Communicating Process Architectures 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

A Communicating Threads (CT) Case Study: JIWY[1]

Dusko JOVANOVIC, Gerald H. HILDERINK, Jan F. BROENINK
Twente Embedded Systems Initiative,
Cornelis J. Drebbel Institute for Mechatronics and Control Laboratory,
Dept. of Electrical Engineering, University of Twente,
P.O.Box 217, 7500 AE, Enschede, the Netherlands
d.jovanovic@utwente.nl

Abstract. This JIWY demonstrator is constructed in the context of the development of a design framework and software tools to efficiently support mechatronic engineers in developing sophisticated control computer code out of a set control laws. We use the CSP-based *Communicating Threads* (CT) library as the software communication layer – our hard real-time *virtual machine*. JIWY (just a soundful name) is a little tabletop robot with 2 rotational degrees of freedom and a camera as its *end effector*, controlled via a joystick on a PC running Real-Time Linux. The control laws are stepwise refined to obtain the control software, enhancing the control-law block diagrams to CSP-diagrams and showing the communication and composition properties of the control software. Enhancements like adding homing and end stops for safety can easily be added in the CSP-diagrams, without adapting the control law design. This shows the orthogonality of the design steps. State charts were not needed to express all functionality, which leaves the design simple. The prize of this elegance, namely performance, has as yet not been investigated.

1 Introduction

We strive to allow mechatronic engineers to design control-computer code, despite lack in skills of software engineering. Main motivation is that nowadays it is impossible to separate control engineering from software engineering: the only efficient way to implement controllers is to transform them into computer code for the chosen computer target (see also the trends mentioned in [1]). In control engineering practice, used software development techniques suffer from insufficiencies in knowledge in disciplines of software modelling, familiarity with concurrency in software, ways of allowing for reusability, software testing and so forth. The resulting code functions correctly in first instance, but is generally not reliable, efficient nor extendable/updateable.

The control-computer systems are *Embedded Control System* (ECS), because the control computer code is specific to the control system, for which the dynamic behaviour of the plant (i.e. the 'machine'-part of the embedded system to be controlled) is essential for the functionality. 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 computing system, because they are often dedicated to the ECS, although not necessary specifically developed. The software part consists of a layered

[1] This research is supported by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

structure of *controllers* and the *user interface*. The computing system is considered heterogeneous and distributed, because modern control systems are often composed of existing subsystems, having their own (low level) control software and processors [2].

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.

Our current research deals with the development of a design framework and a supporting software tool to efficiently support the mechatronic engineer in developing sophisticated control computer code out of a set of control laws.

Normally, mechatronic design engineers start with modelling the dynamic behaviour of the plant, and derive a control law for it. This control law is then gradually transformed via *Stepwise Refinement* towards efficient concurrent algorithms (i.e. the control computer code). During this process, simulation is often used as a means of verification. In the end phase, realization can also be done stepwise, namely by letting parts of the total system stay simulated, and letting other parts be in the final realization [3]. Especially the step from control law design to implementation is recognized as critical and not methodologically covered by existing approaches and tools.

For the modelling control law design parts including their simulations, existing tools [4, 5] suffice. Often, graphical modelling languages, e.g. block diagrams, are used in this design process for structuring the complexity of the control structures.

In this paper we show by means of the JIWY case, that the software development step can conceptually be covered using the CSP parallel processing paradigm. Since tools for modelling and control law design nowadays are graphically oriented, we use for the software development part the CSP-diagrams [6] instead of CSP expressions. CSP diagrams are basically block diagrams of communicating processes. The *Stepwise Refinement* process thus transforms the control law into an execution model that is ready to be translated into control software, whereby one can easily specify concurrency with CSP in a block diagram.

As the process communication abstraction layer, we use the Communicating Threads – CT– library [7].

First, the CT library and the CSP-diagrams are briefly introduced in section 2. Then, the robotic case is described in section 3. Our design solution is presented in section 4.

2 The Target Software Layer

2.1 Communicating Threads (CT) Library

The *Communicating Threads* (CT) library is an implementation of the CSP channel concept in Java, C++ and C [7-9]. Besides the communication facilities, it also manages all process scheduling.

