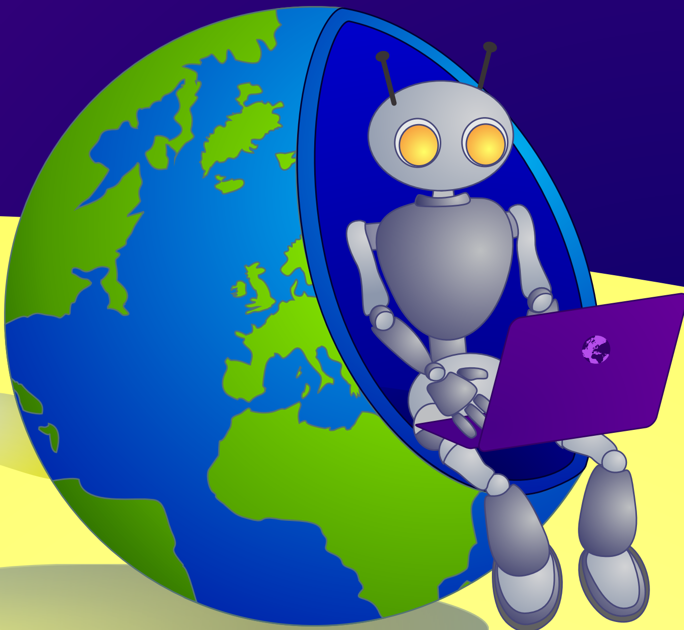
Cyber-Physical Systems Software Development

way of working and tool suite



Maarten M. Bezemer

Cyber-Physical Systems Software Development

way of working and tool suite

M.M. Bezemer

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering Mathematics and Computer Science, University of Twente

Robotics and Mechatronics group

CTIT Ph.D. Thesis Series No. 13-276
Centre for Telematics and Information Technology
P.O. Box 217, 7500 AE
Enschede, The Netherlands.

The research has been conducted as part of the TeleFLEX project, funded by the Dutch Ministry of Economic Affairs and the Province of Overijssel, within the Pieken in de Delta (PIDON) initiative.

Title: Cyber-Physical Systems Software Development way of working and tool suite
Author: M.M. Bezemer
ISBN: 978-90-365-1879-6
ISSN: 1381-3617 (CTIT Ph.D. Thesis Series No. 13-276)
DOI: 10.3990/1.9789036518796

Cover design by Marieke Bezemer-Krijnen.

Copyright © 2013 by M.M. Bezemer, Enschede, The Netherlands.

All rights reserved. No part of this publication may be reproduced by print, photocopy or any other means without the prior written permission from the copyright owner.

Printed by Wöhrmann Print Service, Zutphen, The Netherlands

CYBER-PHYSICAL SYSTEMS SOFTWARE DEVELOPMENT

WAY OF WORKING AND TOOL SUITE

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 14 november 2013 om 14.45 uur

door

Maarten Matthijs Bezemer
geboren op 17 augustus 1982
te Eindhoven

Dit proefschrift is goedgekeurd door:

prof.dr.ir. S. Stramigioli, promotor
dr.ir. J.F. Broenink, assistent promotor

Graduation committee

Chairman and Secretary	
prof.dr.ir. A.J. Mouthaan	University of Twente
Supervisor	
prof.dr.ir. S. Stramigioli	University of Twente
Assistant Supervisor	
dr.ir. J.F. Broenink	University of Twente
Members	
prof.dr.ir. H. Bruyninckx	KU Leuven
prof.dr.ir. J.J.M. Hooman	Radbout University Nijmegen
prof.dr.ir. M. Aksit	University of Twente
prof.dr.ir. G.J.M. Smit	University of Twente

Summary

Designing embedded software for modern cyber-physical systems becomes more and more difficult, because of the increasing amount of requirements, which are also increasing in their complexity. An example of increasing complexity of the requirements is the demand to get a general-purpose cyber-physical system capable of fulfilling a variety of different, ad-hoc tasks or to be suitable in environments with lots of human interactions. A typical example of modern cyber-physical systems, which have these modern requirements, are medical robotic systems used in surgeries.

The essential goal of this research is to provide a *way of working* for the design of the control software for cyber-physical systems, and thereby providing a solution of the problem of the increasing complexity of the control software due to the modern requirements described above. The way of working makes use of model-driven design (MDD) techniques to reduce the complexity of the control software design.

First the models are designed, both the software architecture and the control algorithms. These are combined into the actual control software implementation. Next, the model needs to be verified and simulated to test whether it is behaving as intended.

Ultimately, these steps result in a first-time-right implementation. A more realistic result of these steps is to provide means to get as close as possible to such a first-time-right implementation. Careful design and tests, that are stimulated by these steps, help in preventing damaging the system or its environment due to software problems and result in a higher software quality and reusability.

The way of working is supported by additional elements, which have been developed as part this research. The software architecture model basically is a network of components, which all have the same basic functionality. This basic functionality is provided by a blue-print, called *Generic Architecture Component* (GAC).

The basic implementation of the GAC consists of 4 blocks: Coordination, Computation, Configuration and Safety. These blocks provide the basic behaviour of the GAC, like connectivity to other GACs, a simple component life-cycle and checking signal values for errors. Each block has one or more placeholders, called hooks, to implement the behaviour of the actual component that makes use of the GAC.

LUNA Universal Network Architecture (LUNA) is an execution framework to execute the produced models that result from the way of working. A hardware abstraction layer is available to provide a basis for the execution engine which is platform independent, separating the platform-support code and the execution engine.

The framework consists of multiple components. Each of these components can be enabled or disabled, depending on the requirements of the application that is using the framework. This makes the framework suitable for low resource, embedded systems, as all unused components can be disabled to keep the resource usage as low as possible. On the other hand complex applications, for example to steer medical

robotic systems, are also supported by the framework, as it also has components to perform complex tasks, like model execution.

The *Twente Embedded Real-time Robotic Application* (TERRA) is an MDD tool suite, which supports the way of working. Its tools range from editors to graphically design the models to code generation tools to convert the models into source code.

Models are central in the TERRA tool suite. All structures of the TERRA models are defined by meta-models. These meta-models defined all model elements and their usage. The TERRA tools make use of these defined structures to properly make use of the models and to interpret them correctly.

Like the way of working, the GAC and LUNA, TERRA is also modular. It is a collection of separate components providing meta-models, editors, code generation tools, and so on. This results in means to easily add a new components to include new functionality. The editors for example provide so-called extension points to add new model elements to support for example C++ code blocks in a model.

In the end, it is concluded that the proposed way of working provides design steps for the complete design trajectory, starting at the initial designs up-to and including the deployment of the control software on the target system. It is also concluded that the GAC tightly matches the way of working and increases its value and usability. The execution framework and the tool-suite further increase the usability of the way of working by adding graphical model-driven design support to the way of working, with e.g. model validation and code generation. This all increases the understanding of the complex models and thus decreasing the complexity of the control software design.

