A structured approach to embedded control systems implementation

Jan F. Broenink, *Member, IEEE*, and Gerald H. Hilderink, *Student Member, IEEE*

Control Laboratory, Faculty EE, University of Twente, Enschede, Netherlands, e-mail: J.F.Broenink@el.utwente.nl

***Abstract* – The method presented here, aims at supporting the development of control software for embedded control systems. The method considers the implementation process as a stepwise refinement from physical system models and control laws to efficient control computer code, and that all phases are verified by simulation. Simulation is also used as verification tool during physical–system modeling and control law development. Data flow diagrams are used to describe the control software throughout the whole implementation process.**

Since we aim at heterogeneous distributed processors as target hardware, we use a link driver library based on the CSP channel concept. Communication peculiarities are encapsulated by the link drivers.

***Index terms* – embedded control systems, multiparadigm modeling, software implementation.**

I. INTRODUCTION

Present-day requirements for reliable and efficiently extendable / updateable software for embedded systems, stresses the availability of proper design software, assisting the complete design stretch. Especially, when *embedded control systems* are concerned, having the behavior of the complete system available as dynamic model in the design tool is crucial for effective design work.

We consider *Embedded Control Systems* (ECS) as a separate class of embedded systems: the dynamic behavior of the appliance (i.e. the 'machine'-part of the embedded system) is essential for the functionality of the ES (Figure 1). Furthermore, we separate the I/O interface boards from the computer, because they are often dedicated to the ECS, although not necessary specifically developed. The software part consists of a layered structure of *controllers* and the *user interface* [1]. The *loop controllers* implement the control laws and are *hard real-time*, because missing deadlines mean system failure. *Sequence controllers* implement sequences of activities based on logical actions in time, commanding the loop controllers. Supervisory controllers contain optimization algorithms or expert systems that adapt parameters of the lower controllers.

At an ECS, computational latency must be small compared to the time constants of the appliance. Examples are robots, production machines like wafer steppers, motor management and traction control of automobiles.

The other class of ES is *embedded data systems*, where the relevant behavior of the appliance can completely be described by waiting times between subsequent commands from the software. Missing deadlines decrease the quality of service, but are not fatal: it are *soft real-time* systems.

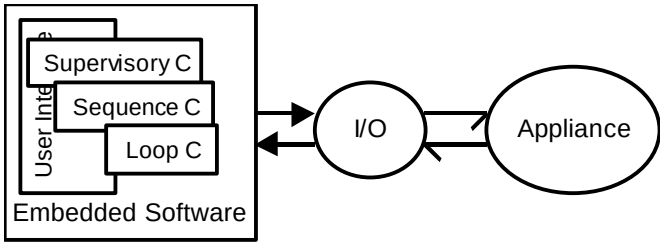

Figure 1 General architecture of embedded control systems

The embedded computer system is considered heterogeneous and distributed, because modern systems are often composed of existing subsystems, having their own control software and processors [2]. Furthermore, systems must be easily *scalable* and *adaptable*, to support ever changing functional specifications and evolution of computer hardware.

Current research deals with the development of a design framework and a tool to efficiently apply the building block approach [3]. We plan to enhance our current modeling, simulation and controller design package 20-SIM [4, 5]. We focus on applications in the field of robotics and mechatronics. Furthermore, we are focusing on the use of heterogeneous networked embedded systems.

The next section treats the different modeling formalisms of the three parts of an ECS. Section three discusses the ECS implementation approach: it is the core of this paper. In section four, a case is presented briefly.

II. USED MODELING PARADIGMS

Since we adhere a *mechatronic* or *systems* approach while designing software for embedded *control* systems, the dynamic properties of the total system, and not only the control software, plays a certain role. Thus, in order to verify the control software, also the dynamics of the appliance should be taken into account. This implies that in an executable (i.e. simulate-able) model, both the control software and the appliance need to be specified. Furthermore, relevant aspects of the computer hardware and interfaces need to be taken into account, to really forecast the behavior of the system.

To support the adaptability of systems and to allow for separate development of reusable parts, we use an object-oriented approach at all the parts. Main reasons are the real plug-and-play capabilities of the parts. Furthermore, concurrent engineering is efficiently supported.

We have applied the following model paradigms to describe all three parts of embedded control systems:

- *Communicating Sequential Processes* (CSP) for the embedded software parts. The interprocess communication is implemented by CSP-based channels [6, 7].
- *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 behavior of the device) for the appliance (i.e. the device to be controlled) [8, 9].