The CT library's philosophy is to 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 library. 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 – and thus portability – of any real-time OS, to the extent that an OS is not even necessary any more. This *is* indeed the case in designs where small and cheap processing units are involved: DSPs, MCUs, FPGAs or in all (typical) cases when it is intended that the design fits into hardware resources as small (cheap) as possible.

The CT library delivers 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 1***, using the CSP *waiting rendezvous* principle. Synchronisation, scheduling and the actual data transfer are encapsulated in the channel. Channels are fully synchronised and basically unbuffered. However, buffers may be added to compensate communication latencies between the hard real-time parts and the soft real-time parts.

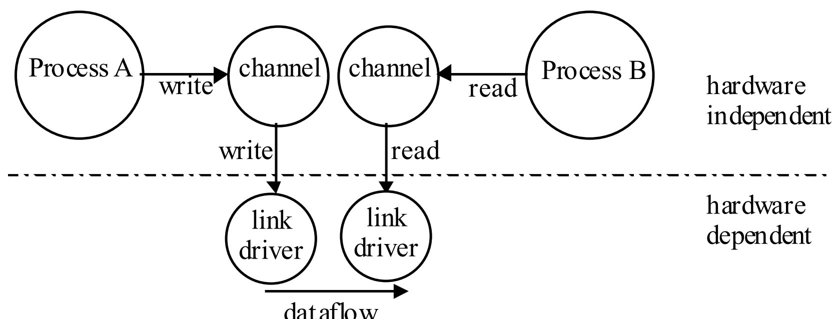

Figure 1. Channel implementation for multiprocessor systems [7]

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 [7]. 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.

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.

The CT library is available from our JavaPP website [10] and is used by several universities and companies. Furthermore, we use it at our 4th year class on Real-Time Software Development.

2.2 CSP Diagrams

As presented in [6], CSP-diagrams show processes and their interrelationships graphically. The execution model of the resulting network of processes is shown: It enables to specify real-time and parallel software architectures. CSP-diagrams are an addition to the block diagrams used in control engineering, which show the data-flow between the different subparts of the control system. Block diagrams can thus be seen as data-flow diagrams.

The extra information we need to add to a data-flow diagram is how the blocks are grouped into processes and what the concurrency of these processes is. This information is put into the newly applied edges between the vertices that remain the data-flow processes. The most essential interesting enhancements are the compositional relationships, showing whether processes run in parallel or in sequence.

3 JIWY: the Robotic Case Sudy

JIWY is a mechatronic set up for orienting some device, a camera in this case. The acronym JIWY is just a soundful name, and does not mean anything. The construction contains two revolute joints that allow the camera to rotate on a horizontal axis and a vertical axis. A user can control the set up and view the pictures from remote via a network connection. Currently, we use a joystick coupled to the JIWY control computer (***Figure 2***).

