flow diagram of the safety controller (below)

However, due to developments in software development and specification methods, the UML syntax will be used in future. Unfortunately, the *Activity Diagram* of UML is more focussed towards control flow than towards data flow, since the UML Activity Diagram is a special kind of Statechart (Douglass, 1998). Now, *solid* lines represent the *control* flow, and dashed lines the data flow. The state boxes with an arrow-like side are logical states where input or output of data is involved. The indicators for the data that is transported are ‘normal’ boxes.

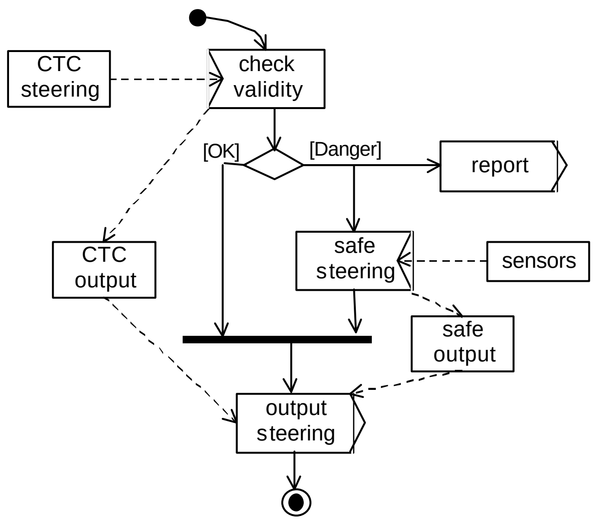

Figure 4 UML activity diagram of the safety controller

For the data communication, we exclusively use *channels*. Channels are simply synchronisation primitives that provide communication between concurrent or distributive processes. Channels control synchronisation and scheduling of processes. Channels are one-way, fully synchronised and basically unbuffered. However, buffers may be added to make the communication asynchronous.

Using channels encapsulates thread programming. Furthermore, priorities need *not* be specified anymore, since the channel also handles this. Moreover, scheduling is no longer a part of the operating system but is hidden in the channels, and thus has become part of the application instead (Hilderink et al., 2000).

A *process* is a group of tasks, and need not necessarily be sequential. Processes may run in parallel, in some sequence or by some choice. CSP specifies fundamental control-flow constructs that describe the sequence of executing processes: PAR, SEQ, or ALT. The SEQ, *sequential*, resp. PAR, *parallel*, constructs mean that processes listed hereunder are executed sequentially resp. in parallel. At the ALT, *alternative*, construct, each process is preceded by a so-called guard determining whether the guarded process will be executed. The ALT construct is a kind of *case* statement.

Since the processes and their communication via channels can be specified in the formal process algebra CSP, reasoning about correctness can be done. So, analysing the CSP description of the software part of an ECS allows for formal checking on deadlock, starvation and life-lock. This gives opportunities to verify the software before it is tested on the real appliance.

We have developed the CTJ library (Communicating Threads for Java™ (Hilderink et al., 1999)) delivering fundamental elements for creating building blocks to implement a communication framework using channels. Besides the prototype in Java, which serves as a *design pattern*, implementations in C++ and C were developed. At this moment, thorough tests on real applications need to be done. The concepts of this CTJ library are discussed next.

Proc. 3rd Workshop on European Scientific and Industrial Collaboration WESIC2001, 27-29 June 2001, Enschede, Netherlands, J. v. Amerongen, B. Jonker and P. Regtien (Ed.), pp. 329-338, ISBN: 90-365-16102

CTJ channel concept

Processes may *only* communicate via channels, using *read* and *write* methods, see Figure 5. When both processes are ready to communicate, a communication event occurs; otherwise one of the processes waits. This synchronisation principle is called *waiting rendezvous*. Synchronisation, scheduling and the actual data transfer (i.e. copying the whole data object) are encapsulated in the channel. Thus, the programmer is freed from complicated synchronisation and scheduling constructs.

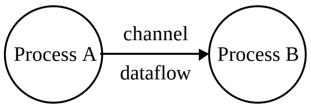

Figure 5: Data flow at channel communication.

