Communicating Process Architectures 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

A multimodal robotic control law modelled and implemented with the CSP/CT framework†

Gerald H. HILDERINK, Dusko S. JOVANOVIC, Jan F. BROENINK
Twente Embedded Systems Initiative,
Drebbel Institute for Mechatronics and Control Engineering,
Faculty of EE-Math-CS, University of Twente,
P.O.Box 217, 7500 AE, Enschede, the Netherlands
g.h.hilderink@utwente.nl

Abstract. We use several formal methodologies for developing control applications at our Control Engineering research group. An important methodology we use for designing and implementing control software architecture is based on CSP concepts. These concepts allow us to glue multidisciplinary activities together and allow for formal stepwise refinement from design down to its implementation. This paper illustrates a trajectory and shows the usefulness of CSP diagrams for a simple mechatronic system. The simulation tool 20-SIM is used for creating the control laws and our CTC++ package is used for coding in C++.

1. Introduction

1.1 The need for a new approach

At the Control Engineering research group of the University of Twente, concurrent architectures are used in control (robotic, mechatronic) applications for more than two decades. Good experiences with applying transputers [1] programmed with occam [2] gave rise to the investigation of building control software with CSP concepts; transputers and occam are founded on CSP. CSP stands for Communicating Sequential Processes, which is a theory in the form of a process-algebra for specifying and analysing concurrent systems [3]. In order to exploit occam/CSP-like concurrent programming (developing concurrent software) in sequential programming languages like C/C++ and Java, the *Communicating Threads* (CT) libraries were developed [4]. CT allows for concurrent programming where multithreading in execution is encapsulated in CSP-like abstractions of processes and channels. Relying on these libraries, a software developer is freed from explicit dealing with low-level synchronization issues in a concurrent software architecture [5]. CT for C is called CTC, for C++ is CTC++ and for Java is CTJ.

Engineers engaged with designing control laws for a control application use block diagrams. Block diagrams provide an abstract graphical notation for modelling control laws offering a data-flow oriented and a plug-and-play design concept.

There is a big gap between a block diagram and its implementation; there are

† 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.

discontinuities between data-flow and control-flow. Furthermore, the engineer has little influence in the flow of control of the execution framework. In order to bridge the gap so that the engineer can specify any executable behaviour of the control software we use an intermediate graphical language, namely CSP diagrams. A CSP diagram specifies data-flow (continues on the block diagram notation) and control-flow in an application and eliminates the discontinuities between the two. This language allows the engineer to refine a design towards the demands of the system. This way the engineer gets a better understanding of concurrency that affects the real world in relation to the real-time software.

This paper reports a stepwise refinement trajectory from a control design to its implementation with the help of CSP diagrams. We use the simulation tool 20-SIM to design block diagrams of the system dynamics and the control laws. 20-SIM allows translating the equations to C-code. Of course we only translate the control laws since these have to be computed by the computer system. The paper mainly focuses on the way CSP diagrams are used for systematically designing control software. We use a simple mechatronic setup, named JIWY, which is a controlled motion device with two joints that rotate according to the movement of a joystick.

The needs for modelling concurrency are most commonly:

- managing complexity with multiple control loops,
- dealing with multiple frequencies,
- priorities and pre-emptive behaviours,
- event-driven control or reactiveness,
- guaranteeing real-time behaviour,
- dealing with distributive and parallel architectures.

The CSP diagrams deal with these issues at a high level of abstraction. We will show that this abstraction directly maps on software using CTC++. The path of code-generation is under control of the engineer *and* under control of the code-generation tool in a formal way.

1.2 The tools used in the refinement trajectory

We use a few languages and tools in the refinement trajectory from control laws to their implementation. As previously mentioned we use block diagrams to specify the control laws, the CSP diagram language is used to specify the concurrent executable framework, and C++ with CTC++ is used to implement the control software. For modelling block diagrams and C-code generation of the control laws we use the tool 20-SIM, see Section 1.2.1. Designing CSP diagrams is done by a drawing tool because there is no a specific design tool for CSP diagrams available. We are working on a specific design tool designing and analysing CSP diagrams. For coding we use a workbench that usually comes with the computer platform or a simple text editor tool for typing the C++ code. We have developed C++ template files that is used by 20-SIM, which takes care of most of the coding. Finally, a C++ compiler-linker and a program loader are used.

We will not elaborate on using these tools in this paper. We are primarily concerned with the design trajectory. This trajectory relies on identifying processes (i.e. physical processes, controller loop processes, control mode processes, and support processes) and identifying communication between these processes (i.e. signals, synchronization, interrupt handling, AD-DA conversion, etc.). The design forms the solution. In order to deal with concurrent software architectures we use the CSP concepts. Concurrency that boils from above affects every stage in the design to the implementation. Using CSP diagrams and CT is our solution to implementing concurrent software architectures.

1.2.1 20-SIM

20-SIM is a modelling and simulation tool developed by ControlLab Products B.V. [6], a spin-off company of the Control Engineering group of the University of Twente. It is a standard MS Windows application consisting of several integrated modules that supports modelling and design of mechatronics products in many aspects.

Modelling of a system can be performed by means of one or more 20-SIM modelling languages: bond graphs [7], block diagrams, iconic diagrams, equations [8] and importing Matlab models is supported as well. The tool complies well with the demand of offering a time-efficient and elaborate feedback to the user on the modelling/design decisions, by means of simulation plots, animated graphs and 3-D animations.

A recent functionality of 20-SIM is to generate C-code from internal equation model derived from a selected submodel. The C-code is sequential and needs to be inserted in a larger framework written in C or C++. This framework is specified in template files. Template files contain target-specific code, like device drivers, and special keywords that are replaced by generated C-code from 20-SIM submodel.

A C++ template is created which extends the 20-SIM code-generation. This C++ template allows us to generate individual submodel objects. A submodel object contains a data structure and methods that operate on this data structure, including the control law equations. The resulting C++ files can be compiled, linked and loaded on the target computer system. In Section 3.2 we discuss aspects of the C++ template for generating control processes, i.e. 20-Processes.

1.2.2 CSP diagrams

Since the introduction of the graphical modelling language based on CSP by Hilderink [9] the language has slightly evolved to become applicable for designing control software in the form of CSP diagrams. CSP diagrams show processes and their interrelationships by a graphical notation. A CSP diagram expresses an execution model of the resulting network of processes. The designer can specify the desired nature of concurrency of the real-time control software.

In this research we investigated the continuity between control models in 20-SIM, CSP diagrams, and the resulting code. The use of CSP diagrams is intended to bridge the gap between control models and the final code. CSP diagrams allow the designer to specify the connectivity between control processes and their execution order. CSP diagrams are based on data-flow concepts and control-flow concepts which can be used as an addition to the block diagrams used in control engineering.

