A DESIGN ENVIRONMENT FOR DEVELOPING AND TESTING CONCURRENT SOFTWARE FOR EMBEDDED CONTROL SYSTEMS *)

Dusko S. Jovanovic, Gerald H. Hilderink and Jan F. Broenink

Electrical Engineering, Control Engineering, Cornelis J. Drebbel Institute for Mechatronics,
Twente Embedded Systems Initiative, University of Twente
P.O.Box 217, 7500 AE, Enschede, the Netherlands

Abstract – *In the context of a trajectory for analysis and design of embedded control systems (ECS), the main focus is put on an approach to concurrent programming in the light of process orientation, in a way which is transparent for the designer – typically a system engineer with a background in control engineering.*

Due to the nature of real-time applications of ECS's, developers resort to concurrent implementations by the means of multithreaded programming, which leads to unavoidable complexity. The approach presented here relies on a paradigm of compositional programming – in essence, an object-oriented philosophy based on properties of encapsulated, reusable building-blocks applicable not only for software engineering, but as well as for modeling controlled plant and hardware design, actually supporting hardware-software co-design. The building-blocks approach is believed to be capable to manage complexity inherent to ECS's.

1. INTRODUCTION

To clarify the strengths of the environment and the approach to analysis, design, testing and implementation of real-time concurrent embedded software, first an overview of the problem context is given.

Here, under ECS, so-called hard real-time control systems are discussed, where missing reaction deadlines means a system failure.

At the Control Laboratory of the University of Twente research is oriented towards mechatronics, an interdisciplinary engineering interplay of electrical and mechanical engineering and computer science. A mechatronics system, a highly sophisticated control system in nature, typically can be represented as a composition of the following three main parts: (concurrent) software components, (embedded) processing hardware along with specific I/O (actuating and sensoring) hardware, and the plant, whose dynamic behavior is essential for the functionality of the ECS [1], figure 1.

*) This research project is supported by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NWO, the Dutch Ministry of Economic Affairs an the Technology Foundation STW.

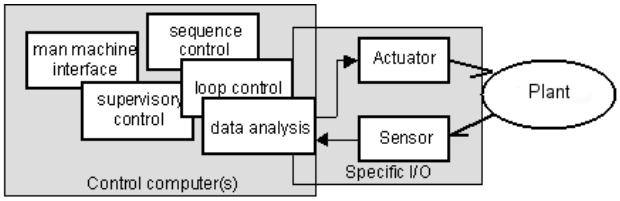

Figure 1. *Typical architecture of modern control systems*

From the years of the research efforts in the mechatronics engineering field, a design paradigm and an environment were established as a basic background for research on more automated, intuitive and user-friendly design tools in one of today's the most prominent, but at the same time the most error- and risk-prone "high-tech" engineering: embedded control systems.

First, a general overview of the established mechatronics development environment in the Control Laboratory is described in section 2; CSP process algebra and its software engineering application are shortly introduced in section 3, which together with section 4 makes the core of the paper – process orientation, CSP based Communicating Threads (CT) kernel and libraries are elaborated there; section 5 gives shortly overview of CSP/CT diagrams. In section 6 current research topics are reported.

2. THE DEVELOPMENT ENVIRONMENT

A. Stepwise refinement (SWR) paradigm

SWR paradigm strives for developing methodologies and tools to support system (in the current context: control) engineers in treatment (analysis and design) a control system as a whole, allowing them to start with a sketch of overall system, and gradually refine the model of solution in the course of understanding the problem at hand. Especially the gap between control laws design and implementing them on the targeted platform(s) is recognized as critical and not methodologically covered by existing approaches and tools. This paradigm in fact is the backbone of this research project [2].

B. ECS design trajectory

Stepwise refinement has an intention to allow the designer freedom of roaming the design space, i.e. arbitrary stepping in the (sub)phases captured by ECS design trajectory. The four main phases in engineering an embedded control system are articulated in figure 2.

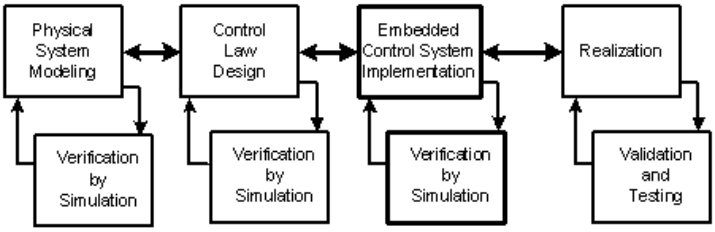

Figure 2. *Main phases in ECS design trajectory*

As it is claimed in subsection *A*, the issue is that complete ECS needs to be considered *in a consistent way*. That is why the intentions in Control Laboratory to apply a building-blocks approach to all parts of an ECS, in all phases presented in figure 2, [3].