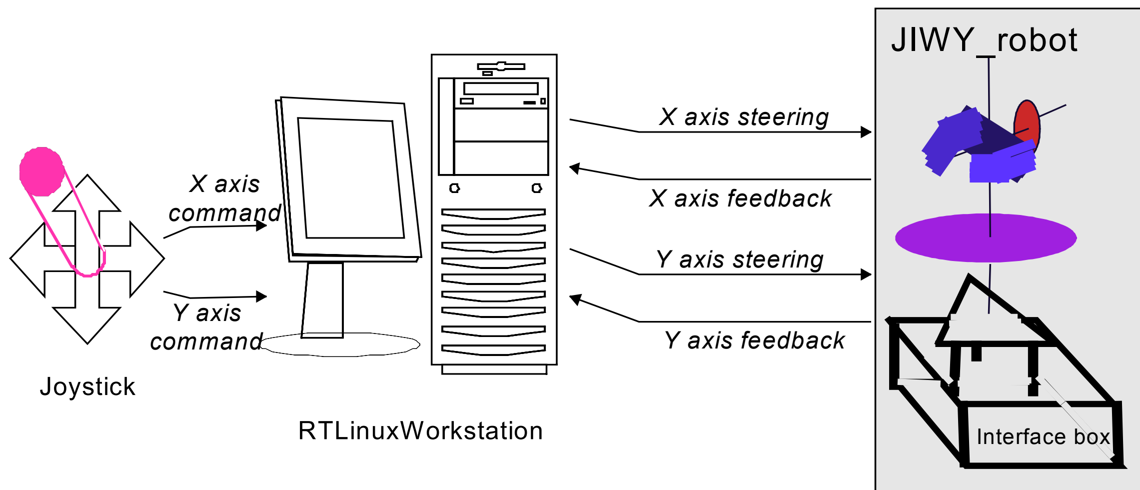

Figure 2. The JIWY set up (20-SIM model)

The operating vertical angle is 165° and the operating horizontal angle is 120°. The maximum swing is limited by end stops. The maximum angles between the end stops are respectively 300° and 150°. 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. Currently, the end switches and watchdog signals are not yet implemented.

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. The controlling PC is a 200MHz Pentium based standard personal computer, running under RT Linux operating system, version 3.1 of FSMlabs with Linux Slackware 8.0 [11]. The I/O card is the National Instruments PCI 6024E with 2 D/A, 16 A/D 12-bits converters (200 Ks/s), 8 digital input/output lines and 3 real-time clocks. See ***Figure 3*** for the architecture of JIWY. This leaves space for additional functionality to further explore and expand the control software.

We expect that this JIWY set up enables a sufficient complex case study for demonstrating multi-loop servo control problems. Besides the controller, a variety of safeguarding can be built-in, like swing limiting via end switches, steering limiting, and cable conditioning monitoring using the watchdog signals.

Using the extensions mentioned above, JIWY is protected by redundant safety-guards. Each safeguard is independent developed from each other as concurrent processes. This way, safeguarding 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.

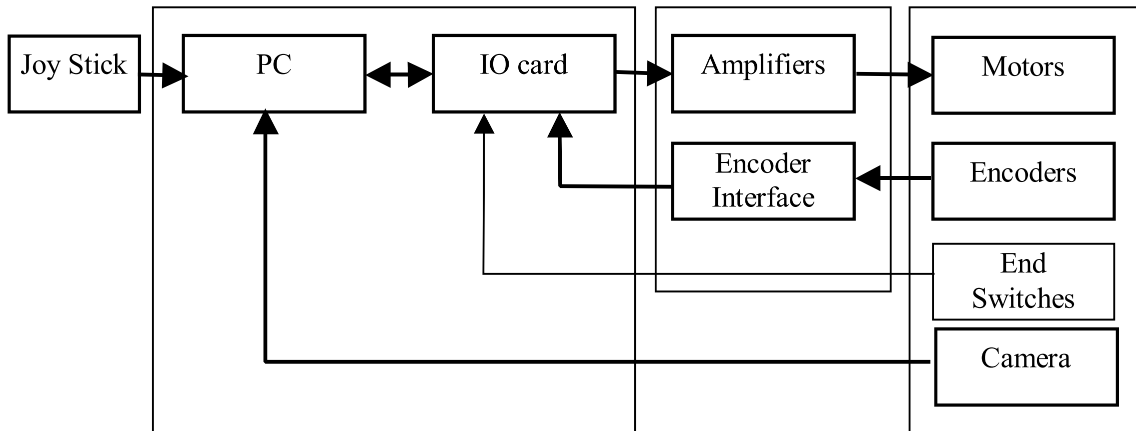

Figure 3. Control Architecture of JIWY

From a software design point of view it is interesting to see what impact each safeguard 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.

4 The Software Design Solution

By notions of *StepWise Refinement* (SWR) of the solution model, the design starts with a coarse sketch of the system in minds. To that end, the modelling tool 20- SIM [4] is used.

4.1 20-SIM Control System Model

The design problem addressed here is engineering of embedded computer system based on two closed control loops. The engineer starts with a sketch of overall system (***Figure 2***), and gradually refines the model of solution in the course of understanding the problem in hand. A typical control system consists of three main subsystems: appliance (industrial process, plant, set up) to be controlled, software which provides all control functionalities and specific I/O.

