Controlling the Production Cell using TERRA-LUNA

F.T. (Frank) Trillhose

MSc Report

Committee:
Prof.dr.ir. S. Stramgioli
Dr.ir. J.F. Broenink
Z. Lu, MSc
Dr.ir. J. Kuper

May 2016

007RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

MIRA CTIT
BIOMEDICAL TECHNOLOGY
AND TECHNICAL MEDICINE

Summary

The aim of this project is to optimize software design of Embedded Control Software (ECS) using the Production Cell demonstration setup in combination with the CSP-based design framework TERRA/LUNA (2015).

It is carrying on the proof-of-concept work of Bezemer (2013) and Hoogendijk (2013) as well as provide improved concepts and implementations that are supposed to overcome the limitations of previous design approaches.

The chosen ECS target (Production Cell Setup) represents the trend of growing needs for more design flexibility and distributed cyber-physical platforms. Additionally, an emphasis is put on energy-efficient software as it is a growing demand in this industrial area.

ECS design frameworks like TERRA/LUNA try to incorporate those trends and requirements by supporting the model-driven design (MDD) idea and leaving the actual implementation (i.e. code generation) to automated routines. This enables the designer not only to get fast test results from model simulations but also to explore more design-space combinations or to detect design flaws at an early stage.

The first part of this project consists of an analysis of the existing work resulting in a selection of optimization criteria. Special attention is given to real-time reliability and design correctness (i.e. by taking full advantage of formal model checking techniques). But also software debugability (incl. adaptability for performance measure), execution efficiency and performance-adjustments (i.e ability to adopt/optimize generated software) of the ECS are evaluated regarding their relevance to the optimization goal.

The results are incorporated into the requirement analysis and used for the design-space exploration in order to decide upon the hardware and software resource allocation. At the end of the first part a design proposal is given how to create an optimized ECS. The to-be proposed approach should incorporate the ideas of the GAC concept with the Way-of-Working of the TERRA/LUNA framework.

The second part of the project is committed to the practical implementation of the design. The goal could be achieved to provide a tool-native ECS design template for a model-driven software project that is applicable to different kinds of mechanical plants – not only the given Production Cell Setup. The template reflects the (in the first part) identified optimization potentials of previous studies while maintaining most of the recommended Way-of-Working principles. Although not as generic as Hoogendijk (2013)'s approach, it still enables future engineers to create easily customized embedded control software that should meet all typical ECS requirements.

Finally an evaluation is given of the project itself as well as the utilized design framework and Way-of-Working. This comprises general ideas of improvement and further recommendations on how to increase functional scope, especially regarding reliability verification issues like formal model checking and real-time execution.

"*A smart machine will first consider which is more worth its while:*
to perform the given task or, instead, to figure some way out of it."

– Stanisław Lem (The Futurological Congress, 1971)

Contents

1 Introduction **2**
1.1 Cyber-Physical Systems 2
1.2 Problem Statement and Solution Demands 4
1.3 Report Outline 5

2 Related Works **6**
2.1 General Background with regard to Software Development for CPSs 6
2.2 Review 9

3 Design Space Exploration **13**
3.1 Requirement Analysis 13
3.2 Feasibility Study 15
3.3 Design Concept 21
3.4 DSE Conclusions 31

4 Implementation **32**
4.1 Software Architecture 32
4.2 Source Code Additions and Adjustments 37

5 Conclusions And Recommendations **40**
5.1 Summary 40
5.2 Requirement Evaluation 40
5.3 Recommendations and Potential Improvements 41

A Appendices **45**

Bibliography **54**

Abbreviations

Abbr.	Explanation	Comment
{...}s	Plural form of the given [abbreviation]	e.g. "bufs" (for buffers)
5C	Separation of concern principle in CPS	Klotzbucher, et al (2013) define 5 functional concerns to be addressed separately
BUF	Buffer	e.g. "COM_buf"
COM	Communication	e.g. "COM port"
CPS	Cyber-Physical System	a.k.a. "Embedded System"
CSP	Communicating Sequential Processes	Formal modeling language
CTRL	Control	e.g. "sys_ctrl"
CTXT	Context	e.g. "CTXT sws"
DSE	Design Space Exploration	Engineering technique
DSP	Digital Signal Processor	Dedicated to massively parallelized tasks
ECS	Embedded Control Software	SW that is supposed to control a CPS
FDR	Failures-Divergence Refinement	Formal model checking tool
FoM	Figure of Merit	Quantity used to characterize the performance of a device
FSM	Finite State Machine	SW modeling technique
GAC	Generic Architecture Component	SW design modeling pattern
HRT	Hard Real-Time	(see definition of RT systems on page 4)
HW	HardWare	
IO-SEQ	Read Write Sequence Principle	Design pattern: sequential data processing, i.e. receive data first, process data and pass it onto the next entity
MDD	Model-Driven Design or Development	System designer guideline[1]
MoSCoW	Requirement Prioritization	Categorized prioritization through 'Must', 'Should', 'Could' and 'Won't'
PCS	Production Cell (Setup)	Simplified demonstration setup
PCS_CU	PCS Computer Unit	HW on which the ECS is running
PCU	Production Cell Unit	A sub-CPS within the PCS
PoV	Point of View	
RaM	Robotics and Mechatronics	UT research group
RT	Real-Time	(see definition of RT systems on page 4)
RTOS	RT Operating System	e.g. QNX Neutrino
SDK	Software Development Kit	set of software development tools
SRT	Soft Real-Time	see definition
SW	SoftWare	
sw	switch	e.g. "context_sw"
sys	system	e.g. "sysCtrl"
var	variable	e.g. "varCtrlValue"
WCET	Worst-Case Execution Time	FoM for Real-time systems
WoW	Way-of-Working	Embedded Systems design guide

1 Introduction

1.1 Cyber-Physical Systems

There are two distinct trends in the digital signal processing field that have characterized the last decades significantly. The first trend is the increasing integration level of computer hardware (i.e. more features and computational performance is concentrated on decreasing area) which can be seen easily by the fulfillment of Moorse's prediction (i.e. Moorse's Law) of 1965. The second trend is the increasing demand for establishing computer-based assistance systems in process automation. Due to those trends, computer systems became so small that they could be fit easily into the environment that they are supposed to control. This is also when the term Embedded Systems became established. At the beginning embedded computer systems where developed solely with the focus on the specific hardware target, making them very unique and difficult to adopt to other targets. Today, those systems have to be very generic and cope with multi-purpose environments, including features like process monitoring and reporting, communication with other distributed units, complex safety measures, etc.

Later on, the term Embedded Systems has been converted into Cyber-Physical Systems (CPS). It emphasizes even more the equal distribution of physical (i.e. mechanical) parts as well as, cyber (i.e software) engineering parts (Marwedel, 2011). Moreover, the physical part, determined by the inexorable passage of time and the intrinsic concurrency of all running processes, has to be managed by the cyber part with its discrete nature and limited capabilities to handle parallel tasks. Furthermore, developing CPSs includes also modeling those physical systems and applying suitable control algorithms. This is then reflected in the development of Embedded Control Software (ECS) which is the main focuses of this study.

With regard to industrial applications of CPS-targeted ECS, several general requirements have to be met. As for most applications rapid and cost-effective design (incl. the automated process of designing as well as design re-usability) is top-most crucial. Closely followed by the demand for dependability, reflected by distinct safety and reliability realization measures which again require proper follow-up evaluation[1]. Due to the increasing complexity of ECS, standardization of design gains equal importance but also becomes a challenge itself (UBM Tech, 2014). A typical example that represents not only the entire range of ECS development but also a paradigm of standardization is the automotive sector. Here several (crucial) car components are getting developed by different vendors using different soft and hardware techniques but have to be entirely compliant and highly dependable when implemented. So far however, this still results in a comparatively high amount of performance loss - "Overdesign is currently the only path to safe and successful system certification and deployment." (Baheti and Gill, 2011)

Thus managing the complexity better would also have positive effects on the design efficiency and dependability. One approach to manage the complexity is Klotzbucher, et al (2013)'s 5C(concerns) principle. Here the idea is to split distinctive fields of concerns such that they can be approached separately but still interact with each other. The principle will be picked up, explained and discussed again in the following chapters in more detail.

Since developing ECS is a central point of this M.Sc. project. The following 2 chapters will discuss the CPS at hand and ECS development more closely.

1.1.1 Project Context - The Production Cell Setup

The application case for this project is a mechanical demo setup called Production Cell Setup (short 'PCS' or simply 'the plant') that simulates in a simple fashion different steps of a typical

[1] or in other words proof of dependability

production line. There are 6 Production Cell Units (PCUs) available at this moment which are meant to simulate production item transportation as well as physical transformation by using different combinations of sensors, belts and actuators. The Feeder unit, for instance, makes sure that the to-be manufactured item gets fed to the Molding unit. Here, two item detectors will sense the arrival and the departure of every item passed by while two other detectors are signaling the start and the end of the performed production step[2]. The chosen ECS control target (see also figure 1.1) represents hereby the trend towards more design flexibility as well as the need for high-performance control accuracy and energy-efficient distributed hardware platforms.

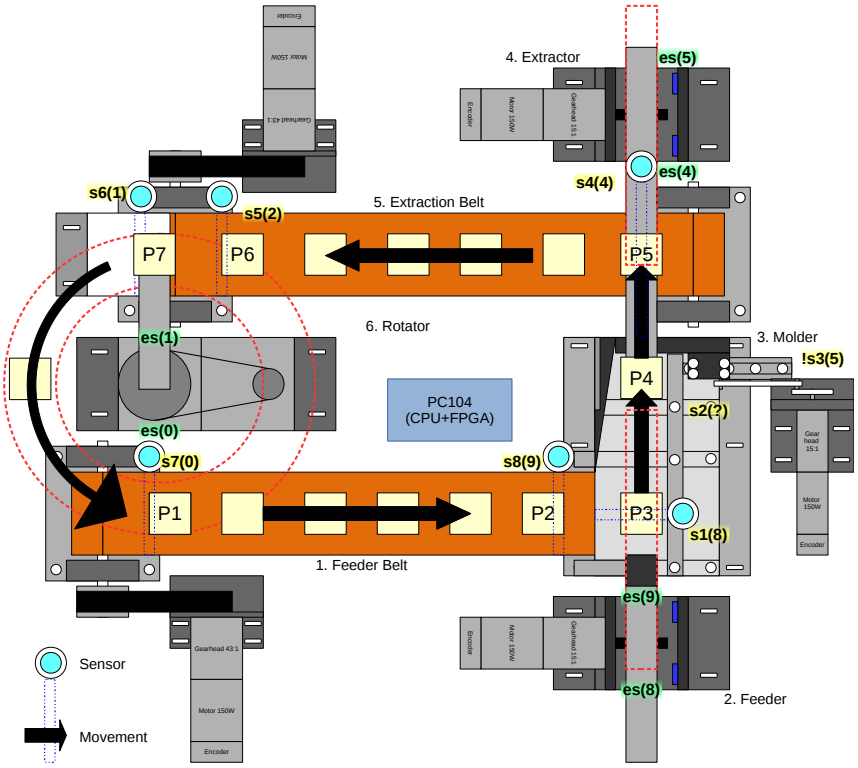

Figure 1.1: Schematic of the Production Cell Setup

1.1.2 ESC Development For The PCS

There have been several attempts to control the PCS whereas the emphasis was always on different requirement sets (thus only considering a limited number of demands) as can be seen in the Literature Study in chapter 2.2 (see also Appendix A.6).

After van den Berg (2006) created the PCS, the main focus of implementing an ECS shifted from a CPU-based solution to a FPGA which can deal with concurrency much better. Despite the fact that there has been always an emphasis on taking advantage of the chosen design automation tool chain which connects tools and design steps seamlessly, many details still had to be added manually to meet all requirements. This was mainly due to focusing more on hardware performance validation than on software architecture improvements. Consequently, certain software development practices became underrepresented (such as design abstraction, modularity or software verification) and did only partially comply with demands connected to ECS development and design quality. Due to increasing complexity of modern ECSs it is becoming crucial to follow consistent Way-of-Working (WoW) guidelines to reduce development cycles and to minimize the risk of design failures.

[2] which is equivalent with the respective actuator movement

A selected set of approaches which focused on improving the ECS of van den Berg (2006) will be discussed in detail in chapter 2.2. Many of them put also emphasis on a model-driven design and therefore used an ECS design tool called gCSP which enabled the designer to check the created CSP models and verify their correctness formally (i.e. to achieve proof of dependability). However, development of gCSP was put to hold, mainly for reasons of falling behind in offering sufficient multi-threading capabilities and due to some software development inconsistencies. Thus, it was decided to invest in a major overhaul of the tool. Consequently, i.e replacing gCSP, a new ECS design framework was created to overcome the inherent drawbacks of its predecessor while still carrying on the fundamental ideas. The new framework now consists of a separate, graphical CSP model editor, called TERRA (Twente Embedded Real-time Robotic Application) and a code base framework library for hard real-time[3]] applications running on multi-core platforms, called LUNA (LUNA Universal Networking Architecture). Both tools are connected by TERRA's model-to-code translator that builds on LUNA's CSP component library.

Hoogendijk (2013) conducted the first attempt to create an QNX-targeted ECS for van den Berg (2006)'s PCS using TERRA/LUNA. This approach especially focused on an aspect that all earlier studies took less into account: the re-usability and adaptability of the resulting ECS to different use-case scenarios or hardware targets. To provide a practical means which would support the intended Way-of-Working Hoogendijk (2013) created an universal architecture template called Generic Architecture Component (GAC) that can be easily re-used and provides enough universality to serve any purpose regarding ECS design. Although seemingly offering a sophisticated architecture solution, an execution performance deficiency was identified but could not be eliminated at the time. In regard to that, Bezemer (2013) recommends to "further evaluate the way of working, by using it to implement different control applications to steer all kinds of different cyber-physical systems." However, this is still limited to a very small range of CPS options due to the fact that the design tool at hand is still under development and does currently only fully support a limited set of computational platform and real-time OS combinations (see also Appendix A.8).

1.2 Problem Statement and Solution Demands

This project has an explicitly practical goal. It is not about proving or applying a specific theory, but instead it is about taking a given system and showing underutilized design potential as well as principal work flow limits. Moreover, under defined circumstances, potential execution performance optimization points have to be detected and exploited while putting the WoW guideline (developed at the RaM group) to the test. As a result, it is intended to provide constructive feedback about how the recommended tool chain can be applied more effectively to aid better throughout software development. The subsequent challenge will be about finding the right balance between using MDD-based measures to maintain dependable ECS development and breaking out of the standardized tool chain by performing manual alteration (e.g. to the source code) to gain performance. Hence there are two aspects of the aimed implementation:

- *functional completeness*
  This means making it work with focus on the best solution no matter the effort.
- *economical efficiency*
  This depicts the fact that making it work in a realistic industrial environment means to deal with resource limitation. In other words, even though there might be a applicable solution it is still considered not feasible to realizes in time.

[3] Definition, Laplante (2004): "A [hard] real-time system is a system that must satisfy explicit (bounded) response-time constraints or risk [causes] severe consequences, including failure."

The latter demand is adopted from a market-driven point-of-view, thus reflecting the practical focus of this project. UBM Tech (2014) states that while software product development cycle duration are kept steady (i.e. increase only very slowly) in practice, the complexity of the developed software increases much more significantly which makes it necessary to counteract with proper automated refinement cycles, including high design re-usability[4] as well as procedures like source code generation and unit testing. However, the main focus of this project lies more on the technological functionality, since, so far and under the determined conditions, a working ECS could not have been achieved, yet.

The previous approaches will be analyzed in depth and will make, to a certain degree, a starting point for different aspects of this project. Especially the ideas of the latest attempt to run van den Berg (2006)'s PCS with an ECS created by Hoogendijk (2013) using TERRA/LUNA will be carried on.

1.3 Report Outline

This assignment carries on the proof-of-concept work of Hoogendijk (2013) and Bezemer (2013) as well as provide improved software design concepts and implementations that are supposed to overcome the limitation of Hoogendijk (2013)'s design approach.

Background information and a summery of previous works regarding ECS development for the PCS are given in the following chapter 2. A discussion of demands[5] that are made by the outside world and which define the overall goal for this assignment is presented in chapter 3. As a result of the in this chapter included requirements analysis, the selected demands are translated into consequent and more explicit requirements to form a design proposal. Practical realization, the derived implementation steps will be discussed in chapter 4. Finalizing this report, conclusions of the achieved results, including recommendation and improvement options as well as a discussion of options how to carry on this project is presented in chapter 5.

[4] which can be e.g. achieved with generic software components or structures that have been proven to be reliable

[5] a fuzzy or very specific requirement from a stakeholder PoV with all sorts of priorities and expressed in general terms like "it has to be fast and efficient"

2 Related Works

2.1 General Background with regard to Software Development for CPSs

The topic of the chapter is placed in the context of Cyber-Physical Systems as object for application of Embedded Control Software (ECS) design. This environment as well as respective software design basics and trends in this area are shortly reviewed in this chapter.

2.1.1 Cyber-Physical Systems (CPS)

Definition: "Cyber-Physical Systems are integrations of computation and physical processes" (Lee, 2007).

The term Cyber-Physical systems is created to emphasize on the actual nature of systems that are linked or even known as Embedded Systems. The latter term is rather non-specific while the former is meant to refer to systems which consist of a mechanical or simply physical part and a signal processing (thus cyber) part which again involves computational hardware executing specific control programs. The physical part consists of all elements (incl. power electronics) forming a machine or robot in which the cyber part can be embedded in. Designing a CPS from scratch often means to cover a large design space where a variety of multi-disciplinary factors have to be taken into account. Deciding on the priority of every factor will also predetermine the route as well as the development time that it will take from an idea to a working solution (Groothuis and Broenink, 2006).

2.1.2 DSE aspect of CPS Development

The exploration of possible architectural options concerning hardware and software solution alike is called Design Space Exploration (DSE). In addition to that the DSE performed here will also include parts of the adopted WoW, i.e. it will be discussed at which point certain tools or concepts are applied that could aid the designer developing an ECS for a CPS. While creating a CPS, performing a good DSE is crucial but can cost a lot of design time. Thus, it is important to perform target-oriented stepwise refinement. Here, the designer has to evaluate after every step the probability of success of each partial solution and find design obstacles as early as possible. This limits the risk of going into the wrong direction and having to start all over again or spending too much time exploring too many options. However, it is still useful to always work out more than one working solution for a final implementation as this will also deliver significant insight into subsequent DSE issues. Even more, with a thoroughly performed DSE, it is possible to make faster and more profound decisions for similarly set follow-up projects. Hence, due to its importance, it is suggested to apply tools or methods that specifically support the DSE step-wise or even iterative processes. Those processes include the handling of design refinement scenarios such as hardware-in-the-loop or co-simulation.
As part of the DSE a thorough Feasibility Study is performed in section 3.2.

2.1.3 ECS Design and Development

ECS development includes many different procedures and tools to support the design and implementation process. According to (Zwikker and Gunsing, 2015), the most well-known are Plan-Do-Check-Act (PDCA), the V-model, Agile, Spiral Design and Unified Modeling Language (UML) as way of documentation. There are several more design methodologies available and most of them have a distinctive scope of application. Zwikker and Gunsing (2015) also state that due to their different methodical strengths it is sometimes beneficial to combine several of them for certain phases of the development process. Although generally different, many methodologies try to incorporate common work steps like requirement analysis, conceptual design,

implementation as well as design verification and validation. Furthermore, it is possible to emphasize on different phases like following a Model-driven, Test-driven or Use-case-driven conception which will be briefly explained in the following sections down below. In regard to ECS and in contrast to hardware-driven development, software-driven projects are more flexible and can be maintained evolutionary, i.e. from simple to complex, most often in a cyclic way.