The object-oriented approaches for modeling all three parts of an embedded control system are:

- Compositional programming techniques for the embedded software parts, using CSP-based channels for information exchange between processes [4]
- VHDL for the specific I/O hardware part, which remain configurable when using FPGA's
- Bond graphs (directed graphs describing both the dynamics structure and dynamic behaviour of the device) for the plant to be controlled [5].

C. RT Linux

As it will be explained in section 4, the discussed software implementation technique does not rely on any real-time OS resources and services. Despite, it turned out that RT Linux can offer a lot of conformity in developing/prototyping outcomes of the phases ruled by concepts stated in subsections *A* and *B*.

RT Linux is a small and efficient real-time kernel which runs Linux as the lowest priority process (task). RT Linux kernel [6] is publicly available as well as Linux, which can be found in a variety of distributions [7]. So, the apparent advantages of relying on (RT) Linux are its deliverability, low price, flexibility and customizability (due to its open-source nature).

Due to property of RT Linux kernel that lower priority task cannot compromise (preempt) execution of higher priority task, the following situation is exploited, in order to efficiently and reliably develop and test concurrent software components: just one of RT Linux tasks is used to shelter CT kernel which is multi-processed itself – while the realm of all Linux (full-featured) services, arbitrary Linux available programming environments, analysis and visualizing tools can be used for testing temporal behavior of the CT processes situated in the RT Linux highest priority process.

However, communication between "CT-hard-real-time" processes and the rest introduces some overhead, but the least expensive and quite natural to CSP-based software structures. Namely, communication is performed through RT Linux fifo queues plugged-in CSP/CT channels on the CT kernel side. That way the results of temporal behaviour analysis are made as less blurred as possible.

But, the former has also an additional substantial virtue: a developed design of a part of an engineered embedded software system can be easily targeted to the other, OS supported or bare-metal target platform, after reasonable adaptation (of channels) to a highly limited extent. More on that can be found in section 4.

3. CSP PROCESS ALGEBRA AND PROCESS ORIENTATION IN SOFTWARE ENGINEERING

CSP is a formal process algebra originated by Hoare in seventies of the last century [8]. It has been reviewed later by Roscoe and defined as "a notation for describing concurrent systems (i.e., ones where there is more than one process existing at a time) whose component processes interact with each other by communication" [9]. A power of applying CSP concepts in concurrent software engineering lies in the fact that processes and their communication via channels can be specified in the CSP and *reasoning about correctness can be done*. So, analyzing the CSP description of the software part of an ECS allows for *formal checking* on deadlock, starvation and livelock. This gives opportunities to verify the software before it is tested in the control loop.

The first practical usefulness of CSP in concurrent software developing had been provided by programming language **occam**, which had been used as implementation language for transputers [10]. Due to movements on the market of VLSI semiconductors (micro-processors/controllers, DSP's), transputers vanished – but good experiences with ease of making concurrent software components implemented in **occam** running on transputers survived.

Shortly, CSP algebra uses abstraction of channels for describing communication between processes; a process is a group of activities, and need not necessarily be sequential. Processes may run in parallel, in some sequence or by some choice. CSP specifies fundamental control-flow operators that describe the sequence of executing processes: "$\rightarrow$" or ";", "$\|$", "$\Box$". The "$\rightarrow$" or ";" – sequential and "$\|$" – parallel operators mean that processes in a expression execute sequentially or in parallel. At the "$\Box$" – alternative operator each process is preceded by a so-called guard determining whether the guarded process will be executed.

For practical use, **occam** introduced following control-flow constructs corresponding to CSP operators: SEQ for sequential, PAR for parallel and ALT for alternative thread of control. In order to deal with prioritized execution of processes, **occam** introduced also PRIPAR and PRIALT constructs (prioritized parallel and alternative, respectively). Feedback of notions of priority had implication to CSP, namely CSPP [11], where the additional "prioritized" operators "$\vec{\|}$" and "$\vec{\Box}$" can be found.

The good experiences with "CSP reasoning" of concurrent software engineering had much broader and more general impact on recent movements in software development theory, which yielded so-called *process orientation* to software engineering. Namely, a number of publications [12, 13, 14] show that object orientation suffers from significant shortcomings when applied to certain engineering fields, among others the concurrent embedded control software engineering.

Process orientation in engineering embedded software has a similar attitude to object orientation like object orientation had to structured software developing philosophy: adopt those concept which turned out to be useful and not compromising new design principles. This means that process-oriented software relies inside on object-oriented realization – but the reasoning in analysis and design starts from even higher abstraction point of view, where object are

seen as useful construct to capture the most of entities in ECS software and to implement their functionalities; however not all entities are recognized just like object, but more. Examples are processes and events.

