HW/SW Design Space Exploration on the Production Cell Setup

Marcel A. GROOTHUIS and Jan F. BROENINK

Control Engineering, Faculty EE-M-CS, University of Twente,
P.O. Box 217 7500 AE Enschede, The Netherlands.

{M.A.Groothuis , J.F.Broenink} @utwente.nl

Abstract. This paper describes and compares five CSP based and two CSP related process-oriented motion control system implementations that are made for our Production Cell demonstration setup. Five implementations are software-based and two are FPGA hardware-based. All implementations were originally made with different purposes and investigating different areas of the design space for embedded control software resulting in an interesting comparison between approaches, tools and software and hardware implementations. Common for all implementations is the usage of a model-driven design method, a communicating process structure, the combination of discrete event and continuous time and that real-time behaviour is essential. This paper shows that many small decisions made during the design of all these embedded control software implementations influence our route through the design space for the same setup, resulting in seven different solutions with different key properties. None of the implementations is perfect, but they give us valuable information for future improvements of our design methods and tools.

Keywords. CSP, embedded systems, Mechatronics, motion control, real-time FPGA, Handel-C, gCSP, 20-sim, POOSL, Ptolemy II, QNX

Introduction

A typical mechatronic system design consists of a combination of a mechanical system, AD/DA, mixed-signal and power electronics, and an embedded control system (examples: cars, printers, robots, airplanes). The goal of the embedded control system is to control the behaviour of the mechanic system in a predefined way. The design of the embedded control system software for these mechatronic systems has a large design space with many multi-disciplinary factors that influence the route from idea to a working realization and the required amount of time to design these systems.

To find an optimal design for the embedded control software, we need to investigate (or explore) the various possible solutions in our design space. This paper describes our *Design Space Exploration* work on the Production Cell setup in our laboratory. This setup consists of a mechanical system with multiple axes that operate in parallel (see section 1.1 for more information). In the past few years we have made seven different designs for the (embedded control) software for this setup, exploring various possible solutions for implementing the software. Common for all these implementations, besides the setup, is the usage of a model-driven design flow and a process-oriented framework for the software that consists of a combination of an event-driven software part and a time-triggered software part. Five different CSP based implementations of the software were made and two other process-oriented, but non-CSP based, implementations were made shown in Table 1. All versions focus on different choices in the design space. This paper compares and evaluates all implementations from

Table 1 by looking at various aspects like computational resource usage, accuracy, amount of design time, the trade-off between CPU-based or an FPGA-based hardware implementation and the choice of real-time operating system.

Table 1. Production Cell Embedded Control System Software implementations.

Nr.	Name	Data type	Target	Explanation	Realization
A	gCSP RTAI	floating point	CPU	[1]	yes
B	POOSL	floating point	CPU	[2]	yes
C	Ptolemy II	floating point	CPU	[3]	yes
D	gCSP QNX	floating point	CPU	[4]	partial
E	gCSP Handel-C int	integer	FPGA	[5, 6]	yes
F	gCSP Handel-C float	floating point	FPGA	[7]	yes
G	SystemCSP	-	-	[8]	no

Section 1 contains background information on the Production Cell setup, embedded control system software, our design method and the languages and tooling used. Section 2 describes the important aspects for an embedded control system software design and how to design choices influence our route through the design space. Sections 3 and 4 describe the CPU and FPGA designs for the production cell setup followed by an evaluation and discussion in section 5 and conclusions and ongoing work in section 6.

1. Background

This section contains a brief description on the Production Cell demonstration setup, followed by a description of the structure of embedded control system software and the design method we use to design the software. The last subsection introduces the languages and tools used for the implementations from Table 1.

1.1. Production Cell Setup

The setup that is used for all process-oriented controller implementations described in this paper is a mock-up of an industrial production line system (in our case a plastics moulding machine). The production cell setup [5, 9] is a circular system that consists of 6 production cells that operate simultaneously and semi-independently. Each of these cells, called *Production Cell Units* (PCUs), executes a single action in the production process. Figure 1 shows an overview of the setup used. Its main goal is to pass along metal blocks and to execute (pseudo)actions on these block like *moulding*, *extraction* from the machine, *transportation* (belts) and *storage*. The storage part is simulated by a *rotation* unit that picks up a block at the end of the production process and transfers it again to the beginning of the setup, resulting in a loop. Sensors (located before and after the PCUs) detect the blocks and generate external events for the PCUs, so the PCUs can perform their jobs. The loop in combination with the sensor-event-triggered PCUs can result in a deadlock when the setup contains 8 or more blocks. The setup needs at least one free buffer position (located next to the sensors) in order to be able to move blocks to the next PCU. When all sensor guarded buffer positions are occupied, the system cannot move anymore resulting in a deadlock. The mechanical setup is connected via power and interface electronics to an embedded PC with an FPGA I/O card, that runs the embedded control software.

1.2. Embedded Control System Software

The combination of a mechanical setup and embedded (control) software for motion control systems and robotics requires a multi-disciplinary and synergistic approach for its design. The

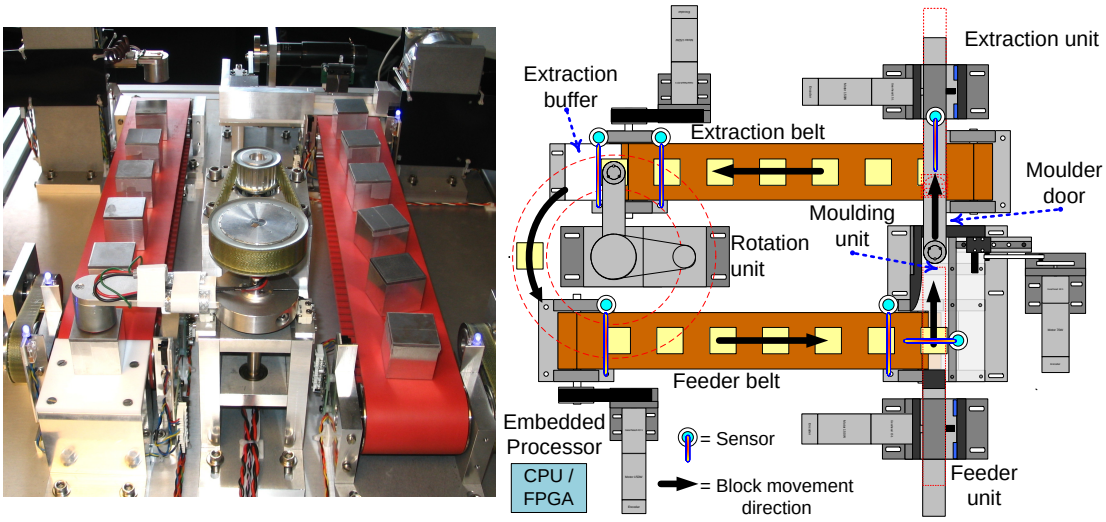

Figure 1. The Production Cell setup.

dynamic behaviour of the mechanics influences the behaviour of the software and vice-versa. Therefore they should be designed together to find an optimal and dependable realization for the entire mechatronic setup.