One of many advantages that come with using CSP diagrams is that the graphical CSP notation offers straightforward code generation. Since our effort to create tools for drawing and analyzing CSP diagrams is in a begin stage, we do not have developed automated code-generation yet. Only 20-SIM can generate code which we will extensively use in this research. Coding of the execution framework is mainly done by handcraft and the results of this research will be used for the development of an automated code-generation tool of CSP diagrams. The coding by hand is not a difficult task since the mapping between a CSP diagram and object-oriented code with our CT library is almost one-on-one. See Section 3.

1.2.3 CT – Communicating Threads package

The Communicating Threads (CT) package consists of an object-oriented framework providing high-level design patterns based on CSP. These design patterns encapsulate multithreading from the user by means of processes, channels, and compositional

constructs. It basically adds process-oriented concepts to the object-oriented paradigm which are useful for reasoning about concurrency and real-time behaviors. Real-time and embedded computer issues have been carefully designed in CT. For example, it integrates real-time scheduling into the application rather than using it from a real-time operating system. In fact, CT does not require a real-time operating system and it can run on bare microprocessors, microcontrollers, or digital signal processors (DSPs). The CT framework has been ported to Java (CTJ), to C (CTC) and to C++ (CTC++). For CTC and CTC++ the memory footprint is low and the scheduling overheads are efficient and fast. We use CTJ for educational purposes and CTC/CTC++ for real-time laboratory setups. The examples presented in this paper are based on CTC++.

1.3 Overview

The refinement trajectory using block diagrams in 20-SIM and CSP diagrams for JIWY is described in Section 2. Section 3 elaborates on the implementation of CSP diagrams and generated 20-SIM C-code using C++ and CTC++. The mapping between CSP diagrams and the resulting code is discussed, while detail on the 20-SIM code-generation process is not discussed. Conclusions are found in Section 4

2. Designing concurrent controllers

2.1 Case study JIWY

As a case study to illustrate the stepwise refinement trajectory, a classical example of a multimodal controller has been chosen, which is a two-degree-of-freedom robotic end-effector, called JIWY, see Figure 1. The construction contains two revolute joints that allow mounted device to rotate on a horizontal axis and a vertical axis. The joints are equipped with DC motors and incremental encoders. We use a PC under DOS, an analog joystick, a amplifier/circuit box to drive the motors and to convert sensor signals. Details on JIWY can be found in [10].

JIWY uses incremental encoders and therefore the centre position for each axis has to be calculated. It is required that the outer boundaries of each axis are measured first in order to determine the centre position of the axis. A joint will first rotate to the left and when it reaches the end stop this process terminates. It remembers the maximum value. The next process rotates the joint with constant velocity to the right until it reaches the end stop. Again this process remembers the maximum value. Then the main servo position motion controller for controlling JIWY takes over. It uses the two maximum values from the alignment processes to determine the exact centre position of the setup. The main controller can be stopped by the user by pressing a specific button on the joystick. After the main controller is stopped by the user it is required that the joints return back to their centre position as a safe position. This process is called homing. After homing the motors will be disabled.

This functional description gives rise to four control modes or sub control laws per joint. One velocity control law for aligning left, one velocity controller for aligning right, a motion controller reacting on the joystick input, and a position controller for homing to the safe centre position.

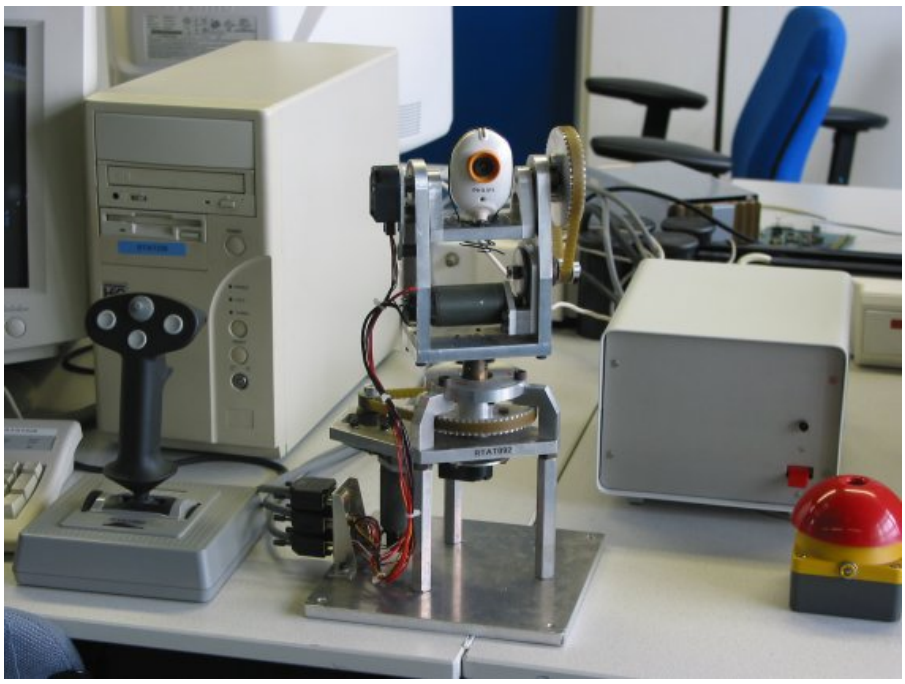


Figure 1 Photo of the JIWY end-effector

2.2 Process design and control law design

The design trajectory is a continuous interaction between process design and control law design. See Figure 2. Each design discipline can be carried out separately, but the whole results in an executable control software framework. The process design embraces the context (or overview of controllers), integration of control processes, and target depending issues. For this, CSP diagrams are used. The control law design focuses on the system dynamics and the control laws. The control laws are exclusively designed in 20-SIM using block diagrams. In the following these two design disciplines are interleaved.