One additional important advantage of process orientation in software engineering is distributing responsibilities over processes (as the coarser-grained software components) in earliest phases of a solution analysis and design, by contrast to classical real-time software engineering, where a concurrent design relied on a chosen real-time OS and where a division of responsibilities was deferred to implementation phases, what had led to preclusion of process-level reasoning in early phases of system developing and even worse what had been preventing easy design migration from one to another underlying real-time OS.

4. COMMUNICATION THREADS (CT) LIBRARIES

For a few years the CT libraries are available from the web site of the Control Engineering of the University of Twente, at the pages concerning JavaPP project [15, 16]. The libraries had been already used by several universities and companies.

The CT philosophy puts all embedded software responsibilities in designer-defined processes, which are supported by OS-independent, CT kernel – a substantial internal part of the CT libraries. This means that process orientation and modeling the system that way is included in the very early phases of reasoning about the problem at hand. This also means independence – thus portability – of/to any real-time OS to the extent that an OS is not even necessary any more, which indeed *is* a case in designs where small and cheap processing units are involved: DSP's, MCU's or moreover programmed FPGA's – in all (typical) cases when it is intended that the design fits into hardware resources as smaller (cheaper) as possible.

Communicating Threads (CT) libraries deliver fundamental elements for creating building-blocks to implement a communication framework using channels. Besides the prototype in Java (CTJ), which serves as a design pattern, implementations in C++ (CTCPP) and C (CTC) were developed.

For the data communications channels are used exclusively. Channels are simply synchronisation primitives that provide communication between concurrent and/or distributive processes. Channels control syncronisation and scheduling of processes. Channels are fully syncronised 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 channels according to **occam** introduced, and CT supported "PRIPAR" and "PRIALT" constructs also handle this. Thus, scheduling is no longer a part of an OS but is hidden in the channels, and thus has become part of the application instead [4].

Processes may only communicate via channels, figure 3, using read and write methods. When both processes are ready to communicate, a communication event occurs; otherwise one of the processes waits. This synchronization principle is called *waiting rendezvous*. Synchronisation, scheduling and the actual data transfer are encapsulated in the channel. Thus, the designer is freed from complicated syncronisation and scheduling constructs.

Since the channel is an object itself, it is shown as a bubble in the implementation diagrams, figures 3 and 4. 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, implementing besides the waiting rendezvous, also buffering, up- or downsampling, resources accessibility etc, figure 4. 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, PCI, USB, RS232 etc. Hence, the distributiveness of the design is also addressed, in a way that can be made rather transparent to the designer.

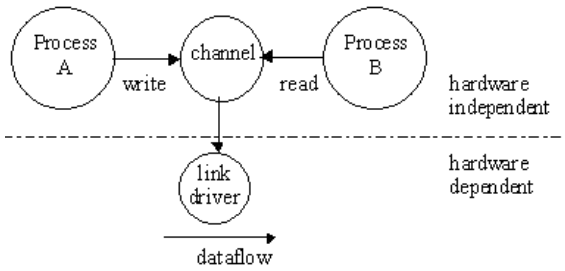

Figure 3. *Channel implementation on a single-processor system*

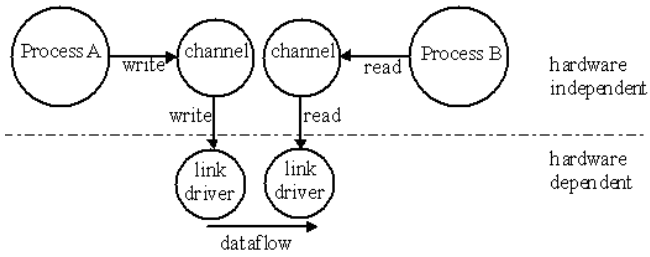

Figure 4. *Channel implementation for multiprocessor (distributed) systems*

5. CSP DIAGRAMS ON A SAFE CONTROL LOOP EXAMPLE

Modeling concurrency by channels and processes (by contrast to bubbles, rectangles on) CSP diagrams will be demonstrated on a simple one loop control problem (figure 5) on a mechanical plant called Linix (not to be confused with the Linux OS!) which was one of the first experiments of applying the CSP concepts in a design tool [17].

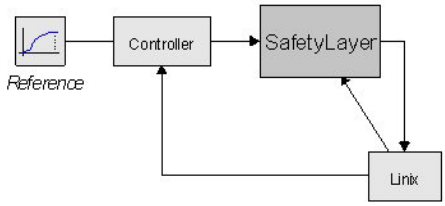

Figure 5. *A block-schema of a safe control loop*

There are two kinds of diagrams relevant to describe two views on CSP/CT architectures: *communication diagrams and composition diagrams*.