These model paradigms are *all* directed graphs, in which the activities occur in the vertices and the ideal (or idealized) connections are shown by the edges. It is, therefore, likely that combining them into one diagram, and also reasoning about such a combination can be done elegantly and naturally. In the following subsections, these modeling paradigms will be discussed briefly.

A. Communicating Sequential Processes (CSP)

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

CSP is the process algebra by which the DFD is described more formally. This allows for more formal checks on the manipulations done with the DFD. The interpretation of the DFD (i.e. code generation) can be checked formally [10]. Furthermore, reasoning about correctness can also be done.

For the data communication, we exclusively use *channels*. Channels control synchronization and scheduling of processes [11]. The use of channels hides threads and priority indexing from the user, thus alleviating the distributed software-writing problem significantly.

We have developed the CTJ library (Communicating Threads for Java™) delivering fundamental elements for creating building blocks to implement a communication framework using channels [12].

B. VHDL for Computer Hardware

We just use VHDL descriptions of the computer hardware. Either realization 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 need to comply with the demands.

C. Bond Graphs

For modeling the machine-part of the embedded system, i.e. the appliance, we use *Bond Graphs* [8, 9, 13]. Bond Graphs are directed graphs, showing the relevant dynamic behavior. Vertices are the submodels and the edges denote the ideal exchange of energy. They are physical-domain independent, due to analogies between these domains on the level of physics. Thus, mechanical, electrical, hydraulic, etc system parts are all modeled with the same graphs.

Entry points of submodels to connect the energy flows (edges) to, are 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. These two properties are essential and ensure true encapsulation.

Bond graphs may be mixed with block diagrams in a natural way to cover the information domain part.

Differential equations are generated after model compilation, where the port variables obtain a computational direction (one as input, the other as output) and the equations are rewritten to assignment statements. This process is rather efficient, because *computational causal analysis* on *graph* level is used. Thus the structure of the graph is exploited, since the computational direction depends on how the submodels are interconnected.

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

III. ECS IMPLEMENTATION

Since the above-mentioned model paradigms are used to denote *one* total model of the system at hand, it is obvious that they are also deployed in our design method. As indicated in section II, this total model is simulate-able, and as such allows for tool support in the verification process.

Exploiting this simulate-ability of the models enables the design work to be done as a *stepwise refinement* process. This implies that the total model will gradually change from a basic functional or conceptual model towards a detailed model from which the code for the control-computer system can straightforwardly be generated.

Since we focus on embedded control systems, the dynamics of the controlled system in total should be used as a starting point for deriving the control-computer code. Therefore, a model structured as in Figure 1 should be used. Furthermore, because of the important role simulation will play in the verification procedures, the model of the appliance should be rather sophisticated as to serve as a real imitation of the appliance.

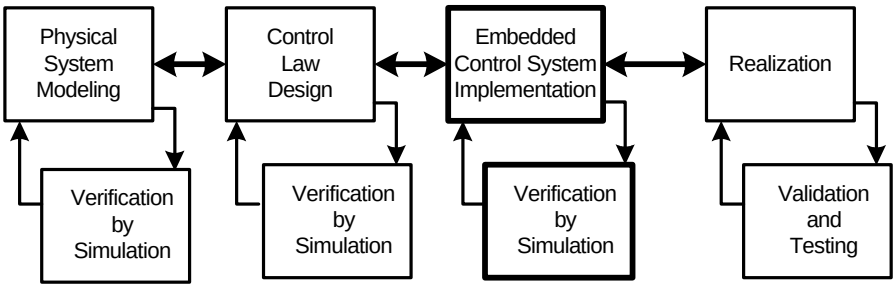

Figure 2 Design trajectory working order

Considering the above, the design trajectory of Embedded Control Systems (ECS) is as follows (Figure 2):

- *Physical Systems Modeling.*
  The dynamic behavior of the system is *object–orientedly* modeled, using bond graphs as a main modeling 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.
- *Realization*
  The realization 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.

After each step, the results are verified by simulation, also in the last phase (realization) when some parts are still a model.

Note that besides this trajectory, there is also a trajectory on the design of the appliance itself, i.e. its physical appearance. It runs in parallel to the control law design and embedded control system implementation parts, as shown in Figure 3. Both parallel design lines, and also the first modeling and realization phases use simulation as a verification instrument.