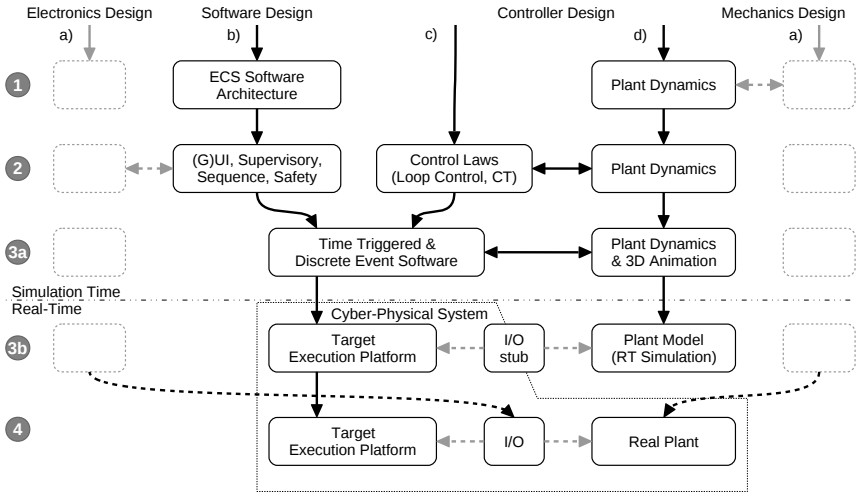

Figure 2.1: Steps of the way of working to design ECS software for CPSs (Bezemer, 2013)

ECS Trends

Accelerated by the rise of the Internet-of-Things (IoT) and accompanied by a general hardware convergence[1], embedded systems have become an omnipresent technology. Typical emerging sectors in the consumer segment are: home appliances, mobile media devices and automotive electronics; whereas the industrial segment (here, using the term Cyber-physical System is most appropriate) focuses more on: assembly line robots, process control, diagnostic devices and infrastructure-level communication electronics. Summarizing the strong trends in the field of embedded or cyber-physical systems development it can be seen that:

In respect to computational hardware, 8- and 16-bit CPUs play recently only a small role and will become eventually more or less obsolete especially in an industrial environment where handling exceedingly complex processes and complying with exacting technological requirements call for advanced hardware resources. In the same time, microcontrollers are getting gradually replaced by 32- or even 64-bit multi- and even many-core general purpose processors. Here, parallel computing is a natural consequence of the ambition to deal with concurrent processes of the external environment more easily and thus addressing the nature of most of the occurring control problems more appropriately. Those problems can and will also be handled by transforming suitable applications into ubiquitous and respectively into distributed systems. This involves consequently the severity of a proper communication between all computational sub-systems. Whereas the consumer segment is going to implement as much as possible wirelessly, the industrial sector still prefers to resort to wired solutions for reasons of dependability and performance. However, due to ease of deployment it is likely that wireless systems will become established despite the drawbacks here, too.

In respect to software development, trends can be identified towards much more complex (in consumer segment often referred to as "smart") designs, which include enhanced on-line optimization algorithms and control process learning capabilities.

[1] in other words and similar to the System-On-Chip (SoC) trend, merging physically mechanical hardware with electronics hardware into one closed unit

Even more, with fast increasing availability of affordable hardware performance the need for code optimization on bit level decreases significantly. At the same time there is a growing demand for rapid prototyping complex but flexible and easily verifiable cyber-physical systems. This extends the workload while development cycles continue to decrease including meeting of deadlines better as shown by UBM Tech (2014). The survey shows also that more than 50% of the interviewed developer worked on an upgrade or improvement to an earlier or existing project and even 80% stated to re-use old in-house produced project code for their current projects.

Those tendencies suggest to make use of highly automatized code generation as is already available in system design suites like Mathworks (2016) Matlab/Simulink, NI LabVIEW (2016), 20-sim (2015), Scicos (2016) and other similar ones. Here, testing and verification becomes an ever growing part of development. Automatized code generation and powerful hardware create opportunities to focus on different implementation aspects separately as proposed by the (Bezemer, 2013)'s WoW in combination with the 5C principle (see also section 2.2.3).

Especially the software development trends will be picked up again and incorporate in this project using dedicated design development tools and guides provided by *RAM* - a Twente university research group.

Model-Driven Development and the TERRA/LUNA framework

Model Driven Development (MDD)[2] was introduced as a systematic and fast - thus cost-effective - software development methodology to enable system modeling and validation independent of a target platform. Instead of producing code manually developers may use domain models[3] to spend more time on design analysis and validation than on actual implementation (i.e. code generation). Furthermore, the models can be used as a language for collaboration between developers of different disciplines making it easier to distribute design tasks and working simultaneously. A major benefit of MDD is accomplished via model transformations which can be used to verify system dependability using formal model checking techniques or it can be used to provide automatically generated, platform-specific code and thus increasing the design portability. By using means of MDD not only up-to-date documentation is immediately provided but also higher design quality can be achieved due to less error-prone development cycles and the enforcement of separation of concerns and skills. However, like all methodologies MDD is not suitable to solve all aspects of software development. Some of the risks when MDD is used solely are evoked by the fact that a desirable design flexibility has to be put into the design on purpose and does not come automatically. A more practical risk comes from the fact that the design will be also always limited by the applied tool thus the importance of determining requirements in accordance of what the used MDD tool is capable of (Den Haan, 2009). Consequently, it is desirable to combine MDD with other methodologies to benefit from several independent strong points.

Test-Driven Design

Test-driven Design (TDD) is a software development technique that aims for improving software testing. Tests or test scenarios are created by the software developer based on beforehand-gathered use-case experience. The tests are carried out beginning with individual modules or functions. Once they passed, all subsystems and finally the entire software as a whole are being tested. The difference to normal software testing is, however, the focus on the design requirements before any code is written. This way TDD specifically encourages simpler or minimalistic designs, similar to the "keep it simple, stupid" (KISS) principle which only implements the

[2] also stands for Model-Driven Design and is synonym for all MD* methodologies like MDA or MDE)

[3] In software engineering a domain model is a conceptual (object) model "of the domain that incorporates both behavior and data". Fowler (2003)

minimum scope of operations[4] (Beck, 2003). Apart from the benefits it is also relevant to take risks into account that come along with TDD such as the significantly increased design phase which means that actual test results are later achieved as they would have been during normal development. It is also worth considering that certain compromises have to be determined with respect on the increasing system complexity that the test environment adds.

Use-Case-Driven Design

In software development functional requirements can be specified with the help of Use Cases[5]. They are used as primary artifacts for deriving architectural abstractions (Aksit, 2001) and provide essential insight as well as assistance in making profound design decisions (Prosman, 2001). Being one of the key activities in requirements analysis, Use Case analysis is a systematic method to determine what users should be able to accomplish when using the to-be-developed software Lethbridge and Laganiere (2005). Applying this principle to parts of a or the entire software architecture is called an Use-case-driven Design (UCDD) approach. A typical start would be to create Use Cases with the fact in mind that most systems are built to interact with external operators or actors, e.g. a common user (Lee and Xue, 1999). All Use Cases and their interactions can be combined in an Use Case model and therefore represent all specified functions of the system under development. Subsequently the model can be seen as system requirements documentation or even as a contract between the customer and the developers (Aksit, 2001). In addition, UCDD helps managing the complexity of a to-be created system due to the attention being always focused on one specific conceptional problem which again can be helpful to create better test cases. Lorenz (1993) also states that due to the key aspects of UCDD designers are encouraged to envision (system) outcomes before attempting to specify them, and thereby helps to develop more effective requirements that will also take otherwise-unconsidered scenarios into account. However, similar to the techniques introduced above, there are certain risks or even drawbacks associated with UCDD. Jerome (2000) criticizes UCDD for its lack of capturing all fault events or unintended outcomes. Even more, Lee and Xue (1999) argue that UCDD is not suitable documenting interactions between requirements since every requirement is handled separately. Consequently, when it comes to non-functional requirements like usefulness and usability involving user motivations, experiences or intention, UCDD will not provide enough expressiveness nor is there a systematic guide how to handle such matters (Prosman, 2001).

2.2 Review

In the following chapter, the analysis of previous studies is presented. Hereby, the project focus of those approaches as well as their strengths or deficiencies is elaborated regarding the chosen WoW and achieved software quality.

The conclusions drawn at the end of this chapter are the starting point of the subsequent chapter 3. They are used to work out the design details of the current approach which should aid adjusting the adopted WoW as well as give ideas on how to achieve exact improvements for the final hard or software implementation.

The summarized reviews are thus the basis for specific optimizations techniques and a list of selected criteria which will be used to guide this new design approach. Furthermore, the set of criteria can be used to evaluate achieved software quality in more detail at the end of the assignment.

The studies have been reviewed w.r.t. the following points:

[4] i.e. as a consequence a software component that is difficult to test can be considered a component with design errors.

[5] here the definition of Jacobson, Booch and Rumbaugh (1999) is adopted: Use Case - "a sequence of actions that the system provides for actors"

- What was the goal or the focus of the study?
- Which domains and requirement specifications have been applied?
- What have been the results and which criteria have been used to evaluate them?
- Which specific software evaluation or performance measuring methods can be adopted?
- Which WoW or design method has been chosen (incl. difference to (Bezemer, 2013))?
- Which tool or toolchain has been used?
- What are the achieved qualitative (e.g. RT guarantee, dependability, deadlock-freedom, adjust-ability, debug-ability, testability, etc.) and quantitative (like, efficiency, performance, sample time, WECT, etc) results, esp. w.r.t. the ECS?
- What is the degree of multi-core or parallelism exploitation?
- What conclusions or recommendations draws the study?

2.2.1 van Zuijlen (2008) - Development of CSP-to-Handel-C-based ECS for FPGAs

After gCSP has been proven by Jovanovic (2006) and Maljaars (2006) to work as a CSP design tool for CPU based hardware, van Zuijlen (2008) managed to implement an ECS on a FPGA using Handel-C-to-HDL code translation.

The study could show that by utilizing a FPGA it was possible to achieve a much higher level of parallel data processing and thus resulting in a significant performance efficiency gain compared to its preceding studies. It was chosen to keep the software architecture simple (including a flat structural hierarchy) and subdivide the core ECS design for every PCU only into 3 components:

- Control (i.e. execution of control algorithm),
- Safety (i.e. error detection) and
- Command (i.e. motion profile and user input).

Due to the lack of proper support in Handel-C, implementing algorithms using floating point precision would require a disproportionate amount of logic cells. Hence it was chosen to translate all algorithms to integer precision by re-designing all controller models. Since the final implementation was mostly done manually, the development wasn't benefiting much from the automatic code generation capabilities of the design framework. In contrast to the increased programming effort, the logic cell utilization could be reduced to an absolute minimum. Furthermore, the study did not show how using CSP models contributed in the ECS evaluation process (e.g. w.r.t. formal model checking). This would have been especially interesting in regard to the low level of design concern separation, i.e. no explicit Finite State Machines or separate communication components, and how this affected the software quality or work flow in general. Mainly directed at the design framework gCSP, (van Zuijlen, 2008) recommends to improve the handling of ECS safety handling as well as usability and code generation capabilities (i.e. Handel-C code generation only supports a sub-set of CSP).

2.2.2 Sassen (2009) - Floating-point Improvements to the Handel-C Approach

Based on the work of (van Zuijlen, 2008) (Sassen, 2009) continued the approach to utilize a FPGA using Handel-C. But in contrast to (van Zuijlen, 2008), the focus of this study was more on safety and control precision efficiency while reducing the design effort at the same time. The increased precision range could be achieved by implementing only a single Safety block whose

functionality is shared by all PCUs which left more hardware resources to spend on executing Floating Point operations[6]. Here, the same principle is adopted for the now more precise but also slower (due to time splitting) control algorithm. Hence, except for the introduced Finite State Machines that runs for every PCU separately, the software architecture was chosen to form a rather centralized structure. This resulted in an increased but better FPGA hardware utilization. However it also increased the risk of single point of failure. Apart from that the performance efficiency of applying Fixed Point algorithms was evaluated as well but deemed to be not superior to the Integer approach of (van Zuijlen, 2008). Since this and (van Zuijlen, 2008)'s approach required the code compilation features of an external proprietary tool (Xilinx Coregen) that is no longer supported by the design framework, the result are not only irreproducible and thus difficult to compare but consequently alternative ways of implementation are necessary. Like (van Zuijlen, 2008), (Sassen, 2009) directs his conclusion at the design framework gCSP and recommends to improve code generation capabilities towards hardware description languages due to lacking support of Handel-C within in developer community.

2.2.3 Hoogendijk (2013) - Proposal for a CSP-based GAC Template

With focus on using the software architecture design tool TERRA, Hoogendijk (2013) provided a sophisticated Generic Architecture Component design proposal called GAC which covers 6 important general design concerns. Those concerns comprise Klotzbucher, et al (2013)'s 5Cs: Composition, Communication, Computation, Configuration and Coordination as well as Safety. Hereby, the last 5 components are implemented as actual, separated model components. Hoogendijk (2013)'s approach is supposed to yield shorter design phases due to the presence of all relevant general functionality. This leaves the option to focus more on an elaborate implementation of specific detail functions, e.g. the control algorithm, error detection or state handling. Although it can be followed that the generic architecture covers many use cases and applications, it was also found that in combination with the TERRA/LUNA framework performance shortcomings (due to significantly increased computational redundancy) arise[7]. When applying the GAC template (in the way it is proposed by Hoogendijk (2013)) to low-performance hardware, like the PC/104, which is accompanying the PCS, an efficient operational mode is difficult to maintain. Hence only 5 of the 6 PCU could be controlled simultaneously. Additionally and for the same reason, it was chosen to implement manually the FSM (here called *GAC life-cycle*) on code level instead of on CSP model level like the GAC encourages to do. Consequently the options to determine the dependability of the ECS by formal model checking techniques was reduced. A way how one could add FSMs correctly to a CSP architecture design was already provided byRan (2012) who created an UML-Statemachines-to-TERRA-CSP-model transformer. Hoogendijk (2013) recommends to improve the integration of the GAC into the TERRA/LUNA framework, including graphical object and code generation optimization like the introduction of multi-data-type channel buses. Furthermore it is necessary to improve the support for design debug-ability (like the real-time logger options) or testability of the interaction of design parts with different real-time levels.

2.2.4 Bezemer (2013) - Introduction to specific Way of Working and correspondent Tool Suite for CPS Development

Bezemer (2013) describes in his work the Way-of-Working which he proposes as best practice to reduce design complexity by separation of concerns (see also preliminary studies of Bezemer, Groothuis and Broenink (2011)) like the 5C principle. Based on a MDD methodology the goal is to achieve a *first-time-right*[8] software deployment for a mechatronic system. Maintaining the designer's point of view it is supposed to decrease design time by giving advice how to

[6] which were added to the Handel-C semantics as part of the study

[7] as is e.g. caused by approach-inherent doubled Safety blocks

[8] i.e. deploying the final software on the hardware target for the first time without major issues

structure the software development as well as its architecture yielding a high level of project re-usability and lower risk of design faults. Furthermore tools and (co-)simulation techniques are proposed which are able to use created models interchangeably. Besides software patterns and real-time layer categorization guidelines, the WoW also suggests to makes use of a specific tool framework that supports the WoW best. In regard to the introduced WoW, Bezemer (2013) emphasis on improvements towards three topics. At first, it is necessary to enable designers keeping better track (i.e. introducing an WoW native model management environment) of different component implementations. Furthermore, it was found that designing graphically is still not optimal even more when too much detail is implemented on model level. Automatic model optimization features could assist the design in keeping the model abstract and simple while (hardware) implementation details or (execution performance) adjustments are added automatically when code is generated. Finally, in order to extend the design verification scope, fully developed (co-)simulation capabilities need to be available for the CSP models as well.

3 Design Space Exploration

The goal of the Design Space Exploration is to achieve a sound project specification which determines all system objectives as well as the way how they can be implemented.

3.1 Requirement Analysis

Given the time restrictions and the fact that this project has to choose for a main focus, not every step of best practice software development can be executed in depth. Consequently, the actual problem solving requirements, not only result directly from the project's motivation (expressed in use cases), but will also be comparatively more limited, i.e. adopted towards the requirements for this Master thesis rather than they would appear under realistic conditions. Even more, it might be that requirements are set using educated guesses where they normally would have been deducted from more advanced analysis and background studies. After introducing requirements and performing a project-based DSE, the set of requirements will be once more analyzed with regard to the scope of the project - i.e. what might be feasible within time. The result of the process of setting up the project's specifications will be presented in the (proposed) design section 3.3.

Starting with the general question in mind what the end user needs to get accomplished to meet their needs, it is crucial to translate vaguely formulated demands into precise requirements, at first. The better the set of requirements is specified, the easier it is to validate if the final solution will suffice. This is a rather iterative process which is also known as requirements engineering. It covers test and feasibility studies as well as simulations and best-practice analysis. Broy (1997) breaks this process into the following sub-phases whereas the first three points belong to the analysis phase and the remaining to "design, implementation, integration, and tests" phases:

- domain analysis and domain modeling
- requirements capture
- requirements validation
- requirements tracing
- requirements verification

Based on the outcome of a system analysis phase (which was carried-out before), the requirements engineering establishes a general understanding which leads to a increasing detailed modeling of the application domain. The work process can be concluded when all stakeholders agree to a certain set of requirements.

Going into more detail and starting with the general software structure, several kinds of requirements have to be complied with, as mentioned in the requirements section. The main reasons (from a designer point of view) for establishing a sophisticated software architecture is the demand for easily executable testing, debugging, documentation and maintenance. Thus most requirements are related to how the software is supposed to be created and come from stakeholders like the software designer or the programmer. As a matter of course, the way the architecture is set up, is also influenced by what the purpose of the software is and the anticipated performance (i.e. giving the expected result within a certain time limit). All stakeholders are sorted by priority. This will determine at which point during the development phase formulated requirements are realized and also to which extend.

With regard to business environments, a particular stakeholder has the overall highest priority rank and that is the investor. His biggest concern is profit which again is linked to costs and the "time-to-market" factor. This issue is often reduced to "time is money". In other words it is crucial that the developed product is as fast as possible finished and can be sold to meet the market's current demands. One way of accomplishing this goal, regarding ECS development, is to provide a generic architecture platform that enables the software designer to take advantage of a predefined structure. This structure could be applied directly or adapted easily for different purposes.

3.1.1 Domain Analysis

The following chapter aims to provide a quick overview of the standard software quality assurance technique, called Domain Analysis which is supposed to identify objects, operations, and relationships of the matter at hand. This makes it easier to understand the project's background which again is needed to break down the main issue into smaller problems and make well-informed decisions. (Lethbridge and Laganiere, 2005)

The following selection shows identified, more specific (system) stakeholders with their corresponding domain:

- User — Usability, availibility, safety
- System Architect — System abstraction (incl. module/component interaction)
- System Modeler — Modeling (transformation) and formal checking (dependability)
- System Designer — (Co-)Design (incl. design standards, patterns and APIs)
Functionality, such as FSM or fault tollerance (safety)
Efficiency and exploitation of parallism
- Software QA — Quality Assurance (i.e. models, pattern and code quality)
- System QA — (Co-)Simulations and tests (of reliability or efficiency)
- Project QA — Project documentation and maintainability/reuseability
- Applications Engineer — Requirement engineering (incl. domain analysis)
- Software Engineer — Code generation and scheduling (parallism and efficiency)
- Specialists
  - Security — Security principles
  - Control Engineering — Hard real-time compliance and control algorithms
  - Hardware Engineer — PCS hardware and hardware interfaces

Hereby, the selection is based on a preparatory criteria elaboration which determines the focuses of the current project (see also Appendix A.2). Further typical stakeholders like *finances* or *management* have been excluded due to lower priority.