It is recommended to further evaluate the way of working, by using it to implement different control applications to steer all kinds of different cyber-physical systems. This likely results in adaptations of the way of working, making it more generic and suitable for a broader range of applications. Including model management is recommended to increase the reusability of the modelled components and to provide sets of components for a specific goal. Simulator should be added to TERRA to let it further support the way of working and to increase the probability of first-time-right implementations.

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Cyber-Physical System Overview . . . 2
1.3 Objectives . . . 4
1.3.1 Design Space Exploration . . . 5
1.3.2 Scalability . . . 5
1.3.3 First Time Right . . . 5
1.4 Approach . . . 6
1.5 Outline . . . 7

2 Terminology and Technologies **9**
2.1 Real-Time Guarantees . . . 9
2.2 Safety . . . 11
2.3 Model-Driven Development . . . 12
2.4 Meta-Models . . . 14
2.5 Component Port Connection . . . 15
2.6 Communicating Sequential Processes . . . 16
2.7 Model Transformations . . . 18
2.8 Co-Simulation . . . 18
2.9 Deployment . . . 20
2.10 Software Frameworks . . . 20

3 Design Approach for Embedded Control Software **23**
3.1 Way of Working . . . 23
3.1.1 Software Architecture Modelling . . . 26
3.1.2 Software Testing . . . 27
3.1.3 Software Deployment . . . 29
3.2 Tool Coverage . . . 30
3.2.1 Graphical Modelling . . . 31
3.2.2 Code Generation . . . 32
3.2.3 Model-to-Model Transformations . . . 33

3.2.4 Co-Simulation 34
3.3 Generic Architecture Components 34
3.4 Execution Framework 35
3.5 Conclusions 36

4 Generic Architecture Component 37

4.1 Requirements 38
4.2 Existing Component Models 40
4.2.1 BRICS Component Model 40
4.2.2 Orocos 40
4.2.3 ROS 41
4.2.4 Conclusion 42
4.3 Design 42
4.3.1 Separation of Concerns 43
4.3.2 Computation 44
4.3.3 Coordination 44
4.3.4 Configuration 46
4.3.5 Communication 46
4.3.6 Composition 46
4.3.7 Safety 48
4.3.8 Discussion 48
4.4 Implementation 49
4.5 Usage of the Generic Architecture Component 51
4.5.1 PCU GAC Design Considerations 53
4.5.2 Production Cell Architecture Implementation 56
4.6 Discussion and Conclusions 58

5 LUNA Universal Network Architecture 61

5.1 Requirements 62
5.2 Existing Solutions 63
5.3 LUNA Architecture 64
5.3.1 Threading Implementation 66
5.3.2 LUNA CSP 68
5.3.3 Channels 71
5.3.4 Alternative 73

5.4 Results 77
5.4.1 Context-Switch Speed 77
5.4.2 Commstime Benchmark 79
5.4.3 Cyber-Physical System Use Case 81
5.5 Conclusions 83

6 Twente Embedded Real-time Robotic Application 85

6.1 Related Work 85
6.1.1 Meta-Models 86
6.1.2 Tooling 87
6.2 Meta-Model Usage 87
6.3 Meta-Model Implementation 89
6.3.1 CPC Meta-Model 89
6.3.2 CSP Meta-Model 92
6.3.3 Architecture Meta-Model 94
6.3.4 Other Meta-Models 95
6.4 Graphical Model Editor 95
6.4.1 20-sim Editor Integration 96
6.5 Model Validation 97
6.6 Model Transformations 98
6.6.1 Model-to-Text Transformation 98
6.6.2 Model-to-Model Transformation 99
6.7 (Co-)Simulation 100
6.8 Evaluation 101
6.8.1 Usage of TERRA 101
6.8.2 Discussion 102
6.9 Conclusions 103

7 Conclusions and Recommendations 105

7.1 Conclusions and Evaluation 105
7.1.1 Way of Working 105
7.1.2 Generic Architecture Component 106
7.1.3 Framework and Tooling 107
7.1.4 Relevance 108
7.2 Recommendations 108

7.2.1 More Evaluation 108
7.2.2 Model Management 109
7.2.3 Model Optimisation 109
7.2.4 Simulation Support 109

Bibliography **113**

Index **119**

1

Introduction

1.1 Context

Designing control software for modern cyber-physical systems becomes more and more difficult because of the increasing amount and complexity of their requirements, as stated in the RoboNED roadmap on robotics (Kranenburg-de Lange, 2012). The mechanical designs of the physical parts of the systems are becoming more and more complex. As a result the systems have a growing amount of sensors and actuators, which are needed by the control software to provide the desired behaviour of the system. For general purpose systems, the flexibility and complexity of their behavioural tasks also increase to make the system suitable for as many situations as possible. Even though the complexity keeps increasing, the systems need to become more and more safe. The increasing need of safety arises due to the increasing interaction with humans and other cyber-physical systems. On top of this, the cyber-physical systems also tend to become more mobile, and therefore need to be as energy efficient as possible.

In contrast with these quality-related requirements, like safety, robustness and energy efficiency, the companies that develop and produce the cyber-physical systems have a time-to-market requirement. They rather do not spend too much time on the quality requirements, as this will slow down the design process and thus the postpone the availability of their product on the market. This is disadvantageous for a company when there are competitors developing a similar product or just because the product revenue is delayed as well.

A typical example of modern cyber-physical systems, that have the requirements mentioned above, are medical robotic systems. Traditional medical systems are well established in hospitals already, for example in the form of imaging systems like MRI or ultra-sonic scanners. Besides these traditional systems, a new type of robotic systems is being introduced into hospitals. These new systems have more and intense interaction with the people in hospitals, both staff and patients. Examples are robotic systems that help raising patients out of or into their beds, automated carts transporting food or laundry around the hospital, or even robotic systems that are directly involved in performing surgery on the patient.

Examples of systems that are directly involved at robotically-assisted surgery are the da Vinci and the Zeus systems (Sung and Gill, 2001). This thesis work is sponsored by the TeleFLEX project. This project aims to research, design and construct a robotized endoscope that can be used to perform biopsies and to perform simple surgeries in the human intestine. These robotic systems aim to assist at general surgeries using a minimal invasive approach, resulting in less trauma for the patient. Modern cyber-physical systems optionally support the possibility of the surgeon being at an other location, so the system is controlled over a distance using the Internet or any other network. Besides improving the advantages for the patients, some of these new systems also improve advantages for the surgeons personally, for example the TeleFLEX project aims to improve the working posture of the surgeons.

The described medical cyber-physical systems are complex, have a lot of interaction with the operator and their environment and therefore require lots of sensors and actuators. The control software needs to process lots of data and communication streams, making sure that all software parts are able to calculate their control algorithms in a timely matter. Robustness and safety aspects are required to be absolutely sure that the system behaves as intended in all situations, otherwise the system might inflict harm to the patient with potentially disastrous results.