2.2.1 Modes of control

The identified control modes are designed as four different control processes. Figure 3 shows these processes and their interrelationships in a CSP diagram. Figure 3a shows the communication diagram representing the data-flow between these four processes; the arrows render the communication relationships or channels between processes. Figure 3b shows the compositional diagram of the control-flow between these processes. Each joint operates in parallel and so there is a parallel relationship between the horizontal and vertical processes. The little circle at the end of the parallel relationship denotes a parenthesis which determines a group of control processes for the horizontal joint. The parenthesis symbol

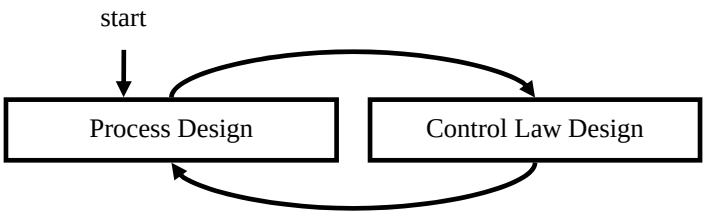
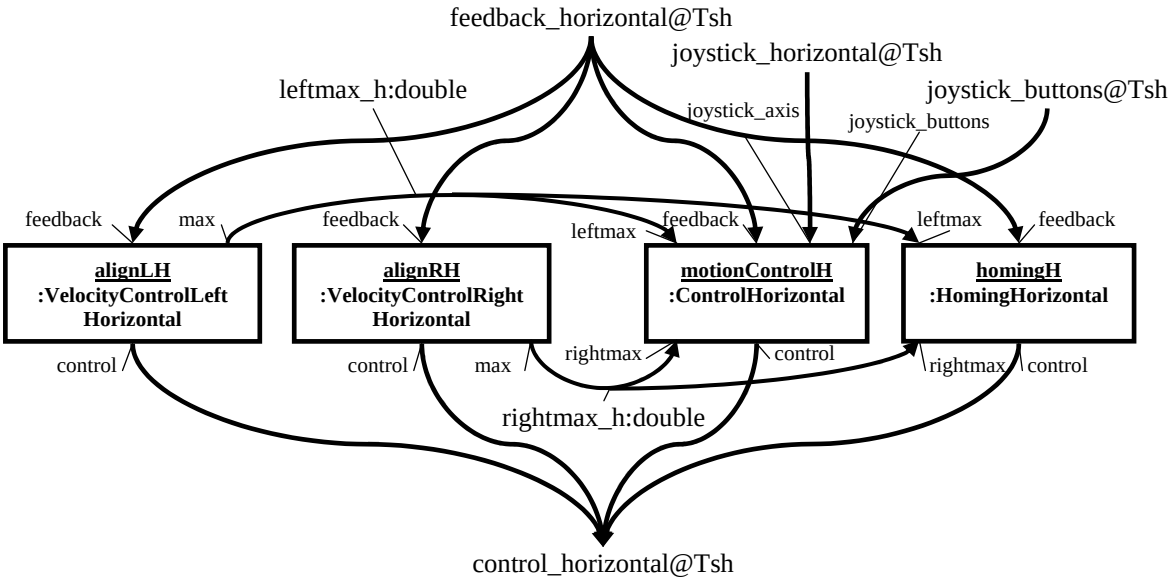
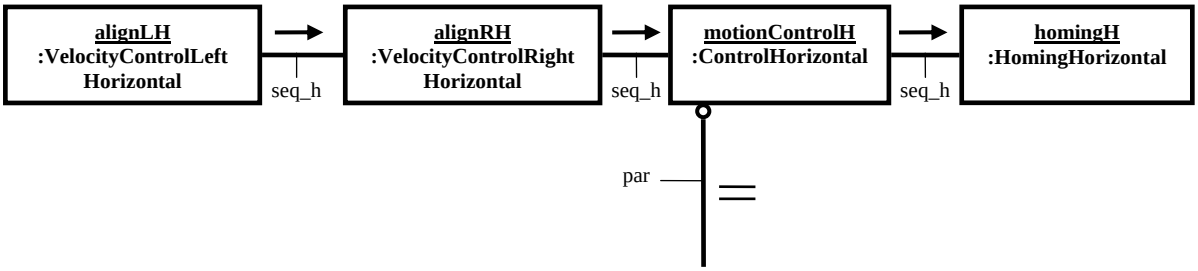


Figure 2 Proces design and control law design



(a) Four sub-controllers and their communication relationships for the horizontal joint



(b) Composition relationships between the four sub-controllers for the horizontal joint

Figure 3 CSP diagram of the horizontal controller

forms one parent anonymous process. This is similar for the vertical joint, which is not shown in the figure. In order to describe this system we will mainly restrict ourselves to the horizontal joint. The vertical joint is identical but with slightly different parameters than the horizontal joint.

Here, suffix "@Tsh" specifies that we deal with a sampled data system, where the samples of measured values arrive at equidistant moments in time. Clearly, each sample must be processed before the next arrival. This suffix means that the external channels are triggered on sampling period T_{sh} for the horizontal control loop. T_{sv} is the sampling period for the vertical control loop, which is not shown here. This information is used to set up timing, see Section 3.7.

A process can be in a different scope than another process when:

- a process resides at a different level in hierarchy than the other process,
- one process is in a different context than the other process, e.g. process in software and process in hardware.

A process that is out of scope is replaced by a *port name* at the end of the arrow. See the names `feedback_horizontal`, `control_horizontal`, `joystick_horizontal`, and `joystick_buttons` in Figure 3a. These names identify process-interface elements or ports of the parent process to which the CSP diagram belongs. These names are used inside the parent process and are connected to communication relationships outside the process. In Figure 3a these external processes reside in hardware. A port of the parent process acts also as a local label of the communication relationship.

In order to show the ports of a process to which communication relationships are

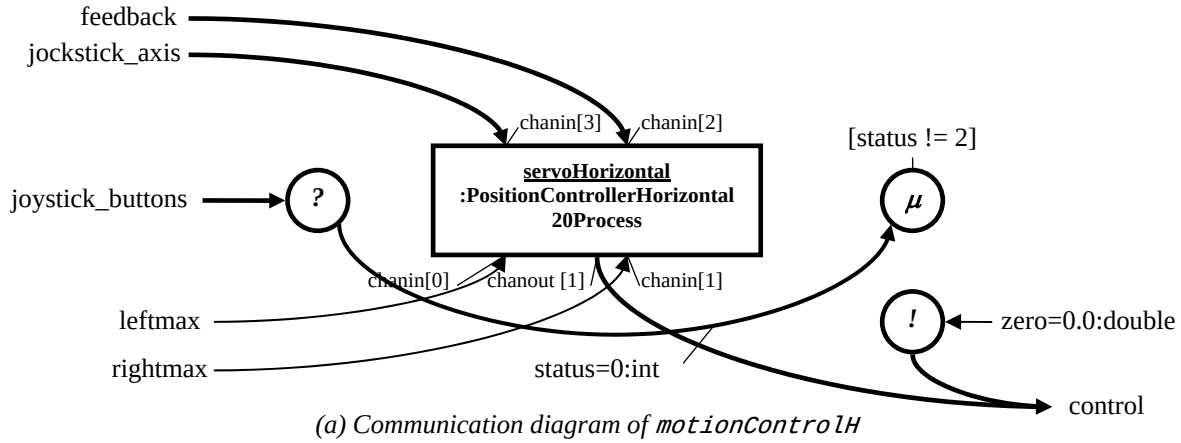
connected, the name of a port can be shown next to the arrow head or arrow tail, e.g. `feedback`, `max`, `leftmax`, etc. These names are used locally by the process at the end of the communication relationship. Each port has properties such as its name, direction and communication type, e.g. `{feedback_horizontal, channelin, double}` and `{control_horizontal, channelout, double}`. The communication types of ports at each end of a communication relationship should be of the same type otherwise they are incompatible and cannot be connected. For example, port {`max`, `channelout`, `double`} on `alignLH` is compatible to port {`leftmax`, `channelin`, `double`} on `motionControlH`. The label `leftmax:double` identifies the actual communication object, i.e. a variable of type double, between the ports `max` and `leftmax`. These labels can be hidden individually in order to hide detail.