20-SIM can be used very well for early phases of SWR, because it allows organizing the model in the abstractions of high levels building blocks; the building blocks are partitioned in hierarchies of simpler and simpler building blocks, which can be easily accompanied with basic default dynamic behaviour. In that manner, simulations can be used as verification means in very early phases of the system design. ***Figure 4*** shows the specification of one level deeper in the hierarchy than ***Figure 2***. Both are simulateable. Note that all the block-diagram blocks are basic building blocks available in 20-SIM.

It turned out that the signals from the analogue joystick were polluted with noise that likely will make JIWY slightly shake. Analogue filters did not have the desired effect of eliminating the noise and closer analysis showed that the noise seemed to be digital spikes generated by the counter logics of the analogue joystick input. In order to eliminate the digital noise we had to add a digital filter for each of the axis. The digital filters are functional blocks that were placed before the controller blocks. The digital filters provided the desired effect.

The next steps of refinement will focus on the *Control* component only – i.e. refinement of the control software.

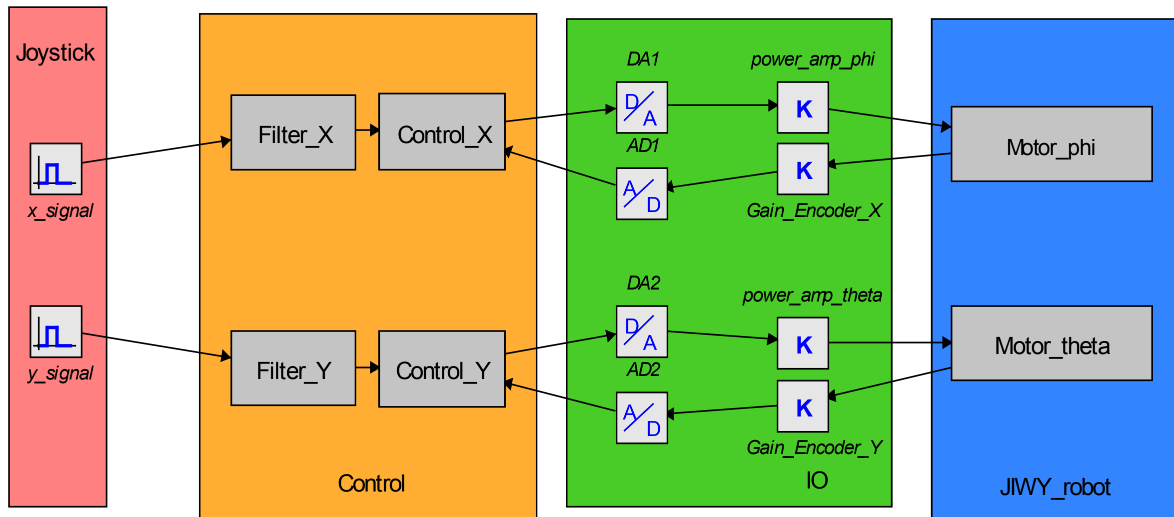

Figure 4. Simulateable JIWY system

4.2 CSP Diagrams of Control Software Components

CSP diagrams capture composition and communication aspects of the concurrent software components designed due to CSP principles. In perspective, CSP diagrams will be recognized as an extension to software modelling language UML.

As in UML, software design is approached via multiple views, captured by families of diagrams. CSP diagrams encompass two orthogonal kinds of views/graphs: communication graphs and composition graphs, as explained in [6].

For the starting point in reasoning about design of control software, ***Figure 5*** can be arranged in a slightly different way, which reveals the possibilities of parallel processing:

