CONTROLLING A MECHATRONIC SET-UP USING REAL-TIME LINUX AND CTC++ *)

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

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

Abstract

The development of control software for mechatronic systems is presented by means of a case study: a 2 DOF mechanical rotational set-up usable as a camera-positioning device.

The control software is generated using the code generation facility of 20-SIM, thus guaranteeing the generated code being the same as verified by simulation during the controller design phase. We use CTC++ (Communicating Threads for C++) as Communication Abstraction Layer *to facilitate the conversion process from the block diagram describing the controller towards the control computer code. Furthermore, CTC++ enables the use of distributed and different processors as target computer, since distributivity and heterogeneity are explicitly accounted for in the CTC++ library. CTC++ provides the tools to make the application real-time. Real-time Linux provides us the real-time operating system resources that are needed to create a real-time environment for the real-time application.*

1 Introduction

Present-day requirements for reliable and efficiently extendable/updateable software for embedded systems, stresses the availability of proper software design tools, assisting the complete design stretch. Especially, when *embedded control systems* are concerned, having the behaviour of the complete system available as a 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 behaviour of the plant (i.e. the 'machine'-part of the embedded system to be

*) 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 an the Technology Foundation STW.

controlled) is essential for the functionality of the *Embedded System* (ES) (Figure 1). Examples are robots, production machines like wafer steppers, motor management and traction control of automobiles. 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*. 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. *Data analysis* checks the measurements from the sensors and the values sent to the actuators for their correctness, compensates nonlinearities, performs filtering etc.

At an ECS, computational latency must be small compared to the sample period, which is of course a factor smaller than the time constants of the plant. This class of ES is considered *hard real-time* since it has to meet its deadlines; otherwise the system fails.

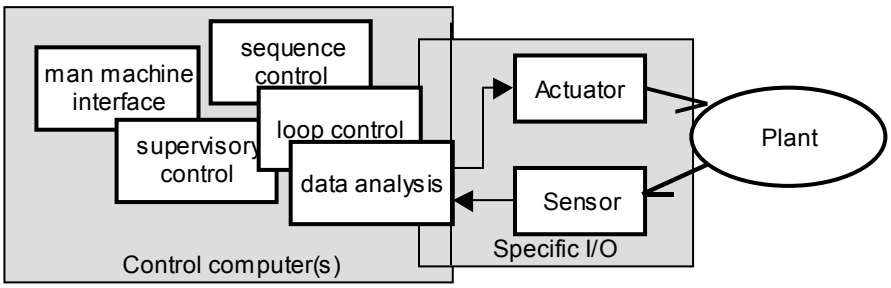

Figure 1 Typical architecture of modern control systems

The other class of ES is *embedded data systems*, where the relevant behaviour of the plant can completely be described by waiting times between subsequent commands from the software. This class of ES is usually *soft real-time* which means that missing deadlines decrease the quality of service, but are not fatal.

The embedded computer system can vary from a single processor system to a distributed and heterogeneous multi-processor system. Furthermore, systems must be easily scalable and adaptable, to support ever changing functional specifications and evolution of computer hardware.

Since we focus on embedded control systems, the dynamics of the controlled system in total along with functional as well as non-functional requirements should be used as a starting point for deriving the control-computer code. Furthermore, simulation plays an important role in the verification procedures and therefore the model of the plant should be rather sophisticated as to serve as a real imitation of the plant.

Considering the above, the design trajectory of Embedded Control Systems (ECS) is as follows (Figure 2) (Broenink *et al.*, 1998):

- *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 (SWR) process.

- *Realization*
  The realization of the ECS is also worked on as a stepwise sequence. This means that parts of the system stay as models while other parts are coded on their target hardware. This gives opportunities to catch variation in development time of parts of the system, and also to do additional verification.

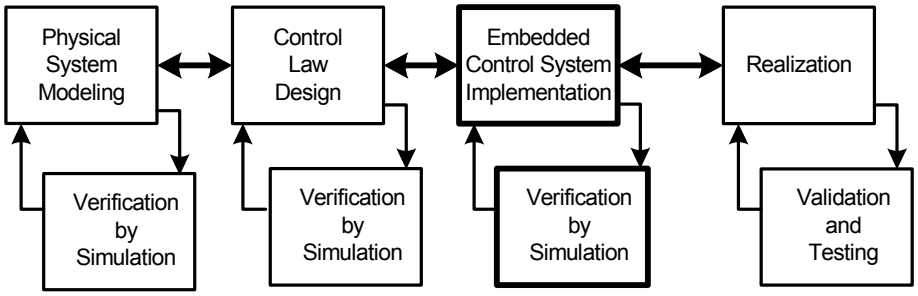

Figure 2 Design trajectory working order

During each step, the results are verified by simulation, also in the last phase (realization) when some parts are still a model. Actually, the methodology in each design step is that of *Stepwise Refinement* (SWR).