A communication diagram is responsible for the declarations of the processes, the internal communication relationships and the ports to external communication relationships. A composition diagram is also responsible for the declarations of processes, the compositional relationships, conditions, and guards. In Section 3 we will see them appearing in the code with the use of CTC++. For this each element requires an identifier.

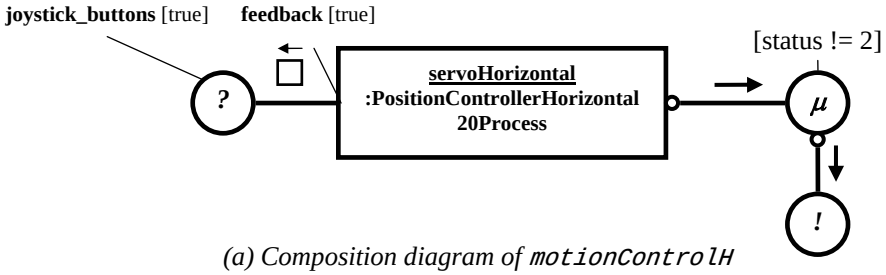
2.2.2 Motion controller

The `motionControlH` process in Figure 3 is the main sub-controller that receives the set points from the joystick that is used by the user. This process only terminates when the stop button on the joystick is pressed. This process contains a 20-SIM process `servoHorizontal` which computes one sample in the control law. See Figure 4. The looping is specified by the µ-process.

In Figure 4a communication relationships between the external ports and the ports of the 20-SIM process `servoHorizontal` are specified. A 20-SIM process can be recognized by the name `20Process` at the end of the process class name, like



(a) Communication diagram of `motionControlH`



(a) Composition diagram of `motionControlH`

Figure 4 CSP diagram of process `motionControlH:ControlHorizontal`

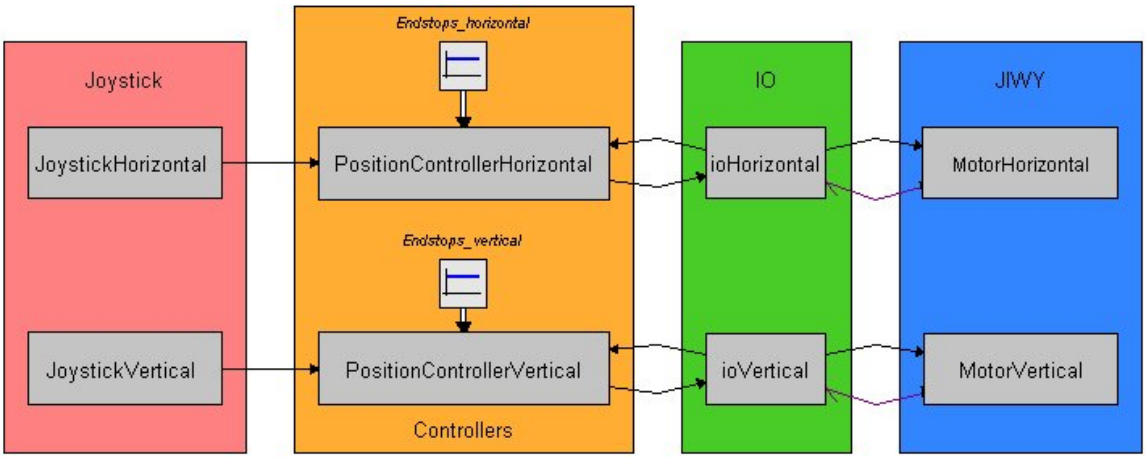


Figure 5 20-SIM model of the position motion controller

`PositionControllerHorizontal20Process`. A 20-SIM process or simply 20-Process is a CSP process with special treatment, namely:

- it is a single-shot process performing a 20-SIM submodel for one sample;
- the process-interface (ports) consists of a channel-input and a channel-output array;
- it is generated by 20-SIM.

The 20-SIM process ports are rendered by indexed channel array names. These channel array names rise from the 20-SIM submodel; chanin[i] → u[i] and y[j] → chanout[j], where u[] is the model input vector $\bar{u}$ and y[] the model output vector $\bar{y}$.

Two communication relationships have been added, namely `status` and `zero`. Here `status` holds the status of the joystick buttons when it is pressed. The looping process μ will continue repeating to perform the alternative relationship '□' until `status` is 2. `zero` is a constant value 0 in the communication diagram, see Figure 4a. Therefore these variables are declared and initialized with the default value in the code.

The alternative construct will wait until the channels `feedback` or `joystick_buttons` become ready. Then it makes a (prioritized) choice between reading the joystick button and computing one sample in the controller. The construct will select the controller when feedback is ready to communicate and when the joystick button is not pressed. If the joystick was pressed and feedback is not yet ready to communicate then the joystick buttons will be read. If both the joystick button was pressed and the feedback is ready then reading the buttons from the joystick button gets a higher priority. In this case the loop terminates right before the entire process terminates, it will send a zero to the actuator in order to release any steering of the joint.

Separate from CSP diagrams, a simulate-able model in 20-SIM has to be designed. A 20-SIM model embraces all relevant dynamics of the system and the controller itself. The model can be simulated and the feedback can be used to understand and improve the dynamics of the system. The identified processes can be found in the 20-SIM model in the form of mathematical blocks.