The purpose of an embedded control system is to control physical processes (like mechatronic setups). For this paper, the purpose is to control and co-ordinate mechanical movements (like position, velocity and acceleration) to get a smooth and precise movement. A typical embedded control system software design contains often a layered structure [10] with layers for: *user-interfacing*, *supervisory control*, *sequence control* (order of actions), *loop control* (control law and motion profiles), *safety* purposes and *measurement and actuation*. The embedded control system software is a combination of an event-driven part and a time-triggered part with different and often challenging (real-)time requirements for the different layers. Hard real-time behaviour is for example required for the last two layers. The control laws for the loop control layer require a periodic time schedule with hard deadlines in which jitter and latency are undesirable.

For the Production Cell setup, the sequence control, loop control and safety layers are essential to the implementation.

1.3. Design Method

The design method used for designing the embedded control system (ECS) software for mechatronic systems is based on model-driven design with a close cooperation between the involved disciplines. Concurrent design techniques are used to shorten the total time from idea to realization. The loop controllers are, for example, designed concurrently to the other ECS software layers [11]. In the design flow for designing the loop controllers, the starting point is a *physical system model* (a model of the mechanical setup). From this model, the control engineer derives the required *control algorithm*, based on the assumption of continuous time and floating point calculations and verified by simulation in, for example 20-sim [12]. The next step is to *incorporate target behaviour* (discrete time, AD/DA effects, signal delays and scaling) via *stepwise refinement* into the design before the loop controllers can be integrated in the ECS software design.

Concurrently, the other ECS software layers are designed, starting from an *abstract top-level model* that is extended via stepwise refinement into a complete ECS software design. In order to prevent integration problems of the software, the control laws and the setup, co-simulation tests between the software and the physical system model can be performed as early integration tests [11].

1.4. Used Tools and Languages

The production cell implementations from Table 1 were made using various modelling and implementation languages and tools. This section introduces them briefly.

The *20-sim* modelling and simulation tool (commercial) is used for implementations *A, B, D* and *F* to model the dynamic system (the mechanical part) and to design the control laws and motion profiles (the trajectory to follow) for the axes movements.

The embedded control system design models for implementations *A, D, E* and *F* were made using our graphical CSP tool (academic), *gCSP* [13], based on the GML language (graphical notation for CSP) [14]. gCSP diagrams contain information about compositional relationships (SEQ, PAR, PRI-PAR, ALT and PRI-ALT) and communication relationships (rendezvous channels). The tool supports animation/simulation [15] of these diagrams and code generation of CSPm code (for deadlock and livelock checking with FDR2 or ProBE), Occam code, C++ code (using the CTC++ library (implementation *A*) [16]) and Handel-C code.

Implementations *E, F* use the Handel-C [17] (commercial) hardware description language to implement the CSP based embedded control system in an FPGA.

The ECS software for implementation *B* is made using the Parallel Object Oriented Specification Language (POOSL) [18] in combination with the Shesim simulator and the Rotalumis execution engine for POOSL (academic). The POOSL language has a formal background based on timed CCS [19].

Implementation *C* is entirely modelled in *Ptolemy II* [20]. Ptolemy II is a heterogeneous modelling and simulation tool (academic) that allows for creating multi-domain models using different Models of Computation (MoC) in a hierarchical model structure, consisting of *actors* (comparable to submodels or processes) and *directors*. The director determines the domain and the model of computation that is used by the simulator for executing an actor. Communication between actors takes place via channels connected to ports. Each port uses a receiver that determines the exact behaviour (FIFO, mailbox or CSP rendezvous) of a channel, according to its domain. Examples of Ptolemy II domains that can be used to model the Production Cell and its ECS software layers are the *continuous-time* (CT), *discrete-event* (DE), *synchronous dataflow* (SDF), *rendezvous/CSP* and *finite state machines* (FSM) domains.

Implementation *G* is modelled using the SystemCSP language. SystemCSP is based on the principles of both component-based design and CSP process algebra, offering a more structured approach and more expressiveness than the occam-like GML approach used by gCSP [8].

2. Design Space Exploration

The optimal design of embedded control software that is flexible, dependable, cost-efficient and takes into account all kind of functional and non-functional constraints, like real-time requirements and time-to-market, is complex and has a large design space. The design pyramid in Figure 2 shows that an idea can be realized in many ways.

During the route from idea to final realization, many design decisions need to be made which all have their own influence on the final result. With realization we mean the software implementation that runs on the setup, so the total system including the embedded control system software. Every decision restricts the design space and starts a new smaller design pyramid. The reachable solutions (feasible design space), whether optimal or not, depend on all these decisions. For example, the architecture choice between a CPU (a) or an FPGA (b) results in different sub-design space with different feasible solutions. Typical decisions for the Production Cell setup that influence the final realization are:

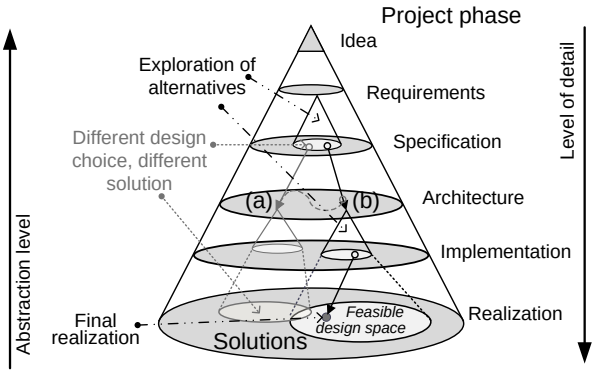

Figure 2. Design Pyramid with different abstraction levels (adapted from [21]).

- Choice of the modelling formalisms and languages;
- Operating system choice: general purpose or dedicated (real-time) operating system;
- Architecture trade-offs: CPU ⇔ FPGA, distributed ⇔ centralized design, parallel ⇔ sequential design;
- FPGA solution: use natural parallelism (high resource usage) ⇔ sequential solution (lower resource usage) and resource usage ⇔ design time.

The next two sections describe the CPU and FPGA based embedded control system implementations followed by an evaluation of the design choices and their effect on the realization.

3. CPU Implementations

This section describes all embedded control system *software* implementations from Table 1 that run on a embedded PC/104 platform with 600 MHz X86 CPU equipped with an FPGA based digital I/O board that connects to the Production Cell setup.

3.1. gCSP RTAI (Implementation A)

The *gCSP RTAI* implementation is the first completely working embedded control system (ECS) software implementation for the Production Cell setup. The ECS software structure is modelled in and generated from gCSP and manually combined with the loop controllers and motion profiles that are modelled in and generated from 20-sim. The compiled code runs under RTAI real-time Linux [22]. The focus of this implementation was a proof-of-concept for gCSP in combination with its CTC++ library in an environment that requires real-time guarantees. LinkDrivers [13] are created and used to provide channel communication with the hardware. In order to provide the required periodic timing behaviour (for the loop controllers) to the (untimed) CSP program, *TimerChannels* are used to synchronize the controller processes with the OS timer (an environmental process that provides periodic tocks [23]). The layered software structure (section 1.2) is implemented using prioritized PAR constructs to be able to prioritize the loop controller above the other layers. The gCSP RTAI version is based on a bottom-up design approach starting with the hardware drivers and loop controllers and extended via a single PCU implementation towards a complete CSP based embedded control software implementation. Figure 3 shows an example of the top-level gCSP model.