Since the channel is an object itself, it is shown as a bubble in the implementation diagrams, see Figure 6 and Figure 7. In order to separate the hardware-dependent details of the communication, a *device-driver* framework for communication channels has been developed. These device drivers, so-called *link drivers*, are hardware-dependent objects that can be plugged into the channel. When the channel is between processes on the *same* processor, the link drivers are *memory link drivers*, implementing besides the waiting rendezvous, also buffering, up– or downsampling, overwriting etc, see Figure 6. When a channel communication occurs between processes on different processors, channel and link-driver objects are present on both processors: the link drivers implement the specific communication protocol used, like CAN, TCP, PVM, PCI, USB, RS232 etc, see Figure 7

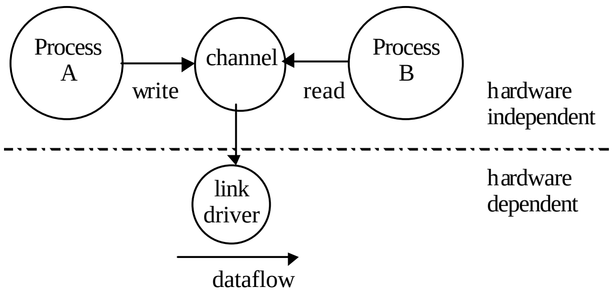

Figure 6 Channel implementation on a single-processor system

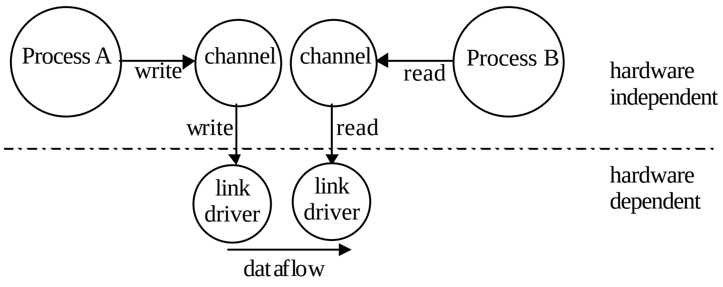

Figure 7 Channel implementation for multiprocessor systems

Proc. 3rd Workshop on European Scientific and Industrial Collaboration WESIC2001, 27-29 June 2001, Enschede, Netherlands, J. v. Amerongen, B. Jonker and P. Regtien (Ed.), pp. 329-338, ISBN: 90-365-16102

3.2 Hardware building blocks

We just use VHDL descriptions of the computer hardware. Either realisation can be done in specific circuits (ASIC) or, to be more flexible, using FPGA chips (Field Programmable Gate Arrays). The development of these hardware components is like software: updates can easily be made. Especially in the design phase, this is a real advantage. Furthermore, it is the solution when the specific chips are not available on the market anymore. However, the performance of FPGA chips needs to comply with the demands.

3.3 Appliance building blocks

For modelling the machine-part of the embedded system, i.e. the appliance, we use *Bond Graphs* (Breedveld, 1985; Broenink, 1990; Karnopp et al., 1990). Bond Graphs are directed graphs, showing the relevant dynamic behaviour. Vertices are the submodels and the edges denote the ideal exchange of energy.

Bond Graphs are *physical-domain independent*, due to the fact that *physical* concepts are analogous for the different physical domains. Thus, mechanical, electrical, hydraulic, etc system parts are all modelled with the same graphs. Six different elementary concepts exist, which each have one or two basic building blocks: storage of energy (C, I), dissipation (R), transduction to other domains (TF, GY), distribution (0, 1), transport (edge of bond graph), input or output of energy (Se, Sf).

Since the amount of basic physical concepts is limited, the number of basic elementary bond graph models is limited too. Furthermore, more complex building blocks have been constructed, mostly for practical reasons of reuse. Setting up a competent taxonomy of those more complex building blocks, is really laborious and troublesome piece of work (Breunese et al., 1998). However, most software packages supporting bond graphs, have model libraries available. Examples are Enport, CAMP and 20-SIM.

Another starting point is that it is possible to write models as *directed graphs*: parts are interconnected by bonds, along which exchange of energy occurs. A bond represents the energy flow between the two connected submodels. This energy flow can be described as the product of two variables (effort and flow), letting a bond be conceived as a *bilateral signal* connection. During modelling, the first interpretation is used, while during analysis and equations generation the second interpretation is used.