It is shown above that modern cyber-physical systems become more and more complex due to the various reasons, as a result the control software also becomes more complex. Additionally, the hardware that executes the software also becomes more capable and feature rich. For example, modern hardware often has a CPU with multiple cores. Software that needs to make efficient use of them is more complex than a single-core implementation. This increasing software complexity requires new design techniques to efficiently manage this increasing complexity.

1.2 Cyber-Physical System Overview

The terms *cyber-physical system* and *robotic system* are both used in the previous section. Additionally, the term *mechatronic system* is a well-known term as well. These three terms are used to describe the same type of system, i.e. a system that consists of a mechanical part, an electronic part and a software (control) part.

- The term mechatronic system does focus more on the mechatronics.
- The term robotic system does focus on the robotic application.
- The term cyber-physical system does focus on the presence of a cyber and a physical part.

The author has chosen to mainly use the term *cyber-physical system*, because it has a focus on the presence of the cyber part of the system, which is the main subject of this thesis. However, in certain situations the term robotic system is more applicable, so it is used in those situations. The actual components of these systems are explained in the remainder of this section.

A cyber-physical system consists of a *cyber domain* and a *physical domain* as shown in Figure 1.1. The physical domain usually consists of the mechanical and electrical components of the system. Actuators and sensors convert signals between the *electrical domain* and the *mechanical domain*. The cyber domain consists of software that controls the system, it resides in a (embedded) *computing platform*. The electrical signals are converted by digital-analog converters (DACs) and analog-digital converters (ADCs) to communicate from and to the cyber domain and its software components, respectively.

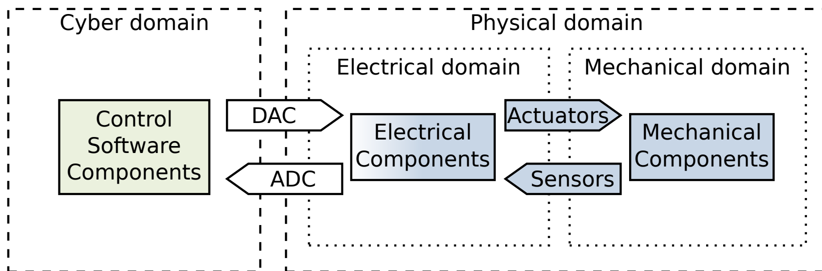


Figure 1.1: System overview of a cyber-physical system as used in this thesis.

There are multiple software solutions to steer a cyber-physical system. Agent-based controllers use intelligent, autonomous entities (agents) that interact with the environment to work towards a specific task. Multi-agent systems (MAS) use multiple agents to perform more complex tasks (van Breemen, 2001; Dao, 2011).

Another way to structure control software is to implement a subsumption architecture (Brooks, 1986). Such an architecture consists of multiple modules that are organised in layers on top of each other, each implementing its particular goal. The modules and layers are able to suppress or inhibit the outputs of their higher layers.

The layered approach that is used in this thesis is depicted in Figure 1.2. Each layer has its own set of properties which influences the behaviour of the control components that are placed on them. Components either directly steer the cyber-physical system or provide tasks for other components to influence the overall behaviour of the cyber-physical system. As this thesis focuses on the cyber domain of cyber-physical systems, the physical domain part is simplified by the *Plant* and part of the *I/O Hardware* blocks. The cyber domain, or *Embedded Control Software*, is shown in more detail at the left of the figure. Further details of the embedded control software component of Figure 1.2 are discussed in Section 2.1.

A typical usage of the architecture is a combination of an embedded computing platform combined with an FPGA, of which an example is shown in Figure 5.10. The FPGA is used to get access to a lot of easily usable I/O pins, that are used for the connection with the actuators and sensors. A software driver handles the communication with the FPGA, so the control software has direct access to the FPGA pins and thus to the actuator and sensor signals.

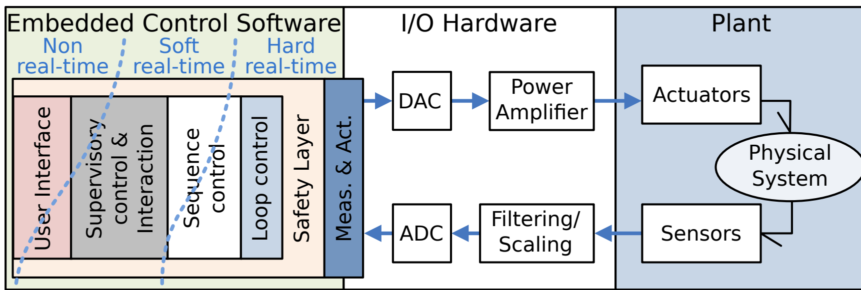


Figure 1.2: Detailed system architecture of a cyber-physical system, Broenink et al. (2010b)

As mentioned earlier, this thesis focuses on the embedded control software design and implementation. However, the other domains need to be kept in mind as well during the software design and implementation, as knowledge of the these domains is required during the design of the software to get the optimal result.

1.3 Objectives

This thesis has the focus on three main objectives. These objectives result in better control software quality for cyber-physical systems and in a faster, more structured design trajectory.

- The essential goal of this thesis is to provide a *way of working* for the design of the control software for cyber-physical systems, and thereby providing a solution of the problem of the increasing complexity of the control software, described in Section 1.1. This way of working needs to be generically usable for all kinds of cyber-physical systems, like the medical systems that are involved with surgeries, industrial systems that are used for assembly lines, service robots having interaction with humans, or small research-orientated robotic systems. The way of working must cover the whole design trajectory, from the initial system architecture design to the deployment of the control software.

- The second objective of this thesis is to provide a blue-print for a more suitable generic software component, as required by the proposed way of working. Earlier work attempted to design such a component (Groothuis et al., 2008), but that was not flexible and reusable enough to make it suitable for the different types of cyber-physical systems.

- The third objective is to describe and provide an execution framework and tooling support that is compliant with the way of working. The way of working strongly suggests the usage of models to design the control software, as this helps managing the complexity of modern cyber-physical systems. These models require framework and tooling support to become fully usable, hence increasing the efficiency and usability of the way of working.

Together these objectives should result in a decrease of development time for (complex) cyber-physical control software. Some additional requirements for these objectives are discussed in the following sections.

1.3.1 Design Space Exploration

During the development trajectory of a cyber-physical system, multiple designs are often tried out to find out the optimal solution. Other components could temporarily get replaced with prototypes, e.g. for testing reasons, or due to the lack of the availability of the final component. Partially using prototypes for testing is called hardware-in-the-loop testing.

The way of working must support such an iterative design trajectory. For example, it should be possible to try multiple software controller implementations, so they keep matching the different mechanical, prototyped components, or to try out different control algorithms to see which one is suitable. Therefore, the modelled components need to be interchangeable with similar component models. Splitting the components into an interface and the actual implementation helps with this requirement, as the designer is able to select a different implementation for each (component) interface used in the system architecture.