Figure 6 is an example of communication diagrams. In essence, they have much in common with data-flow diagrams known from the ages of structured ways of software engineering [18]. A communication diagram is a graph of processes and their communication relations, which are to be instantiated by message passing over channels. A communication relationship is defined as a directed relationship, which represents message flow between a sender and a receiver process.

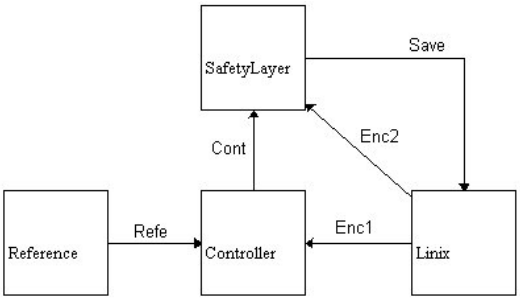

Figure 6. *Communication graph for the safe control loop*

A composition graph shows the processes with their compositional relationships. A composition graph typically shows the processes in the same topography as a corresponding communication graph. A composition graph is control-flow oriented and expresses concurrent behavior, figure 7.

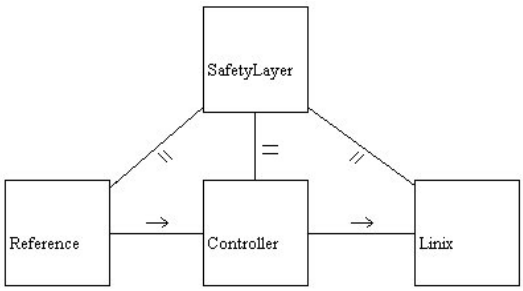

Figure 7. *Composition graph for the safe control loop*

Compositional relationships in a CSP composition diagram are based on the CSP and CSPP set of operators: $\{\rightarrow, \|, \vec{\|}, \Box, \vec{\Box}\}$.

6. CURRENT RESEARCH

At the moment, the research focuses on the following topics:

- Unifying the paradigm and design trajectory by means of integrating supervisory, sequence, loop control, safety layers and corresponding concurrent real-time software components and make the stepwise framework concepts and tools close to the practice
- Measuring and assessment of the real-time behaviour of the systems based on CT libraries and its temporal analysis
- Finalizing a multiloop control case study on a 2DOF robot (JIWY project)
- Looking for refinement of the method in the course of integration with the plant modeling tool 20-SIM, UML CASE tools (I-Logics' Rhapsody, Rational RT Rose), automatic code generation and methods for hardware-software co-design.
- "UMLization" of all underlying concepts of the method
- Extending expressive capabilities of the framework to support reasoning in terms of heterogeneous distributed *networked* embedded systems.

REFERENCES

[1] J.F. Broenink, G.H. Hilderink, *A structured approach to embedded control systems implementation,* in proc. of IEEE Computer Society Conference, 2001, Mexico

[2] http://www.rt.el.utwente.nl/djov/Swr

[3] J.F. Broenink, G.H. Hilderink, *Building blocks for control system software,* in proc. of the 3rd Workshop of European Scientific and Industrial Collaboration WESIC, 2001, The Netherlands

[4] G.H. Hilderink, A.W.P. Bakkers, J.F. Broenink, *A distributed Real-Time Java system based on CSP*, in proc. of the 3rd IEEE Int. Symp. On Object Oriented Real-Time distributed Computing ISORC, 2000, USA

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

[6] http://www.fsmlabs.com

[7] http://www.linux.com

[8] C.A.R. Hoare, *Communicating Sequential Processes,* Prentice Hall International, 1985.

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

[10] Inmos Limited, *occam 2 Reference Manual,* Englewood Cliffs, NJ: Prentice Hall International, 1988.

[11] A.E. Lawrence, *CSPP and Event Priority*, in proc. of the Communication Process Architectures conference, 2001, UK

[12] T. Locke, *Towards a Viable Alternative to OO – Extending the occam/CSP Programming Model*, in proc. of the Communication Process Architectures conference, 2001, UK

[13] C. Bryce, C. Razafimahefa, *An Approach To Safe Object Sharing*, in proc. of ADM SIGPLAN conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2000), 2000.

[14] J. Hogg, *Islands: Aliasing Protection in Object-Oriented Languages,* in proc. of OOPSLA 1991.

[15] http://www.rt.el.utwente.nl/javapp/

[16] G.H. Hilderink, J.F. Broenink, A.W.P. Bakkers, *Communicating threads for Java,* in proc. of World Occam and Transputer User Group Technical Meeting, UK, 1999.

[17] J.P.A. Hendriks, *Realization of Tool Support for CSP Diagrams and Generation of Concurrent Java Software,* M.Sc. Thesis, University of Twente, 2001.

[18] T. DeMarco, *Structured Analysis and System Specification,* Yourdon Press / Prentice Hall, 1978.