Encapsulation is granted because:

- The interfaces of bond-graph submodels consist of so-called *ports*, consisting of two variables, whose product is the power exchanged through the port. For each physical domain, such a pair can be specified, for example voltage and current, force and velocity.

- The submodel equations are specified as real equalities, and not as assignments.

Differential equations are generated after model processing, where the port variables obtain a computational direction (one as input, the other as output) and the equations are rewritten to assignment statements.

Simulation of bond-graph models to study the dynamic behaviour is in fact repeatedly executing the model statements.

Proc. 3rd Workshop on European Scientific and Industrial Collaboration WESIC2001, 27-29 June 2001, Enschede, Netherlands, J. v. Amerongen, B. Jonker and P. Regtien (Ed.), pp. 329-338, ISBN: 90-365-16102

4 Simulation and ECS-Implementation

Since the ECS descriptions presented here, are simulate-able, we use simulation to verify our designs. Furthermore, we advocate a stepwise refinement from specification to (software) implementation (see Figure 8). This way, checking design alternatives can be done efficiently (Broenink et al., 1998).

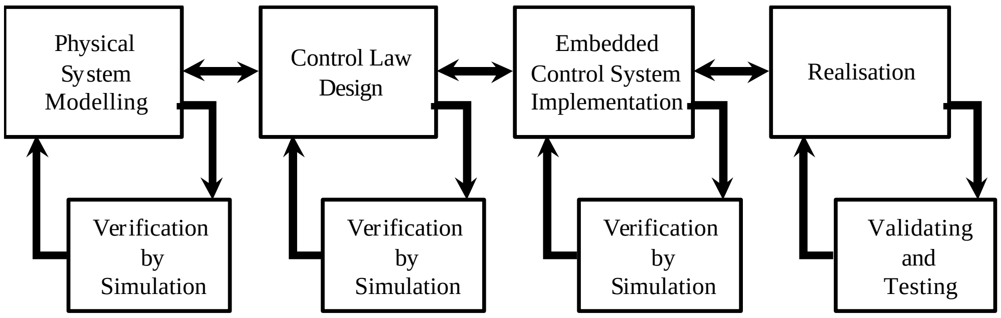
Physical System Modelling
Control Law Design
Embedded Control System Implementation
Realisation
Verification by Simulation
Verification by Simulation
Verification by Simulation
Validating and Testing

Figure 8 Design trajectory working order

Furthermore, different parts of an ECS can be developed separately, provided that the overall model is competent for testing. This implies that development design process can be organised as a concurrent engineering activity. For modern system development, this is an essential feature (Blanchard and Fabrycky, 1998).

The design trajectory of ECS is as follows:

- *Physical Systems Modelling.*
  The dynamic behaviour of the system is *object–orientedly* modelled, using bond graphs as a main modelling paradigm.

- *Control law Design.*
  Using the model acquired in the previous step or a simplified version of it, control laws are designed.

- *Embedded Control System Implementation*
  Transforming the control laws to efficient concurrent algorithms (i.e. computer code) is guided via a stepwise refinement process. After each step, the results are verified by simulation.

- *Realisation*
  The realisation of the ECS is also worked on as a stepwise sequence. Parts of the system stay as models while other parts are coded on their target hardware. Besides catching variation in development time of parts of the system, also additional verification can be done.

The stepwise refinement procedure for the embedded software consists of the following steps:

Proc. 3rd Workshop on European Scientific and Industrial Collaboration WESIC2001, 27-29 June 2001, Enschede, Netherlands, J. v. Amerongen, B. Jonker and P. Regtien (Ed.), pp. 329-338, ISBN: 90-365-16102

- *Control laws only*
  The implementation is assumed to be ideal: sensors, actuators and algorithms do *not* have any effects on the performance of the ECS.

- *Non-ideal components*
  Those components, being considered ideal in the previous step, are modelled now more precisely by considering their relevant dynamic effects.

- *Safety, and command interfacing*
  Reaction to external commands, like from the operator or from connected systems is specified. In addition, safety measures are accounted for (like reaction on external events from like emergency stops and end switches, etc.). Furthermore, facilities for maintenance processing can be added here.

- *Effects due to non-idealness of computer hardware*
  The control computer hardware and software architecture are added. Effects of computational latency and accuracy can be checked. Scheduling techniques and / or algorithm optimisation techniques may be used to obtain a viable realisation.