1.3.2 Scalability

The way of working needs to be scalable for the design trajectories of all kinds of cyber-physical systems: It needs to support both small embedded cyber-physical system designs and large cyber-physical system designs. Support for small embedded systems consists of keeping the resource usage in mind in order to keep it as low as possible. Software design for larger systems does not require strict resource housekeeping, so additional features and flexibility can be provided.

Therefore, the way of working must be flexible to support the complete range of design trajectories from the embedded systems design to the large systems design. This is incorporated in the way of working by providing additional steps or by making steps optional. The execution framework must be suitable for embedded systems with a low amount of resources, but also must provide the additional features and flexibility when required. This requirement is fulfilled by making the framework modular, so features can be optionally enabled or disabled, depending on the requirements of the system that is being developed.

1.3.3 First Time Right

The proposed way of working must be dependable and result in a *first-time-right* design. A first-time-right solution would be ideal, but nearly impossible to attain. However, Verhoef et al. (2012) show that a 30% productivity improvement is within the possibilities. Aiming at a first-time-right solution, is important as it is easier to incorporate design changes in an initial stage of the development than in a later stage, when a product is sold already. From a company's point of view, the time to market must be as low as possible, so a solid way of working helps decreasing this time.

Increasing the probability of a first-time-right design is obtained by using model-driven design (MDD) techniques. The models that result from these techniques are used for all kinds of tasks. For example, they can be formally checked or used for simulations to minimize the design flaws and detect problems at an earlier stage. Generating code from these models, instead of manually providing the code, further decreases unforeseen problems. This all increases the probability of a first-time-right design, reduces required debugging efforts and results in a software design that is working as intended for the cyber-physical system it was designed for.

1.4 Approach

The approach to fulfill the objectives, described in Section 1.3, is provided in this section. It is based on structural design of the control software for cyber-physical systems. This helps managing the increasing complexity of this control software and solves the problem described in Section 1.1.

Using a structural approach helps preventing to make unnecessary mistakes or to find them in an early development phase, so the resulting cyber-physical systems are as safe as possible. The Borderc project (Heemels and Muller, 2006) poses the *Formalisms, Techniques, Methods and Tools* approach. It provides a separation of concerns during design as each part is used separated from the other three. This separation is used throughout this thesis, to provide generic solutions that are not limited by a specific language (formalism), design technique/method or tool.

Model-driven design techniques are used to model the system overview at early stages and to provide the resulting implementation at a later stage. Two types of meta-models are used to provide the semantics of the models: an architectural meta-model to model the software architecture and a Communicating Sequential Processes (CSP) meta-model to model implementation details. The designed models can be formally verified, simulated and used for model transformations. The continuous reuse of the models is part of the structural design and prevents problems due to reimplementation of the system for a different formalism.

The described model-driven design techniques require support in the form of an execution framework and tooling. Models themselves cannot be executed directly and thus need to be converted to another formalism, i.e. a computing language. The execution framework provides, for convenience reasons, the static part of the converted formalism, so it does not require to get included over and over.

Tooling support is required to automate the model-driven design techniques, reducing the complexity of the design process. Model editors provide support in constructing the models, which can be done graphically or textually. Transformation tools transform the models from one formalism into another, resulting in for example models that are optimised for a specific goal or in the executable computer language. Other tooling support provides means of verifying, simulating or managing the models. In general tools supporting a specific type of models, are working together forming a tool suite capable of providing support for the complete design trajectory.

This thesis focuses on all of these aspects, combined in a proposed way of working

describing the required trajectory to design control software. The requirements mentioned in the previous sections are incorporated into the way of working, making it suitable to design control software for all types of cyber-physical systems.

1.5 Outline

Detailed information on model-driven design techniques and the other terminology and techniques used in this thesis is provided in Chapter 2.

The proposed way of working is described in Chapter 3. Model-driven design techniques are used to systematically design the models which are, in the end, used to obtained the control software. Generic components are recommended to provide basic functionality for the control software, for example to provide communication and safety support. It is concluded that the way of working requires an execution framework and dedicated tool support to become fully functional. The chapter is based on the following publications:

- Broenink, J. F., M. A. Groothuis, P. M. Visser and M. M. Bezemer (2010a), Model-Driven Robot-Software Design Using Template-Based Target Descriptions, in *ICRA 2010 workshop on Innovative Robot Control Architectures for Demanding (Research) Applications*, Eds. D. Kubus, K. Nilsson and R. S. Johansson, IEEE, IEEE, pp. 73 – 77

- Bezemer, M. M., M. A. Groothuis and J. F. Broenink (2011a), Way of Working for Embedded Control Software using Model-Driven Development Techniques, in *IEEE ICRA Workshop on Software Development and Integration in Robotics (SDIR VI)*, Eds. D. Brugali, C. Schlegel and J. F. Broenink, IEEE, IEEE, pp. 6 – 11

Next, the Generic Architecture Component (GAC) is discussed in Chapter 4. Separation of concerns is applied to the design of these components, so their structure is kept clear and usable in all kinds of situations. A template GAC, a sort of blue-print, is designed containing the basic functionalities of the component, which then can be used to design specialised GACs for the specific tasks of the control software. An initial implementation is modelled using CSP.

The LUNA execution framework is described in Chapter 5. It is an example of an execution framework that provides support to execute the models that are in the design steps of the way of working. Additionally, the GAC models are designed using CSP, so LUNA needs to provide a CSP execution engine. LUNA builds its provided execution engines on top of a platform abstraction layer, making the engines suitable for all kinds of platforms. The chapter is based on the following publication:

- Bezemer, M. M., R. J. W. Wilterdink and J. F. Broenink (2011b), LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework, in *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, Eds. P. H. Welch, A. T. Sampson, J. B. Pedersen, J. M. Kerridge, J. F. Broenink and F. R. M. Barnes, IOS Press, Amsterdam, pp. 157 – 175, ISBN 978-1-60750-773-4, ISSN 1383-7575, doi: 10.3233/978-1-60750-774-1-157

The overall ideas of LUNA are still the same even though LUNA has been improved since then.

Details about the TERRA tool suite and the tools it consists of, are described in Chapter 6. The TERRA tools provides methods of editing the models, checking them for consistency and transforming them into C++ LUNA code. The chapter is based on the following publication:

- Bezemer, M. M., R. J. W. Wilterdink and J. F. Broenink (2012), Design and Use of CSP Meta-Model for Embedded Control Software Development, in *Communicating Process Architectures 2012, Dundee*, volume 69 of *Concurrent System Engineering Series*, Eds. P. H. Welch, F. R. M. Barnes, K. Chalmers, J. B. Pedersen and A. T. Sampson, Open Channel Publishing, pp. 185 – 199, ISBN 978-0-9565409-5-9

Note that TERRA has be developed much further since then, so large parts of the original paper contents are changed.

The results of this thesis are discussed in Chapter 7.