The SWR paradigm strives for developing methodologies and tools to support system engineers (here: control engineers) in designing 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.

The focus in this research is on the third step (Figure 2): Embedded Control System Implementation using Stepwise Refinement. This is discussed in more detail below.

First, an overview of the controller implementation method used is described in section 2. The CT-library, being the Communication Abstraction Layer, facilitating the conversion from the block diagram description of the control law towards the concurrent code on the control computer, is treated in section 3. The test case setup, called JIWY, is dedicated to experiment with extending the controller code with surrounding safeguarding. Special care is taken for separation of concerns in order to manage complexity. This means that safeguarding does not influence the structure of the controller part – like it is discussed in section 4.

2 ECS implementation

After the control law(s) have been designed, they need to be implemented on the embedded computer. The starting point of this phase is that the control laws have been verified by simulation using the detailed model, assuming ideal devices for implementation: sensors, actuators and algorithms do *not* have any effects on the performance of the ECS.

We advocate a SWR 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. It 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 modelled 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 (i.e. safeguarding) can be centralized in one module or distributed among the components. 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 behaviour 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 optimisation 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 (Douglass, 1998) and controller design.

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; control software, specific I/O, and the plant, as indicated in Figure 1.

3 Communicating Threads Library

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 (Hilderink *et al.*, 2000), (Hilderink, 2002) . 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 the OS-independent real-time kernel – a substantial internal part of the CT libraries. This means that process orientation and modelling the system that way are 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 small (cheap) 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. Processes may only communicate through the channels using read and write methods, as shown in Figure 4. 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 synchronisation and scheduling constructs. Channels are fully synchronised and basically unbuffered. However, buffers may be added to make the communication asynchronous.

Using channels encapsulates thread programming. Scheduling is no longer a part of an OS but is hidden by the channels, and thus has become part of the application instead (Hilderink *et al.*, 2000). Thus, the application schedules itself to guarantee real-time behaviour independent of the underlying real-time OS; it is the application that must be real-time in the first place. Since the channel is an object itself, it is shown as a bubble in the implementation diagrams, Figure 3 and Figure 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.

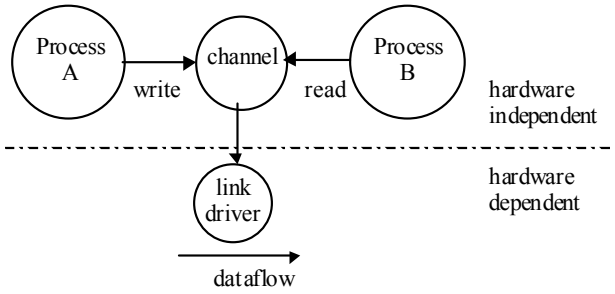

Figure 3 Channel implementation on a single-processor system

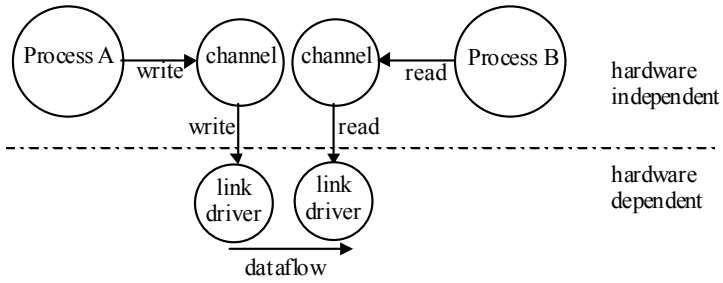

Figure 4 Channel implementation for multiprocessor (distributed) systems

4 Case Study

JIWY is a mechatronic setup for holding a camera. The construction contains two joints that allow the camera to rotate on a horizontal axis and a vertical axis. A user can control the camera setup and view the pictures from remote via a network connection (Figure 6).