In the following subsections, the four phases of the design trajectory are discussed. Since the focus of this paper is on the embedded control system software, the *Embedded Control System Implementation* part and the *Realization* part are discussed in more detail.

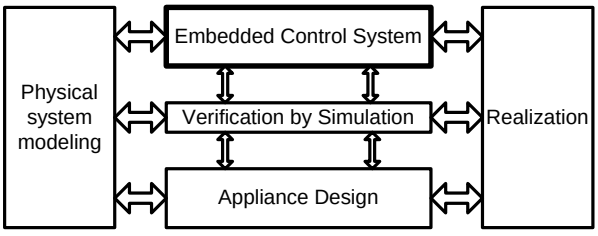

Figure 3 Embedded control system design in the large

A. *Physical systems modeling*

Obviously, the purpose is to create a *competent* model of the appliance part of the system under study. So, only relevant and dominant aspects with respect to the modeling goals need to be considered. However, *three* main modeling goals are the case here:

- Understanding the dynamics of the physical system
- Deriving control laws
- Testing the system, while the appliance model is involved instead of the appliance itself.

This clearly shows the refinement process of modeling throughout the whole design trajectory. Actually, each phase has its own model (the model in the test goal is used in the last two design phases). Feedback does not only come from verification via simulation, but also from other design phases (Figure 1).
These models are mostly derived from each other via some transformation process. Normally, the model used when deriving control laws, is a simplified, often linearized version of the model used for understanding the dynamics.

To be more concrete, we phrase the following, rather common, procedure:

1. *Generate a detailed model*
   This model is a rather detailed model of the physical system, in order to understand the dynamic behavior of that system. It can serve as a kind of physical–system *replacement*: It is later used as a substitute for the physical system when the control laws and control algorithms are tested.
2. *Verify the detailed model*
   Simulate the detailed model to check whether the model satisfies its goals. If the total model is rather complex, first parts of it may be tested separately. If possible, the model can be validated (i.e. compared with measurements on the real system), to check the correctness, and also whether the assumptions are applicable in the particular situation.

B. *Control law design*

Using standard procedures, a control law is now designed, using the model of the previous phase, or a simplified version of it, as starting point. It is also possible to derive a *set* of control laws, each having its own operating conditions. This can make each individual control law more

simple or give a better performance. Additionally, switching from one control law to the other must be designed, and should the system behave smoothly when switching from one control law to another (so-called bumpless transfer is necessary).

We suggest the following, rather common procedure:

1. *Derive a simplified model*
   Starting from the detailed model, either reduce it automatically (e.g. by linearization and/or order reduction) or diminish it by hand to obtain a model suitable for control law design. It may be necessary that more than one simplified model is needed to cover the whole workspace of the control system.
2. *Verify the simplified model(s)*
   Verify the simplified model by performing the same simulations as with the detailed model. Results should not differ significantly.
3. *Derive the control law(s)*
   Derive control laws, using the simplified model(s) acquired in the previous steps. This is common practice in controller design. Currently, external software such as Matlab is used.
4. *Verify the control law(s)*
   Construct a test bed in which the control law is connected to the detailed model. Verify the control laws by performing simulations. Run such experiments that the demands on the controller performance can be checked.

Arriving at this stage, the control law(s) together with the detailed model can be used in the process of embedded system implementation.

C. *Embedded control system implementation*

After the control law(s) has been designed, and verified by simulation, it needs to be implemented on the embedded computer.

Classically, this implementation work is one single step in the design process, thus resulting in a gap between the control law design and the final product. This makes the implementation phase error-prone, and often resulting in wrong behavior of the system at hand. But even worse, there is a break point in the way of working: the results of the previous design phases cannot smoothly be used in the implementation phase.

Therefore, we advocate a stepwise refinement procedure, in order to gradually enhance the control laws towards a description from which computer code can be generated, such that it can be run directly on the chosen target computer.

The starting point of this phase is that the control laws have been tested using the detailed model, assuming ideal devices for implementation: sensors, actuators and algorithms do *not* have any effects on the performance of the ECS.