2

Terminology and Technologies

This thesis contains a specific terminology and throughout this thesis multiple technologies are used. They might be ambiguous, unknown or unclear in the specific context of this thesis. This chapter provides the explanation of the used terminology and technologies and discusses them.

2.1 Real-Time Guarantees

Each cyber-physical system needs its software components, for example containing control algorithms, to run at their specific frequency in order to have a stable behaviour. The (embedded) control software must comply to the timeliness requirement of each component and thus needs to have the results of the component ready at the required interval. Cooling (2003) splits this timeliness requirement into two sub-requirements:

- *Criticality requirement*
  This requirement defines the criticality of the problem when the deadline is missed. To fulfill this requirement the system needs to provide *real-time guarantees* for its components.
- *Time requirement*
  This requirement specifies the required time periods of the components, which defines the speed and responsiveness of the component.

The focus in this thesis is mainly on the criticality requirement and the resulting real-time guarantees, when looking at the timeliness of a component. The time requirement barely influences the design and execution of a software component, it matters more when designing the software computing platform that is used to execute the software of the cyber-physical system.

There are two different types of real-time guarantees: hard real-time and soft real-time. Kopetz (1997) defines their difference as follows:

> "If a result has utility even after the deadline has passed, the deadline is classified as *soft* (...) If a catastrophe could result if a deadline is missed, the deadline is called *hard*".

Components without one of these two real-time guarantees are non *mission-critical* components, i.e. they are not part of the control software but provide other, additional features.

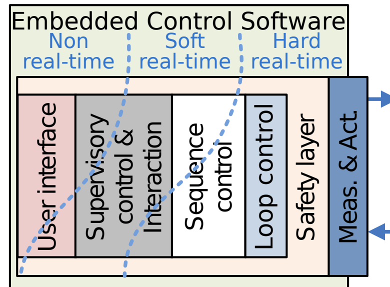


Figure 2.1: Embedded control software domain of a cyber-physical system.

Figure 2.1 shows the layered design for embedded control software part of Figure 1.2. Each layer, like User interface or Sequence control, is used for a certain goal and might contain one or more software components. Each software layer requires at least one type of real-time guarantees, varying from no real-time at all to hard real-time. However, most layers do not have a strict real-time guarantees classification, but provide a choice to their components, shown in the figure by the curved boundaries of the real-time guarantees. Even within a component, two different classifications might be required.

The *Loop control* layer is the software part that is directly responsible for steering the physical system and therefore it requires hard real-time guarantees. Hard real-time guarantees provide strict timing properties, guaranteeing that the given deadlines are always met, assuming that the system has sufficient resources to begin with. These hard real-time guarantees are required to make sure that the control algorithm calculations are finished in time, which in general is required to keep the periodic control loops stable. Therefore, if this for whatever reason fails, the system is considered to be unsafe and catastrophic accidents might happen with the physical system or its surroundings due to wrongly moving parts.

Sequence control components provide support for longer running tasks, like path planning or environment mapping. Depending on the software requirements, these tasks require hard real-time or soft real-time guarantees. The components with the soft real-time guarantees try to meet their deadlines, without giving any hard guarantees. Assuming that the design is correct, nothing bad should happen in case such a deadline is occasionally not met.

The *Supervisory control & Interaction* layer contains components that require even lower real-time guarantees compared to the *Sequence control* layer components, like planning the overall activities or communicating with other (cyber-physical) systems. Their algorithms are typically running for a longer period of time due the nature to their activities and only once in a while they need to change or influence the sequence

controllers to update their tasks. These components are provided with either soft or non real-time guarantees. The left-over resources of the system are used for these tasks, without giving any guarantees of the availability of these resources.

The *User interface* layer and to some extend the *Supervisory control & interaction* layer are (partially) non real-time as they used to provide results that are directly influencing the safety critical system parts. As mentioned earlier these real-time guarantee boundaries are not strict, for example there are situations where a component requires other real-time guarantees than regularly are used.

The hard real-time guarantees provide the best guarantees to meet the given deadlines, so at a first glance it would the easiest solution from a design perspective to let the complete software have hard real-time guarantees. Unfortunately, this is not possible for most systems, as this strains resource usage too much and requires a lot of development effort.

Depending on the class of the cyber-physical system, the choices for real-time type are different. For example, a user interface which displays logging information might not be important for a humanoid robot and can be placed on a non real-time type. The user interface of an industrial robot is probably more important and therefore needs to have the soft real-time guarantees, especially if an operator needs it to respond on particular situations. On the other hand, giving the user interface of the industrial robot hard real-time guarantees is basically wasting resources, as the human operator is not responding with hard real-time guarantees as well.

Due to differences of the deadline guarantees of the real-time type, communication between components on different real-time type is not straightforward. The properties of both layers need to be maintained while communicating data through the layer boundaries. For example, a communication channel between a soft real-time component and hard real-time component, might mess up the guarantees of the hard real-time component. This would happen when the hard real-time component needs to wait on the availability of the data, as the soft real-time component might not meet its guarantees, which is not a problem for *that* component, but making the hard real-time component miss its deadline, which *is* a problem.

A simple solution is to make use of a buffer which is located on the boundary between the two real-time types. The soft real-time component updates the buffer when its data is available or calculated. The hard real-time component reads the data from the buffer, which is always available and can be read in a deterministic manner. The same goes when the hard real-time component needs to send data to the soft real-time component. When using such a solution the (control) algorithms need to accept values that might not yet be updated, otherwise this scheme still results in system failures.

2.2 Safety

Even though control software is designed as careful as possible and has been tested thoroughly, *safety* measures are still required to handle unexpected and undesired situations, as described by Sözer (2009) in more detail. These situations could be

caused the by environment or by last-minute (hasted) fixes that were required. In fact safety aspects of a cyber-physical system are very important, so they need to be redundantly implemented in different domains, so when one domain is failing the other is still able to provide the safety measures.

It is impossible to cover all safety problems by thorough testing only, for example environmental-related causes are hard to predict and to cover completely. The system must react on such situations to prevent accidents with the cyber-physical system and/or its environment. The robot-control software architecture has a large role in properly handling such situations.

Assuming that the control software consists of a network of components, a problem can be handled on a local level, a global level or both. This depends the origin, severity and other properties of each problem. Handling the problem *locally*, in the component that detected the problem, could prevent a complete system shut-down. Examples that could possibly be handled locally are:

- a broken physical component that is not required for the basic functioning of the system.
- a sensor is not returning correct measurement values anymore, but these measurements can be compensated by other sensors
- an undesired region about to be entered with a robotic arm

Global error handling is the other possibility and is required in more severe situations that cannot be solved locally anymore. For example if multiple or important physical components are broken, the system needs to shutdown in a safe way without increasing the damage.

Local error handling is preferred, as local safety handling influences the rest of the system as little as possible, allowing the rest of the system continue working. Nonetheless, most cyber-physical systems are likely to have a *hybrid* solution of local and global error handling.