This work proved that gCSP and CSP processes and channels are usable and suitable to create ECS software that is formally verified. Integration of external 20-sim code was straightforward due to the usage of a special "20-sim-to-gCSP-process" code generation template. The final realization worked reasonably well, but showed serious timing problems (missed deadline) when many (>15) blocks in the system. The generated code in combina-

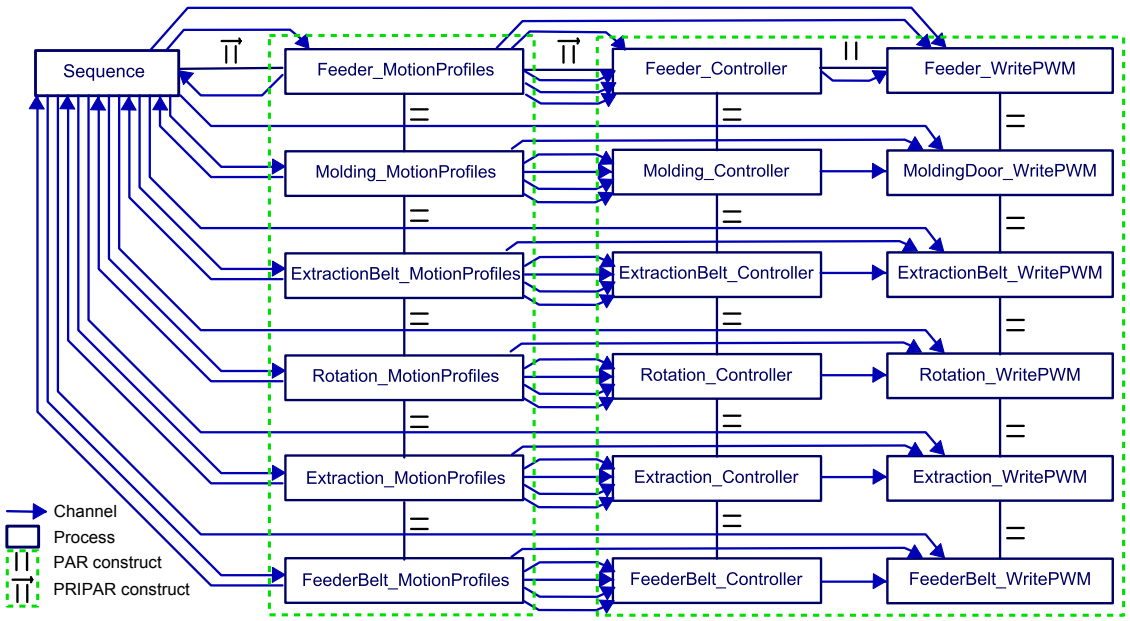

Figure 3. gCSP Production Cell software top-level model.

tion with the CTC++ library shows quite some process switching overhead (many small processes), resulting in a high CPU load. This is now partially solved via optimizations in the CTC++ library. This implementation revealed some serious shortcomings in the gCSP graphical modelling (GML) language and the gCSP tool namely the lack of support to draw state machine constructions for implementing a sequence controller and a (currently) incomplete CSPm code generator. As a consequence, the formal verification is limited to the drawn process network. Contents of code blocks (non graphical processes) cannot be checked directly without writing a corresponding CSPm implementation by hand. Another missing feature was the ability to simulate (animate or graphically debug) the created gCSP process network to see its (time-dependent) scheduling behaviour. This was solved last year with the implementation of a gCSP animation facility [15], which allows for visual debugging of the process network using colours for the channels and processes (as depicted in Figure 3) to indicate the status of all processes and channels (e.g. for processes: blocked, running, finished and for channels: free, reader waiting, writer waiting, rendezvous). Furthermore, the animation framework allows for setting breakpoints on processes and it shows the contents of the CSP scheduler queues.

3.2. POOSL (Implementation B)

The non-deterministic timing behaviour of the system under load for the gCSP-RTAI implementation was the starting point of the POOSL implementation [2]. One of the strong features of the POOSL language is its predictable timing behaviour (with formal background, see [24]). The POOSL version is made using a top-down design approach starting with a top-level discrete-event *concurrency model* for the process interactions in the system. This model is extended via stepwise and local (within process) refinement towards a *multi-model-of-computation model* (discrete-event (DE) and continuous-time (CT) equations) that is still untimed. The last refinement is the addition of (real-)time information to the discrete event part of the model and to integrate the continuous time parts. The latter runs in parallel with the event-driven part and generated from 20-sim using a POOSL code generation template. This results in a *real-time model*. The channel interaction of the PCUs is explicitly specified by using two-way, three-way and four-way handshaking pattern for the PCU synchronization.

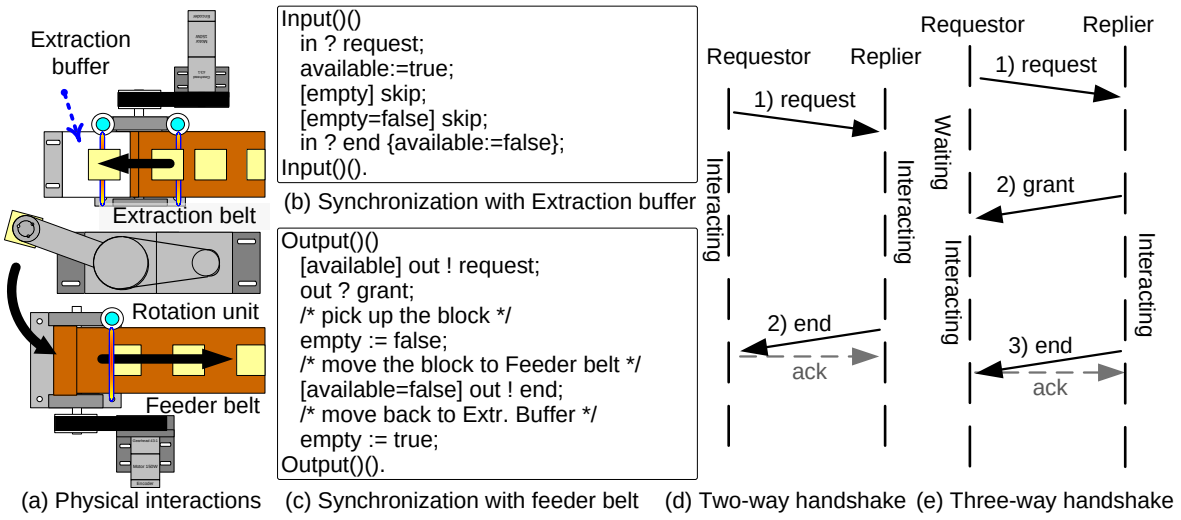

Figure 4. Handshake synchronization in POOSL (left) and schematic (right).

See Figure 4 for a schematic and POOSL code example for the *Rotation* unit synchronization (Input()() and Output()() side run in parallel).

The behaviour of the (untimed) discrete event ECS part was completely verified via simulation in Shesim. This revealed also the possible > 8-blocks deadlock in the Production Cell system. Although formal verification for POOSL models is possible, using for example UPPAAL [25], it is not used here, because the translation is not yet automated.