3.1.2 Requirements

The project specification is derived from a heterogeneous set of general as well as specific requirements which again has been resulted from the problem description and demand analysis. In regard to the current ECS project the main demands have been determined to essentially concern technological functionality. Consequently, most requirements will be specified from

a system designer point-of-view and might be expressed in engineering quantities like "operation X has to terminate within a certain amount of time (i.e. cycle period or sampling rate)" - which stands here for the requirement of performance or hardware-dependent efficiency (R1).

Continuing Bezemer (2013) and Hoogendijk (2013)'s WoW approach and since the latest ECS project provides only reduced functionality (see also section 2.2.3), the requirement analysis is partially based on those studies, adopted and extended to create an ECS that runs all necessary operations within a guaranteed real-time period limit. However, the demand for functionality does not only include executing the program once, but repetitively executing it an unlimited amount of times. This demand can be reflected by the requirement of dependability or reliability (R2) which can be functionally expressed with explicit fault tolerating error handling operations[1] (R2f). Special emphasis is set on the contradictory nature of the first two requirements: A fast execution time usually requires a reduction of software complexity, while reliability measures (especially regarding fault tolerance) increase the ECS complexity. Since meeting the performance requirement is of top priority, the aim is to establish satisfying compromises to keep reliability as high as possible.

Additionally relevant is the economical demand. It aims at a high return on investment, meaning that the invested development should pay off by being able to re-use the currently produced ECS for future projects. Therefore the requirement "software re-usability" which can be can be also connected to ease of maintainability (R3) is introduced to this project with increased priority.

Those three key demands (see also Appendix A.2) can be then translated into functional requirements like:

- real-time guaranteed execution (performance)
- fault tolerance (reliability)
- structural modularity (re-usability)

The process of translation is performed when carrying out a Feasibility Study which will be done in more detail in the following section. Here, each relevant non-functional requirement is assigned to a set of available functional requirements which would satisfy determined needs. Conclusions regarding what is found feasible is presented in the design section (including derived software specifications) where the identified best solutions or feasible compromises are discussed. Regarding this project, it is also worth noting, that the suggested solutions should always meet the requirements in combination with the given ECS design framework TERRA/LUNA (and its closely related software development environment tool chain) as a use-case-based proof-of-concept.

Table 3.1 (further down below) lists the key requirements and assigns them a priority according to the MoSCoW method (Clegg and Barker, 1994).

3.2 Feasibility Study

The process of designing software is influenced by several constraints or limitations. Still, dependent on the degree of freedom there may be a considerable number of appropriate design implementations - each with different benefits and drawbacks. Analyzing combinatorial alternatives and selecting optimal (architectural) configurations is called *Design Space Exploration* (Popvici, Rousseau, Jerraya and Wolf, 2004). After formulating and exploring different

[1] in other words, due to the fact that identifying every possible operational error or system failure is hard to maintain, it is equally important to focus on preparing the system for unidentified issues and enable it to continue or finish the current operation as much as possible rather than failing completely and changing its state into uncontrollable

#	Requirements	Prioritization
R1	**Efficiency optimization to fit targeted HW**	**Must**
R1f1	Less context switches	Should
R1f2	Load distribution on several hw platforms	Could/Won't
R2	**Dependability/Reliability measures**	**Must**
R1f1	Proper fault tolerance / error handling	Must
R1f2	FSM-based execution coordination	Should
R1f3	Reliability has to be verified	Must
R3	**Design for re-usability/maintainability**	**Must**
R3f1	MDD: TERRA/LUNA has to be used	Must
R3f2	Platform independence	Should
R3f3	Towards generic/scalability	Should
R3f4	Design for testability	Should

Table 3.1: Requirement Analysis (MoSCoW principle)

approaches and system variations, decisions regarding the hardware and software architecture have to be made. Those decisions represent the final system specification within the feasibility phase. They will not only provide general functionality but also determine the overall performance level of the system. This concerns typical aspects like execution time, power consumption or even development expenses. Hence, it is also crucial to provide design options that can be adopted in different ways or adjusted to change, for instance, the system's efficiency according to its implementation priorities. With respect to subsequent or previous projects, creating benchmark tests aid to help to evaluate the achievements and expose the quantitative differences to other approaches. Previous studies, regarding ECS development for the PCS, however, offered generally only little measurements or performance data that could help comparing results[2]. Apart from existing software performance-related figures like cycle time and deadlock-freedom or hardware utilization, further benchmarks like real-time guarantee, WCET, ECS memory footprint, exploitation of parallelism or modularity should be considered to be used as well. Especially in regard to the WoW, development figures like degree of project maintainability, re-usability or work-flow integration provide essential insight into achieved software quality level.
The following sub-sections will give a summery of the DSE process executed for this project and state feasible implementations or potential optimization points next to ways of their efficiency evaluation.

3.2.1 Hardware

Initially, it seem there is a wide range of signal processing hardware options due to the fact that there is no requirement about which hardware architecture the ECS has to run on. Even more, by definition, the ECS design framework TERRA/LUNA does not compel any specific hardware platform (e.g. ARM / x86 based CPUS or FPGA). Consequently, the software architecture can and should be designed independent of the eventually targeted hardware (Bezemer, 2013).

However from a demonstration point of view, although very limited in signal processing resources the currently used HW option (referred to as PC/104) has shown to work best controlling the PCS with the FPGA-based implementations. This is due to the fact that it features several control aspects that benefit significantly from being handled simultaneously as shown by van Zuijlen (2008). Consequently utilizing a FPGA would be a plausible consideration. On

[2] which is also due to there diverse nature of implementation

the other hand, the stacked hardware showed also very significant limitations[3] using the FPGA when it comes to more complex ECS implementations as demonstrated by Sassen (2009).

Moreover, although older tool chains (including gCSP) provided partial integration of specialized hardware into the WoW design flow, the TERRA/LUNA-framework-based tool chain support is still very limited (see also Appendix A.8). So far, the new framework is only capable of generating exhaustive source code mainly for ARM or x86 architectures[4] [5].

This is why the in-house developed RaMstix platform (housing a ARM/based Gumstix board) is considered an hardware alternative next to the PC/104. Benefits would be slightly more clock cycles per time which could translate in faster processing[6] and thus increasing the chance to meet the RT time requirements better. Another important factor is the easily accessible expert knowledge which would reduce the implementation time. Similar, those advantages, but to a lower extend, would be also true for an Arduino or Rasperry PI based board. However, by default, such boards are often limited in the way they provide interfaces to other hardware. Thus considerably high effort is assumed to fix this drawback as well as finding working BSPs and writing drivers. For this reason, only the two ARM-based versions of the Gumstix/RaMstix board have been included in the DSE next to the x86-based PC/104 system that was already used mainly for controlling the PCS.

Whereas as Groothuis and Broenink (2006) concluded that there are enough resources left on the given FPGA (AnyIO board) to do all the necessary operations within 1 ms, seemed Hoogendijk (2013) to have exposed the (CPU) hardware's limitations[7]. As a consequence it is also considered to distribute to computational load over multiple hardware boards. However, this raises the issue of handling hard real-time communication between independent boards. One way of solving this and avoiding to develop a dedicated hardware link and communication protocol, is by defining the communication as soft real-time[8]. Here, the idea is to send frequently data via Ethernet using TCP/IP (see also Appendix A.7), including verifiable or error-correcting-coded process data. This data might come from sensors or other control relevant instances and is essentially a positive status streaming[9].

The drawback of having the communication defined this way and sending constantly all information (even those which have not changed) would be the increased process load. Additionally, if any error (e.g. wrong sensor data or a message was not received in time) is detected, the entire setup had to be stopped due to the fact that the actual error cannot be identified with absolute certainty. Consequently, this wouldn't make it an error-free but still a fault-tolerant design.

3.2.2 Operating Systems and SDKs

Although there exist simple, often micro-controller or FPGA-based systems for which it is not essential, but most embedded setups need the support of an operating system due to the application complexity. Hereby, the OS takes over tasks like scheduling, synchronization, context switching as well as I/O and memory operations. Since many embedded systems serve as real-time (RT) systems (UBM Tech, 2014), the utilized OS must be real-time compliant making it a

[3] with respect to floating point calculations and enhanced safety handling

[4] i.e. this highly depends on the applied compiler which currently either gcc or a special version of it

[5] no support for HDL code generation at all, yet

[6] since x86 and ARM differ widely w.r.t. the instruction set it is difficult to make a sophisticated evaluation of their potentials solely based on the clock cycles and without knowledge about the ECS

[7] though, he was additionally separating functional concerns and while adding abstraction also implicitly adding more software complexity to the application design

[8] i.e. something similar to Xenomai's RTnet approach

[9] In other words sending the confirmation that everything is operational and no errors have been occurred.

RTOS (Marwedel, 2011)[10]. The RTOS needs to be capable of limiting the task execution time jitters deterministically (i.e. in a predictable or reproducible thus reliable way) to an extend such that task deadlines cannot be missed (hard real-time requirement).

Benefiting from the reduced development time on one hand, choosing the right OS will also reduce the designer's flexibility in the way how the embedded application can be configured (e.g. limited portability) and reduce the overall system performance (due to more load on the CPU). To counteract this effect some RTOS like QNX or FreeRTOS are based on micro-kernels which limit the OS support to a minimum.

For the DSE the following operating systems have been checked for suitability, considered factors like availability, resource load, hardware and framework support[11]:

OS	availability	kernel type	hardware support
FreeRTOS	available	microkernel	n/a
Linux + RTAI	available	twin kernel	n/a
Linux + Xenomai	available	twin kernel	n/a
Linux + PREEMPT_RT	available	monolithic	n/a
QNX	available	microkernel	PC104 / RaMstix(partially)
Windows Embedded	licensed	hybrid	n/a
Keil RTX	available	n/a	n/a
VxWorks	licensed	monolithic	RaMstix(partially)

In the course of selecting the right OS, properties like the types of supported scheduler (preemptive, cooperative, etc) as well as the algorithm used for the tasks management (FIFO, round robin, priority-based, etc.) would be important, too. Being apply to deploy different task schedules enables the software designer to optimize on the cycle period. However, currently LUNA only supports FIFO preemption which all of the OS offer.

Many of the vendors or institutions which provide those operating systems also provide supporting SDKs. Here, further aspects have been added to the DSE like how BSPs (if available) could be deployed and if or to what level built-in profiling tools were provided.

As stated by Hoogendijk (2013), so far only implementations for QNX[12] exist and are known to work (incl. tool chain availability). Although other options might work as well[13], in regard to project time limitations having a working option at hand becomes a predetermining factor.

3.2.3 Software Architecture

The architecture of the ESC is the domain with the greatest potential of optimization. Hoogendijk (2013) chose to create an ESC using his developed Generic Architecture Components (GAC) design template. This structural component is meant to be very general and thus use-case independent. It represents the opposite approach of creating a highly customized specific architecture for a very particular application. The major benefit of the Generic Architecture Components lies in the re-usability. A created universal architecture can be the starting point to many use-cases and the modularity of the structure and standardized design patterns make functional refinements easier. With respect to the hard real-time requirement (and indirectly to the performance efficiency), the GAC approach has the disadvantage of creating too

[10] here the definition of Takada (2001) is adopted: RTOS - "[..] is an operating system that supports the construction of real-time systems"

[11] Here hardware support has to be at least in form of Board Support Packages (BSPs)

[12] Initially QNX seemed to support the idea of using channels to link processes which is why it was identified to work best with TERRA/LUNA Wilterdink (2011). Later however this was found to be not true thus reducing the appeal to use it Bezemer (2013)

[13] Bezemer (2013) states that there exists at least partially native support for Linux + Xenomai targets by TERRA/LUNA

hierarchically expanded structures and might introduce significant performance loss. This becomes especially evident when analyzing the GAC template with focus on functional redundancy (i.e. components, which are necessary to achieve generality and universality, but which also introduce unnecessary duplications when used more than once). Individual designed architectures could avoid these redundant or even irrelevant components, however, on the cost of design re-usability. Regarding software composition, the modularity of the structure and the use of standardized design patterns support the reliability of the GAC architecture. Not only are standardized modules less prone to manually inserted errors or inconsistencies, but they also support unit testing, making it easier to prove reliability on the level of individual components. This is best shown by the separation of safety and coordination measures which is included in each component of the architecture. However, it is to argue whether reliability actually would or would not diminish if the repeated elements (like error detection component) were to be reduced. Compared to Hoogendijk (2013), van Zuijlen (2008) approach was less generic and comprised only 3 components for each PCU (i.e. a Controller, Safety and a Command Block, see also Fig. 3.1). However, the component could be used to represent each of the PCS's PCUs without adjustments to its structure - yielding high efficiency. It also has to be analyzed how both approaches can be brought together, combining both beneficial sides.

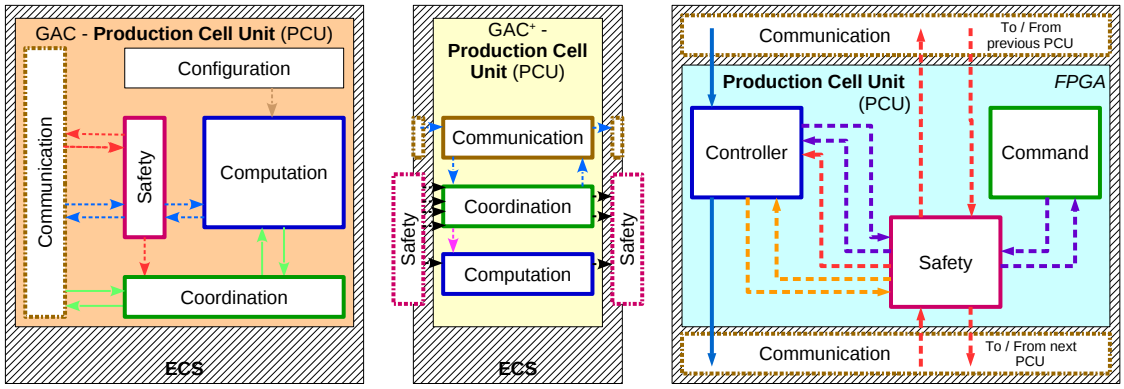

Figure 3.1: ECS structure attempts: GAC (left), optimized GAC (middle) and FPGA (right)

Since it was found that the Safety feature is always dependent on the specific implementation of an (control) algorithm or function it always has to be adjusted accordingly to meet the exact characteristics. This is why a separation of concerns is not always entirely feasible or necessary (w.r.t to functional redundancy) on every design level. Hence, this concern was reduced to be only present at the interface to the hardware on the top design level where it matters the most. Furthermore it was found that there is no need for the Configuration concern since there is only one simple set of configuration during the initialization.

The new and modified GAC now only implements only the Communication, Coordination and Computation components which generally should reduce the computational load compared to Hoogendijk (2013)'s approach.

FSM For reasons of stability, fault tolerance, debug-ability and documentation, it is preferred to implement explicit FSMs into the software design. Hoogendijk (2013) recommends to embed the respective FSM algorithms into a separate area of the software architecture, i.e. using the proposed TERRA/CSP-based GAC approach on software model level. In addition to that, Meijer (2013) describes the integration of typical FSM constructs into the software architecture using TERRA. An alternative to implementing a FSM on model level would be to provide respective source code manually. Both options have different benefits and drawbacks. Using Meijer (2013) approach would follow the standardized WoW which would yield source code that is automatically generated. While this would fit exactly into the recommended tool chain,

it would also result in a significantly increased overhead which is caused by how LUNA generates code from CSP models. The alternative, however, would break out of the automated and thus safe tool chain but yielding more performance while facing increased effort to be spent on testing and debugging.

3.2.4 Performance Efficiency Optimization

As stated above, this project has a strong emphasis on the practical aspect of developing an ECS. This is due to the rigorous limits set by the project specification which again is based on the expected (small form factor and low performance) computer hardware. Consequently, different optimization techniques will be presented that are meant to reflect explicitly the hardware nature by either removing or simplifying several parts of what would be or has been previously implemented in a ECS (that was controlling the PCS). In addition to that, Rutgers (2014) claims that "in practical solutions, streamlining programming by abstractions is only viable when these abstractions still allow performance optimizations or cost analysis, which can achieve an equivalent performance as hand-written or hand-optimized code". In accordance to this, the WoW as well as the MDD-based tool chain around TERRA/LUNA will also be analyzed and checked in regard to if the framework meets the set requirements and which specific performance improvements are actually supported by it.

3.2.5 Software Generation

In one of the latest studies, Hoogendijk (2013) was building the ESC architecture using TERRA. Here, next to the built-in code generation features, 20sim has been used to create code for specific tasks (i.e. tasks that included the control algorithm and motion profile set point creation), as well. Within TERRA, a specific 20sim block component provides a link between the CSP (architecture) model and the externally generated source code. The entire code is then compiled using links to the LUNA components library. However elements, like the FSM have been added manually into the program code for efficiency reasons. So far, there is no further alternative way available besides the combination of these 3 software generation methods. Consequently and in order to determine the best compromise between ECS performance[14] and software design abstraction[15], each method (incl. the manual code manipulation) will be evaluated regarding its project requirements. For instance, with respect to a single core CPU, 20sim does allow to create more efficient code and algorithms than TERRA/LUNA. This is mainly due to generating only one thread which will handle all 20sim operations. However, being represented only as one component block in the architecture model, this code is not formally verifiable. The same is true for manual adoption to the code which would offer even more potential for optimizing the programs performance. All code adaptations however might result in a setback regarding reliability. Not only is there a risk for introducing accidentally errors and inconsistencies very high, but also is the option for reliability checking limited to plain code review. Consequently, it is crucial to achieve a balance between all methods according to their impact on the final design.

At this moment automatically generated code for certain functional topics is not yet covered fully by the MDD tools. Consequently they need to be programmed manually. Those topics are motion profile generation and the implementation of the FSM as is explained down below.

MotionProfile Currently, for most of the created ECSs that are based on motion profiles generated by 20sim, each set point is calculated with every new cycle (i.e. at runtime). This produces an avoidable high computational load on the hardware. Since the expected values can be considered as fixed, i.e. if value X1 assigned to time step Y1 then X2 will be the next value for

[14] in other words, achieving real-time compliance
[15] i.e. verifiability of reliability

time step Y2, an alternative implementation could use simple LUTs (Look-Up-Tables). So far, 20sim is not able to generate LUTs. This is why they will be generated during the initialization phase of the to-be created ECS using a simple sine-function-based algorithm. Consequently, this approach will allocate much more memory than a runtime solution.

Although this is a work step outside the standardized tool chain, the risk of introducing errors by doing it manually is considered quite low and correctness verification is straightforward (i.e. checking LUT for wrong values).