2.3 Model-Driven Development

This thesis mainly revolves around the *Model-Driven Development* (MDD) software development methodology, Model-Driven Development is also known as Model-Driven Design or Model-Driven Engineering (MDE). The difference between these three terms is neglectable for the use in this thesis, if there is any difference at all.

France and Rumpe (2007) describe MDE as a method to

> “(reduce) the gap between problem and software implementation domains, using systematic transformations from the problem-level abstractions to software implementations”

They furthermore state that the complexity of designing software for systems is reduced

> "through the use of models that describe complex systems at multiple levels of abstraction and from a variety of perspectives, and through automated support for transforming and analyzing models"

Basically, MDE (and thus MDD) is an engineering technique that uses model construction and model-based transformations to reach a desired end-result, in this thesis this end-result is the (embedded) control software for cyber-physical systems.

Models are the basis for the MDD methodology, they are used to perform all kinds of complex or tedious tasks, that otherwise need to be manually performed by the developer. Typical examples of such tasks are model transformations (Section 2.7) or (co-)simulations (Section 2.8).

The need for model transformations rises, for example, when a model can be formulated in a formal language. When transformed into such a formal language, the model quality and consistency issues can be rigorously checked using a formal verification tool. The *Failure Divergence Refinement* (FDR2) tool (Formal Systems (Europe) Limited, 2012) is such a tool that is able to formally check the CSP process algebra (Hoare, 1985). So the original model needs to be transformed into CSP algebra to make use of the FDR2 capabilities. Of course the principles behind the original model needs to be compatible with the CSP algebra in order to be able to transform the model. More information on CSP and its derived models is provided in Section 2.6.

Different types of models are used when a cyber-physical system is modelled. Each model type has its own advantages and disadvantages and is usable for a specific task or domain. Model transformations are used to transform these different types of models into a form that is compatible with all model types in order to combine them. For example, model-to-code transformation is used to obtain the actual control software of the cyber-physical system.

Model-driven design techniques are typically combined with tool support, providing means to construct the required models or to perform the (transformation) tasks using these models. Integrating these tools into a tool suite (also called toolchain) and automating these tasks further, streamlines the development process even further, preventing unnecessary human-based errors and reducing the development time of the control software. This makes small and quick iterative cycles more accessible, as the introduced manual labor of performing a cycle, like testing, validating, simulating and so on, are taken care of by the MDD supporting tool suite.

TERRA is an example of such a MDD tool suite and is part of this research. Details about TERRA are provided in Chapter 6, especially Figure 6.1 is useful in this context, as it shows how the (MDD) tool suite revolves around the models. *20-sim* (Controllab Products, 2012) is another example of a MDD tool, it provides means to design, analyse and simulate models of mechatronic systems. These models range from the plant to control laws or a combination of these. 20-sim provides means to generate C++ code of the models to embed them into control software applications.

The models that are constructed and used with the MDD techniques can be presented in a visual/graphical or in a textual way. Both ways have their own advantages and disadvantages. The main disadvantage of visual modelling is that a (complex) diagram can be interpreted differently by each person, often depending on the skills of this person, as described by Petre (1995). Visual models also require additional information for the visual aspect, thus cluttering the information of the 'pure model'.

On the other hand, visual models have a limited set of symbols, in contrast to a textual model which can basically consist of any number of symbols (words made by individual letters), which is convenient to novice modellers. When using visual models, careful design of the visual representation and symbols improves the understanding of the models. This is debated by Moody and Hillegersberg (2009) for UML diagrams, as the authors pose that the visual representations of the UML diagrams contain flaws, but this discussion is applicable to visual model representations in general. Either way of modelling and presenting the resulting models is applicable when using MDD techniques and does not matter (much) within the scope of this thesis.

The MDD methodologies, as described above, are inevitable to comply with both the technical and business needs of modern designs. Quality control and automatic consistency checking are crucial here, to support an effective design process. The use of MDD tools makes developing for complex cyber-physical systems a less complex and more maintainable task (Groothuis et al., 2009).

The ultimate goal for MDD is to create designs that are first-time right, i.e. have the model(s) verified at multiple stages in the design process, implementation (code) correct by (model) specification and satisfying all requirements targeted by the design.

2.4 Meta-Models

The MDD software development methodology extensively makes use of models: All of the accompanying tools require to be able to understand these models. Therefore, the models need to comply to a definition, so their content is understood correctly and as intended. Such a definition could solely exist in the mind of the tool developer and gets directly implemented in the tools. A big disadvantage is that this definition is loosely constructed, i.e. when additional model data is required, the developer just adds it to the definition and tools, that require this additional model data, get updated. After a couple of these iterations, the model definition in the mind of the developer becomes more hazy and deviates more and more from the implementation in various tools. Of course when multiple developers are present, the model definition becomes even more blurred.

The same goes for reusing models and their definitions of third-party tools. Their model semantics are not publicly available or only live in the minds of the tool developers. This limits the extensibility and transparency of the models used by these tools. This research will therefore not make use of the implicit model information of these third-party tools for modelling purposes to prevent (future) problems.

Meta-models on the other hand strictly define the model structure. A meta-model is basically a model that describes the semantics of the *target model*, hence the name

meta-model. This can be continued to any desired level, for example a meta-meta-model describes the semantics of a meta-model.

As meta-models are basically models themselves, they can also be developed using the MDD methodology. For example, EMF (Steinberg et al., 2009) provides the Ecore meta-model which can be used to model custom meta-models. These meta-models can be used as a basis for graphical editors and other model based tools. The Ecore meta-model is an extreme example of meta-modelling: the Ecore meta-model is described using itself as a meta-model.

Using the MDD tools, a meta-model can be transformed into a piece of software that can be used by the MDD tools that needs to understand the models that are described by the meta-model. Regularly, but at least in the case of EMF, the generated software contains means to store and retrieve the model. Due to the strictness of the meta-model, and thus the software, only the meta-model defined data is handled, resulting in a clean model without undefined data.

Compared to the 'model definition in the mind of a developer' implementation, tools using the generated software meta-model, directly use a derived version of the meta-model itself. Any changes in the meta-model are reflected in the resulting software and thus in the tools. If such a definition changes, all tools automatically use the newest version. Even when a tool does not require the updated meta-model definitions, it is still able to handle the model data without changes. So, after a meta-model update all MDD tools are capable of using the new model definitions, without much effort.

2.5 Component Port Connection

Architectural models describing the control software for cyber-physical systems, see Section 3.1.1, provide information about the system on a high abstraction level. Typically, a cyber-physical system can be seen as a collection of units that have a specific functionality. The same goes for control software, which can also be seen as a collection of units taking care of their specific task. On an architectural model level only the interfaces of these units are designed, the actual implementations are left out at this stage of the design process.