Compared to the gCSP implementation in section 3.1, the Shesim simulator allowed us to better predict and verify the behaviour of the software (discrete event + continuous time part). The top-down refinement approach allowed us to design the discrete event (DE) and the continuous time (CT) part separately and to integrate them later, by only specifying the required DE-CT interactions and the DE-CT channel interfaces on beforehand. The Rotalumis POOSL execution engine, used for the final implementation on the setup, did not allow us to run it in a real-time environment (RTAI real-time Linux in our case). This is a major shortcoming for this implementation. The highest priority Linux process was the best we could achieve, so we could not give any real-time guarantees. The timing behaviour was however surprisingly stable and predictable (under the given non-real-time conditions), which allowed us to run our loop controllers without too much trouble (no stability problems), even under heavy load (about 15 blocks in the system). A minor issue with the POOSL language is that it did not allow us to specify process priorities. In case of a high system load, the loop controllers are, for example, more important than the graphical user interface.

3.3. Ptolemy II (Implementation C)

The focus of the Ptolemy II implementation was on the feasibility of a single tool solution for modelling the dynamic system behaviour, the motion profiles and control laws and the embedded software in one single tool and model. Essential for our model-driven design flow is the requirement for code generation, preferably without manual adaptations in order to run it on the target. As all other implementations require multiple tools and models and often manual integration of generated code from these models, the Ptolemy approach can significantly reduce the integration effort and the required design time.

The Ptolemy II implementation is based on a top-down approach design approach, similar to the POOSL approach, but now with the entire mechatronic system as top-level and the embedded control system as one of its components. The relevant behaviour of the setup (mechanics, electronics, embedded control system software and even a 3D model with the kinematic behaviour) are modelled via local stepwise refinement in one single Ptolemy II

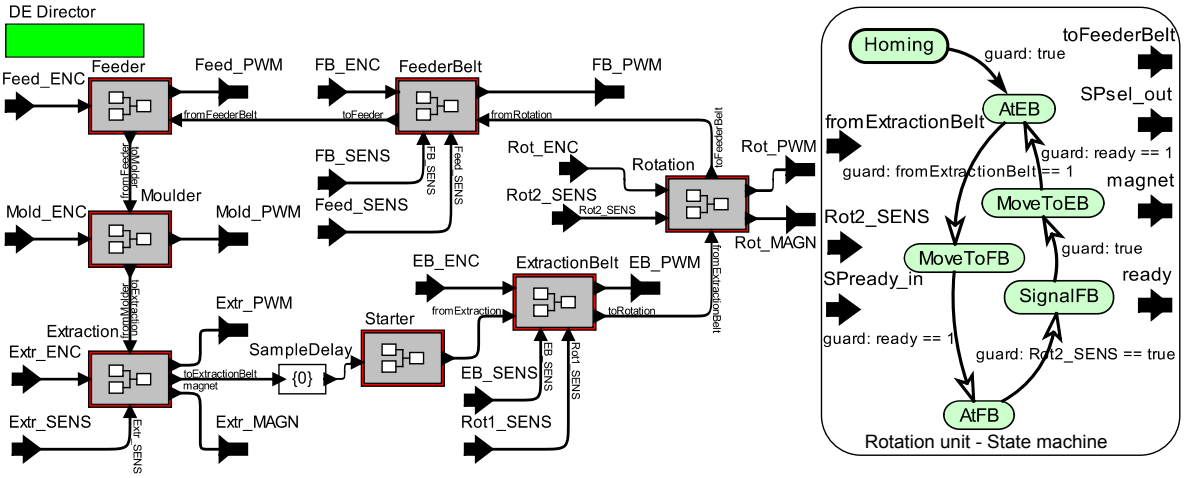

Figure 5. ECS top-level implementation in Ptolemy II and a state machine example.

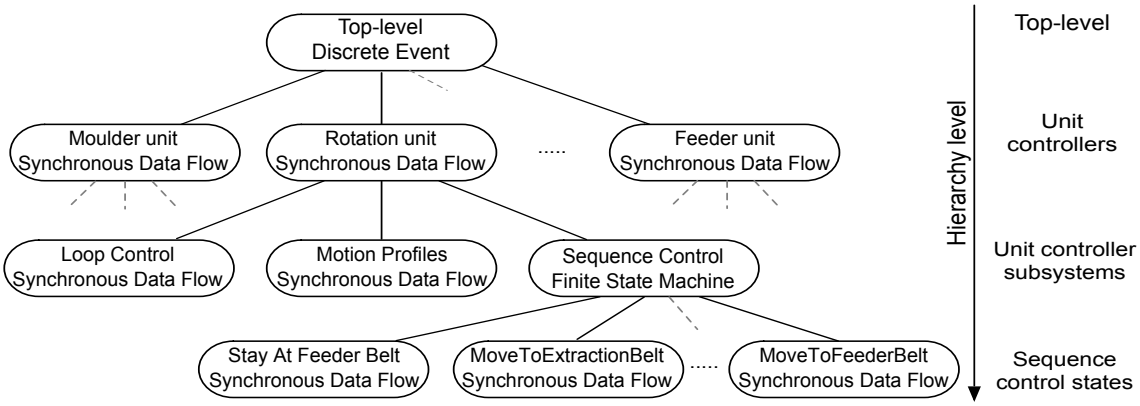

Figure 6. ECS hierarchy and used models of computation.

model as multiple (composite) actors (processes). These actors are implemented using multiple different domains (and the corresponding models of computation). Figure 5 shows the structure of the embedded control system actor and an example of the *Rotation* unit state machine, which we could not implement graphically in gCSP yet. Figure 6 shows the different models of computation that are used for the embedded control system software layers, zoomed in on the *Rotation* unit. Formal verification via automated exports to model checkers for the various Ptolemy II domains is not yet possible. The entire setup, including the ECS behaviour, was verified by simulations in Ptolemy II. The embedded control system software implementation was generated from Ptolemy as ANSI-C code.

Not all available domains in Ptolemy II are mature enough for practical use, as the tool is still under development. The Ptolemy II CSP/Rendezvous domain is, compared to gCSP, of limited use. It has no support for SEQ, no priority support and no code generation support yet. In order to model the dynamic behaviour of the setup in the continuous time (CT) domain, we had to use transfer function equations, instead of graphical diagrams like “ideal physical models” or bond-graphs in 20-sim. We have extended the Ptolemy II library with self developed Ptolemy building blocks in order to implement motion profiles and PID controllers. For the final implementation, an extension to the ANSI-C code generation facility was necessary in order to get the required real-time behaviour under a real-time operating system (in our case RTAI Linux). On the graphical modelling level, the channel communication through ports between different models of computation was not straightforward. The ports only transfer data, but they do not specify sampling behaviour at the boundary of continuous time and discrete time, which can result in unexpected behaviour unless the modeller

adds these required "conversion" actors to the model by hand.

The all-in-one model approach proved to be time-saving and easy for early integration testing, however Ptolemy II needs quite some extensions to be able to use it for a real setup like the Production Cell.

3.4. gCSP QNX (Implementation D)

The QNX real-time operating system [26] is a POSIX compliant microkernel based operating system with advanced scheduling capabilities and extensive run-time tracing and profiling capabilities dedicated for the development of deterministic systems with hard real-time demands.