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

1. *Integrate control laws*
   Combine the control law(s) with the sequence and supervisory control layers.
   Reaction to external commands, like from the operator or from connected systems is taken into account.
   Design and test the bumpless transfer when switching from one control law to another.
   Design and test protocols on machine level (e.g. homing to ensure proper repeatability).
   The implementation is still assumed to be ideal.
2. *Capture non-ideal components*
   Those components, being considered ideal in the previous step, are now modeled more precisely by augmenting the specification with their relevant dynamic effects (i.e. adding non-idealness of components).
   Also, add algorithms to process signals to obtain other signals which could not be measured directly in the practical situation (e.g. add an estimator to derive an internal variable, for which no sensor will be available).
3. *Incorporate safety, error and maintenance facilities*
   Facilities for safety of the system are specified and designed (like reaction on external events from emergency stops and end switches, etc.).
   Safety and error handling can be centralized in one module or distributed among the components.
   A centralized module enables easier assessment of the safety measures, as is proposed in the Safety Kernel Design Pattern [14]. Safety handling distributed among the components allow for reusable components, which are safe.
   Furthermore, facilities for maintenance processing can be added here.
   The impact of these additions on the behavior of the ECS can be checked by means of simulation.
4. *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 optimization techniques may be used to obtain a viable realization.

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.

By stimulating an iterative approach, which is a quite natural way of working, tool support becomes inevitable. This motivates our research on the design framework and tool development. Note that iterative ways of development is also performed in the separate areas of software

development for embedded systems [14] and controller design.

D. Realization

After the control algorithms have been derived as a result of the refinement procedure of the previous step, one can work towards realization on the target computer and appliance.

To make a stepwise approach possible, the *real* embedded control system is divided into three parts, as indicated in Figure 1.

- *The embedded computer and its software*
  This consists of *all* computer functionality, namely the software developed in the previous step and the mostly standard computer hardware on which this software runs.
- *I/O interfacing*
  These are the interface boards to connect the appliance to the computer system. Specific operating system resources, like device drivers running on these boards belong also to this part. The application-specific software running on these boards, implementing a part of the control algorithms is allotted to the software part.
- *The appliance itself*
  This is the physical device to be controlled: actuators including power amplifiers, the appliance itself and the sensors with their specific signal processing hardware allocated on or close to the sensor.

In the procedure of transforming the results (i.e. models) of the implementation phase to the *real* embedded system, any of the three parts can be realized first, leaving the other two as simulated models. This verification process is a form of *hardware-in-the-loop simulation*. Which one to chose first, depends on the specific situation of the project: Is the target hardware available? Is the appliance or a prototype of it ready? Etc.

A next step is to have *two* parts as real system parts and only one simulated. It depends on the progress of this realization phase whether this intermediate step is useful. It must contribute to the realization phase either as an efficiency enhancement or as an extra check to be sure that the complete system will work right the first time.

In this protocol, simulation again plays a relevant role, especially when the design project is set up in a concurrent engineering fashion. Doing so, the different trajectories of developing subsystems are decoupled (cf. Figure 3). The first available part of an ECS can be tested together with the other parts, which are still simulated models.

Some examples to illustrate different hardware-in-the-loop simulation situations:

- *Appliance simulated*
  In a concurrent design setting, the device might not be ready when the embedded software is available for testing. Using the model of the device instead of the device itself allows the control computer hardware and software to be tested before the device is available.
- *Computer code simulated*
  In a situation where the embedded computer will be a dedicated processor, or even an ASIC, the embedded software needs to be optimal. Testing the embedded software running on a 'normal' computer coupled to the real I/O and the real appliance can be a step in fine–tuning the embedded software before it actually gets 'burned' into the embedded computer. Often, optimization of the code towards minimal size is necessary to be able to use the smallest processor possible.

Note that the test set up as used in the first example can also be used in a training situation.

IV. Case Study

This example illustrates the stepwise refinement process, while performing the embedded control system implementation.

The robot is an industrial robot of SCARA type, controlled by a digital controller, often used for pick and place tasks [15]. The I/O is embodied by a standard board. The robot has two vertical revolute joints and one vertical translational joint. It is driven by three servomotors. In this case, a basic single-axis control scheme is used to control a point-to-point motion, whereby the steering voltages are limited to resemble the real situation. Figure 4 shows the structure of the robot and its control. The content of the I/O block is shown in Figure 5. Note that the figures are direct copies of the models in 20-SIM.