The position motion controller is the main operational mode of this servo system. The model renders the context of close-loop system where the two axes are servo-controlled by a joystick as a position set-point generator. Figure 5 presents a 20-SIM model of this control mode (for both axes). The joystick input is simulated by some function as described in submodel `JoystickHorizontal` and `JoystickVertical`. The `ioHorizontal` and `ioVertical` submodels model and simulate the hardware input/output interfacing between the physical system and the control software to make simulation more realistic compared to

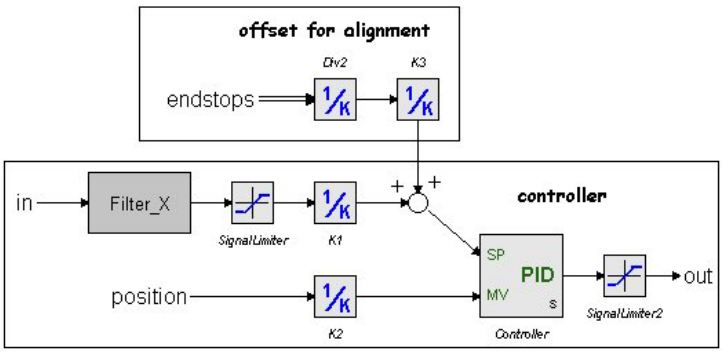


Figure 6 Block diagram of a position controller

the real setup. The physical system is modelled by the `MotorHorizontal` and the `MotorVertical` submodels using bond-graphs.

The submodels `PositionContollerHorizontal` and `PositionControllerVertical` are the software controllers that will be code generated by 20-SIM as 20-SIM processes. The block diagrams of these submodels slightly differ with different parameter values. One block diagram of the control law is depicted in Figure 6.

2.2.3 Alignment controller

The alignment modes are based on velocity controllers. For each rotation and joint the same velocity controller model is used. Again these controllers use slightly different parameter values.

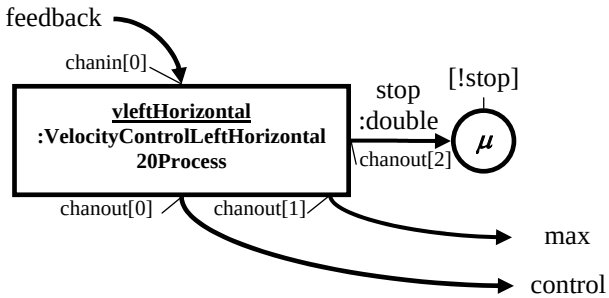
Process `alignLH` has been worked out in Figure 7. This process contains a 20-SIM process `vleftHorizontal` and a loop construct which repeats itself until `stop` is true. The variable `stop` becomes true when the end-stop has been reached. This is specified by the 20-SIM submodel. Process `alignRH` is identical to `alignLH` except that the motor rotates right.

We separated the classes `VelocityControlLeftHorizontal` and `VelocityControlRightHorizontal` because the submodels are also separated in 20-SIM. This allows changing parameters independently of each other as a result of parameter fine-tuning. For example, the resistance at rotating left could be different than when rotating right. The 20-SIM model for left alignment and for right alignment is depicted in Figure 8.

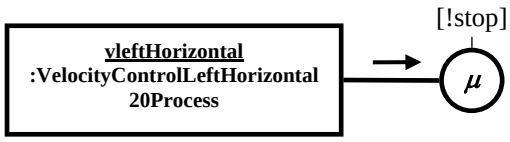
The submodels `VelocityContolLeftHorizontal`, `VelocityContolLeftVertical`, `VelocityContolRightHorizontal`, `VelocityContolRightVertical` are the controllers that will be code generated by 20-SIM, called 20-SIM processes or 20-Processes. One of the block diagrams is depicted in Figure 9.

2.2.4 Homing controller

The homing controller uses the same position controller process as in the motion controller, see Figure 4. This process requires a set-point that is set to the centre position (i.e. zero). Figure 10 shows the CSP diagram. The 20-SIM model is given in Figure 5.