Because we had some serious timing problems with the existing non-pre-emptive CTC++ library (see also section 3.1), the focus of this ECS design was mainly on the creation of a QNX version of our CTC++ library that is API compatible with our existing Windows/(RTAI)Linux/DOS CTC++ library, but now with a good timing foundation. The mapping of CSP constructs, processes and channels onto QNX proved to be straightforward. The PAR and PRIPAR constructs are, for example, implemented using QNX POSIX threads in combination with prioritized pre-emptive scheduling and the QNX channels (message passing) are used to provide CSP channels. To allow transparent distributed processing, QNX provides its own distributed networking protocol *QNET*. The QNX CTC++ library uses QNET to provide network channels. A TimerChannel and time-out guard are implemented to provide timing (periodic and time-outs) to CTC++ processes by letting them synchronize with a (discrete) *tock* event from the environment (the operating system timer). The QNX *adaptive partition scheduler* is used to guarantee a set of processes (partition) an amount (e.g. 80%) of CPU cycles. For more information about this library, see [4].

The current gCSP QNX ECS design is not complete. It only contains the *Rotation* unit and its interactions (by coincidence the part shown in Figure 4). This partial design was generated from gCSP to test the backward compatibility of the new library. Oscilloscope measurements and instrumented QNX kernel traces provided exact scheduling and timing information on our embedded control system software design. The required timing was reached with almost no jitter (2 μs for a 1 ms period). The extensive traces revealed also that the operating system overhead for channel communication and thread switching is relatively high (140 μs) compared to our process calculation times (70 μs). So, although the timing is reliable, the mapping of (small) processes onto operating system threads needs some optimizations to be useful for the entire setup.

3.5. SystemCSP (Implementation G)

The SystemCSP design of the ECS software for the Production Cell setup can be found in chapter 6 of [8]. This design is made for demonstration purposes of the SystemCSP language. This design is *not* implemented on the Production Cell setup, because a tool and an execution engine for the SystemCSP language do not yet exist. However, the SystemCSP design in [8] does show how the ECS software can be modelled using the SystemCSP notation. The GML language has no support for drawing state machines (related to CSP primitives), which is possible in SystemCSP. An example of a graphical CSP based state machine construction with interactions (CSP events) in SystemCSP is shown in Figure 7.

4. FPGA Implementations

Our Production Cell setup consists of several Production Cell units running in parallel and because deterministic real-time timing behaviour is important for an embedded control sys-

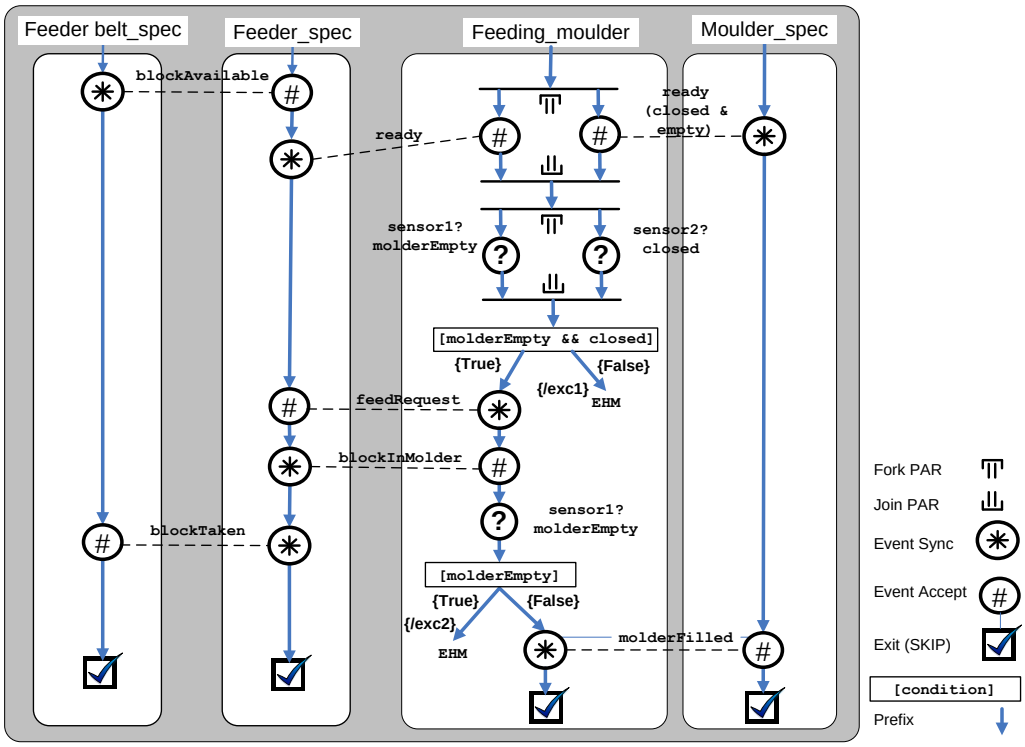

Figure 7. SystemCSP interaction contract between feederbelt, feeder and moulder (adapted from [8]).

tem, this requires a careful choice of the scheduling algorithm used to emulate the parallelism in software on a sequential processor, while keeping the observable timing behaviour deterministic. Because the Production Cell setup is also equipped with an FPGA based digital I/O board and FPGAs can provide deterministic timing and native parallelism, we made also two completely FPGA-based embedded control system implementations: *E* and *F* from Table 1. The purpose of these implementations was to investigate the feasibility of an FPGA-based solution and to look at the trade-off between a CPU-based and an FPGA-based solution. This section describes the embedded control system *hardware* implementations.

4.1. gCSP Handel-C integer (Implementation E)

The *gCSP Handel-C integer* version of the Production Cell motion control software was the result of a feasibility study on FPGA based motion control (see for more information our CPA 2008 paper [5]). The main characteristics of this implementation are:

- FPGA choice: exploit inherent parallelism and accurate timing; no usage of a soft core CPU;
- Usage of Handel-C as hardware description language;
- Design of a decentralized process-oriented layered structure and communication framework for motion control (see Figure 9, Figure 8 and [5]). FDR2 was used to check that the framework is free of deadlocks;
- ECS framework designed in gCSP (see Figure 8). Implementation generated using Handel-C code generation;
- Control laws and motion profiles designed in 20-sim (floating point). Integer used as native data type for motion profile and loop controller implementation;
- Implementation on a low cost Xilinx Spartan III 3s1500 FPGA;
- Integer PID loop controllers run at 1 ms with idle time of 99.95%;
- Combination of top-down design (ECS framework) and bottom-up design (PID loop controllers).