FSM From a designer's point of view crucial functional components like FSMs should be implemented on an abstract, architecture model level. However, at this point and with respect to the TERRA/LUNA framework, it is not only found that "representing a state-machine in pure CSP results in in-efficient code" (Meijer, 2013) but is also still difficult to debug due to a lack of proper simulation and verification capabilities (Ran, 2015). As a temporary solution, tools like UPPAAL (2015) which provide also the option to specify formal queries to check created Timed Automata could be used. However, with regard to UPPAAL, there is no built-in option to generate source code from the created models which means that they have to be ported manually into the ECS. A brief research showed that there are (ongoing) projects which try to provide a third-party tool that is capable of adding this feature to the UPPAAL framework (see also Appendix A.10). However, they have been found to not completely meet the project's requirements (i.e. not matching intended target language, not being matured enough or just not accessible)

3.2.6 Tool Chain and WoW

The established tool chain reflects the ideas of the WoW created by Bezemer, Groothuis and Broenink (2011). However, although the core set of tools is already an elaborate choice, it can be relevant to search for alternative or additional tools that provide different or more benefits to the final result than the currently used one do.

Benchmarking and Profiling Next to the considered reliability checking tool UPPAAL, one further crucial tool chain feature is the profiling of the to-be developed ECS. Profiling does not only help to verify if functional requirements have been met but also supports the designer during the development with identifying performance deficiencies and to compare quantitatively with previous approaches. Especially the latter point would improve the necessary[16] documentation of the ECS development evolution and help to interpret and classify (previously) achieved results better.

3.3 Design Concept

Concluding the DSE, the following sections briefly outline the main design decisions and optimization hypotheses of the project.

Several conceptual design steps and topics have been highlighted. One of the major concern right at the beginning (when modeling the ECS) was finding the best compromise between structural completeness (that supports all desirable features of high quality software) and execution performance. In other words, it is preferred (by the system designer) to realize and model as much as possible within the top abstraction layer such that the evaluation (e.g. by performing CSP model checking) considers as much as possible. But so far this has been known to be a constant source of process overhead which degrades the execution efficiency and thus has to be compensated (Bezemer, 2013; Hoogendijk, 2013; Maljaars, 2006). Due to the fact, that the aimed hardware and software development tools are rather limited in their performance adjustment capabilities, additional measures have to be taken. To solve the problem at hand, the

[16] i.e. as a mean and knowledge base of future DSEs

following attempts could be conducted to improve the current status such that the deficiencies can be overcome entirely. In accordance to the prepared ECS criteria list (see also Appendix A.2) the following ECS-related and potential optimization topics have been identified:

3.3.1 Computation and Communication Hardware

On this note, it would be also possible to resume the work of van Zuijlen (2008) or Sassen (2009) which implemented the ECS on a FPGA. This would yield high performance and efficiency. However, currently the TERRA/LUNA framework and linked ECS tool chain do not support directly the FPGA as a deployment target. In other words, the currently given low-performance (hardware) target platform and design tools tend to limit the quality of abstraction options significantly. This condition requires to optimize selectively, i.e. for instance by replacing abstract constructs with specific implementations that reflect the tool's or hardware's capabilities. Ultimately, this leads to an inevitable loss of design generality and reduction of model checking coverage.

The PC/104 provides a sufficiently sized hardware interface to the PCS. At the moment three separate connectors are used to connect to all sensors, encoders and H-bridges using at least 72 general purpose I/O ports via the extension board (Figure 36, van den Berg, 2006). Due to this parallel I/O interface between the Anything I/O board and the computer unit other alternatives to the computer unit would have to provide the same physical interface capabilities.

Hence, it was analyzed how much effort it would take to establish the same status that the PC/104 provides on different platforms. A potential hardware target option would be the RaMstix board. It offers small hardware performance improvements. But it does not provide such a wide parallel port interface like the PC/104 does. The lack of a suitable parallel I/O interfaces makes it necessary to establish a serial data bus link which again has to be compliant with real-time requirements. This only leaves a distributed computation option, including an Ethernet connection as the most likely to implement communication link. Finally, the expected amount of effort to provide a working AnyIO Linkdriver or BSP, like the existing one for the PC/104, is not feasible within the project's time scope. Hence, a RaMstix-based unit option seems not feasible at the moment and thus had to be excluded from further implementation attempts.

Building a distributed system would solve several problems related to limited hardware performance. A specific solution to handle real-time communication adequately was explained in the DSE section. Here, a constant "heart-beat" signal or data object is passed every cycle if everything is in operational mode. However, besides the drawbacks of increased process load and reduced error handling, this distributed processing scenario is not yet fully supported by LUNA nor TERRA.

Subsequently, it seems apparent to utilize only a single PC/104 stack and consequently run the ECS on one CPU which makes multi-threading less crucial to exploit. The benefit is that development time can be reduced by adopting a known-to-work platform which provides tested interfaces to the PCS. However, due to the limited computation capabilities of the PC/104 CPU and memory components, a small (memory footprint) single-thread application would be preferred over a largely complex multi-threaded program. This (goal) again, is in contrast to achieving a high level of re-usability, adaptability and similar related requirements which are meant to support the designer but are likely to yield less computational efficiency. As a consequence of the conflict of interest and to provide a back-up solution, time has been spent analyzing the distributed option for the PC/104 stacks. However, currently LUNA does not provide the full support to setup a Ethernet-based network between several stacks.

3.3.2 Operating System and SDK support

Two generally feasible RTOS options have been identified that would run on the selected hardware and support the intended software (development). The first one is using the commercial QNX RTOS, Neutrino. Its micro-kernel aims specifically at low performance RT applications on embedded systems whereas at the same time supporting many small form factor hardware (i.e. with custom BSPs). The second options is to use an open-source Linux-based OS in combination with RT compatibility-adding enhancements like Xenomai, RTAI or the PREEMPT_RT patch. With respect to supporting SDK tools, only the Xenomai project offers additional simple profiling script similar to QNX's Momentics profiling suite.

Generally, there are even more RTOSs available, but the two mentioned above have been the easiest to access, and have been proven to work with the given prospective hardware [17]. Furthermore, an important aspect to consider regarding performance is how the OS will schedule or manage multi-threaded software applications. LUNA is capable of assigning CSP models to both UThreads or OSThreads (see also figure 3.2) (Bezemer, 2013).

Here, whereas exact definitions and ways of implementation might vary, Wilterdink (2011) defines UThreads (i.e. user threads as opposed to OS or kernel threads) as execution entities which share all resources (memory allocation, data objects, etc) which have been assigned by the OS to a single process [18]. However UThreads are handled by the user space (i.e. specific thread libraries) in contrast to OS threads which are handled directly by the OS. This definition is also similar to how the POSIX standard describes the separation of processes and threads (McCracken, 2002). According to Wilterdink (2011) (see also his table 5.4) Linux (esp. in combination with Xenomai) and QNX Neutrino differ in the way POSIX is implemented and thus also will differ in performance for application generated with TERRA/LUNA.

However, comparing those two, one can find a major difference in the way the thread management is set up using a QNX or Linux-based OS. Wilterdink (2011) demonstrates by using a simple test program[19] that "UThreads can potentially switch 7 times faster than OS threads on a uniprocessor Linux machine"[20], still 11% of the CPU load is already generated by scheduling the software's tasks alone[21].

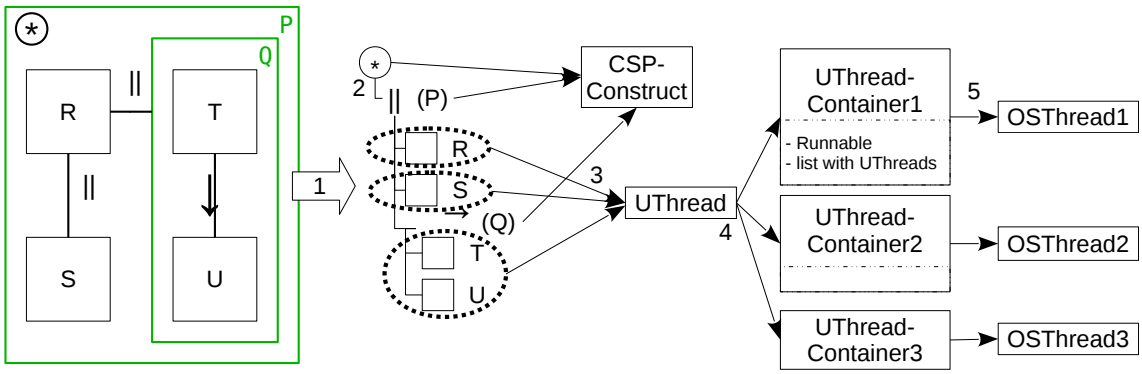

Figure 3.2: Mapping a (graphical) TERRA CSP model (left) onto OS threads (right) step-by-step (1..5) (adopted from Bezemer (2013))

Consequently, choosing a Linux-based OS would require much more hardware resources than currently available. In addition to that and similar to the RaMstix situation, using a Linux-based

[17] i.e. at least at some point in time
[18] which can be seen as a thread container
[19] whose purpose and work load is to perform thread context switches
[20] i.e. "overhead costs for UThread scheduling are not included in this measurement"
[21] test setup: PC/104, 600MHz, 1kHz sampling rate

OS would require providing custom drivers[22] for the computer hardware indented to use in the end. However the expected amount of effort to provide these is considered not feasible within the project's time scope.

This is why preference is given to the QNX Neutrino RTOS.

3.3.3 Software Architecture

Providing a sound software architecture concept is an important part of software development. The more effort is spent on this topic the more it will reduce the chance of introducing errors when implementing the design (i.e. creating the source code).

It is claimed that the proposed GAC implementation can be improved towards more efficient execution but on the account of generality and future customizability. It is even more argued that the high level of customizability of the current GAC implementation is not necessary to maintain, especially with respect to the introduced, semantic redundancy in combination with the aimed use-case[23].

With respect the current approach, three different major concerns have been chosen to focus on when creating a software architecture of an ECS. In other words, by forming the final software structure, those concerns represent distinctive levels of integration in contrast to crucial areas like Safety. First, the general (top level design) structure, this determines how the ECS part will be placed into the overall design and what needs to be added aside to make it a complete application. Next to the ECS core functionality, this includes topics like a GUI, APIs or hardware I/O (see also figure 4.1). The second concern is about integration of a mature communication between all the implemented components (being part of the 5C and can be seen as data flow management). This determines how components can cooperate with each other and share information (see also figure 4.2). The last concern is about the detailed (lower level) features or the components which will represent the ECS and perform the major operations. This is also the level where the remaining concerns of the 5C principle (as introduced above) should be applied when suitable (see also figure 4.3).

The following potential points of optimization with regard to the software architecture have been identified:

- *"Copy and Paste" feature of the GAC*

  One fundamental aspects of the GAC is its generic nature which allows to "copy and multiply" an implemented GAC such that it can serve as another processing unit. This way there's a significant development time reduction as well as a lower chance of introducing systematic failures. This idea has been adopted to the new ECS design as well as, i.e. especially the structure of the PCU components will be kept identically.

- *flattening and thinning out design levels yielding less functional redundancy and thus less overhead*

  Initially, the GAC is particularly meant to be used in a nested way. Hence, for instance, the Coordination component is set up as GAC as well and consequently consists of the same 6 main components which were mentioned above. As a result, this introduces a high level of redundancy. In many cases, where safety is the most important requirement, redundancy is highly intended. However in case of the PCS where computation

[22] which should offer access to the AnyIO board and periodic timer

[23] General use case: A system designer using the ECS template will only specify parameter within the software components rather than their structural relations (as they are fixed being the GAC)

resources are very limited, it presents an unused performance potential. In regard to that, the redundancy feature has been removed from the GAC. As a consequence there is only one Safety layer that is directly put at the hardware I/O interface of the software, i.e. where bit information is read from or written to the processor ports which are connected to the plant. The motivation behind this approach is the assumption that a computation error only presents a serious danger (to the plant hardware or its environment) if the error is not filtered out before the hardware I/O interface or entirely unaltered passed to actuators. The benefit of such a less detailed safety handling is the reduced computational overhead (during normal operational state). On the other hand, having less redundancy means errors cannot be caught at a very early stage any more which again reduces the time to take proper counter measures. Hence error handling has to be designed in a broader sense (e.g. "if any error then do the one most secure counteraction").

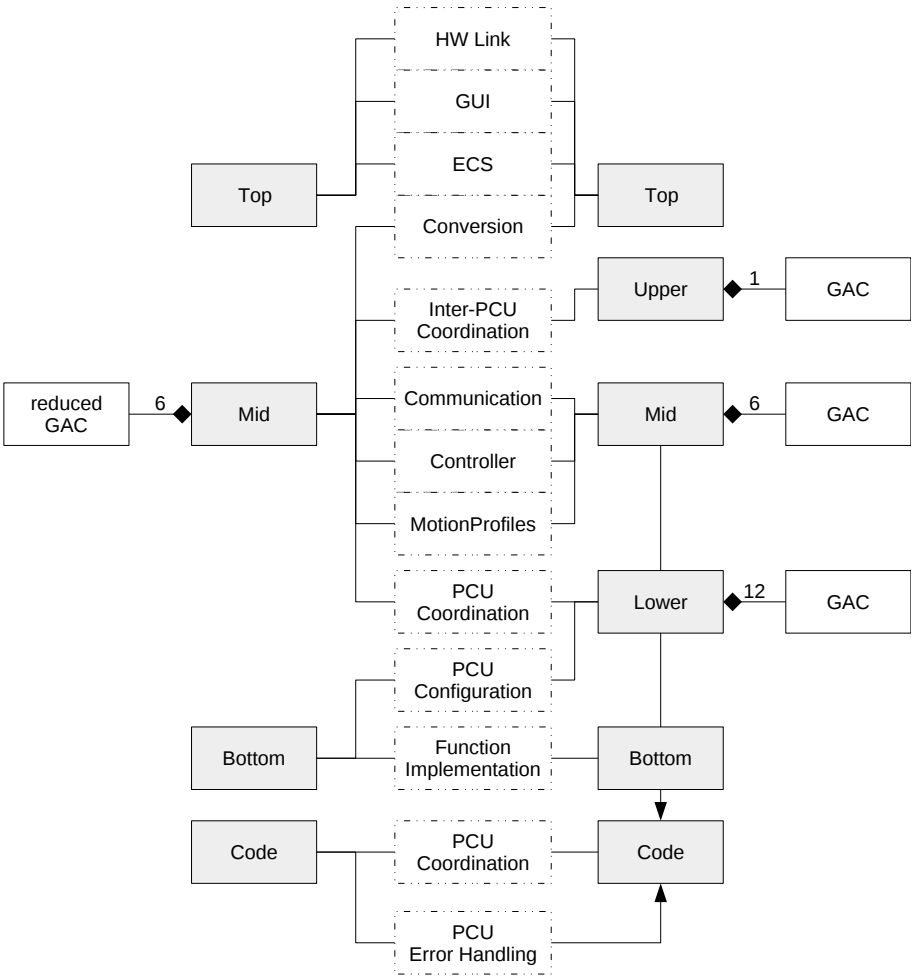

Figure 3.3: Model structure optimization - current (left) implementation hierarchy and functionality distribution compared to previous study of Hoogendijk (2013) (right)

  - Realization: Final implementation consists of 3 layers (Hoogendijk (2013) used 5)
    * top layer (PAR arrangement)
      · ECS Core Application [24]
      · Hardware Interface [25]
    * mid layer (PAR arrangement) - represents the PCUs and their relationship
      · InterPCU COM Link [26]
      · FSM MotionProfile [27]
      · Control [28]
    * bottom layer (SEQ arrangement)
      · IO SEQ structure [29]

- *Serialization of tasks*

  A crucial part of designing software architecture is to manage how tasks or processes will be scheduled. Since hardware resources are limited and often various kinds of data have to be processed simultaneously, the order in which contributing processes are executed becomes essential. This becomes even more a core issue when those concurrent processes are dependent from each other in the way that they share or pass on data. Often this data has to be processed in a certain order including feedback to preceding processes. This is where it is easy to introduce data flow deadlocks. Here, the TERRA/LUNA tool chain assists the designer by fixing the order of execution by using channels to forward data. After creating a first version of the software architecture or when adapting a previous design, it is possible to analyze the path the data takes for additional performance optimization.

  The GAC template, as specified by Hoogendijk (2013), basically handles different kinds of data mostly in series, although all processes are set up completely in parallel (see also figure 3.4). The advantage of an entirely parallel construct is that only inter-task data-dependencies will determine which of the concurrent tasks is to be executed. However, if there are many tasks without dependencies, this can result in an increased (CPU) work load caused by many issued context switches which the OS scheduler then has to work off. Due to the fact that the potential HW target[30] will be most likely a single-core processor[31] it might be beneficial to explicitly align certain concerns of the GAC in a series of consecutive tasks. This approach could be seen as cooperative scheduling in which task gets finished before it yields to the next task. Another step could be reducing the architectural overhead even more and merge several components into one component. In other words, if processes or tasks turn out to provide no feedback to their predecessors or any different component they can be merged to one process which just performs all the operations on the data at once. This way additional context switching load can be reduced by using less reader and writer processes. Furthermore, it is even possible to move certain parts of the structural design to code level and gain more performance while reducing abstraction and the ability to verify the software architecture formally.

[24] Consists of the actual composition of all PCUs
[25] Handles the conversion and data distribution
[26] Handles communication between PCUs
[27] Handles coordination of a PCU
[28] Handles core computations of a PCU
[29] Execution sequence: Reader, code block, Writer
[30] i.e. for this project
[31] in other words, reduced capability of handling task concurrency

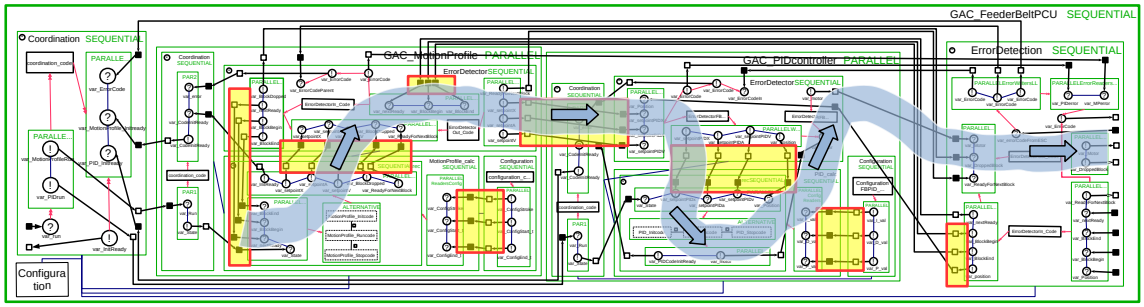

Figure 3.4: General Data Flow through Hoogendijk (2013) GAC implementation (here: Feeder Belt)

- *ChannelMerge and data object grouping*

  Similar to the point mentioned above and after thorough analysis of the data flow, different data objects can be merged to one single multi-data-type object (hence, the generic type of that object could be then e.g. a *struct* type). This kind of object structuring is not yet supported by TERRA and can only be achieved by manually altering the generated source code. In order to estimate the potential work load reduction of merging channels several tests[32] have been performed (see also Appendix A.3.3).

- *Using a single multi-data-type communication link between PCUs*

  Basically, a GAC has no specific inter-GAC communication protocol. Every data object has its own data link (a.k.a channel) which connects the same type of sub-components and data types. E.g. the Coordination components have type-dedicated channels which let them exchange only status data as well as the ErrorDetection components can only exchange specific error occurrences. Generally, this represents a good separation of concerns and makes token-based Data Flow analysis straightforward[33] but also introduces a large number of reader and writer processes which contribute to a higher context switching load (thus computational overhead). In order to reduce that load a single (soft-real) time communication link (called InterLink) between the respective PCUs has been implemented which passes several types of information from different data sources.