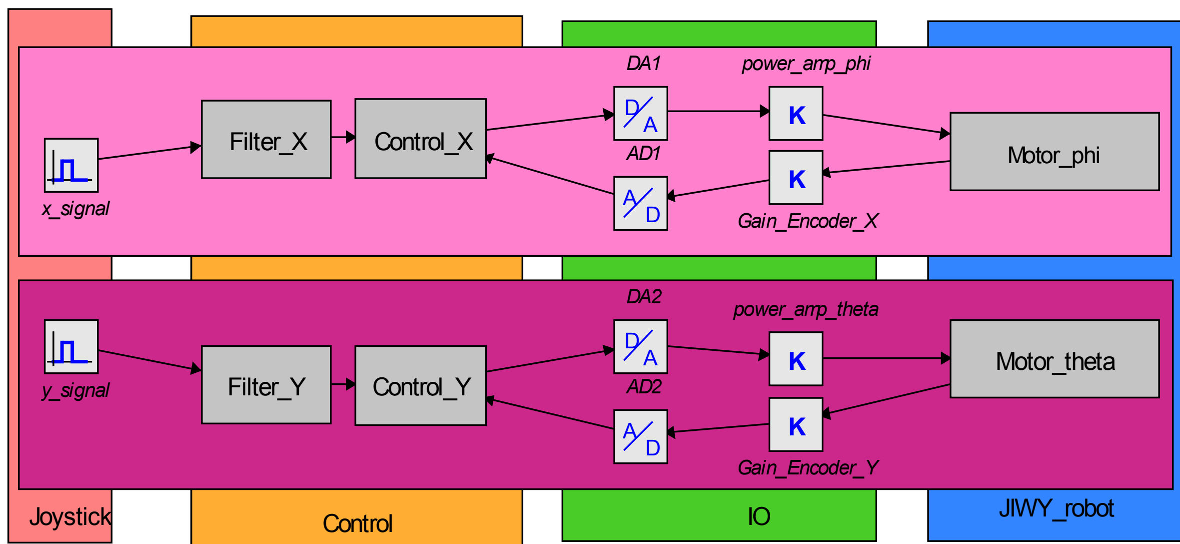

Figure 5. Simulateable JIWY prepared for parallel processing

The code generated by the current version of 20-SIM is placed in one single loop, such that parallelism is not possible. It is a sequential implementation, which is optimised for simulation. By using the CSP diagrams, we can specify exactly what we mean and how the execution framework should fulfil our requirements. CSP allows us to overcome the limitations of the previous mentioned simulator framework. In short, by designing CSP diagrams we will have complete control over the execution framework of the controller software and we can extent the execution model with additional processes that cannot (yet) be dealt with by 20-SIM.

The communication diagram of the controller is given in ***Figure 6***, and the composition diagram is given in ***Figure 7***, showing that all processes run in parallel. The processes in both diagrams are the same. Furthermore, layout of these processes is also kept the same, to optimally show the relation between the two diagrams.

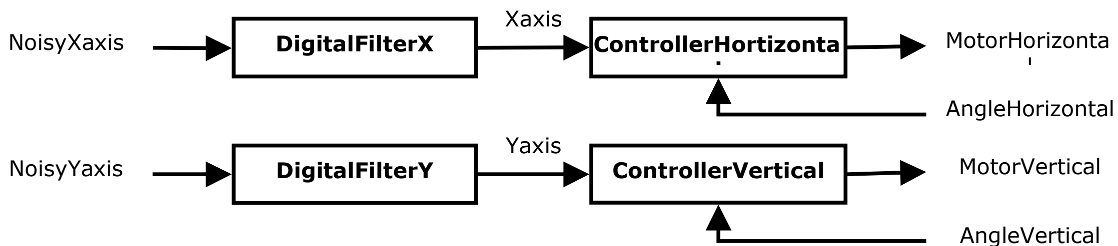

Figure 6. Communication graph of the controller

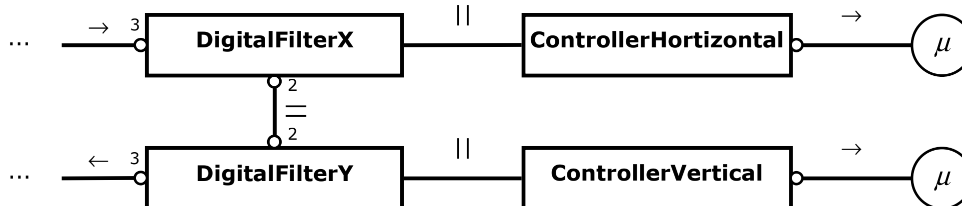

Figure 7. Composition graph of the controller

The μ-processes in ***Figure 7*** indicate that the process at the other end of the relationship, to which it is connected, will be executed in an infinite loop. Via the parenthesis symbols 'o' and their indices, it is indicated that the upper two processes will be executed independently from the lower two processes. This way, the decoupling of the two axes is indicated (See [6] for explanation of this syntax).