Figure 2.2: Simple model based on the Component Port Connection elements

The Component Port Connection (CPC) paradigm, described by Klotzbücher et al. (2013), provides such a level of abstraction. Figure 2.2 shows a simple example model consisting of the elements provided by the CPC paradigm.

A *component* defines the unit that provides a designated, functional part of the system. The interfaces of these components are designed using ports. A *port* defines that a connection to another component is required for data communication, like sensor signals or calculated control values. Both components in the figure contain one port.

Connections are used to connect ports to each other, the example ports of the figure are connected to each other. A connection has two sides, one side of the connection provides the communication data, also called the producer side, and the other side requires the data, the consumer side. In the example figure, the open rectangle of *Component1* indicates a producer port and the closed rectangle of *Component2* is the consumer port. The arrow head of the connection also shows the data flow direction.

The components that are modelled using the CPC paradigm can be seen as placeholders for the actual component implementation. The collection of ports that is part of a component, provides a so called *component interface*. This interface must match to the *implementation interface*, i.e. the implementation must provide matching ports. These matching ports must be in revered directions, as an incoming port of the component interface is similar to an outgoing port of the implementation, due to the differences in their points of view.

2.6 Communicating Sequential Processes

Chapter 1 introduces *Communicating Sequential Processes* (CSP) (Hoare, 1985) and shows that it is suitable to describe communication flows within a model. Describing these communication flows is exactly what is needed when modelling the architecture of a cyber-physical system.

Additionally, CSP provides the possibility to synchronise processes based on their communication flow. For example, when a *process A* requires a certain value from a *process B*, a *rendezvous communication* channel can be used to make sure that process A is blocked until process B is able to provide the required value. Another solution would be to place both processes in a sequential order, which makes sure that process B is finished before process A and the value is available when process A is executed. *Buffered communication* channels are the opposite of rendezvous communication channels: they do not provide synchronisation as the data is buffered in the channel. Both the reading and writing process are able to interact with the channel without getting blocked. This channel type is typically used for communication between components that have different real-time guarantees.

Furthermore, CSP seamlessly adheres to the CPC paradigm. The CSP processes can be seen as components and the CSP channels as the connection between these processes. The CPC ports can be seen as part of the CSP process that specifies which channels are required to be connected to a process. For example a *reader* or a *writer* process reads or writes data from a channel respectively. Therefore both require a port to connect the channel to. The TERRA meta-models that are described in Chapter 6 contain port elements for exactly this reason.

Another advantage of using CSP as a modelling methodology, is that CSP is suitable to formally check on correctness of the model. *Formal verification* of a model provides insights on modelling problems, like deadlocks or livelocks. FDR2 is a tool that is able to formally check (machine-readable) CSP models for these kinds of problems.

Deadlocks are design flaws, where two or more components are indefinitely waiting on each other to provide information in order to be able to continue executing and that

part of the program comes to a halt. *Livelocks* are design flaws, where a group of processes never stop executing and thus preventing the execution of the other processes called starvation.

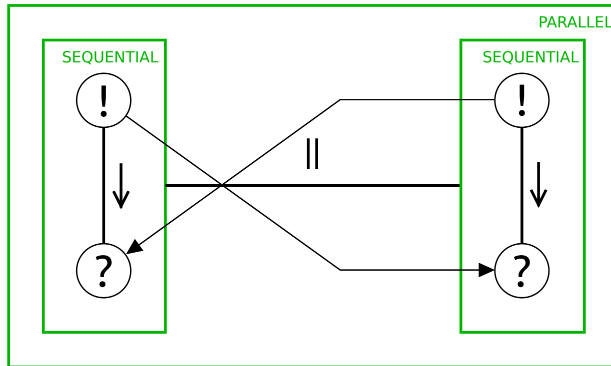


Figure 2.3: Example of an obvious deadlock situation

Figure 2.3 shows a simple example of a deadlock situation. It shows two sequential groups containing a writer process (circle with the exclamation mark) and a reader process (circle with question mark). A sequential group specifies the execution order of the processes it contains. The arrow in the figure shows this order, both sequential groups have their writer executed first and next their reader. The CSP channels between the writer/reader pairs are rendezvous channels. As mentioned before, the processes of both ends of such channels must be active to be able to communicate data. Due to the blocking writers in both sequential groups, the readers never become active resulting in a deadlock situation. The cause of the deadlock in this example is very obvious and can be found easily. A simple solution to prevent this deadlock is to swap the reader and writer of one of the sequential groups, or to change the sequential groups by parallel groups. For more complex models the use of formal checking tools provide means to find similar problematic situations that otherwise not would have been noticed.

The (mathematical) notation of CSP cannot be written using the ASCII encoding, so it not possible to use this notation as an input for CSP related tools. Therefore, machine-readable CSP is designed (Scattergood, 1998; Roscoe et al., 1997). It uses the ASCII character set to describe CSP models. For example the machine-readable CSP representation of the previous figure is as follows:

```
channel c1, c2
SEQ1 = c1!var_writer1 ; c2?var_reader1
SEQ2 = c2!var_writer2 ; c1?var_reader2
PAR = SEQ1 [| {| c1, c2 |} |] SEQ2
```

2.7 Model Transformations

Meta-models and their use-cases are discussed in the previous sections. Modern MDD tool suites provide graphical editors to manually construct models, based on these meta-models. Another model construction method is *model-to-model transformation* (M2M): A source model is used to construct a resulting model. The resulting model could conform to either the same meta-model as the source model or to another meta-model. An example of the first is *model optimisation*, the source model is optimised focusing on a certain aspect and an example of the latter is the transformation from one domain to another. An example implementation and additional details of model-to-model transformations is provided in Section 6.6.2.

Text-to-model transformation (T2M) is another way to construct models, it could be used to (re)create a model from plain text information, e.g. the source code, to make its structure more clear. For example, C++ code could be converted to a CSP model to get a (better) understanding of the concurrent architecture of the code. As code files do not contain all required information that (graphical) models require, like co-ordinates and object dimensions, this needs to be added by either the transformation algorithm or the user. This form of model construction is not seen often, probably due to complexity issues, missing model data and the lack of functional use.

Model transformations are also used to convert a model into the final, required object. The model is used as an easy means to construct or collect the required information for this final object. For example, a CSP model can be transformed into a machine-readable CSP file so it can be used by formal verification tools, like FDR2. This model transformation is called a *model-to-text transformation* (M2T), as the resulting file is a plain text file. If the transformation result is in the form of a computer language, or source code, the transformation is also called a *model-to-code transformation* (M2C) or *code generation*. The differences between M2T and M2C transformation techniques are minimal, if there are any at all.

2.8 Co-Simulation

Formal verification of a model only shows architectural design problems, as discussed in Section 2.6. Additional checks are required to see whether the model is behaving as expected. Executing the software directly on the actual cyber-physical system might be hazardous. Especially when dangerous environmental situations might occur or the system gets damaged due to improper and untested software. *Software simulations* provide means to safely check whether the architectural model and its implemented components contain errors.