The operating vertical angle is 165^{o} and the operating horizontal angle is 120^{o}. The maximum swing is limited by end stops. The maximum angles between the end stops are respectively 300^{o} and 150^{o}. These end stops prevents full swings so that the wires cannot be twisted or damaged. Each joint is equipped with one DC motor and one incremental encoder. The wires between these devices and the I/O-interface are bundled in one cable together with a watchdog signal lead. The watchdog signal is a clock signal that is used for detecting whether the cable is damaged or disconnected. Thus, each joint is separately controlled and independently connected by a separate cable.

JIWY enables a sufficient complex case study for demonstrating multi-loop servo control problems. Besides the controller, a variety of safeguarding can be built-in, for example:

1. Maximum swing is limited by physical end stops
2. Maximum swing is limited by end switches
3. Operating swing limited by angle-position monitoring
4. Motor protection by steering limiting
5. Motor protection by rotation monitoring
6. Motor protection by time-out
7. Human protection by limiting torque by steering limiting
8. Human protection by limiting torque by rotation monitoring
9. Human protection by limiting torque by time-out
10. Cable condition monitoring
11. Exception handling

The mechanical setup, the motors and the human operator are protected by three redundant safety-guards. Each safety-guard is independent developed from each other as concurrent processes. This way, safe-guarding becomes fault-tolerant; thus, if one guard fails then another pops in. Concurrency allows us to build fault-tolerance software that is reliable, robust, and that is scalable with complexity.

From a software design point of view it is interesting to see what impact each safe-guard has on the entire software. We try to eliminate anomalies as much as possible so that we end up with a clean design that is easily maintainable, extendable and reusable.

The software is designed using CSP (Communicating Sequential Processes) concepts (Hoare, 1988; Roscoe, 1997) and these concepts are implemented using our CT libraries. RT-Linux provides a real-time environment for the application. The CSP concepts allow us to design real-time behavior that is inherently part of the application – it is the application that must be real-time in the first place. The CSP paradigm provides guidelines to manage complexity in a simple and elegant way. This is far more powerful and simpler than using purely objects, sequential programming with or without multithreading.

Between the JIWY set up and a PC/RT-Linux are the amplifiers for steering the motors, electronics for the sensors, and a power supply to supply JIWY with power are placed in a separate box (Figure 5 and Figure 5). The camera is not yet placed on the top holder.