3.3.4 Special Algorithms

There are two kinds of algorithms which have to be taken care of. One generic algorithm has to be developed which handles the controlling of the actuator. The second algorithm concerns the monitoring and state handling of the entire ECS. In contrast to the control algorithm the intended FSMs have to be created and checked from the ground up.

Control algorithm Within the intended tool chain the creation and implementing of the control algorithm is supposed to be handled by 20sim[34]). One PID control algorithm was designed by Maljaars (2006) and has been proven to work generally efficient by succeeding studies. This PID controller would also be sufficient for the current application [35], i.e. no optimization potential has been identified that would improve the execution performance nor the design ab-

[32] i.e. a simple Producer/Consumer program running on multi-cores as well as single-core hardware and with several combinations of amounts of channels or Writer/Reader process pairs has been compared w.r.t timing, memory allocation and thread count to the same program but with the respective merged channels implementation

[33] i.e. one token represents only one kind of data

[34] other MDD supporting and TERRA/LUNA compatible tools are generally also possible, however are not yet available

[35] provided that the controller coefficients have been adjusted according to the achieved sample rate

straction level. However due to the current combination of TERRA/LUNA and the given QNX compiler certain C++ arithmetical functions that 20sim uses cause runtime errors. The actual source of the issue could not be eliminated completely within the time scope of this project. Alternatively, a simpler (thus easier to verify) and suitable control function has to be added manually to the source code. Since the general speed of the PCS can be adjusted seamlessly and the setup provides enough steering room for maneuvering a feed-forward controller can be used. Naturally this also involves lower steering accuracy and limits the maximal speed an actuator can be controlled, but does not pose an issue for the setup operability, in general.

FSM As Ran (2012) states and in order to reduced the communication overhead and the related context switching all FSM should be implemented manually. Due to its importance for the ECS the consistency of the FSM has to be checked formally. However, this can only be done externally from the established tool chain, as is discussed in the respective section 3.3.6 down below.

Even though for more advanced applications it would be preferable (esp. for debugging and monitoring) to cover many different states a PCU could be in, it is aimed to implement only as few states as necessary (e.g. init, run/turn/move left, run/turn/move right, stop) in order to keep calculation load and state complexity low.

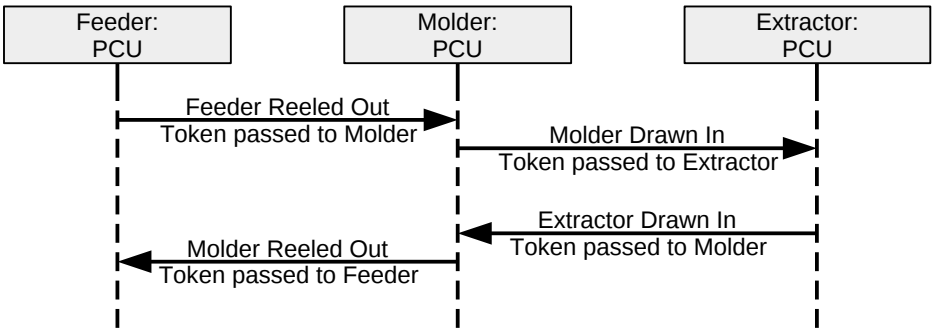

Figure 3.5: Sequence Diagram for the token passing scenario

3.3.5 Code Generation and Manipulation

Several implementation details have been chosen to be realized manually on code level due to missing integration support of the design framework or in order to gain more performance efficiency.

Motion Profile Generation Previous studies have been using functions provided by 20sim's code generator. Those functions use extended set of math functions which are called every control cycle. However, it is argued that in order to reduce the regular process load the motion profile generation can be reduced to a single trigonometric function which is only called once during the initialization phase. Furthermore, up until now acceleration and deceleration have been calculated independently but can be simplified to only one calculation [36] and followed by mirroring the calculated values. All results will be stored in a LUT and read according to the current status of the to-be controlled actuator. Apart from that, it was also found that when compiling 20sim code additions would cause compile errors which seem to be originated in an unsupported version of the math library[37].

ChannelMerge Several options have been evaluated how to optimize the utilization of Reader, Writer and Channel instances. It is found that reducing the communication overhead

[36] see also Figure 4.7, phase 1 where the sin function is only calculated for 0 to pi/2 and then mirrored to phase 3; phase 2 is a fixed value

[37] *i.e. _Sinx* error is given when compiling with linked 20sim code

and the related context switching can improve the ECS execution efficiency. One way to do this would be to perform post-code-generation operations on the source code files. An example is given in the Appendix A.11 which groups data objects similar to the principle of a multiplexer. However, that would include substantial changes to the software architecture and might render the design model difficult to verify as it evolves into quite a divergent functional representation compared to the code.

Due this fact, including the general lack of the framework support for this kind of optimization, a simpler way has been chosen to give a proof-of-concept of this concept. Information exchanged between the PCUs are reduced to boolean type and merged into one multi-bit data object by design. The information is passed in a ring topology using only one channel respectively (see also figure 4.2).

The decision for using this topology was made due to a two-way or duplex ring network introducing race conditions. Furthermore a centralized network would represent a central point of failure and decrease performance as well as the level of scalability (i.e. due to decentralized design it would be easier to distribute the ECS load on several devices).

Controller Previous studies showed in detail that the (loop) control algorithm developed by Maljaars (2006) is working fast and stable. The algorithm can be realized using 20sim's code generation feature or by manually implementing a C++ code based function. However, in order to achieve a maximum of performance efficiency the controller has been reduced to a simple feed-forward controller.

3.3.6 Tool Chain and WoW

As starting point the WoW proposed by (Bezemer, 2013) is adopted for most parts. This design methodology focuses on Model-Driven Design (MDD) in combination with model (co-)simulation and iteratively executed stepwise refinement and is best supported by model-based tools like the TERRA/LUNA framework, 20sim or FDR3 (2016).

In regard to the stepwise refinement, the V-model has been chosen to be applied to this project including the focus on model-driven, test-driven and use-case-driven approaches [38]. This is done in accordance to Zwikker and Gunsing (2015) who state "the V-model is a much-used and well-structured method" when developing CPS-like systems (i.e. which involve carrying out mechanical and electronic hardware concepts alongside software design). A reason for this is its flexible "development of decomposed system elements" and its immediate support for documentation[39].

The V-model, named after it is shape, can be divided in 3 work phases: Development, Realization and Testing. A system designer would initially stepped through the procedure from left to right. Each steps allows to iterate over a certain sub-set of steps. The goal is to finish a project with a positive validation of the final (software) product.

[38] which is also the order of priority

[39] but which is also seen as a potential risk by Zwikker and Gunsing (2015)

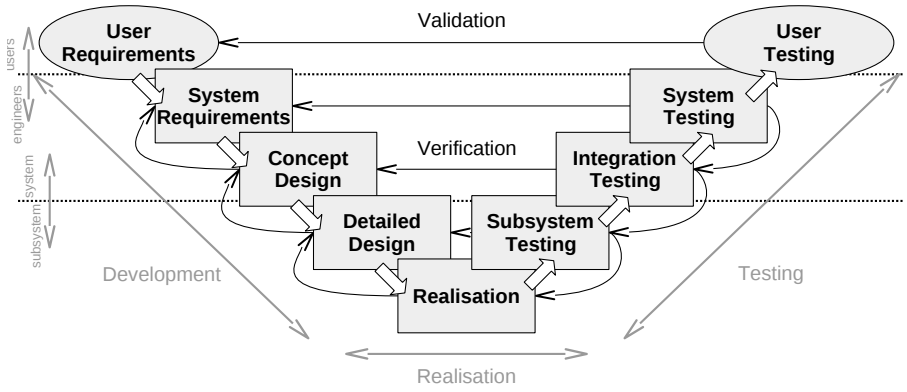

Figure 3.6: V-model with iteration loops between subsequent phases (Zwikker and Gunsing, 2015)

In accordance with the requirements the following tool chain is proposed to be used to achieve a reproducible as well as easily adjustable and verifiable ECS program.

FSM and UPPAAL Normally UPPAAL (see top of figure 3.7) and FDR represent similar purposes, i.e. model checking features. However due to the fact that there is no native FSM development support in TERRA yet (like selecting guards, events or invariant conditions), it was chosen for the ease of designing to use UPPAAL over FDR. This comes at the cost of loosing model incompatibility between TERRA (CSP) and UPPAAL (Timed Automata)[40]. Nevertheless, implementing any FSM on model level has to be excluded from the tool chain due to lowered performance as a result of communication overhead introduced by how the CPP LUNA plugin is generating code from CSP models. As a temporary solution the UPPAAL models will be manually implemented on source code level.

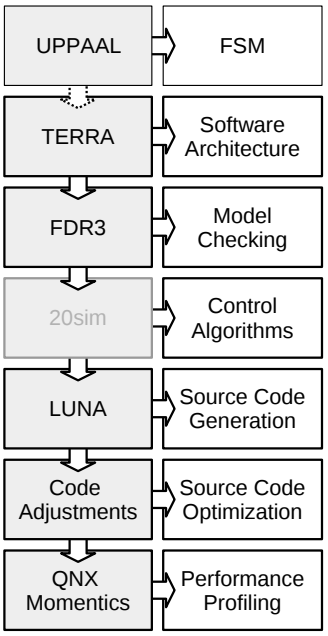

Figure 3.7: ECS tool chain chart

Dependency Validation and CSP Model Verification In order to formally check the integrity of the ECS design, the created CSP models will be analyzed with the tool FDR3. The results can be used to determine if the application is life and deadlock-free.

Benchmarking and Profiling As Fraleigh and Shulman (2004) state, the fundamental goal when designing (real-time) embedded system software is to make sure that the final product meets the user's demands. This is why every requirement or specification has to be checked upon its conformance. In addition to validating functionality, determining performance in order to compare with alternative solutions can be crucial as well. This is called benchmarking. To improve the software execution, performance bottlenecks have to be identified by profiling communication and computation during early design phases. Typical ways to determine this is to use (system) profilers like the software analysis framework *Valgrind*, Google's CPU profiler or QNX Momentics profiler suite. Here, performing a dynamic program analysis, profilers measure essentially where and how long a CPU would spend time executing parts (or threads) of an

[40]The study of Ouaknine and Worrell (2002) indicates that automated translation from UPPAAL to TERRA models could be implemented with a manageable amount of effort. Besides both models are being stored in XML format it is understood that Closed Timed Automata can be translated directly into Timed CSP.

application. System profilers fall into two categories: those that report actual or exact measurements and those that report statistical data. Both methods provide important information and the decision about which type to use will depend on the purpose of the analysis.

Here, the following figures will be used to draw conclusions about the (performance) efficiency:

- Since it is argued that most performance can be gained from reducing the number of processing overhead like context switches, the *number of running or created threads* is good indicator[41]
- Execution time spent and sample time verification [42]
- Using the *file size* of the ECS program gives information about the complexity of the ECS (here, less would be better to meet real-time requirements more assured)

3.3.7 Behavioral Correctness Checking

In order to work correctly the control software needs to be guaranteed to operate even during hard-real time conditions. The requirement "real-time guarantee" is always connected with a specific time scope that represents the ECS execution cycle period or sample time. Previous studies accomplished sample cycles of 1ms. This figure was generally accepted by all other studies without further verification. Since the PCS represents a rather slow process due to the high inertia of the actors and the demo metal block objects it is estimated that larger sample times are still applicable in accordance to the motion profiles and less accurate movements. This consideration provides which will become necessary when finding feasible solutions in the design space exploration.

3.4 DSE Conclusions

Due to the design framework supporting different HW and OS options only very limited or not at all, the completed DSE does not provide as much design flexibility as is needed to meet all requirements completely. The main issue is that software produced with TERRA/LUNA (in combination with the AnyIO drivers) only works sufficiently on a PC/104 system with QNX. Although in general the OS can be considered a good choice, the computer hardware, it is running on, does not provide the enough signal processing resources.

Hence, the realization of the introduced system specifications is entirely determined by the capabilities of the design framework. However, so far MDD abstraction and execution efficiency are not yet fully compatible and cannot be realized to the same extend at the same time.

This can only be compensated if the most complex and resource demanding functions of the ECS, i.e. control algorithm and FSM handling, are implemented on code level (at the expense of design abstraction) to increase the execution efficiency.

Since those functions are also the most crucial ones in regard to reliability, additional formal checking measures are necessary.

Apart from that Hoogendijk (2013)'s initial GAC structure offers also potential to be further adjusted according to the remaining requirements. While aiming still for a generic design template, an optimized GAC can be achieved by reducing several areas of hierarchical complexity, functional redundancy and inefficient data separation.

[41] Whereas in a Linux for every running processes there is status file generated that provides information about *Non/voluntary context switches* (see also Appendix A.13), there seems to be no similar method like this for QNX, this is why the LUNA debugging feature is enabled to count started processes.

[42] e.g. with QNX Momentics profiling tool System Profiler as presented in Appendix A.15 or FDR3's CSPM profiling feature that provides the number of times a function was called (see also Appendix A.1)

4 Implementation

This chapter deals with the realization (i.e. specific implementation and validation) of the proposed ECS design concept, separated by model (i.e. abstract software architecture) and code level. Several implementation details are picked up and examined more closely according to the adopted MDD and (co-)simulation-based WoW (see also figure 2.1). Hereby, the following concerns have been used to reflect the steps taken during the development process:

1. General software structure/architecture design
2. Algorithm design (complex algorithms/control algorithms)
3. (a) (Model) Simulation and verification of the parts of the design concepts
   (b) Hardware/Software in the loop simulation
4. Realization (deployment)

As described above in chapter 3.3.6, the development process was carried out by going through the steps of the V-model, i.e. for every issue or concern, regarding a subsystem, a respective subset of requirements has been extracted to create a detailed design of a sub-solution. Here, the produced documentation of the chosen design concept in form of models or other ways of design specification is used to generate hardware-specific source code. In parallel to the realization of each subsystem and in order to determine to which level requirements have been met, several kinds of validation tests have been performed ranging from single unit testing to testing the complete system as a whole (i.e including the new subsystem).

4.1 Software Architecture

4.1.1 ECS Model Design Implementation

The final control software for the Production Cell Setup was designed in reference to Hoogendijk (2013)'s generic design pattern (GAC). The now adjusted implementation comprises however significant structural reductions or rearrangements. The main tool for developing the software architecture of the ECS remained the TERRA model editor. For reasons of performance, the Top Level architecture design is kept as simple as possible - only consisting of two process blocks, as shown in figure 4.1 down below. It is also the only entity that the recursion attribute is assigned to which again is requiring every (sub-) part of it to be essential (i.e. any functional redundancy is to be avoided otherwise the computational load would increase unnecessarily). The first implemented operational block, called `modelECS`, contains the core functions of the complete ECS programs which will be discussed in more detail further down below. The second block, called `HWInterface`, abstracts away the FPGA hardware register access details from the actual PCS control implementation as a result of separation of concerns. In addition to that it also serves as One-to-any channel workaround [1](see also Figure A.1). This feature is not yet implemented in TERRA/LUNA and could have reduced the number of writer processes in the current ECS program noticeably (similar to the ChannelMerge optimization, see also Appendix A.3.3 for context switching test results). Furthermore, this process block is also meant to handle data conversion but currently serves only as bit value interpreter in this function for specific output data. Optionally, a third UI process block could be added in combination with an external host PC running a GUI. However, there is no adequate communication bus available yet that would connect the control[2] and the user PC[3].

[1] i.e. a channel that can be connected to one writer and several readers
[2] running the ECS
[3] running the GUI application

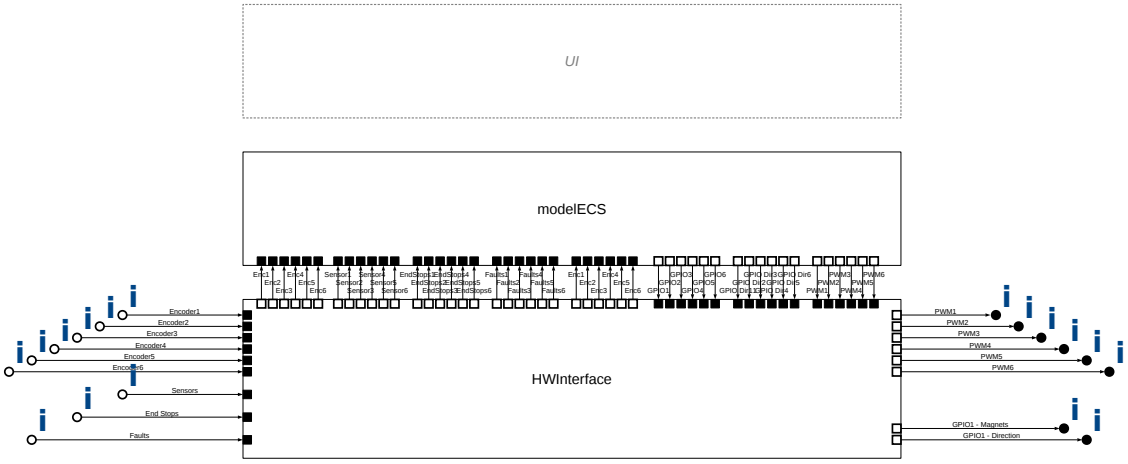

Figure 4.1: Top Design Level showing ECS block (top) and the Hardware Interface (bottom)

Every functional entity is supposed to execute exactly once per cycle, thus, the repetition property is solely reserved for the top level. This is done in order to avoid structural redundancy and to follow TERRA/LUNA WoW preferences. It can also be followed that less fragmented software structures (i.e. reduced utilization of parallel and repetition attributes) will consequently result in less fragmented task scheduling as well as reducing the number of task activation checks and context switches per cycle, yielding more performance especially on single core CPUs. Apart from that, every cycle start is triggered by the timer process [4] which was chosen to be implemented in the `modelECS` block (see also figure 4.2 on the left). This way, the ECS process is always called first and the Hardware Interface follows according to the channel relations as it is less resource demanding and of lower priority.

The lower architecture level of the `modelECS` block consists of 6 PCU group blocks. Each group block is identical with the others and consists of a controller block `PCU_*_Ctrl`, a motion profile block `PCU_*_MP` and a inter-PCU communication block `PCU_*_InterLink`. As indicated in the DSE chapter (see also section 3.3.3), particular attention was paid when creating a software structure (see also section that would not only fit the demands of the PCUs but also every other mechanical unit based on what was proposed as feasible design 3.2.3). This includes also the capability of communicating amongst the PCUs to pass information about the status of a unit or a critical situation. The communication between the PCUs is buffered to show that information passed are not time critical which imposes less constraints on the communication bus (see InterLink components in figure 4.2). The chosen structural composition would also allow to distribute the ECS program such that every PCU group block could run on a separate computer hardware. This would make it more fault tolerant with respect to hardware failures while being able to minimize computational resources at the same time.