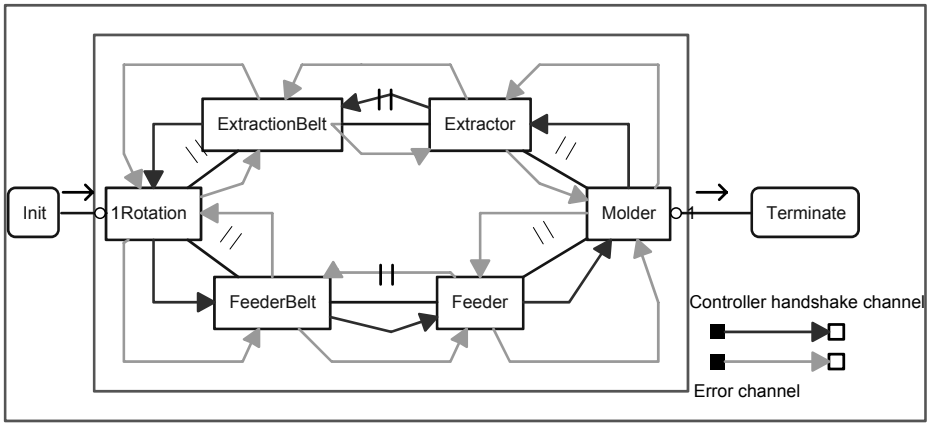

Figure 8. gCSP Handel-C top level (from [5]).

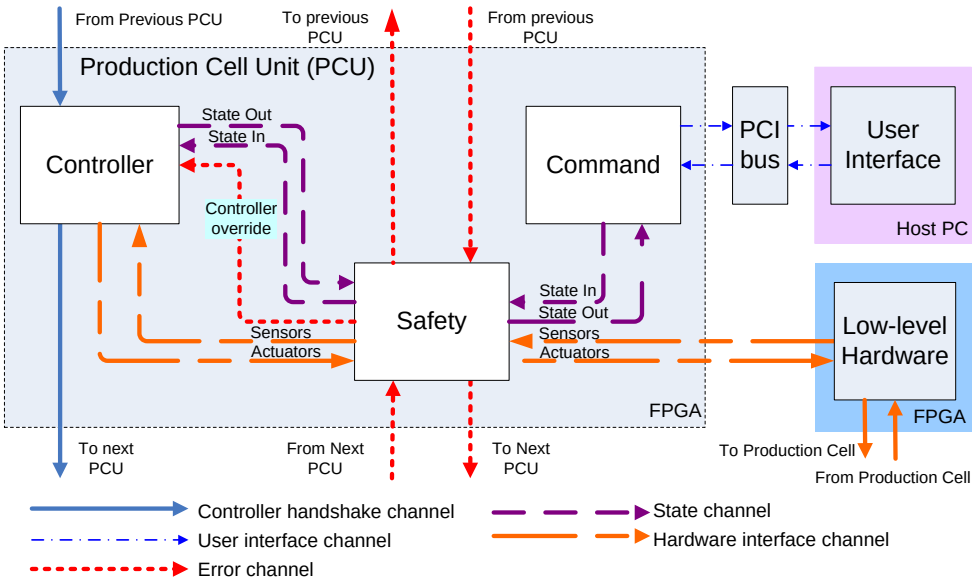

Figure 9. Production Cell unit structure (from [5]).

The result is a completely working and successful FPGA based embedded control system realization for the Production Cell with a much better performance with respect to the timing accuracy and the system load than all CPU based realizations. This implementation fits also in a relatively small Spartan III FPGA. The FPGA resource usage is measured in the amount of internal logic blocks (lookup tables (LUTs, flip-flops (FF), memory (MEM) and arithmetic logic units (ALUs)) that are needed for the implementation of the design. Table 2 shows the resource usage on a Xilinx Spartan III 3s1500 FPGA for this design. The designed software framework structure turned out to be useful for embedded control system software for this class of mechatronic systems. The same structure is also used for the Ptolemy II and the gCSP QNX version (sections 3.3 and 3.4).

The main disadvantage of the integer implementation was the large design time required for manual translation of the motion profiles and loop controllers from a floating point implementation (20-sim) towards an integer implementation. This route does not directly fit into our existing (floating point based) model-driven design flow.

4.2. gCSP Handel-C floating point (Implementation F)

The design gap between a floating point loop controller design and an integer based loop controller implementation in the previous implementation (section 4.1) is rather large and

Table 2. Estimated FPGA resource usage for the integer version (adapted from [5]).

Element	LUTs	(amount)	Flipflops	(amount)	Memory	Used ALUs
PID controllers	13.5%	(4038)	0.4%	(126)	0.0%	0
Motion profiles	0.9%	(278)	0.2%	(72)	0.0%	0
I/O + PCI	3.6%	(1090)	1.6%	(471)	2.3%	0
S&C Framework	10.3%	(3089)	8.7%	(2616)	0.6%	0
Free	71.7%	(21457)	89.1%	(26667)	97.1%	32

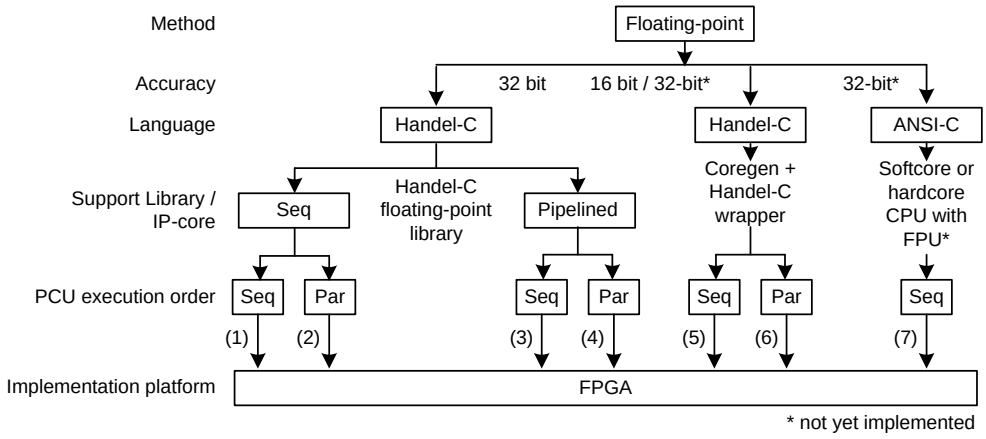

Figure 10. Routes for floating point on an FPGA (adapted from [7]).

requires more design iterations to ensure correct behaviour and a stable loop controller. The choice for the integer datatype for FPGAs is a logical choice because it is the native datatype. However, from a model-driven design flow point of view (including the usage of code generation) a floating point FPGA implementation for the loop controllers is preferable.

A quick and naive attempt to use floating point, in combination with the Handel-C floating point library, during the design of the integer version resulted in a FPGA resource usage explosion and an implementation that did not fit in the used Spartan III FPGA. This was mainly due to our choice to fully exploit the FPGA's parallelism. At the same time, there was plenty of idle time left in each loop controller calculation loop (idle time 99.95%, period 1 ms) to do other things. Handing in some of the parallelism, frees FPGA resources that can be used for a floating point implementation. There are trade-offs between FPGA resource usage, amount of parallelism, calculation time, design time and loop controller calculation accuracy. The focus of this Production Cell design was to investigate these trade-offs to see if we could fit a floating point implementation in this FPGA.