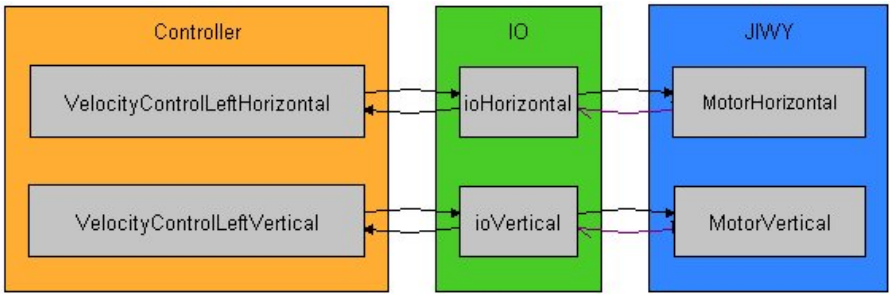


(a) Communication diagram of `alignLH`

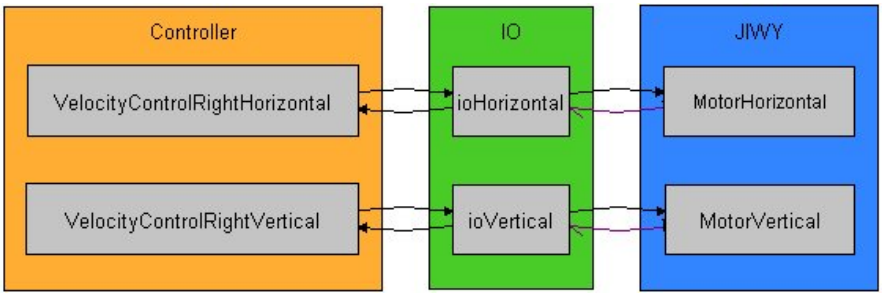


(b) Composition diagram of `alignLH`

Figure 7 CSP diagram of `alignLH:VelocityLeftHorizontal`



(a) Alignment mode for left rotation



(b) Alignment mode for right rotation

Figure 8 Alignment modes for left and right rotations

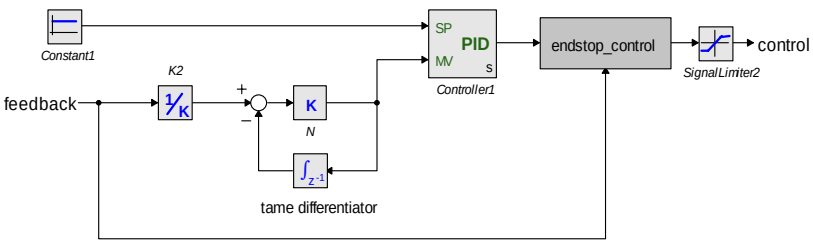
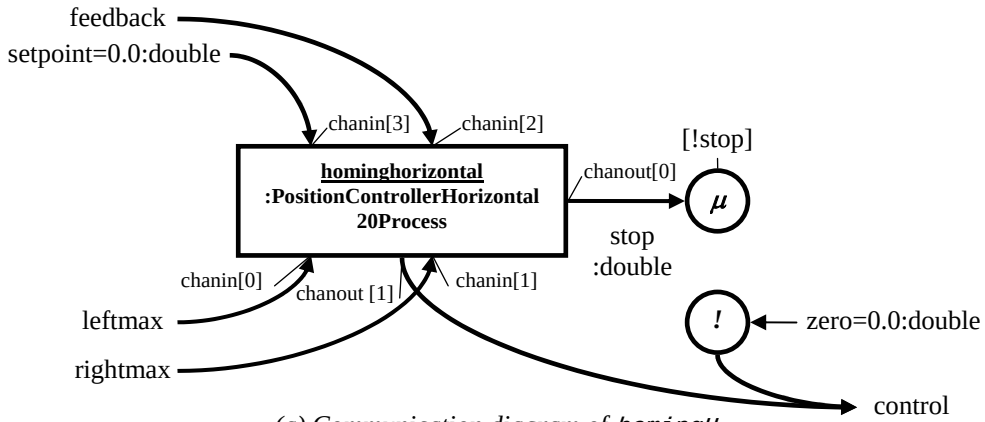
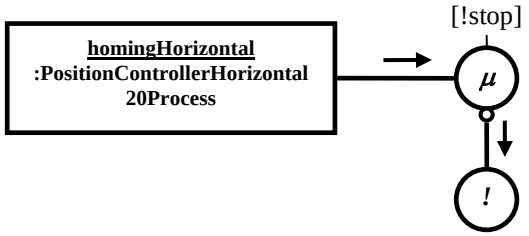


Figure 9 Block diagram of a velocity controller



(a) Communication diagram of `homingH`



(b) Composition diagram of `homingH`

Figure 10 CSP diagram of `homingH:HomingHorizontal`

3. Implementation

3.1 20-SIM submodels

A system is usually designed by separating concerns in submodels in 20-SIM. The submodels which are essential to the control software must be code-generated in C-code. In our case, a submodel is code generated with 20-SIM using the C++ template which results in a C++ class: `MySubmodel` class.

A 20-SIM submodel requires a vector of input signals $\bar{u}$ and a vector of output signals $\bar{y}$. These signals are mapped on array of variables `u[]` for input-only and `y[]` for output-only.

```
double u[n], y[m];
```

The constructor requires no arguments and the constructor sets up the data structure.

```
submodel = new MySubmodel;
```

After construction of the object the begin state of the model needs to be initialized by

```
submodel->Initialize(u, y, 0);
```

Once the model has been initialized the control law equations can be performed by invoking the `Calculate()` method. This method requires the input vector and the output vector of signals (i.e. arrays of variables). This method must be called for each sampling periode.

```
submodel->Calculate (u, y);
```

The 20-SIM submodel is a black box which behavior can be observed and studied by simulating the submodel in 20-SIM. No immediate knowledge of the generated code is required since modifications to control laws are done in block diagrams and not directly in C-code. Therefore we omit the implementation of `MySubmodel`.

3.2 20-SIM process

A 20-SIM process is a CSP process that is generated by 20-SIM. A 20-SIM process has the task to invoke the equations of the submodel object when input data is available and in a particular order of execution. A 20-SIM process is a single-shot process, see Section 2.2.2. This means that the output values are calculated once according to the control laws and the process immediately terminates afterwards. A 20-SIM process can be executed successively according to its compositional relationships with other processes. A 20-SIM process can be used in any kind of compositional relationship; in parallel, in sequence or by choice. In case the 20-SIM process is executed in parallel it must pass data and synchronize on CSP channels (or blocking channels) in order to prevent any race-hazards between process input and output. These channels provide reactiveness to internal and external events. In case a 20-SIM process is in sequence to another process then unblocking channels (`ChannelVar`) should be used to pass data between these processes because blocking channels will cause deadlock.

A 20-SIM process of submodel `MySubmodel` has the following default constructor:

```
MySubmodel20Process::MySubmodel20Process(ChannelIn<double> **chanin,
ChannelOut<double> **chanout)
{
  this->chanin  = chanin;
  this->chanout = chanout;
  this->u       = (double *) malloc (n * sizeof (double));
  this->y       = (double *) malloc (m * sizeof (double));

  /--- Execute Initialize submodel
  submodel = new MySubmodel;
  submodel->Initialize(this->u, this->y, 0);
}
```

The constructor requires two channel arrays (or vectors); an array of channel-inputs and an array of channel-outputs. The arrays and each array element must be declared outside the process. The constant number *n* is the number of input-channels and *m* is the number of output-channels. These constants are determined and filled-in by the 20-SIM code-generator.

The 20-SIM process performs a standard procedure:

- read from channels to $\bar{u}$
- calculate submodel with inputs $\bar{u}$ and output $\bar{y}$
- write to channels from $\bar{y}$
- terminate

The code of this procedure is given in the `run()` method of the process below:

```
void MySubmodel20Process::run(void)
{
  //--- input
  for(int i=0; i<1; i++)
    if (chanin[i] != NULL)
      chanin[i]->read(&(u[i]));

  //--- perform calculation
  submodel->Calculate (u, y);

  //--- output
  for(int i=0; i<1; i++)
    if (chanout[i] != NULL)
      chanout[i]->write(&(y[i]));

  //--- done and now terminate
}
```

The arrays `u[]` and `y[]` are used internally by the submodel and the 20-SIM process and their content is not influenced other than by channel inputs. Therefore, the constructor declares these arrays exclusively for the 20-SIM process.

3.3 Top-level construct

As mentioned in Section 2 the CSP diagram declares all entities. The declarations of the top-level CSP diagram in Figure 3 and the vertical part can be found in the main source file of JIWY. Since we do *not* have a code-generator this is done by hand. The class names and identifiers specified in the CSP diagram are found in the source code.