[4] i.e. a writer process writing to a special Periodic Timer Port

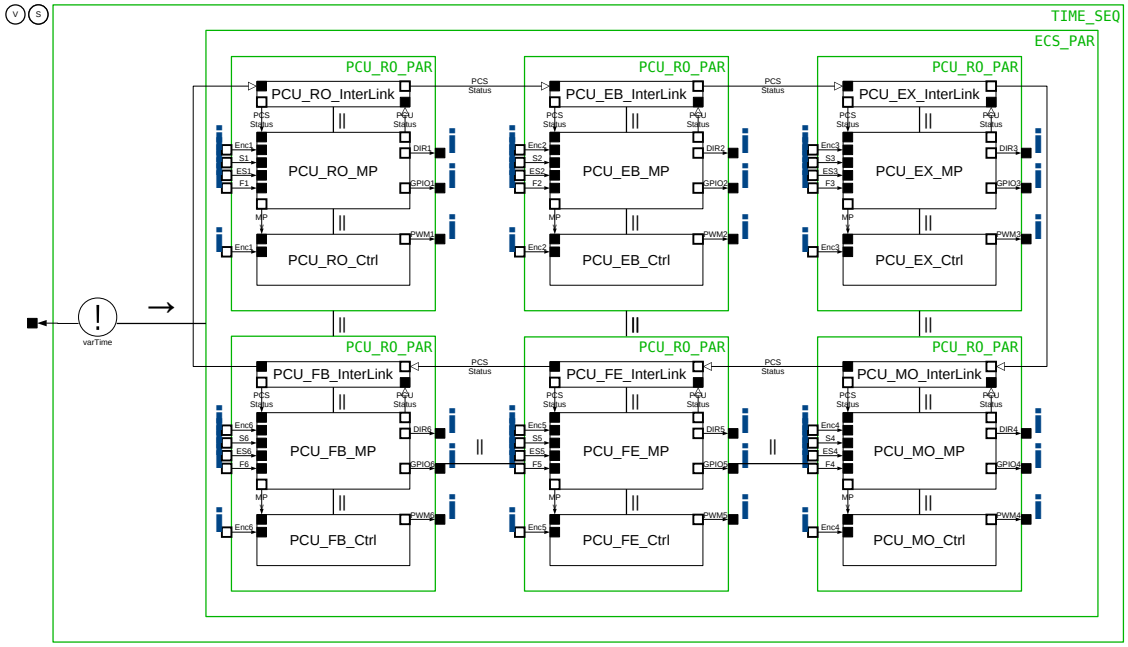

Figure 4.2: ECS Design Level showing all 6 PCU objects

The lowest architecture level for each of the three functional blocks (as can be seen in figure 4.3) completes not only the intentionally flat overall architecture but solely consists of Readers, Writers and Code process blocks. Here the IO SEQ structure (i.e. an execution sequence starting with Reader processes, followed by a code block and terminated by a set of Writer processes) has been applied to the model composition to enforce a preferred data flow.

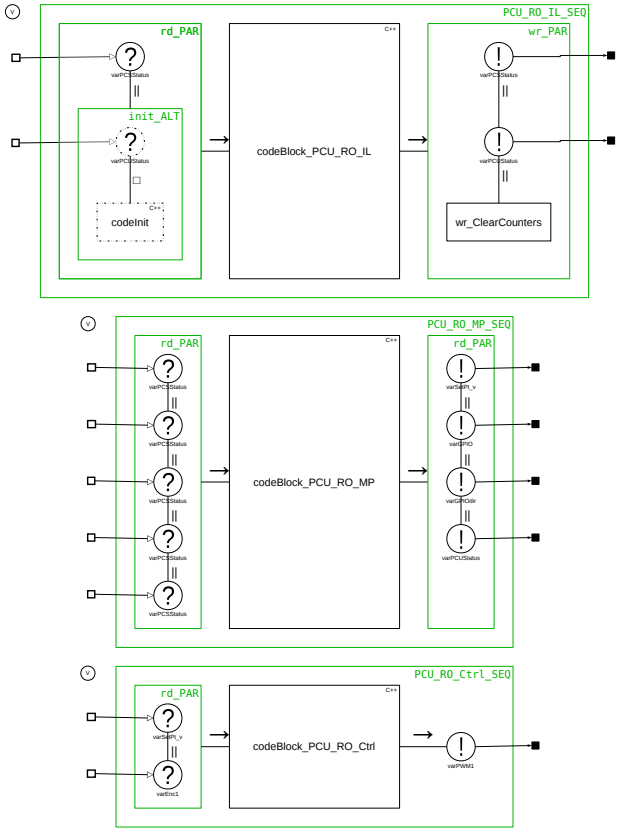

Figure 4.3: Bottom Design Level of InterLink (top), MotionProfiler (middle) and Controller (bottom)

Regarding the overall ECS architecture, there are several elements which have been not implemented in a detached way or have been omitted completely compared to the proposed GAC component (see also figure 3.1 and 3.3 in chapter 3). The main reason to do so is in regard to functional insignificance of those blocks as result of the current ECS specifications. Hoogendijk (2013) recommends to separate the Configuration concern structurally from the other concerns. The current ECS implementation, however, only requires configuration during the initialization phase which is why all relevant assignments have been realized in the respective constructor sections of the Computation-like blocks, reducing structural redundancy and the overall computational payload.

4.1.2 Testing / Experiments / Measurement

A thorough verification of the model as well as functional correctness has been performed assisted by formal model checking tool FDR3. However, only parts of the final model could be proven to be consistent. When checking the complete model the tool FDR3 crashes [5]. This test has been repeated with different, mostly simpler versions of the ECS which still causes crashing the tool. The same behavior occurs also for Hoogendijk (2013) implementation. So far the precise cause could not be

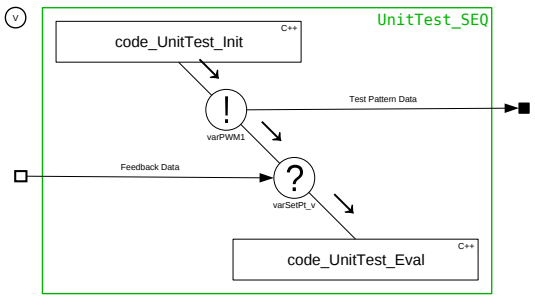

Figure 4.4: Example of Unit Testing on model level

[5] i.e. it gets stuck in an infinite loop that eventually consumes the complete main memory of the development PC and thus crashes the entire OS when not terminated before.

identified. It seems though, that the problem is linked to the usage of serial relations between blocks[6]. At this point it can only be assumed that a certain construct of relations triggers a state space explosion that the tool can not handle or catch as a fault. Further tests also indicated that it does not seem to be linked directly to the application of buffered channels, alternative constructs or the number of process blocks in general.

In addition to the formal checking a test-bench-like setup, using the TERRA model editor to simulate a variety of test scenarios (see also figure 4.4), has been created and applied to the ECS model.

With regard to communication overhead a significant reduction of potential context switches could be achieved while still maintaining a high level of software abstraction and performance quality.

CSP model	Writer	Processes	Readers	total
`modelECS`	*43*	*49*	*54*	
`HWInterface`	*38*	*2*	*27*	
`modelArch`	-	*2*	-	
sum	81	53	81	**215** (current approach)
`SyncProdCell`	351	175	351	**877** (Hoogendijk, 2013)

Table 4.1: Comparison of the number of introduced CSP processes for different approaches

4.1.3 Evaluation

To this point, a complete formal check of an ECS could not yet be performed[7]. It is recommended to spend more time on the investigation of what exactly causes the failing of the tool and if it can be prevented from a modeling point-of-view without changing structural dependencies completely. This seems crucial, otherwise FDR3 has to be excluded from the standardized tool chain as formal model checker and replaced with an alternative. Furthermore, it has been shown that using the test-bench like setup on model level is very time consuming. Especially when performing unit tests on components which are parts of a larger composition this can turn into a laborious task. The issue is mainly that every port or Writer/Reader pair needs to be connected properly even when not meaningful to the test. A feature known from textual programming languages, called "commenting out", but which is also available in graphical model editors like Mathworks (2016) Matlab/Simulink[8], could decrease significantly the amount of time spent on preparing the entire ECS to test only a single part of it. Another tool-inherent difficulty is how to deal with the ring initialization which describes a situation of circular dependent units[9] (like it is shown in figure 4.2). Here, all PCU units depend on the fact that the preceding unit, which they're linked to, gets initialized first. However, there is no actual start in a ring-dependency-like structure. The issue is caused when all components make use of the IO-SEQ principle[10]. There are many ways how to solve this. The easiest way is to not make use of the IO-SEQ principle or alter it for a specific part of the ECS. However that is not always possible or preferred. It would mean for instance for the current ECS design that a PCU component cannot be kept generic as it has to be altered completely to work in a ring composition.

[6] if they get replaced by parallel relations FDR will not crash and proves the ECS to be deadlock free which is expected when every relation is parallel

[7] as it keeps allocating memory when it is tried to analyze CSP models of the ECS (see also bug reports)

[8] Here, a system designer can hide (i.e. it becomes greyed out) a model from simulation.

[9] in other words, channels need to be ready to execute during initialization although not enabled by the Writer process

[10] A typical way in processing data sequentially, i.e. data is first received by a CSP Reader process, then processed and finally passed onto the next entity by a CSP Writer process

This is why a solution was chosen that can be kept generic, i.e. easily duplicated without alterations. IO-SEQ Readers can be adjusted such that they will be placed in an *Alternative* construct which catches the first execution of the ECS and replaces the Reader with a dummy process that does not depend on another Writer process to be executed first. However this is more of a workaround than a real solution to the issue. First, it presents an unnecessary addition to the architecture which does not contribute to the actual purpose of the ECS and second, it might cover design or implementation flaws as FDR3 is for instance not capable of detecting certain lifelocks due to the fact that the *Alternative* property has to be handled on code level[11]. As a prospective feature of TERRA it might be useful to enhance Writer and Reader processes such that they have an initialization property that allows to get passed when executed the first time. Another solution could be to be able to give channels an initialization value[12] that they can pass to the respective Readers when checked for the first time.

4.2 Source Code Additions and Adjustments

Several concerns are implemented on code level to reduce the communication overhead between the Computation and the Coordination units.

4.2.1 Implementation

Finite State Machine One of the concerns involve the FSM structures that have been developed, simulated as well as verified using UPPAAL. Two different kinds of FSMs have been implemented in the respective Motion Profile units of each PCU. One type FSM handles the state management of the PCU covering universal states like 'Initialization' or 'Homing' as well as specific states that describe a certain movement or rest position (as can be seen exemplary for the Rotator unit in figure 4.5). Another FSM (representing the communication between the PCUs) is handling the token that is passed between the 3 PCUs 'Feeder', 'Molder' and 'Extractor'. Appendix A.16 shows the complete simulation of all created models of the PCS.

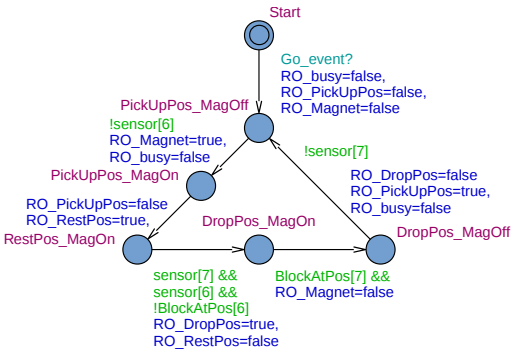

Figure 4.5: FSM example model using UPPAAL (Rotator PCU)

ChannelMerge As indicated in the feasibility section it was not possible due to various issues to introduce a complete design optimization step to the workflow that guarantees structural coherence of the model as well as functional coherence of the produced source code and yet is still easily maintainable (i.e. altogether, resembling an automated post-design optimization routine).

However, to be still able to evaluate the performance gain, the principle idea of merging channels has been implemented in the *InterLink* components of the design. Here, only one data object is passed which holds different kind of information from different contributers. In this way the number of applied channels could be reduced while keeping source code and model functionally identical. The figure below shows how specific information (like the status of ac-

[11] e.g. there's a counter which should but never gets iterated on code level, thus the program ends up in a lifelock due to the value 0 of the counter being a valid value to proceed i.e. idle wait. Such design flaws cannot not be checked by FDR3.

[12] similar to dataflow analysis where one can pre-set egdes between certain nodes with initial tokens

tuators or progression of the *FEMOEX* safety token[13] passing) has been grouped and stored within one 32-bit data object.

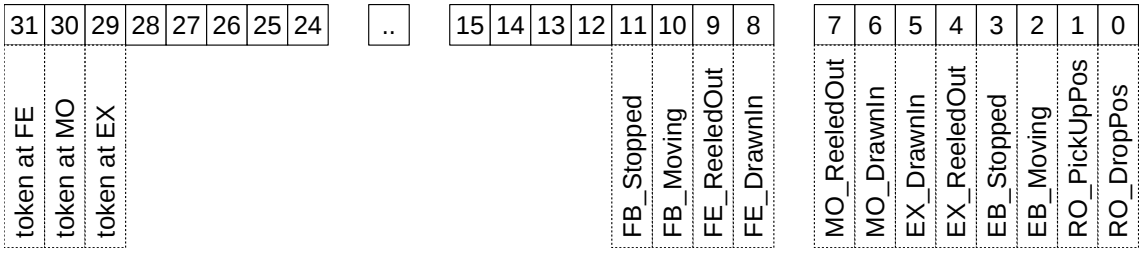

Figure 4.6: InterLink shared data object bit representation chart

Motion Profile and Homing As proposed, the handling of the Motion Profiles is implemented such that (prior to the transition into operational mode) all relevant set points are generated and stored in a LUT. To reduce allocated memory size of the LUTs, only phase 1 of the complete motion profile (see also figure 4.7) is generated. The remaining phases are either represented as fixed numbers (i.e. no additional LUT entry) or read in reverse order (in other words mirroring phase 1 to get phase 3)[14].

In addition to using the motion profiles in operational mode to let the actuators move the demo metal blocks, all actuators will be steered to a safe homing position during the initialization phase, i.e. when the ECS programs starts for the first time.

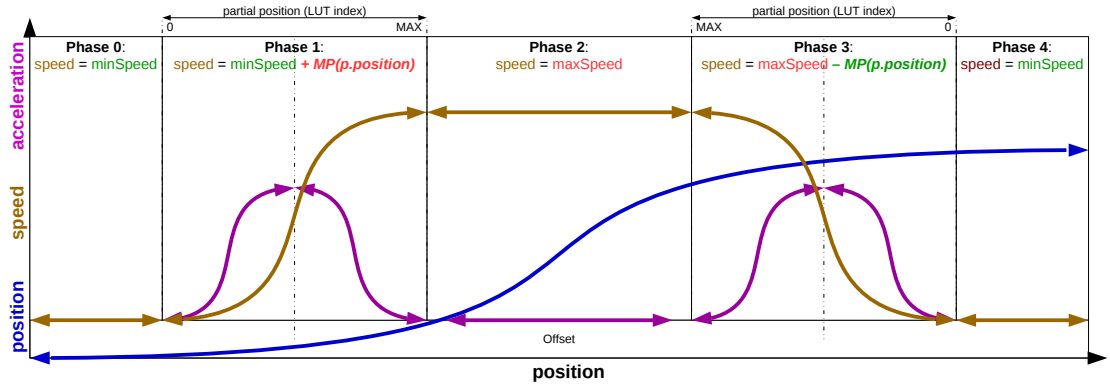

Figure 4.7: Diagram of how Motion Profiles are composed

4.2.2 Testing / Experiments / Measurement

With regard to evaluating the created program several techniques and tools have been used:

- FDR3 Determining deadlock freedom of CSP design
- LUNA Determining number of active threads and CSP process debugging
- QNX Momentics Timing and performance evaluation
- UPPAAL FSM simulation and verification

[13] a token that is passed between the three PCUs: Feeder (FE), Molder (MO), Extractor (EX); to enforce a specific execution sequence. The tokens indicates which PCU is enabled to act.

[14] The LUT array is consequently read in the following way
Phase 1: *0* .. *0* ; Phase 2: *0* .. *Max* ; Phase 3: *Max* .. *Max* ; Phase 3: *Max* .. *0* ; Phase 4: *0* .. *0*.

With respect to the last point, every individual FSM as well as their interaction with each other has been verified. Assisted by the model checking tool UPPAAL, a FSM can be simulated as well as formally tested using property expressions like "`E<> deadlock`" (here, it is verified wether "*eventually for all paths there is a deadlock*" which should and did return *Property is not satisfied*) for the created model.

4.2.3 Conclusions

Overall, a significant execution load reduction, especially compared to Hoogendijk (2013)'s approach, could be achieved.

In addition to reducing the number of activated threads to a quarter, the sampling time (or cycle period) could be lowered from 10ms to 2ms, occupying the CPU now to approx. 66% on average. However, to this point the given tool chain does not yet support the validation of hard real-time guarantees (i.e. determining an accurate WCET) or obtain a sufficiently large time sample which would allow to draw conclusions on the accuracy of the cycle period. As a result this has to be compensated by taking particularly large execution time tolerances (also known as "time jitter") into account. Lastly, the program size could be reduced by more than a half of the original size of Hoogendijk (2013)'s approach which becomes crucial when taking into account that the entire program is kept in memory and frequently read.

5 Conclusions And Recommendations

5.1 Summary

Up to this point, several previous approaches providing a comprehensive solution to the cyber-physical applications have been discussed. For most of them it was possible to use supporting design tools like gCSP. However, all of them put emphasis on different requirements like safety, efficiency or real-time reliability while giving them even varying priorities. This is where the present project takes up previous accomplishments.

The achieved results show that it was feasible to create a consistent, working ECS solution using the current TERRA/LUNA design framework. The DSE, performed at the beginning of the project, however revealed that the currently necessary combination of HW and design framework can be identified as the main limiting factors on how the system specifications can be realized. As consequence, increased execution efficiency can only be achieved at the expense of (model) design abstraction. A complete separation of concerns (i.e. 5C) was therefore not possible to be realized exclusively as system models, i.e. highest abstraction level. Instead a wide range of manual low-level as well as high-level optimization options have been proposed and applied during the process of software development where possible. Respectively for the model (i.e. abstract software architecture) and code level, an optimized GAC could be achieved by reducing several areas of hierarchical complexity, functional redundancy and inefficient data separation. Hereby, the created generic design template represents the best compromise between necessary abstraction and performance optimization.

Finally, the result of this project can be used to further influence the on-going development of the TERRA/LUNA framework and how it can be used efficiently.

5.2 Requirement Evaluation

Further emphasis of the project was set on providing comparable results which would enable prospective designers to elaborate better on achievable system implementation quality in prospective projects. Therefore, a general set of requirements, derived from a specific use case, has been determined and prioritized using the MoSCoW method. All key requirements which were also set to be a Must have been achieved except for the Dependability/Reliability measures requirement (R2). Here, the sub-item Verification (R2f3) was only partially realized as will be discussed in more detail down below.

5.2.1 Performance and Efficiency (R1)

With regard to performance and as show by Hoogendijk (2013), source code created with TERRA/LUNA is still likely to produce software that puts a rather high load on computing resources in comparison to manually written programs. Yet, an efficiency improvement was achieved by several optimization techniques that yielded also an significant reduction in Context Switching (Should requirement R1f1). Now, executing more PCS control tasks within lesser time than initially would have been possible. This led to a sampling rate of 2ms which is the best result for programs created with the TERRA/LUNA ECS development framework, so far. Compared to previous studies which again used completely different tools or target hardware, however, it is still twice the time as achieved then.

Several optimization techniques have been proposed and shown to effectively aid in general performance gain. However, it was also shown that the way they have been applied is still to be improved in order to meet other requirements like project re-usability or maintainability (which is picked up again by the Re-usability and Maintainability (R3) section down below) better.

It was also examined to what degree it is possible to distribute the ECS processing load over several (independently working) hardware platforms (Could/Won't requirement R1f2). Using an Ethernet network and a non-hard-Real-Time (error) communication protocol the selected hardware would have been able to achieve this. Yet, it was found that the given design framework is not yet capable of integrating this feature into the software generation seamlessly. Hence, it was not further pursued to be implemented.

5.2.2 Dependability and Reliability (R2)

One of the most important requirements to satisfy is the aspect of dependability when it comes to industrial data processing. This is why several methods have been used to ensure a high level of dependability (which was a Must requirement) for the developed ECS.