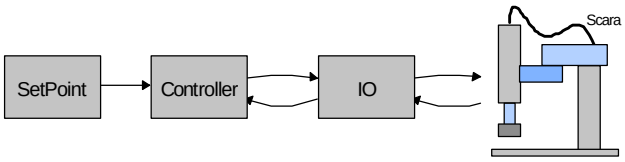

Figure 4 Structure of the Robot case

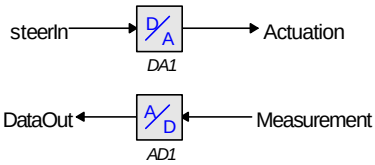

Figure 5 Contents of the I/O block of Figure 4

In the refinement process of embedded control system implementation, the A/D and D/A converters can be specialized as follows:

1. *Essential behavior only*
   Only the time discretization resp. reconstruction process is taken into account. The converters behave like ideal elements.
2. *Add functional behavior*
   The quantization (real -> integer) is added, thus the accuracy of the converters. Note that a high accuracy often means a high price.
   Quantization effects can influence the behavior of the control loop.

3. *Add nonlinearities*
   The windowing and nonlinear conversion effects are added. Normally, the A/D converter will not clip signals since the sensors deliver a limited signal. The control algorithm has to take care of not sending too large signals to the D/A converter.
   Nonlinear conversion effects are generally rather small, thus these additions are only useful when detailed simulations are needed.
   Unexpected clipping can severely influence the behavior of the control loop, so it must be mastered.
4. *Add conversion times*
   Especially, conversion times of A/D converters can be considerable, and might consume a too large part of the time budget.

Specialization step 1 is the starting point during control law design (phase 2), although step 2 is often also taken into account at control law design. Specialization step 2 and 3 belong to step 2 of the stepwise refinement procedure. Specialization step 4 clearly is part of the last step of the stepwise refinement procedure.

Clearly, when the type of converters are known, the complete model (at least with quantization and windowing) can be used in the control law design phase, in order to be as precise as possible.

V. 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.

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

References

[1] S. Bennett, *Real-Time Computer Control: An Introduction*. London, UK: Prentice-Hall, 1988.

[2] H. Kopetz, *Real-Time Systems: design principles for distributed embedded systems*: Kluwer Academic Publishers, 1997.

[3] J. F. Broenink and G. H. Hilderink, “Building blocks for control system software,” presented at Proc. 3rd Workshop on European Scientific and Industrial Collaboration WESIC2001, Enschede, Netherlands, 2001.

[4] J. F. Broenink, “Computer-aided physical-systems modeling and simulation: a bondgraph approach,” in *Faculty of Electrical Engineering*. Enschede, Netherlands: University of Twente, 1990.

[5] J. v. Amerongen, “Modelling, Simulation and Controller Design for Mechatronic Systems with 20-Sim 3.0,” presented at 1st IFAC conference on Mechatronic Systems, Darmstadt, Germany, 2000.

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

[7] A. W. Roscoe, *The Theory and Practice of Concurrency*: Prentice Hall, 1997.

[8] D. C. Karnopp, D. L. Margolis, and R. C. Rosenberg, *System dynamics, a unified approach*, 2nd ed. New York, NY: J Wiley, 1990.

[9] P. C. Breedveld, “Multibond-graph elements in physical systems theory,” *Journal of the Franklin Institute*, vol. 319, pp. 1-36, 1985.

[10] H. J. Volkerink, G. H. Hilderink, J. F. Broenink, W. A. Vervoort, and A. W. P. Bakkers, “CSP Design Model and Tool Support,” presented at Communicating Process Architectures 2000, WoTUG-23, Canterbury, United Kingdom , 2000.

[11] G. H. Hilderink, A. W. P. Bakkers, and J. F. Broenink, “A distributed Real-Time Java system based on CSP,” presented at Proc. Third IEEE Int. Symp. On Object Oriented Real-Time Distributed Computing ISORC'2000, Newport Beach, CA, USA, 2000.

[12] G. H. Hilderink, J. F. Broenink, and A. W. P. Bakkers, “Communicating threads for Java,” presented at Proc. 22nd World Occam and Transputer User Group Technical Meeting, Keele, UK, 1999.

[13] J. F. Broenink, “Object-oriented modeling with bond graphs and Modelica,” presented at Proc. 1999 Western Simulation Multiconference, Conf. on Bond Graph Modeling and Simulation ICBGM'99, San Francisco, USA, 1999.

[14] B. P. Douglass, *Real-Time UML: developing efficient objects for embedded systems*: Addison Wesley Longman, 1998.

[15] H. Ecker, “Comparison 11: Scara Robot - definition; solution in ACSL,” *Eurosim - Simulation News Europe*, pp. 30-33, 1998.