The scheduling of the processes will be determined by the data-flow that is depicted by the communication diagram. The channels between the concurrent processes perform scheduling and synchronized data-transfers between the processes. Thus, the reactivity and responsiveness is determined by the network of communicating processes.

The sampling rate is set for the input-channels and output-channels and the depending processes automatically adapt to that rate. This makes it easy to add multiple sampling rates with or without multiplication to the model. The input-channels and output-channels are concerned with interrupt handling, timer control, and other hardware issues. As a consequence, the processes become hardware independent and highly portable. This is conform to ***Figure 1***.

4.3 Enhancing the controller

In certain systems it could be desired to stop the controller when a stop-button is pressed. For example, how can we update the model so that we can stop a controller when we press a joystick button? When we press a joystick button we should be able to terminate a controller loop. The model as depicted in ***Figure 8*** specifies this feature. We added for each loop a choice operator '[]' that executes the control loop until a joystick button to be pressed. When a button is pressed a corresponding Boolean variable is set to true so that the condition of μ-process becomes false and the μ-process terminates. Furthermore, the indexes of the parenthesis have been incremented to correct the compositions of processes and relationships.

The *?*-process is a primitive reader process. The reader process is conditionally related to the choice operator. The reader will read from the channel `buttonchan1` when `buttonchan1` is ready (i.e. a writer writes to the channel) and condition is true (i.e. `[true]`); otherwise the reader will not be selected. If both the channel is ready and the expression is true then the choice operator may select the reader. The reader will immediately read channel `buttonchan1`. Also Boolean `stoph` will be set to true. This is similar for `buttonchan2`.

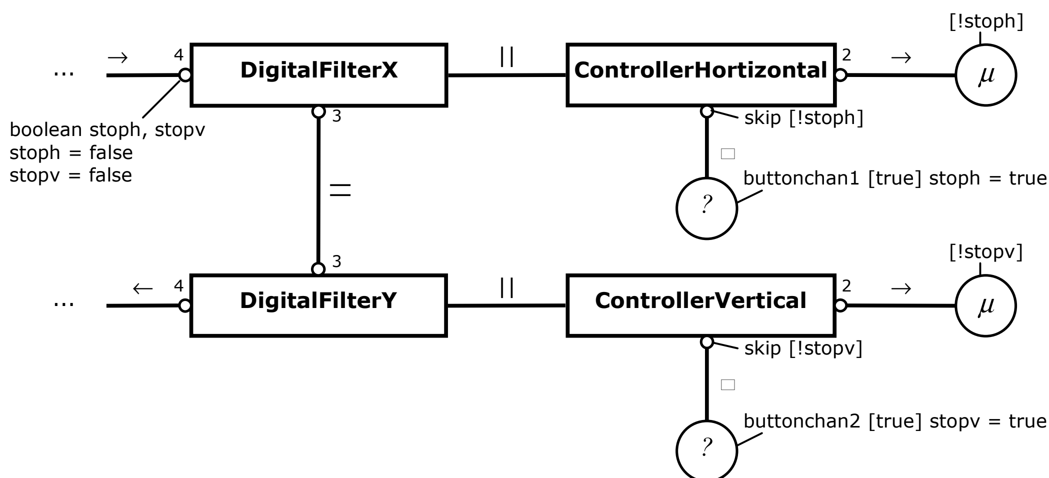

Figure 8. Stop button added

After a joystick button is pressed some process should take care of the stopped joint. For example, a homing process could be a last action that moves the join to its initial position. The processes `HomingHorizontal` and `HomingVertical` can easily be added in the communication diagram and composition diagram (**Figure 9** and ***Figure 10***).