Next to taking precautionary measures by adding fault tolerant characteristics or error detecting services to the ECS, even more effort was spent on determining the level of operational reliability.

Since TERRA/LUNA enables the designer to create graphical models that are translated into CSP, one is also able to use consecutive tools like FDR3 to check and verify the architectural design at hand.

Due to being still under heavy development itself and some CSP-related translation issues, the final model in this project, created with TERRA/LUNA, could not be checked as a whole. Instead only distinctive sub-parts have been checked alongside general unit testing. Particularly crucial parts like FSM algorithms (Should requirement R2f2) which were not realized on model level have been checked (Must requirement R2f3) using out-of-line tools like UPPAAL [1].

5.2.3 Re-usability and Maintainability (R3)

Using the TERRA/LUNA framework (Must requirement R3f1), a heterogeneous model[2] was achieved that combined a high level of abstraction, including enough universality to serve as template for different control applications (Should requirement R3f3). However, consisting of several sub-models, the one that represents a general PCU had to be reduced to abstract only its core functions: communication, coordination and control. Other concerns like safety matters or the actual implementation of FSMs were realized only on the lowest (i.e. code) level. While the latter is done to achieve a performance increase, it results inevitably in a reduction of project re-usability and maintainability.

Regarding platform independence (Must requirement R3f1) only the created models achieved to be entirely independent from the targeted platform. Still, code generation was only possible for QNX-based systems due to the current limitations of TERRA/LUNA and the provided platform drivers. The modularity of the design is part of the testibility requirement (Should, R3f4) and meant to support especially unit testing. However setting up respective tests was found to take too much effort when done manually. Here the issue mainly originates in creating and (re-)naming great numbers of new data objects and data links (channels).

5.3 Recommendations and Potential Improvements

Throughout the project several points have been identified that need further improvements or general consideration but which were not within this project's scope.

[1] which are not part of the seamlessly linked model translation tool chain and are likely to introduce a risk of translation errors

[2] a single model containing multiple hierarchically grouped sub-models

5.3.1 GAC

In regard to separation of concerns, it has been found that actual modeling of functional components like (system) configuration can result in avoidable overhead. Especially when executed every cycle [3] alongside the remaining operational payload (like controller or sensor handling), this can take over most of the hardware resources the more complex a system gets. This is why a reduced alternative design template is being proposed that focuses more on control and safety rather than covering the whole separation of concerns (i.e. including the modeling of very specific implementation details) on the most abstract level. In this regard, there are examples of crucial operations like FSM-related tasks which also contribute significantly to the communication overhead but cannot be excluded from being executed every cycle. Here, based on the necessity for performance gain and in exchange for the level of abstraction, maintainability and verifiability, a range of low-level optimization proposals have been given and evaluated on their effectiveness and justification for application.

Generally, however, the concept of having GACs is understood to aid significantly in the course of software development despite the downsides of needing more hardware resources. It is also found that the demand for earlier and more dependable prototypes out-weight the decreasing hardware cost factor.

5.3.2 Hardware

Due to several limiting factors (like lack of drivers or BSPs and hardware interfaces) it was not possible to test different computational hardware platforms in depth other than the PC/104 stack. In correspondence with what is said above in the GAC section, the typical hardware target for ECS developed with TERRA/LUNA should be generally considered at least in the range of modern Mini-ITX platforms. Since CSP models are meant to run concurrently, they can benefit more from those multi-core platforms which offer not only more hardware resources but are able to execute tasks in parallel. Thus more complex RTOS systems can be used while still providing developers with enough performance margin to experiment (i.e. function integration without the need for immediate optimization). Another option with more potential to exploit parallel computing, addresses the current embedded systems trend called "Internet of Things" (or short "IoT") even better. Here, instead of one single hardware platform several very small units are used to distribute the ECS's diversity of functions and can be easily scaled to the software's hardware resource demands. This is why development towards more support of rather smaller platforms like the RaMstix is also recommended, including the focus on implementing universal communication links like USB, Ethernet or even wireless solutions.

5.3.3 WoW and Tool Chain

Dependability verification is a crucial part of ECS development, this is why it was also a focus of this project. So far, this had not been the main focus of the framework. Consequently, this work showed that there are still work flow obstacles to overcome with regard to formal model checking[4] or specific software adjustments like performance optimization (incl. model optimization as well as code optimization). In addition to functional checking (i.e. like formal model checking and simulations), being able to check temporal accuracy (e.g. time jitter evaluation) is needed as well. This is why it is recommended to expand the tool chain such that it meets the requirement for more options to profile or benchmark and to enable the designer to do more system behavior property checking. Another aspect which should be considered to become a part of the WoW philosophy is the design for optimization which should be also reflect in the proposed tool chain. The idea is to, in a first step, create a model-driven design that represents

[3] but not contributing to the system's activity

[4] e.g. created mCSP is not fully compatible with the format FDR3 expects

the designers non-functional requirements [5] and then in a second step leaves the option to meet functional requirements [6] using additional optimization techniques or tools. An example how this is not yet fully addressed, is in regard to one of the most important figures when it comes to RT property evaluation. The WCET that is used to give a hard RT guarantee should be easier to determine than it is at the moment to give the designer a better understanding of the status of his current implementation. Giving a guarantee involves knowledge about the hardware the software is running on as well as a sufficient amount of explicit statistical performance data. Especially the latter is difficult to acquire and thus also difficult to integrate into the automated work flow. Nonetheless, it is recommended to put more emphasis on this issue as it is a crucial part of developing real-time systems.

5.3.4 TERRA/LUNA Design Framework

Since this project also aims to evaluate the current status of main design tool TERRA/LUNA with respect to usability and software generation quality, several aspects have been picked up and are elaborated in more detail (for complete list of favored improvements or a report of encountered bugs see Appendix A.12).

Stepwise Refinement Generally, it was found that making repeatedly alteration to the (model) design introduces a constant high amount of repetitive manual work and thus has to be improved. Testing, validation and performance optimization or model simplification (e.g. like ChannelMerge) have to become more significant features of the framework as it is a crucial part of the WoW and the very idea of what the frameworks tries to accomplish. In fact, after every design refinement step it should be possible to perform not only correctness verification but also seamlessly and automated profiling and optimization tasks. The TERRA editor specifically could assist in model refinement better by providing automated (re-)naming of variables and process blocks[7] as well as introducing the option to 'disable' parts of the main model[8]. Next to stability improvements regarding Copy & Paste and the mentioned CSP model translation compatibility with FDR3, model simulation and automated test-bench creation could be features of the TERRA editor, too.

Design Analysis Usually a new software design is created by starting with a high-level representation of the various functional components comprising the system in mind. Following that and for optimization purposes, often the system under development has to be profiled to identify the functions which consume the most processing resources and/or time.(Maxfield, 2008) In regard to this project, the latter step was performed using several tools like QNX Momentics or simple code analysis. However, in order to achieve a better understanding of the created software design, it is recommended to add more support towards code analysis[9] and simple profiling like determining the number of components (which is linked to thread count) or how often certain channels are used.

Design Distribution Furthermore, TERRA/LUNA could assist in setting up distributed ECS hardware units (i.e. like proposed above; ECS working on different HW targets communicating with each other) while integrating virtual environment like the KVM (ARM-emulation) or VirtualBox (x86-emulation) better in the simulation setup. Here, a special type of channel that

[5] like Modularity, Testability or even Safety

[6] i.e. explicit implementation that target a certain sampling time, control algorithm or fault tolerance mechanism

[7] this is especially useful when just creating a test system that is meant to be a proof of concept test rather than an actual implementation

[8] in analogy to 'out commenting' written source code in common programming languages

[9] i.e. adding automatically specification language code that helps not only measuring the time certain functions take but also if they are executed correctly similar to the Java's JML or C's Frama-C

represents an Ethernet link (i.e. buffered socket-based link) could be added to the standard components of TERRA.

OS support Another aspect that needs more attention concerns the support of a wide variety of RTOS versions. So far, only one commercial, micro-kernel-based RTOS is fully supported by the current framework. In accordance to Gupta (2002) who claims that "[for] complex embedded systems, these kernels [incl. QNX] are inadequate as they are designed to be fast rather than to be predictable in every aspect.", it is recommended to increase the support (and thus platform independence) for more open-source and freely available (hybrid) RTOS solutions like GNU/Linux+Xenomai or GNU/Linux+PREEMPT_RT. Although it could be claimed that the results might not result in high performance solution (compared to commercial products), it would allow much more researcher to join and participate in the development of TERRA/LUNA.

Code Generation Next to creating graphical models, code generation the other crucial core functionality of TERRA/LUNA. Although, C++ serves as an adequate programming language that is covered by many compilers and different processor architectures, it can be still beneficial to extend to a wider selection of supported languages that the models can be translated into. Supporting code translation to VHDL, Handel-C or Haskell might make it easier to deploy to different system and benefit from their different languages characteristics in terms of efficiency, safety or system analysis.

5.3.5 ECS

LinkDriver Generally, one of the reasons why elaborating on different computational hardware targets was quite limited is the fact that the original QNX AnyIO LinkDriver source code was not found. The only working driver was installed (as binaries) on one PC/104 stack and was not extractable from the OS and thus also not portable to other (OS) systems like Linux. Consequently, it is recommended to create new LinkDrivers, taking new hardware platforms and Operating Systems into account as indicated above.

GUI One of the missing features of the ECS is yet a meaningful GUI that aids in evaluating and debugging even more use case or fault scenarios. Typical control functions could include changing speed or direction (e.g. rewinding the mechanical process to a certain point) up to the point where the use can select certain operational modes like jogging or even repeat specific movements or fault scenarios[10]. In anticipation of a simulation mode within TERRA[11] syncing the mechanical process to validate its accuracy might be interesting as well.

[10] i.e. assuming fault scenarios can be stored as templates or pre-loaded within the (debug) system
[11] this feature was still under development and not available for the current project

A Appendices

A.1 FDR Profiling capabilities

Using the commands:

- options set cspm.profiling.active On
- load test.csp
- graph (show)
- profiling_data (show)

FDR can aid to pin point bottlenecks, i.e. model areas where most of CPU load is generated.

A.2 Project Criteria List And Corresponding Fields Of Scope Of Application

The following table lists all criteria which have been selected from a complete set of possible software project key aspects.

ECS Criteria	Area Of Application
Resource Efficiency (R1)	
• Less context switches	SW architecture (GAC) : simple FSM
	SW architecture (GAC) : Channel bus
	SW architecture (GAC) : data clustering
(multi → single core)	SW architecture (GAC) : PAR → SEQ
	no OS (e.g. process outsourcing → FPGA)
• Resource usage (distributivity)	centralized → decentralized
	HW: FPGA (incl. soft core), CPU,..
• Code generation	Tool: LUNA / 20sim / VHDL? / ...
Reliability (R2)	
• Safety Layer (HW protection) - Implementation	SW architecture (GAC)
◦ State-oriented	
◦ Event-handling	
• Real-time - Implementation	RT design Tool
◦ Concurrency	
◦ Synchronization and Communication	
• Testability / Debug-ability - Implementation	SW architecture (GAC)
• Testability / Debug-ability - Check/Proof	Formal Model Checking : CSP + FDR
• Simulation (domain specific model execution)	Partially : 20sim
Re-usability Increase / Design Effort Reduction (R3)	SW architecture (MDD / GAC)
	SW architecture (domain separation / 5C)
	Tool usage (WOW)
	SW architecture (FSM)
	SW architecture (design readability)
	OS support increase
◦ Platform independence	Multi-OS support / HW abstraction
◦ Scalability	SW architecture
Accuracy / Precision of operations	Data type / calculation cycle time

A.3 Translation of non-functional into functional requirements

In order to validate if the demands can be satisfied with the achieved results several non-functional requirements can be translated into functional requirements. Benchmark test can then provide information about how much a certain goal is accomplished (see also thesis documentation file *[MSc] [M] DSE Chart - Requirements Dependency Tree*).

A.3.1 ECS qualities

- Reliability, Safety, Integratity -> Verificable via FDR3, 20sim, UPPAAL (implies simulate-ability, testibility)
- Maintainability, Re-usability -> MDD (software modularity, structural hierarchy)

A.3.2 Functional determinism

- state space completeness -> FSM should cover all important states
- formal correctness -> no formal errors like deadlocks
- behavioral correctness -> operations should executed in the right way

A.3.3 Context Switching test for ChannelMerge optimization

In order to estimate the potential work load reduction for the ChannelMerge code optimization several test have been run that covered combinations of multi/single-core deployment as well as amount of channels on a Linux-based OS. Results have been compared with respect to memory allocation, timing and thread count. They also indicate that there is a linear relationship between the amount of reader/writer process pairs and the scheduled amount of context switches. (see thesis documentation file *[MSc] [M] Context Switching Test* for complete test results)

A.4 DSE Chart - Requirements

In the course of the Feasibility Study the main requirements have been picked up and analyzed towards their dependencies and potential realization.
(see thesis documentation file *[MSc] [M] DSE Chart - Requirements Dependency Tree*)

A.5 DSE Chart - Implementation considerations

The specific domains (or areas of concern) that have been identified for optimization are: processor hardware, operating system, software architecture and code generation. In all domains the current realization is being compared against alternative options while evaluating them in regard to the defined requirements as well as practical aspects. They're all inherently interdependent which makes it difficult to maintain an order of making decisions. On some level the number of alternative options will determine the priority of the concern. However it is possible that decisions made at some point in time have to be overruled later by a different decision. This could be a consequence of an initially lower ranked concern that had be rated higher later on due to eliminated cross-incompatible alternatives. Consequently, a recurrent way of evaluating certain decisions is inevitable.

In the course of the Feasibility Study the main requirements have been picked up and analyzed towards their dependencies and potential realization.
(see thesis documentation file *[MSc] [M] DSE Chart - Implementation Tree*)

A.6 DSE Review - Database

Several approaches have been compared with each other regarding:
(see thesis documentation file *[MSc] [M] DSE Review Database*)

- design programming language and meta-languages
- hardware target (processor type) as well as OS target
- used tool (chain)
- implementation details (e.g. Motion Profiles and Loop Controller)
- benefits and drawbacks

A.7 DSE - Communication (Protocol) considerations

	TCP	UDP
Reliability	+ reliable (incl. error-checked)	− unreliable
Overhead	−− big header	+ smaller
Processing	+ ordered	− no order
Application	+ streaming	+ datagrams
Packaging	+ equally long parts of packages	− whole packages)

A.8 DSE - ARM hardware comparison

The following table summarizes the found (small form factor[1]) computer alternatives by giving a short overview of the PROs and CONs:

[1]The property of complying with a small form factor is adopted from the fact that the cyber part of the system is meant to be embedded into the physical part as it is common in an industrial environment. Still, it is a soft requirement rather than a hard one.

	RaMstix	Arduino	Raspberry PI	Beaglebone
Description:	ARM	ARM	ARM	ARM
PROs:	- actively used and applied in the research group, thus expert knowledge available	- huge developer community (potential expert knowledge)	- huge developer community (potential expert knowledge)	- official QNX BSP available for QNX 6.5.0 as well as QNX 6.6.0
	- provides almost double the computation power (1 GHz over 600 MHz).[2]	- potentially easily extendible to meet the interface requirements to the AnyIO FPGA board	- provides some more computation power (900 MHz over 600 MHz)	- Linux support
	- Linux support			
CONs:	- does not really provide significantly more computation power (700 MHz over 600 MHz) than the PC104.[3]	- less expert knowledge than PC104 and RaMstix	- does not really provide significantly more computation power (720 MHz over 600 MHz) than the PC104	- less expert knowledge than PC104 and RaMstix
	- no supported QNX BSP[4]	- no native parallel interfaces that matches with the AnyIO FPGA board	- less expert knowledge than PC104 and RaMstix	- no native parallel interfaces that matches with the AnyIO board
	- no native parallel interfaces that matches with the AnyIO FPGA board	- no AnyIO driver implementation so far	- no native parallel interfaces that matches with the AnyIO board	- no AnyIO driver implementation so far
	- no AnyIO driver implementation so far		- no AnyIO driver implementation so far	
In Question		- QNX and Linux support	- QNX and Linux support	

As a summarizing of the table above it can be seen that there are significant risks and drawbacks of all options. Hereby, the closest favorable, but still not acceptable, alternative option is the RaMstix platform. However, due to the lack of an already implemented interface to the PCS (incl. the driver development part), it is unlikely that a requirement compliant platform could be achieved in the aimed time frame.

Note: The table presented above already includes cross-dependencies to the software part of the cyber implementation. In other words, the effect of certain design considerations which will be discussed later, are already taken care off here.

A.9 DSE - Programming language conversion

Due to the lack of HDL conversion capabilities of the TERRA/LUNA framework, a short feasibility check has been made concerning options to use external tools to convert:

- C/C++
- Haskell (CEAS group)

- CSP models
- SystemC

.. to VHDL, Verilog or SystemVerilog in order to use FPGA deployment tools like Altera Quartus or Xilinx's SDK. More precisely, it was researched if a respective tool exists and if there has been documentation about its conversion quality and reliability. The result of this check showed that even though some conversation combinations do exist (like the Haskell-to-VHDL CλaSH tool developed by the CAES group), none of them are meant for a productive environment, i.e. reportedly they lack significantly efficiency and/or can only convert very specifics subsets of the given input language and thus serve only academical purpose.

A.10 DSE - UPPAAL model code generation

Authors	Target Language	Availability	Comment
Pajic, Lee, Sokolsky and Mangharam (2013)	Matlab/ Simulink models	n/a	Although code can be generated from Matlab/Simulink models it is not considered a straightforward solution and might introduce inconsistency
Kristensen, Mejlholm and Pedersen (2005)	C	available[5]	Even though C and C++ are related language there are still incompatibilities. Additionally there seems to be no experiences with the tool from other research communities
Opp, Caspar and Hardt (2011)	C	n/a	Even though C and C++ are related language there are still incompatibilities. Additionally there seems to be no experiences with the tool from other research communities
Rensink and Stoelinga	RT-Java	n/a	Project has not started yet[6].

A.11 Optimizations - ChannelMerge script

So far, there exists only one kind of channel communication between TERRA CSP process blocks, the one-to-one channel (see also figure A.1). It is possible, though, to add this feature later manually on code level.

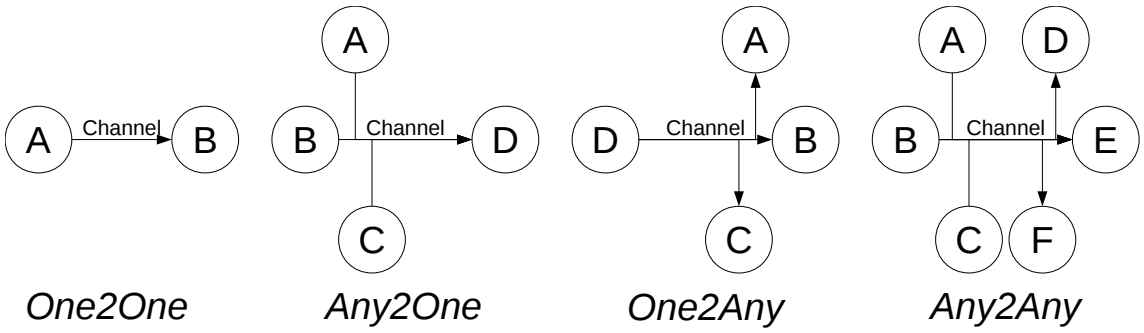

Figure A.1: Overview of all different kinds of shared communication (i.e. types of CSP channels) (Broenink, 2014)

The idea of ChannelMerge takes up the Any2Any option by combining several channels two one channel and forwarding different data objects at once. A script that would realize the channel and object merging operation on the source code should execute the following steps. However, a drawback is the extensive range of preparations which have to be performed on the software model in TERRA.