The communication relationships, processes, and constructs are declared as follows:

```
//--- channel pointer declarations
ChannelIn<double>  *feedback_horizontal = NULL;
ChannelIn<double>  *feedback_vertical   = NULL;
ChannelOut<double> *control_horizontal  = NULL;
ChannelOut<double> *control_vertical    = NULL;
ChannelIn<double> *joystick_horizontal  = NULL;
ChannelIn<double> *joystick_vertical    = NULL;
ChannelOut<int>   *joystick_buttons     = NULL;

//--- internal channel declarations ('chan' prefix to variable names)
ChannelVar<double> chanleftmax_h  = new ChannelVar<double>;
ChannelVar<double> chanrightmax_h = new ChannelVar<double>;
ChannelVar<double> chanleftmax_v  = new ChannelVar<double>;
ChannelVar<double> chanrightmax_v = new ChannelVar<double>;

//--- Analog joystick
AnalogJoystick *joystick = new AnalogJoystick();
joystick_horizontal = new AnalogJoystickX(joystick);
joystick_vertical   = new AnalogJoystickY(joystick);
```

```
joystick_buttons    = new AnalogJoystickButtons(joystick);

//--- DAQSTC, National Instruments 6024E IO Board
DAQSTC *daqstc = new DAQSTC();
daqstc->Initialise();

//--- Analog Output
daqstc->SetAOTM(AOTM::Primary, AOTM::CPUDriven);
control_horizontal = daqstc->GetDAC(DAC::DAC0);
control_vertical   = daqstc->GetDAC(DAC::DAC1);

//--- Two counters for sensors
feedback_horizontal = daqstc->GetCounter(GPC::Counter0);
feedback_vertical   = daqstc->GetCounter(GPC::Counter1);
```

Firstly, the pointers to the external channels and the internal channels are declared. Secondly, the external channels are declared and are assigned to the external channel pointers, see italic code. The italic code cannot be derived from the CSP diagram and is added apart from diagram-to-code translation. These external channels are based on hardware-specific drivers with channel-interfaces and with CSP-valid channel semantics. Therefore, this code is considered to be target dependent, where as the control processes remain hardware-indepedent. The communication relationships `leftmax_h`, `rightmax_h`, `leftmax_v`, and `rightmax_v` are prefixed by `chan`. We will see later that this is required to distinguish a channel from a variable used in conditional expression.

Usually CSP channel will cause deadlock when they are used between processes in sequential composition. Therefore to avoid deadlock we use unblocking channels only for communication between processes in sequential composition. These communication relationships are `leftmax_h`, `rightmax_h`, `leftmax_v`, and `rightmax_v`. These become `ChannelVar` channels which are the unblocking versions of `Channel`. `ChannelVar` is a subclass of `Channel` and therefore a `ChannelVar` can replace a `Channel` in code; they can be intertwined because they have the same channel-interface. See listing above. These unblocking channels can be analysed in CSP as an one-place override buffer within the communication relationship.

The processes are declared in the main source file as follows:

```
//--- declare all processes
ControlHorizontal *motionControlH =
   new ControlHorizontal (joystick_horizontal, joystick_buttons,
feedback_horizontal, control_horizontal, chanleftmax_h, chanrightmax_h);
ControlVertical   *motionControlV =
   new ControlVertical (joystick_vertical, joystick_buttons, feedback_vertical,
control_vertical, chanleftmax_v, chanrightmax_v);
VelocityControlLeftHorizontalProcess  *alignLH =
   new VelocityControlLeftHorizontalProcess(feedback_horizontal,
control_horizontal, chanleftmax_h);
VelocityControlRightHorizontalProcess *alignRH =
   new VelocityControlRightHorizontalProcess(feedback_horizontal,
control_horizontal, chanrightmax_h);
VelocityControlLeftVerticalProcess    *alignLV =
   new VelocityControlLeftVerticalProcess(feedback_vertical, control_vertical,
```

```
chanleftmax_v);
VelocityControlRightVerticalProcess   *alignRV =
   new VelocityControlRightVerticalProcess(feedback_vertical, control_vertical,
chanrightmax_v);
HomingHorizontal *homingH =
   new HomingHorizontal(feedback_horizontal, control_horizontal, chanleftmax_h,
chanrightmax_h);
HomingVertical   *homingV =
   new HomingVertical(feedback_vertical, control_vertical, chanleftmax_v,
chanrightmax_v);
```

The arguments correspond to the port names of the process-interface. These names can be found in the interface description of the process classes. See for example the process-interface specified by the constructor-interface in Section 3.4. Here the port names are substituted by the local names of the channel pointers.

The compositional construct of the Figure 4 with CTC++ is:

```
//--- create CSP relationships
Sequential *seq_h = new Sequential(alignLH,alignRH,motionControlH,homingH,NULL);
Sequential *seq_v = new Sequential(alignLV,alignRV,motionControlV,homingV,NULL);
Parallel   *par   = new Parallel(seq_h, seq_v, NULL);

//--- timer code, see Section 3.7
```

The entire process will be executed by invoking `run()` on the top-level construct:

```
par->run();
```

Thus before the `run()` method of the top-level construct is invoked, all processes, channels, constructs, timing, and objects have been created. Once the `run()` method is invoked the real-time run bodies perform according to the communication relationships *and* the compositional relationships in the CSP diagrams. After the top `run()` method terminates the declared entities can be deleted and the program can gracefully terminate. This top-level process is called a *network building process*. In our approach this top-level process is the only hardware dependent process in the software since it is the only one that sets up hardware dependent objects. All other processes are hardware independent since they solely use channels and external channels access hardware devices. Whether channels are internal or external is invisible to the processes since channels have a common and most simplified fundamental interface for channel communication. Of course, processes may depend on the data that flows through channels.

3.4 Motion controller process

The process-interface of `motionControlH` is specified by the constructor below. This constructor assigns its ports to the ports of the child processes. The internal channels for `setpoint`, `stop`, and `zero` are declared. Of course the 20-SIM process `servoHorizontal` is declared.

```
ControlHorizontal::ControlHorizontal(ChannelIn<double> *joystick_axis,
ChannelIn<double> *joystick_buttons, ChannelIn<double> *feedback,
ChannelOut<double> *control, ChannelIn<double> *leftmax, ChannelIn<double>
*rightmax)
{
  //--- create channel-input array and a channel-output array
  this->chanin  = new ChannelIn<double> * [4];
  this->chanout = new ChannelOut<double> * [2];

  chanstatus = new ChannelVar<int>(0); //(!)

  chansetpoint = new ChannelVar<double>(0.0);
  chanstop = new ChannelVar<double>;
  chanzero = new ChannelVar<double>(0.0);

  chanin[0]  = leftmax;
  chanin[1]  = rightmax;
  chanin[2]  = feedback;
  chanin[3]  = joystick_axis;
  chanout[0] = NULL;
  chanout[1] = control;

  //--- create the 20-SIM process
  servoHorizontal = new PositionControllerHorizontal20Process(chanin, chanout);

  //--- set up the alternative construct process
  alt = new Alternative();
  alt->add(new Guard(feedback));
  alt->add(new Guard(joystick_buttons));
}
```

The `run()` method performs the CSP diagram of Figure 4. Here we see that `status` and `chanstatus` are used. We need to distinguish between a channel and a variable. This is similar for `zero` and `chanzero`.

```
void ControlHorizontal::run(void)
{
  int status = 0;
  double zero = 0;

  do
  {
    switch(alt->select())
    {
      case 0: //--- controller process
          servoHorizontal->run();
        break;
      case 1: //--- stop button process
        joystick_buttons->read(&status);
        chanstatus->write(&status); // (!)
```

```
      break;
    }
    chanstatus->read(&status); // (!)
  } while (status != 2);
  //--- release output by setting to zero
  chanzero->read(&zero); // (!)
  control->write(&zero);
}
```

The `switch(alt->select()) {}` clause performs the alternative construct. The lines marked with "*// (!)*" can be eliminated as part of optimization. This optimization increases the performance of the application by using variables instead of `ChannelVar` channels. This optimization requires further research and is not discussed in this paper. Anyway, CSP diagrams contain enough information to allow these kinds of optimizations.