Figure 10 shows the possible implementation routes for a floating point FPGA implementation of the Production Cell motion profiles and loop controllers:

1. Sequential Handel-C floating point library with sequential PCU execution;
2. Sequential Handel-C floating point library with parallel PCU execution;
3. Pipelined Handel-C floating point library with sequential PCU execution;
4. Pipelined Handel-C floating point library with parallel PCU execution;
5. Using a 16-bit floating point core from Xilinx Coregen with parallel PCU execution;
6. Using a 16-bit floating point core from Xilinx Coregen with sequential PCU execution;
7. Soft-core or hard-core CPU with floating point unit.

Route 1, 3, 6 and 7 use sequential PCU execution which means that the controllers for each PCU run after another and share the same PID loop controller and motion profile generator FPGA processes, only using different parameters. The necessary motion profiles

Table 3. Test results for floating-point loop controller implementation routes 1-4 (MP = motion profile).

Route	LUT	FF	Mem	ALUs	LUT %	Mem %	Fits	t_{calc}	t_{loop} %
(1) MP in blockram	5669	3825	671744	4	21.29	113.89	-	-	-
(1) *MP during runtime*	*6385*	*4531*	*0*	*4*	*23.98*	*0.00*	•	*41.54 μs*	*4.15*
(2) MP in blockram	32530	21492	868352	24	122.18	147.22	-	-	-
(2) MP during runtime	33967	22648	0	24	127.58	0.00	-	6.92 μs	0.69
(3) MP in blockram	6508	4127	671744	4	24.44	113.89	-	-	-
(3) *MP during runtime*	*6816*	*4818*	*0*	*4*	*25.60*	*0.00*	•	*38.41 μs*	*3.84*
(4) MP in blockram	31181	21937	868352	24	117.12	147.22	-	-	-
(4) MP during runtime	32407	23792	0	24	121.72	0.00	-	6.4 μs	0.64
Maximum	26624	26624	589824	32				$< 1000\mu s$	

Table 4. Estimated FPGA usage for the floating point version.

Element	LUTs	(amount)	Flipflops	(amount)	Memory	Used ALUs
Floating point library + wrapper	27.4%	(8191)	19.7%	(5909)	0.0%	4
PID controllers	4.2%	(1251)	0.3%	(91)	0.0%	0
Motion profiles	1.1%	(314)	0.5%	(163)	0.0%	0
I/O + PCI	4.1%	(1250)	1.8%	(534)	2.3%	0
S&C Framework	5.6%	(1666)	4.2%	(1250)	0.3%	0
Free	57.6%	(17280)	73.5%	(22005)	97.4%	28

provide the loop controller with a predefined trajectory (position, speed and acceleration set-points) that the PCU axes should follow. They can be calculated at runtime or stored (hard-coded) in FPGA blockram, resulting in a trade-off between FPGA resources (ram or LUTs). The Handel-C floating point library supports pipelined calculation and sequential calculation which results in another resource usage optimization possibility. Another route (5,6) for optimization is to lower the floating point precision from 32 bit to 16 bit at the cost of calculation accuracy. This is not possible with the Handel-C floating point library, but it is possible to generate a similar core with the Xilinx Coregen tool. The resource usage results of the different floating point possibilities are given in Table 3, which is based on the result from [7]). This table does not show routes 5 and 6 because it turned out that the generated 16-bit Coregen floating-point core was already bigger than the 32-bit Handel-C library. These routes were not further investigated. Route 7 is also not shown because it is still future work. The usage of a soft-core requires external RAM, which is not available on our Production Cell FPGA board.

Table 3 shows only two feasible routes (italic) for the given FPGA: 1 and 3 with motion profile calculation during runtime. The result from Table 3 contain only 6 controllers and 6 motion profiles and not a complete implementation. The complete FPGA embedded control system with floating point controllers is implemented based on route 1. The resource usage results of this version are given in Table 4. This version is around 50% larger than the integer version (implementation *E*), but it has a better calculation accuracy, resulting in a slightly smoother movement. A disadvantage of sharing the PID loop controller and motion profiles with all PCUs is that it requires more effort to use it together with the PCU structure from Figure 9.

A minor disadvantage of the Handel-C floating point library in combination with C code generation is that it is not ANSI-C compatible, because the Handel-C floating point library uses functions instead of standard operators and it uses a floating-point structure, containing the integer representation of the sign, exponent and mantissa separated by commas, instead of a float datatype, also see Table 5. These differences require manual changes in the generated code from 20-sim or changes in the 20-sim code generator.

Table 5. ANSI-C float versus Handel-C float declaration.

ANSI-C	Handel-C
`float a = 0.007;`	`Float a = {0, 119, 6643777}; //sign, exponent, mantissa`
`float b = -0.31;`	`Float b = {1, 125, 2013265};`

5. Evaluation

All presented Production Cell embedded control system (ECS) software implementations were made after each other by different people with a different amount of experience. Some results from one version were used for the others (e.g. the 20-sim controllers). Furthermore, the used tools (mainly academic) have a different maturity for our purpose (ECS software). This makes it difficult to give a precise and fair comparison of all these approaches when looking at the required design time, which operating system to choose and which tool or method is the best. However it is possible to give global observations and guidelines for future embedded control system implementations based on the presented methods, tools and the setup used.

Common for all CPU and FPGA implementations is a hierarchical process-oriented implementation. The (CSP) process abstraction together with a layered structure and standardized building blocks (like Figure 9) is perfectly suitable for (ECS designs with a combination of discrete event and continuous/discrete time parts.

Accurate timing is essential for real-time ECS software, however the combination of (untimed) CSP and timing poses some questions about how to implement this in practice (see also [8]) to get an efficient and deterministic timing realization. The POOSL language (implementation B) can provide accurate timing, but without real-time guarantees, the implementation is of little value at the moment. Implementation *A* uses our (existing) CTC++ library without pre-emption support and using user-level threading. The timing accuracy here is limited to the channel communication frequency (scheduling is only possible on channel communication). The new QNX CTC++ library, made for implementation *D*, does use pre-emptive scheduling and provides more accurate timing, but on the other hand, the usage of operating system threads results in a much larger context switch overhead, especially with many channel communications. When we compare all implementations, we see that they all contain many small processes with multiple channels to the same neighbour to communicate small amounts of data (simple variables). To become more resource efficient it is necessary to turn these channel communications into bus transfers or to send multiple variables with a single write action. Furthermore, the small processes should be combined into larger ones with the same behaviour while translating the model into an implementation.

The usage of formal checking of the created (graphical) models in combination with automated model-to-formal-language translation reveals that none of the used approaches can currently provide an intuitive and user-friendly way of using formal methods to ensure the correctness of the designed ECS structure. It is possible for implementations *A, B, D, E, F*, but not yet without manual translation or adaptation/extension of (gCSP) generated formal descriptions (CSPm).

The Ptolemy II all-in-one tool approach is promising with respect to shortening the design time and doing early integration, but this academic tool is not yet mature enough for daily usage in the mechatronics field.

The FPGA implementations provide an interesting alternative for the commonly used CPU-based embedded control system implementations in industry, especially when accurate timing and more parallelism than CPU-based solutions is required. The FPGA implementation allow for a single chip solution, containing both the ECS “software” and the required (digital) I/O hardware for actuation and sensing. It also allows us to reach faster reaction times than possible on X86 PCs. The main disadvantage of an FPGA based ECS implement-

ation is the required design effort. The design gap between model-driven ECS design and an FPGA implementation is rather large with the current tooling, especially for implementation E (integer), where our floating point based controllers needed to be translated into an integer implementation. Implementation F (floating point) makes this design gap smaller, at the cost of additional FPGA resources which may require again a sequential implementation, a larger FPGA or an FPGA with DSP blocks (e.g. the Xilinx Virtex series) that can be used for floating point calculations.