In ***Figure 10***, two exception-handling processes are added: `ExceptionVertical` and `ExceptionHorizontal`. The exception operator '$\overrightarrow{\Delta}$' deals with exceptions in the system. Exception handling could take care of many things such as sending the user a message, retying to establish connection, or eventually terminating the loop. The process `ExceptionHorizontal` catches exceptions caused in `DigitalFilterX`, in `ControllerHorizontal`, or in reading the joystick button. The process `ExceptionHorizontal` does not cover process `HomingHorizontal`. A source of exception could for example be *division-by-zero* or a failure in an external channel (such as a DA-converter has stopped converting data or the joystick cable has been disconnected). This is similar for `ExceptionVertical`.

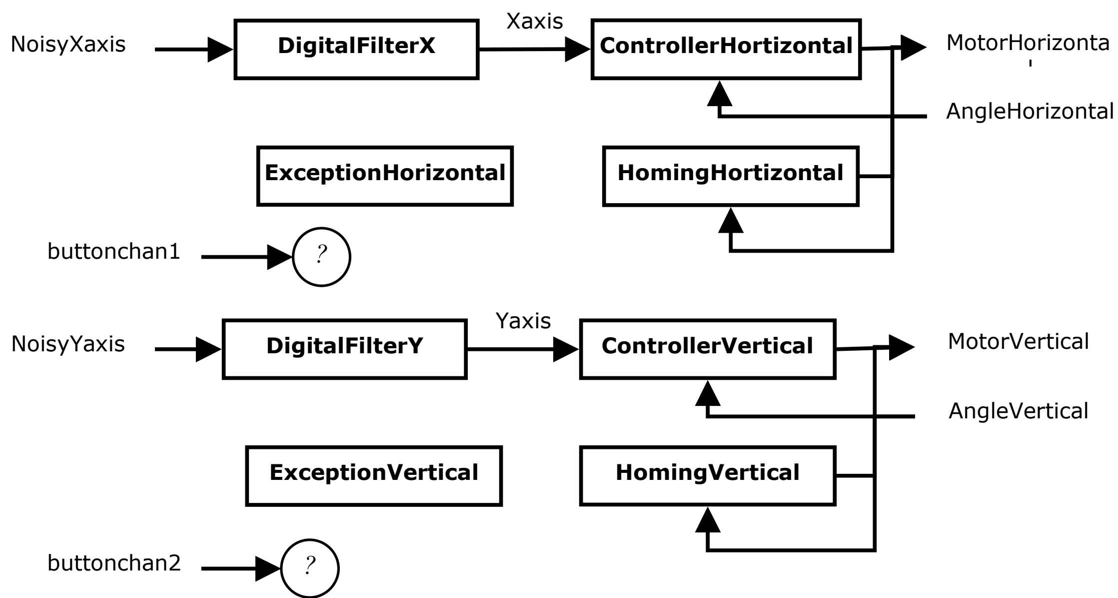

Figure 9. Communication diagram with Stop buttons and Homing

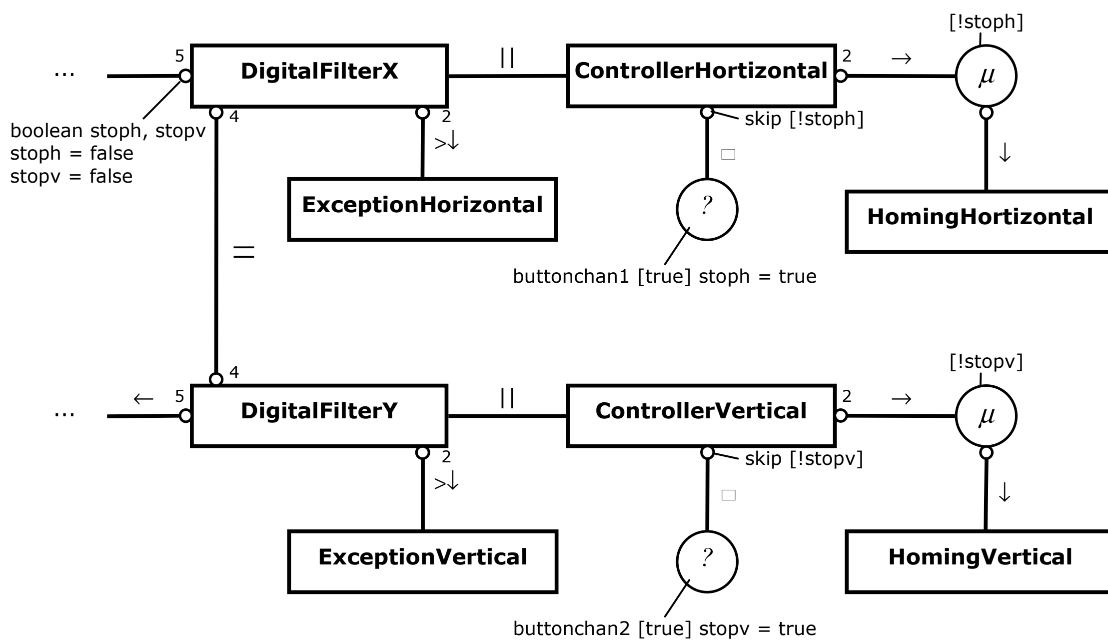

Figure 10. Composition diagram with Stop buttons and Homing

5 Conclusions and Further Work

In the work described above, we met the following advantages:

- The CSP diagram extensions only concern the edges of the graphs, and can as such be done *without* necessity to change the vertices (i.e. processes / submodels). This allows for easy switching between the different graphical formalisms. Automation can therefore be straightforward.

- The refinements per step are rather small. This implies that the SWR process can be performed really stepwise.
- Separate control loops (having may be different sample frequencies) can be shown clearly. Also hierarchies can be used in CSP diagrams. This can make the overall drawings more clear.
- State charts are not necessary to express concurrent behaviour and choices. So combinatorial explosion of number of states is not an issue here.

Further research will deal with the following aspects:

- Code generation itself from the CSP diagrams to CT code. In most cases the target will be either CTC++ or CTC (i.e. C++ code or C code), because of availability of compilers for the target platform.