3.5 Alignment controller process

The process-interface of `alignLH` is specified by the constructor below

```
VelocityControlLeftHorizontal::VelocityControlLeftHorizontal(ChannelIn<double>
*feedback, ChannelOut<double> *control, ChannelIn<double> *max)
{
  this->chanin  = new ChannelIn<double> * [1];
  this->chanout = new ChannelOut<double> * [3];

  chanstop = new ChannelVar<double>;

  chanin[0]  = feedback;
  chanout[0] = control;
  chanout[1] = max;
  chanout[2] = chanstop;

  vleftHorizontal = new VelocityControlLeftHorizontal20Process(chanin, chanout);
}
```

Figure 7b and some information of Figure 7a translates to the following process body.

```
void VelocityControlLeftHorizontal::run(void)
{
  do
  {
    vleftHorizontal->run();
    chanstop->read(&stop);
  } while (!stop);
}
```

3.6 Homing controller process

The CSP diagram in Figure 10 is translated to:

```
HomingHorizontal::HomingHorizontal(ChannelIn<double> *feedback,
ChannelOut<double> *control, ChannelIn<double> *leftmax, ChannelIn<double>
*rightmax)
{
  this->chanin  = new ChannelIn<double> * [4];
  this->chanout = new ChannelOut<double> * [2];

  chansetpoint = new ChannelVar<double>(0.0);
  chanstop = new ChannelVar<double>;
  chanzero = new ChannelVar<double>(0.0);

  chanin[0]  = leftmax;
  chanin[1]  = rightmax;
  chanin[2]  = feedback;
  chanin[3]  = chansetpoint;
  chanout[0] = chanstop;
  chanout[1] = control;

  homingHorizontal = new PositionControllerHorizontal20Process(chanin, chanout);
}

void HomingHorizontal::run(void)
{
  do
  {
    homingHorizontal->run();
    chanstop->read(&stop);
  } while (!stop);
  //--- release output by setting to zero
  chanzero->read(&zero);   // (!)
  control->write(&zero);
}
```

3.7 Timing and sampling

The CSP diagrams are based on untimed CSP. We have created an environmental process which takes part accepting every communication event in the software. We can command the environmental process to accept events at certain moment in time with periodical interval. This will be based on external channels. The environmental process will control a timer and on interrupts of the timer the drivers inside the external channels will be executed in sequence and at highest priority (atomic).

```
Environment::at(feedback_horizontal, starttime, Tsh);
Environment::at(joystick_horizontal, starttime, Tsh);
Environment::at(control_horizontal, starttime, Tsh);
Environment::at(feedback_vertical, starttime, Tsv);
Environment::at(joystick_vertical, starttime, Tsv);
Environment::at(control_vertical, starttime, Tsv);
```

The `starttime` is set to a value that is long enough for all initializations to be finished and until it waits for the first events to occur. Usually we set the `starttime` to `10000` (=10000 µsec, or 100 ms). The sampling time for the horizontal control loop `Tsh` and for the vertical control loop `Tsv` are set for the timed external channels. See @Tsh for the external channels `feedback_horizontal`, `joystick_horizontal`, and `control_horizontal` in Figure 3. When processes do not arrive on the timed channel before the environmental process want to accept the communication event, then a `TimeoutException` will be thrown by the channels and the processes will be released and terminate unsuccessfully. In this paper we omit exception handling. JIWY is hard real-time and timeout-exceptions do not occur so far. In further research we will investigate the exception framework under various kinds of errors in the system.

4. Conclusions

Guidance for transformation of the control engineering block diagram language to the control software processes and composition with other vital software components starts coming in sight. The CSP diagrams provide links for refinements of the substantial control system objectives towards complete runnable computer code and language for composing the computer code as a concurrent ensemble. The CT libraries catch the necessary constructs for building the compositional network structure, timing aspects and a CSP-consistent hardware access framework. The control code is generated from CSP diagrams and 20-SIM submodels.

This way, following an intuitive reasoning path, the interdisciplinary communication in development phases of a mechatronic project, is freed from serious design flow discontinuities.

References

[1] INMOS. Inmos Web Site, www.inmos.com,
[2] INMOS. occam 2 Reference Manual. C. A. R. Hoare *International Series in Computer Science*, Prentice Hall. 1988. ISBN 0-13-629312-3.
[3] A.W. Roscoe. The Theory and Practice of Concurrency. R. Bird *Series in Computer Sciences*, Prentice-Hall. 1998. 0-13-674409-5.
[4] G.H. Hilderink. Communicating Threads for Java (CTJ) home page, http://www.rt.el.utwente.nl/javapp, 2002.
[5] G.H. Hilderink, A.W.P. Bakkers and J.F. Broenink. A Distributed Real-Time Java System Based on CSP. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC-2000)*, IEEE Computer Society. Newport Beach, California**:** pp. 400-407, 2000.
[6] 20-SIM. Control Labs Products, www.20sim.com, 2003.
[7] P.C. Breedveld. Multibond graph elements in physical systems theory. In *J. Franklin Inst.* **319:** pp. 1-36, 1985.
[8] J.F. Broenink. 20-Sim software for hierarchical bond-graph/block-diagram models. In *Simulation Practice and Theory*. **7:** pp. 481-492, 1999.
[9] G.H. Hilderink. A Graphical Modelling Language for Specifying Concurrency based on CSP. In *IEE*

Proceedings Software, IEE. **150:** 108-120, 2002. ISSN 1462-5970.

[10] D.S. Jovanovic, G.H. Hilderink and J.F. Broenink. A Communicating Threads (CT) Case Study: JIWY. In V. S. Sunderam *Communicating Process Architecture 2002*, IOS Press. University of Reading, UK. **60:** pp. 311-320, 2002. ISSN 1383-7575.