6. Conclusions and Future Work

The comparison of different design methods and tools for embedded control system software (ECS) for the same setup gives us growing insights in the maturity of the used design tools, that have mainly an academic background, for ECS software design and realization. The different ways of designing the process-oriented ECS software lead to a standardized layered structure which we can add a building blocks into a (g)CSP library.

Having both software and hardware realizations of the ECS "software" for the same setup provides us with useful information about the design trade-off between a CPU-based and an FPGA-based solution. The FPGA solution requires more design time but it can provide accurate timing without the usage of a real-time operating system.

The comparison of all ECS realizations shows that many small decisions made during the design of all these realizations influence our route through the design space, resulting in seven different solutions with different key properties. None of the realizations is perfect, but they give us valuable information for future improvements of our design methods and tools. We are currently working on version 2 of gCSP with suggested improvements like state machines and language elements from SystemCSP and with a better CSPm translation. We are working also on an extended version of the ECS software framework from Figure 9 to incorporate also vision processing and other Human-Machine-Interface (HMI) features for the usage in our Humanoid soccer robot and our robotic head setup.

Acknowledgements

We would like to thank our former MSc students Bert van den Berg, Pieter Maljaars, Kees Verhaar, Jasper van Zuijlen, Bart Veldhuijzen and Thijs Sassen for their final MSc project contributions on the Production Cell setup, its software and hardware motion control implementations. Furthermore, we would like to thank our ViewCorrect project partner and colleague Jinfeng Huang from Eindhoven University for the joint work on the POOSL implementation of the Production Cell control software.

References

[1] P. Maljaars. Controllers for the production cell set up. MSc thesis 039CE2006, Control Engineering, University of Twente, The Netherlands, December 2006. URL `http://www.ce.utwente.nl/rtweb/publications/MSc2006/pdf-files/039CE2006_Maljaars.pdf`.

[2] Jinfeng Huang, Jeroen P. M. Voeten, M.A. Groothuis, J.F. Broenink, and Henk Corporaal. A model-driven approach for mechatronic systems. In *Seventh International Conference on Application of Concurrency to System Design, 2007, Bratislava, Slovakia*, pages 127–136, Los Alamitos, July 2007. IEEE Computer Society Press. ISBN 978-0-7695-2902-8. doi: 10.1109/acsd.2007.40.

[3] C. A. Verhaar. An integrated embedded control software design case study using Ptolemy II. MSc thesis 011CE2008, Control Engineering, University of Twente, The Netherlands, May 2008. URL `http://purl.org/utwente/e58154`.

[4] Bart Veldhuijzen. Redesign of the CSP execution engine. MSc thesis 036CE2008, Control Engineering, University of Twente, February 2009. URL http://purl.org/utwente/e58514.

[5] M.A. Groothuis, J. J. P. van Zuijlen, and J.F. Broenink. FPGA based control of a production cell system. In *Communication Process Architectures 2008, York, United Kingdom*, volume 66 of *Concurrent Systems Engineering Series*, pages 135–148, Amsterdam, September 2008. IOS Press. ISBN 978-1-58603-907-3. doi: 10.3233/978-1-58603-907-3-135.

[6] J. J. P. van Zuijlen. FPGA-based control of the production cell using Handel-C. MSc thesis 008CE2008, Control Engineering, University of Twente, April 2008. URL http://purl.org/utwente/e58152.

[7] Thijs Sassen. Floating-point based control of the Production Cell using an FPGA with Handel-C. MSc thesis 009CE2009, Control Engineering, University of Twente, June 2009. URL http://www.ce.utwente.nl/rtweb/publications/MSc2009/pdf-files/009CE2009_Sassen.pdf.

[8] Bojan Orlic. *SystemCSP, A graphical language for designing concurrent component-based embedded control systems*. PhD Thesis, Control Engineering, University of Twente, The Netherlands, September 2007.

[9] Bert van den Berg. Design of a production cell setup. MSc thesis 016CE2006, University of Twente, Control Engineering, 2006. URL http://www.ce.utwente.nl/rtweb/publications/MSc2006/pdf-files/016CE2006_vdBerg.pdf.

[10] S Bennet. *Real-Time computer control: An introduction*. Prentice-Hall, New York, NY, 1988. ISBN 0137641761.

[11] M.A. Groothuis, A.S. Damstra, and J.F. Broenink. Virtual prototyping through co-simulation of a cartesian plotter. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2008.*, number 08HT8968C in ETFA, pages 697–700. IEEE Industrial Electronics Society, September 2008. ISBN 978-1-4244-1505-2. doi: 10.1109/etfa.2008.4638472.

[12] Controllab Products. 20-sim website, 2009. URL http://www.20sim.com.

[13] D.S. Jovanovic. *Designing dependable process-oriented software, a CSP approach*. PhD thesis, University of Twente, Enschede, The Netherlands, 2006.

[14] G.H. Hilderink. *Managing complexity of control software through concurrency*. PhD thesis, University of Twente, Enschede, The Netherlands, May 2005. URL http://doc.utwente.nl/50746/1/thesis_Hilderink.pdf.

[15] T.T.J. van der Steen. Design of animation and debug facilities for gCSP. MSc thesis, Control Engineering, University of Twente, June 2008. URL http://purl.org/utwente/e58120.

[16] Bojan Orlic and Jan F. Broenink. Redesign of the C++ Communicating Threads library for embedded control systems. In Frank Karelse, editor, *5th PROGRESS Symposium on Embedded Systems*, pages 141–156. STW, Nieuwegein, NL, 2004.

[17] Agility Design Systems. Handel-C, 2008. URL http://www.agilityds.com.

[18] B. D. Theelen, O. Florescu, M. C. W. Geilen, J. Huang, P. H. A. van der Putten, and J. P. M. Voeten. Software/hardware engineering with the parallel object-oriented specification language. In *5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 139 – 148. IEEE, 2007.

[19] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, 1989. ISBN 978-0131149847.

[20] Ptolemy. Ptolemy II website, 2009. URL http://ptolemy.berkeley.edu/ptolemeyII.

[21] Henk Corporaal. Embedded system design. In Frank Karelse, editor, *Progress White Papers 2006*, pages 7–27. STW, Utrecht, 2006.

[22] RTAI. RTAI website, 2009. URL http://www.rtai.org.

[23] G.H. Hilderink and J.F. Broenink. Sampling and timing a task for the environmental process. In *Communicating Process Architectures 2003*, pages 111–124. IOS press, 2003. ISBN 1 58603 3816.

[24] Jinfeng Huang. *Predictability in Real-Time System Design*. PhD thesis, Technische Universiteit Eindhoven, The Netherlands, September 2005.

[25] Uppaal. UPPAAL model checker. Website, July 2009. URL http://www.uppaal.com/.

[26] QNX Software Systems. QNX real-time operating system (RTOS) software. Website, June 2009. URL http://www.qnx.com/.