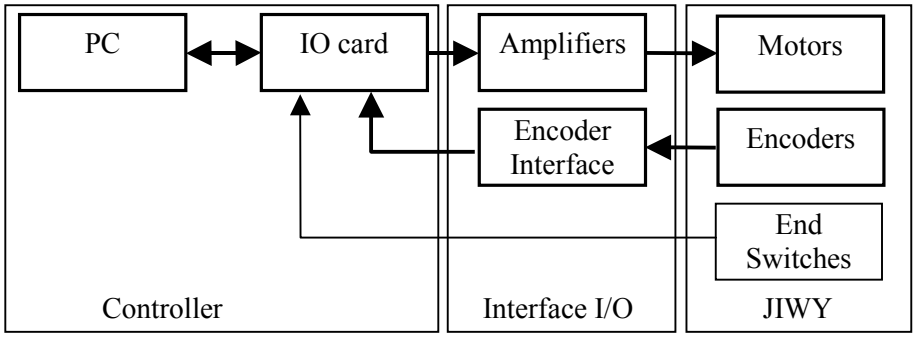

Figure 5 Architecture of JIWY

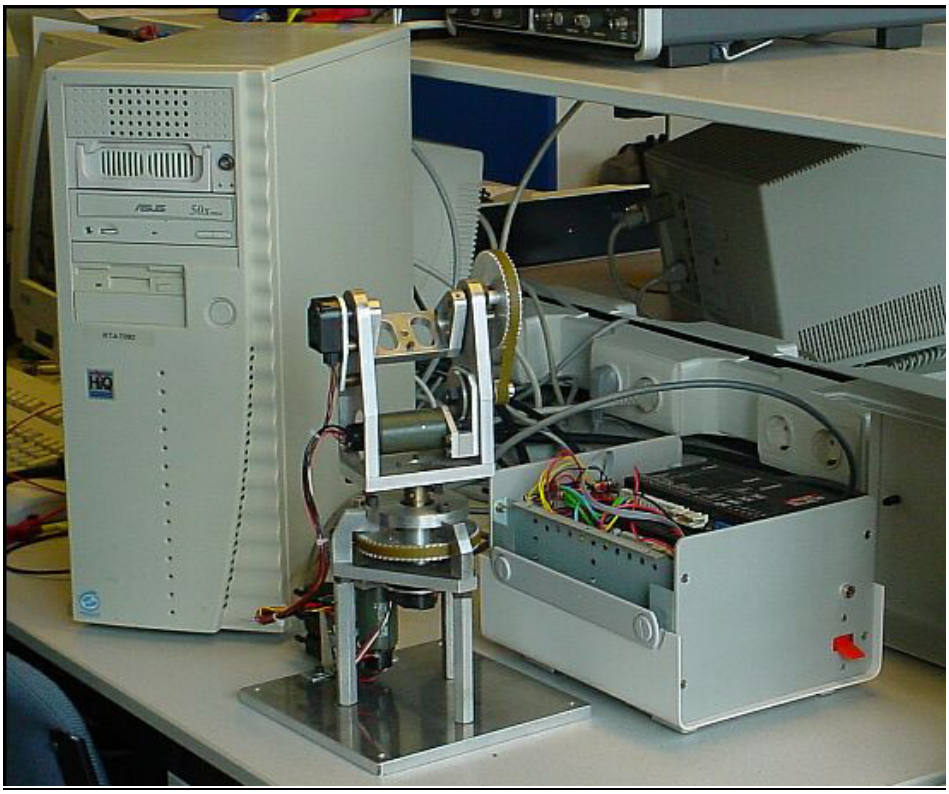

Figure 6 Photo of the JIWY robotic set up and its controller box

The processing unit is a 200MHz Pentium based standard personal computer, running under RT Linux operating system, version 3.1 of FSMlabs with Linux Slackware 8.0 (FSMlabs, 2002). RT Linux has been chosen in order to evaluate CT design of overall control system, although CT based designs themselves do not require a OS underneath. RT Linux is a real-time kernel for Linux, which runs Linux as the lowest priority process (task). This RT Linux kernel is publicly available, low priced, flexible and customisable (due to its open-source nature).

In fact, the minimal possible hardware/software configuration is augmented with additional hardware as well as software resources in order to provide easy, and at the same time reliable insight in the functioning of the control system based on CT software development methodology. That way, instead of embedded control CT software running on an embedded processing unit (a DSP or a MCU possibly supported with a programmed FPGA), the following monitoring and testing software and hardware components are added:

- A Linux full-featured operating system able to ease monitoring of activities inside CT software components, to make possible distribution of the video frames from the camera and eventually commands to the set up supplied from the TCP/IP network (intra- or internet).

- National Instruments PCI 6024E I/O card with 2 D/A, 16 A/D 12-bits converters (200 Ks/s), 8 digital input/output lines, 3 real-time clocks which

together with I/O amplifying/acquisition interface in the box allows: easy accessibility to relevant signals, testing on hardware communication faults, monitoring values of relevant hardware registers and so on.

The existing experiences from the JIWY project show that it is possible to reason about real-time software for (embedded) control systems in a way more natural to the human designer then it is provided by existing methodologies, namely, to preserve the concurrent nature of the control system and its environment in the ECS software. But, so far, there were no simple and practical methods and tools to express this natural concurrency in an elegant way, and moreover use the structures conceived in that way for further refinement towards real-life implementations *smoothly*. The research efforts trace a route of integration of the overall analysis and design trajectory by means of combination of prototyped and existing commercial CAD tools and case studies.

5 Conclusions

At the moment of submitting the text, the JIWY set up was built up and initial tests showed that it functioned well. We expect to show JIWY fully working at the conference.

The case study shows possibilities of extending the control software with additional features like safeguarding, without compromising the architecture of the controllers.

Further work comprises of using the experiences gained in this case study to sharpen the paradigm and design trajectory of embedded control system design and implementation. Emphasis is on the Stepwise Refinement of the description from control laws to embedded code.

References

FSMlabs (2002), http://www.fsmlabs.com, pp. ISSN:

Hilderink, G.H. (2002), http://www.ce.utwente.nl/javaPP/, pp. ISSN:

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

Broenink, J.F., G.H. Hilderink and A.W.P. Bakkers (1998), Conceptual design for controller software of mechatronic systems*, Proc. Proc. Lancaster Int. Workshop on Engineering Design CACSD'98,* Lancaster, United Kingdom, (Ed.), pp. 215-229, ISBN: 1-86220-057-2.

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

Roscoe, A.W. (1997), *The Theory and Practice of Concurrency,* Prentice Hall,

Hoare, C.A.R. (1988), *INMOS Limited Occam 2 Reference Manual,* Prentice Hall International Series in Computer Science, 0-13-629312-3.