Preparations A generic `struct` data type (in addition to `int`, `boolean`, `uint`) has to be added to the TERRA editor that indicates that data objects are classified to be grouped or merged. The main reason to do this is the ease of parsing through the source code files.

Apart from that the system designer would have to specify a group object with that data type for every group of objects and link it to a single, dedicated channel.

Additionally, the system designer would have to follow a specific data object naming convention that selects or marks each data object that belongs to a specific group.

- e.g. `varName_cobjAB12` which would give:
  - for 'AB' max. 26^2 group identifiers
  - for '12' max. 10^2 data object identifiers (per group)

All objects with the pattern 'AB' would be then grouped into an object called `varName_cobjAB`.

Script algorithm

- Start at lowest hierarchical (TERRA) level
- Parse Header file and count data objects per group
- Add `struct` with identified objects as members
- Remove remaining object entries which have been grouped

A.12 TERRA/LUNA framework - tool improvements

Being still under heavy development the tools which build up the TERRA/LUNA software design framework showed occasionally malfunctioning or lack of enhanced functionality. (for a complete list of detailed improvements or bug reports see thesis documentation file *[MSc] [TERRA] [LUNA] bug + improvement report*)

A.13 Performance Evaluation - Linux built-in profiling capabilities

- First, locking the process to a single processor to simulated the same environment as PC104:
  `$ sudo nice -n -20 taskset 0x1 ./path/to/bin/modelname`
- Then, get the PID of the respective process
- Finally search in the process' status file for the context switches section:
  `$ ./path/to/bin/modelname & grep ctxt /proc/$!/status`
- The result should look like this:
```
voluntary_ctxt_switches:     41
nonvoluntary_ctxt_switches:  16
```

A.14 Performance Evaluation - CSP model profiling preparations with FDR3

The formal model checker tool FDR3 offers simple performance profiling capabilities. In order to identify performance difficulties, the FDR profiler can be applied on a CSPM script. Aiding by spotting so called performance bottlenecks, the tool returns the number of times a function was called to be extracted. The following preparations have to be made:

- `:options set cspm.profiling.active On`
- `:load test.csp`
- `:graph (show)`
- `:profiling_data (show)`

A.15 QNX Momentics profiler suite

QNX Momentics provides a range of different profiler.

At first the QNX softare project under consideration, has to be set up such that the *Build Options* enable Profiling (for performance reason, preferably "Call Count Instrumentation").

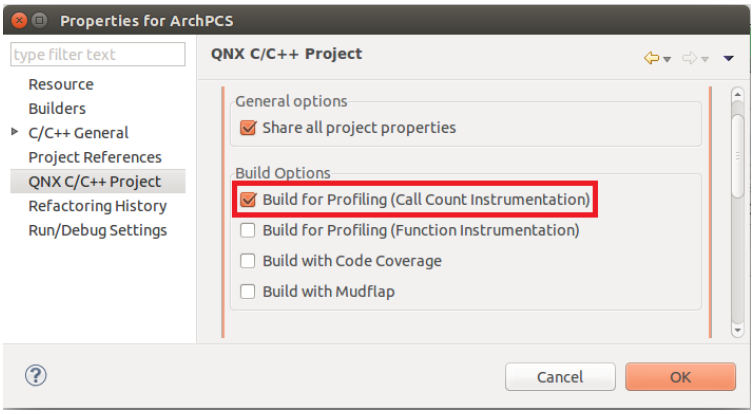

Figure A.2: QNX Momentics project build preparations

The next step is to select a profiling tool (like the "Application Profiler or the System Profiler".

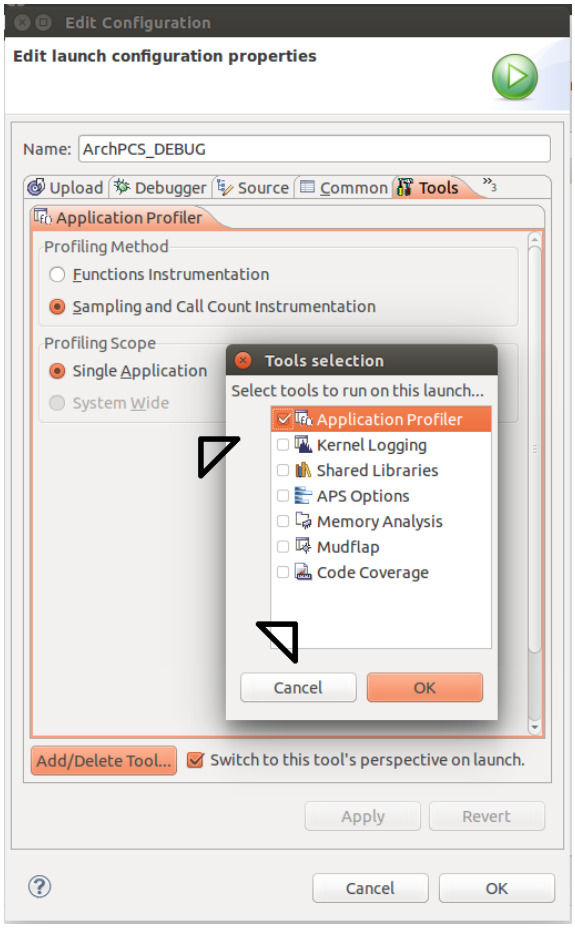

Figure A.3: QNX Momentics profiling tool selection

After running and collecting data, the results can be analyzed by opening the respective *Perspective* (e.g. QNX system profiler perspective).

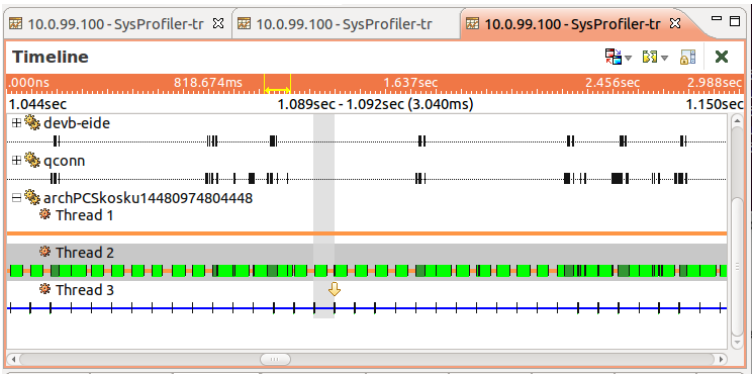

Figure A.4: QNX Momentics profiling data analysis

A.16 Complete PCS simulation using UPPAAL

In order to verify the correctness of the FSMs implemented in the PCU models a complete model of the PCS was created, simulated and formally checked. (for the actual UPPAAL models see thesis documentation file *[MSc] [UPPAAL] PCS model.xml*)

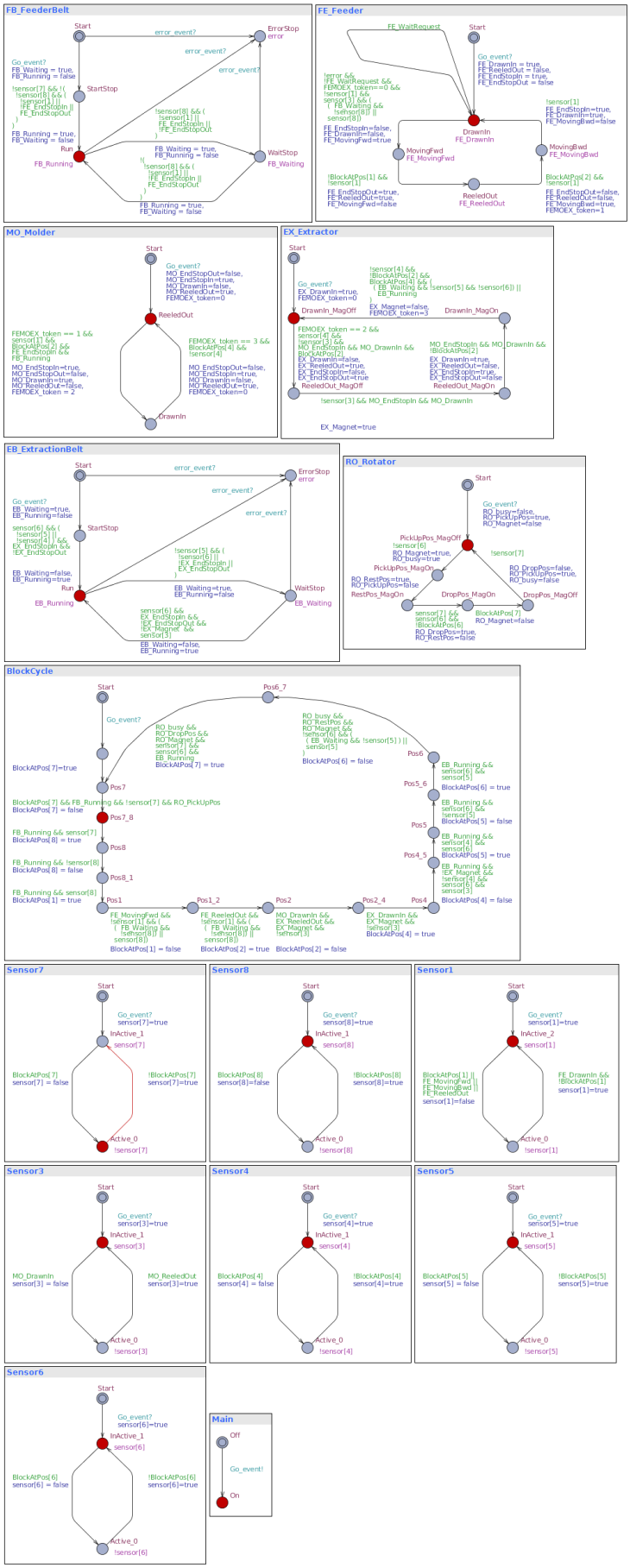

Figure A.5: Complete simulation of PCS using UPPAAL

Bibliography

20-sim (2015), Merging Process Structure And Design Freedom.
http://www.20sim.com/

Aksit (2001), *Software Architectures And Component Technology*, Kluwer.
http://www.springer.com/us/book/9780792375760

Baheti and Gill (2011), *The Impact of Control Technology*, IEEE Control Systems Society.
https://ieeecss.org/general/impact-control-technology

Beck (2003), *Test-Driven Development by Example*, Addison Wesley.
https://books.google.de/books/about/Test_driven_Development.html?id=gFgnde_vwMAC&redir_esc=y

Bezemer (2013), *Cyber-Physical Systems Software Development*, University of Twente.
https://www.ce.utwente.nl/aigaion/attachments/single/1159

Bezemer, Groothuis and Broenink (2011), *Way of Working for Embedded Control Software using Model-Driven Development Techniques*, University of Twente.
http://doc.utwente.nl/79349/

Broenink (2014), *Lecture Slides - Real-Time Software Development*, University of Twente.
https://osiris.utwente.nl/student/OnderwijsCatalogusSelect.do?selectie=cursus&taal=en&collegejaar=2015&cursus=191211090

Broy (1997), *Requirements Engineering for Embedded Systems*, University of Technology Munich.
http://www4.in.tum.de/publ/papers/femsys_boesswet_1997_Conference.pdf

Clegg and Barker (1994), *Case method fast-track: a RAD approach*, Addison-Wesley.
http://catalogue.pearsoned.co.uk/educator/product/Case-Method-Fast-Track-A-Rad-Approach/9780201624328.page

Den Haan (2009), *An Enterprise Ontology-based Approach To Model-Driven Engineering*, Delft University of Technology.
http://repository.tudelft.nl/assets/uuid:e2093132-9db7-4cba-bc68-9355f93cb9e3/Johan_den_Haan_-_An_Enterprise_Ontology_based_approach_to_Model_Driven_Engineering_MDEE.pdf

FDR3 (2016), *FDR3 - CSP refinement checker*, university of Oxford.
www.cs.ox.ac.uk/projects/fdr

Fowler (2003), *Patterns of Enterprise Application Architecture*, Addison Wesley.
https://bd.eduweb.hhs.nl/es/2014-embedded-market-study-then-now-whats-next.pdf

Fraleigh and Shulman (2004), *Use ProïňĄling to Meet Application Performance Goals*, RTC Magazine.
http://rtcmagazine.com/articles/view/100033

Groothuis and Broenink (2006), *HW/SW Design Space Exploration on the Production Cell Setup*, University of Twente.
https://www.ce.utwente.nl/aigaion/attachments/single/22

Gupta (2002), *Tasks and Task Management (lecture slides)*, UC Irvine.
http://www.cas.mcmaster.ca/~sancheg/EE_UCU2006_thesis/biblio/Tasks%20and%20Task%20Management%20%28slides%29.pdf

Haugen (2010), *Advanced Dynamics and Control*, TechTeach.

https://www.hioa.no/tilsatt/finnhaug

Hoogendijk (2013), *Design Of A Generic Software Component For Embedded Control Software Using CSP*, University of Twente.
https://www.ce.utwente.nl/aigaion/attachments/single/1145

Jacobson, Booch and Rumbaugh (1999), *The Unified Software Development Process*, Addison-Wesley.
http://www.pearsonhighered.com/educator/product/Unified-Software-Development-Process-Paperback-The/9780321822000.page

Jerome (2000), *Building Better Business Systems with Scenarios*, Software Development.
http://www.drdobbs.com/building-better-business-systems-with-sc/184414618

Jovanovic (2006), *Designing Dependable Process-oriented Software - A CSP-based Approach*, University of Twente.
http://doc.utwente.nl/55447/1/thesis_Jovanovic.pdf

Klotzbucher, et al (2013), *The BRICS Component Model - A Model-based Development Paradigm For Complex Roobotics Software Systems*, ACM.
http://lucagherardi.it/wp-content/papercite-data/pdf/brics2013component.pdf

Kristensen, Mejlholm and Pedersen (2005), *Automatic Translation From UPPAAL To C*, University of Aalborg.
http://mejlholm.org/uni/pdfs/dat4.pdf

LabVIEW (2016), *LabVIEW System Design Software*, National Instruments.
http://www.ni.com/labview/

Laplante (2004), *Real-Time Systems Design and Analysis*, John Wiley and Sons, Berkeley.
http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471648280.html

Lee (2007), *Computing foundations and practice for cyber-physical systems: A preliminary report*, EECS Department, University of California, Berkeley.
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-72.pdf

Lee and Xue (1999), *Analyzing User Requirements by Use Cases: A Goal-Driven Approach*, IEEE Software. v.16 n.4.
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=776956

Lethbridge and Laganiere (2005), *Object-Oriented Software Engineering - Practical Software Development Using UML And Java*, McGrawHill.
http://www.pearsonhighered.com/educator/product/ObjectOriented-Software-Development-A-Practical-Guide/9780137269280.page

Lorenz (1993), *Object-Oriented Software Development: A Practical Guide*, PTR Prentice Hall.
http://www.pearsonhighered.com/educator/product/ObjectOriented-Software-Development-A-Practical-Guide/9780137269280.page

Maljaars (2006), *Controllers for the Production Cell Set Up*, University of Twente.
https://www.ce.utwente.nl/aigaion/attachments/single/1057

Marwedel (2011), *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems*, Springer.

http://www.springer.com/us/book/9789400702561

Mathworks (2016), *Product: Matlab and Simulink*, Mathworks.
https://www.mathworks.com

Maxfield (2008), *FPGAs - Instant Access*, Elsevier.
http://www.sciencedirect.com/science/book/9780750689748

McCracken (2002), *POSIX Threads and the Linux Kernel*, Ottawa Linux Symposium.
https://www.kernel.org/doc/ols/2002/ols2002-pages-330-337.pdf

Meijer (2013), *Finite State Machines And The Embedded Systems Design Flow*, University of Twente.
https://www.ce.utwente.nl/aigaion/attachments/single/1164

Opp, Caspar and Hardt (2011), *Code Generation For Timed Automata System Specifications Considering Target Platform Resource-Restrictions*, Information Technology Journal Vol. 7, No. 14, July - December.
http://www.it.kmutnb.ac.th/journal/pdf/vol14/ch07.pdf

Ouaknine and Worrell (2002), *Timed CSP = Closed Timed Automata*, Tulane University.
http://www.cs.ox.ac.uk/joel.ouaknine/publications/express02.pdf

Pajic, Lee, Sokolsky and Mangharam (2013), *UPP2SF: Translating UPPAAL Models To Simulink*, University of Pennsylvania.
http://www.seas.upenn.edu/~pajic/TEMP/UPP2SF_report.pdf

Popvici, Rousseau, Jerraya and Wolf (2004), *Embedded Software Design and Programming of Multiprocessor System-on-Chip - Simulink and SystemC Case Studies*, Springer.
http://www.springer.com/us/book/9781441955661

Prosman (2001), *Defining User Requirements During the Analysis Phase: A Look at Use Cases*, University of Missouri - St. Louis.
http://www.umsl.edu/~sauterv/analysis/488_f01_papers/Prossman/

Ran (2012), *Design Of A Transformation From UML Statemachines To TERRA CSP models*, University of Twente.
https://www.ce.utwente.nl/aigaion/attachments/single/939

Ran (2015), *Design Of Animation Facilities For Analysing Cyber-Physical System Software Architectures*, University of Twente.
https://www.ram.ewi.utwente.nl/aigaion/attachments/single/1302

Rutgers (2014), *Programming Models for Many-Core Architectures - A Co-design Approach*, University of Twente.
http://doc.utwente.nl/90661/1/thesis_J_Rutgers.pdf

Sassen (2009), *Floating-point Based Control Of The Production Cell Using An FPGA With Handel-C*, University of Twente.
https://www.ce.utwente.nl/aigaion/attachments/single/1028

Scicos (2016), *Scicos - a graphical dynamical system modeler and simulator*, Metalau team, INRIA.
http://www.scicos.org/

Takada (2001), *Real-time operating system for embedded systems*, ASP-DAC.
http://www.aspdac.com/2001/eng/tutorial/07.html

TERRA/LUNA (2015), *TERRA/LUNA - model-driven design (MDD) tool suite for designing (control) software for cyber-physical systems*, University of Twente / RaM group.
https://www.ram.ewi.utwente.nl/ECSSoftware/

UBM Tech (2014), *2014 Embedded Market Study - Then, Now: What's Next?*, UBM Tech.
https://bd.eduweb.hhs.nl/es/2014-embedded-market-study-then-now-whats-next.pdf

UPPAAL (2015), *UPPAAL - an integrated tool environment for modeling, validation and verification of real-time systems*, Uppsala University, Aalborg University.
http://www.uppaal.org/

van den Berg (2006), *Design Of A Production Cell Setup*, University of Twente.
https://www.ce.utwente.nl/aigaion/attachments/single/1074

van Zuijlen (2008), *FPGA-based control of the production cell using Handel-C*, University of Twente.
https://www.ce.utwente.nl/aigaion/attachments/single/1032

Wilterdink (2011), *Design Of A Hard Real-Time, Multi-Threaded And CSP-Capable Execution Framework*, University of Twente.
https://www.ce.utwente.nl/aigaion/attachments/single/883

Zwikker and Gunsing (2015), *Merging Process Structure And Design Freedom*, Mikroniek.
http://www.dspe.nl/mikroniek/