- Prototyping a graphical tool to demonstrate the CSP diagrams and support checks.
- Performance analysis of resulting code (which is highly dependent on the target, the quality of the target compiler and the suitability of the target to run the CT library).

Indeed, the prize of this elegant embedded software development design methodology, namely performance and memory footprint, has as yet not been investigated. Especially, in the more high-volume embedded systems, performance and memory sizes are essential design criteria, since material costs are a crucial factor.

References

[1] L. D. J. Eggermond, *Embedded Systems Roadmap 2002*: STW Technology Foundation, 2002.

[2] H. Kopetz, *Real-Time Systems, Design principles for Distributed Embedded Applications*. Boston: Kluwer Academic Publishers, 1997.

[3] J. F. Broenink and G. H. Hilderink, *A Structured Approach to Embedded Control Systems Implementation*, presented at 2001 IEEE International Conference on Control Applications, México City, México, 2001.

[4] Controllab Products BV, *20-SIM Home Page,* http://www.20sim.com: 2002.

[5] The MathWorks, Inc., *Matlab and Simulink*, http://www.mathworks.com: 2002.

[6] G. H. Hilderink, *A Graphical Modeling Language for Specifying Concurrency based on CSP*, Communicating Process Architectures 2002, pp 265-294, IOS Press (Amsterdam), 2002.

[7] G. H. Hilderink, A. W. P. Bakkers, and J. F. Broenink, *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, 2000.

[8] G. H. Hilderink, J. F. Broenink, and A. W. P. Bakkers, *Communicating Threads for Java*, Proc. WoTUG-22, pp 243-261, IOS Press (Amsterdam), ISBN 90 5199 480 X, 1999.

[9] G. H. Hilderink, *Communicating Java Threads Reference Manual*, Proc. WoTUG-20, Parallel Programming and Java, , pp 283-325, IOS Press (Amsterdam), ISBN 90 5199 336 6, 1997.

[10] G. H. Hilderink, *JavaPP Project at UT*, http://www.ce.utwente.nl/JavaPP: 2002.

[11] Finite State Machine Labs, Inc.,, *FSMlabs Home Page*, http://fsmlabs.com: 2002.