The impact of these additions on the behaviour of the ECS can be checked by means of simulation. These steps need not be performed in the order specified here. The designer has the freedom to tackle the individual subproblems in any order. This is a major difference with the traditional design methods, which are basically waterfall like. For example, a top–down decomposition may be applied first to define the global architecture of the system, after which those control algorithms in which problems are expected may be developed. Also parts of the controller can be developed incrementally and combined to obtain the description of the total controller. In short, the designer has the option to apply the most appropriate technique to each problem.

In the realisation step, simulation can play a relevant role, especially when the design project is set up in a concurrent engineering fashion. The first available part of an ECS can be tested together with the other parts, which are still simulated models. This verification process is a form of *hardware-in-the-loop simulation.*

5 Conclusion

Embedded (control) systems can *completely* be described by object-oriented techniques, using a building-block approach: for all parts (software, hardware, and appliance) we use such techniques, namely bond graphs, VHDL and component-based software using channels. Advantages are the possibility to use a concurrent engineering approach, to use simulation as a means for verification, and to use a mechatronic or systems approach during design. The latter truly supports *flexible hardware-software co-design*, which becomes crucial in modern embedded system development.

We have developed the CTJ library – Communicating Threads for Java™, resulting in fundamental elements for creating building blocks to implement communication using channels. Besides the prototype in Java, implementations in C++ and C were developed. Firstly implementing in Java, and using it as a design pattern, turned out to be a convenient approach. At this moment, thorough tests on real applications need to be done.

Proc. 3rd Workshop on European Scientific and Industrial Collaboration WESIC2001, 27-29 June 2001, Enschede, Netherlands, J. v. Amerongen, B. Jonker and P. Regtien (Ed.), pp. 329-338, ISBN: 90-365-16102

Current research deals with the development of a design framework and a tool to efficiently apply the building block approach. We will use applications in the field of robotics and mechatronics to test the tool. Currently we are focussing on the use of heterogeneous networked embedded systems.

References

Blanchard, B.S. and Fabrycky, W.J. (1998), *Systems Engineering and Analyis,* Prentice Hall, 0-13-135047-1.

Breedveld, P.C. (1985), Multibond-graph elements in physical systems theory, *Journal of the Franklin Institute,* **319,** (1/2), 1-36.

Breunese, A.P.J., Broenink, J.F., Top, J.L. and Akkermans, J.M. (1998), Libraries of reusable models: theory and application, *Simulation,* **71,** (1), 7-22.

Broenink, J.F. (1990), *Computer-aided physical-systems modeling and simulation: a bondgraph approach,* PhD thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands.

Broenink, J.F. (1999a), 20-Sim software for hierarchical bond-graph/block-diagram models, *Simulation Practice and Theory,* **7,** 481-492.

Broenink, J.F. (1999b), Object-oriented modeling with bond graphs and Modelica, *Proc. 1999 Western Simulation Multiconference, Conf. on Bond Graph Modeling and Simulation ICBGM'99,* San Francisco, USA, (Ed.), 163-168.

Broenink, J.F., Hilderink, G.H. and Bakkers, A.W.P. (1998), Conceptual design for controller software of mechatronic systems, *Proc. Lancaster Int. Workshop on Engineering Design CACSD'98,* Lancaster, United Kingdom, (Ed.), 215-229.

Douglass, B.P. (1998), *Real-Time UML: developing efficient objects for embedded systems,* Addison Wesley Longman, 0-201-32579-9.

Hilderink, G.H., Bakkers, A.W.P. and Broenink, J.F. (2000), A distributed Real-Time Java system based on CSP, *Proc. Third IEEE Int. Symp. On Object Oriented Real-Time Distributed Computing ISORC'2000,* Newport Beach, CA, USA, (Ed.), 400-407.

Hilderink, G.H., Broenink, J.F. and Bakkers, A.W.P. (1999), Communicating threads for Java, *Proc. 22nd World Occam and Transputer User Group Technical Meeting,* Keele, UK, (Ed.), 243-261.

Karnopp, D.C., Margolis, D.L. and Rosenberg, R.C. (1990), *System dynamics, a unified approach,* J Wiley, New York, NY, 0 471 45940 2.

Yourdon, E.N. (1989), *Modern Structured Analysis,* Prentice Hall,