The models resulting from the MDD techniques can also be used for the simulations. These models are based on meta-models, so the definitions of all model elements are known. The simulator uses these known definitions to determine how the software behaves when being executed on the real computing platform. For example the CSP meta-model implicitly contains definitions of the execution order of processes. These definitions can be used by the simulator to determine the execution flow of CSP a model. The actual accuracy of the simulation depends on the simulator and could

vary from just determining the process execution order to a complete simulation as if the software would be have executed on a specific hardware platform.

The described simulations are practically unusable for models that contain components which implementation is provided by external tools, since knowledge about the format and behaviour these implementation is required, i.e. the meta-model definitions of them are unknown to the simulator. This problem can be overcome when the external tool has simulation capabilities, which can be accessed from outside of the tool. The simulation capabilities of the tool need to be used in these cases, so explicit knowledge of the external meta-model is not required. This technique of combining the simulators of several tools is called *co-simulation*.

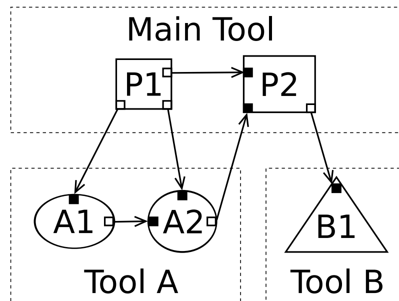


Figure 2.4: Co-simulation example

Figure 2.4 shows a co-simulation example. One tool needs to take the lead during the co-simulation, this tool is called the The Main Tool in the example, making sure that all simulating tools stay synchronised. Synchronisation between simulation tools consists for example of issuing commands to take a single simulation step to the other simulators making sure that the correct execution order is maintained. Another synchronisation task is providing the required data to a simulation tool that is going to perform its simulation. In the example processes A1 and A2 require data from P1, so the Main Tool needs to provide this data to Tool A, so Tool A is able to simulate the next step. The result of this simulation is then sent to P2, which sends its result to another external Tool B.

The leading tool typically is the tool that handles the simulation of the system architectural model. It has information about the external modelled components that is required for synchronisation. This information is required to perform co-simulations as it determines the execution order at the highest system level.

A variant of co-simulation is *hardware-in-the-loop simulation* (Isermann et al., 1999). As the term already indicates, this type of simulation (partially) uses actual hardware to perform the simulations, instead of an external simulation tool. This type of simulating is particularly useful in situations where parts of the hardware are available, and other parts are not. Advantage over regular co-simulations is, that the hardware that is available can be used to obtain more realistic results of a higher quality. Hardware-in-the-loop simulations save development time, as the designers do not need to wait

until the complete cyber-physical system is constructed and available, but still are able to increase the accuracy of the simulations and thus the software quality.

Another advantage of hardware-in-the-loop simulation is that the system can be further checked in a safe environment. For example, software timing constraints can be measured and checked using the actual target platform but still using a plant simulation. In situations where these constrains are not yet met, resulting in problematic situations, the actual plant does not get damaged.

2.9 Deployment

After formal checks and (co-)simulations showed that the model is designed and properly working, the model needs to be deployed to the target platform. It is obvious that the model cannot be executed on the target directly, as it needs to be converted to executable code that is understood by the target platform.

First the model is transformed into *source code*, using the model-to-text transformations, which then needs to be *compiled* into an executable application. Compiling source code for another target (compared to the development environment) is called *cross-compiling*. Cyber-physical systems mostly have embedded computers, which are low on resources and typically do not have (much) peripherals attached. Due to the lack of resources, these embedded computers generally do not have development tools installed. So a development computer is required to develop the embedded control software and cross-compilation is necessary to provide the actual executable application for the embedded computer.

Next, the compiled executable needs to be transferred from the development computer to the target. If this is not too often required, this could be done manually, but most of the time this deployment needs to be done more than once and deployment tools provide services to ease this repeating task.

Deployment tool suites handle cross-compilation, transferring the executable to the target and the actual on-target execution. Services like collecting log data and hardware-in-the-loop simulations could also be provided by the deployment tool, further easing the tasks of the developer.

Obviously, the deployment tool suite requires additional information for its tasks. For example, target information of hardware-specific parts is required, so this can be used when generating the executable for a specific target platform. Such hardware-specific details could provide the details on access the sensors and actuators of the cyber-physical system from within the software. The actually required details depend on the deployment tool and the services it provides.

2.10 Software Frameworks

The model-to-code transformation tools are able to convert the model into source code, as explained in Section 2.7. Actual execution of models requires a lot of additional code to support all used meta-model elements. For example, a CSP process requires an *execution engine* which schedules the processes in a correct execution

order. Used channel elements require actual implementations to actually send data from their producing to their consuming processes. These required features can be provided by the transformations as well, but as this is mostly static code it makes more sense to make use of a software *framework* that provides this required, static code. It makes the transformations less complicated and the static code easier to maintain.

Besides providing high level components, like an execution engine, a framework is also used to provide an operating system and/or target independent software platform by means of a *hardware abstraction layer* (HAL). It typically hides the platform dependent system calls with a public *application programming interface* (API). An API provides an interface that has one or more implementations, which are hidden from the application. Depending on certain properties, in this case the selected target platform, the correct implementation is used without active interaction of the application. This results in a platform independent application being unaware of the underlying OS or hardware. Ideally, such an application be case executed on different platforms without the need to apply platform dependent modifications.

3

Design Approach for Embedded Control Software

Design patterns help managing the complexity of designing (embedded) control software for the modern cyber-physical systems that are described in Chapter 1. The *way of working* that is proposed in this chapter, uses the model-driven development approach combined with separation of concerns to reduce the complexity. It covers the complete design trajectory, starting at initial system architecture designs to the deployment of the finished control software on the target.

The ultimate goal of the way of working is to provide a *first-time-right* approach for the software implementation of a cyber-physical system. Preventing excessive design time, while maintaining the designers point of view and preferred techniques and tools, is a more realistic goal of the way of working. This is accomplished by defining a structured way of designing the software, reusability of previously designed models and components reduce design time further.

Several design methods are being researched that seem promising to increase the quality and decrease the design time of control software development. For example, the BRICS project defined the *BRICS Component Model* (BCM) and mapped it on the 4 layers of model abstractions (meta-models) defined by OMG (Klotzbücher et al., 2013). The DESTECS project focuses on co-simulation (Pierce et al., 2012) and model management (Zhang and Broenink, 2013). The way of working must integrate, or at least tolerate, these researched design methods, in order to let the way of working become as universal applicable as possible.

3.1 Way of Working

Figure 3.1 shows the required software steps when developing embedded control software for a cyber-physical system. It starts with designing the *software architecture* (*I-a*) and the *control algorithms* (*I-b*). Both are typically designed using model-driven engineering tools and result in models of the architecture and algorithms. Using verification and simulation techniques (*II*) these models are continuously refined until their required functionality is met.

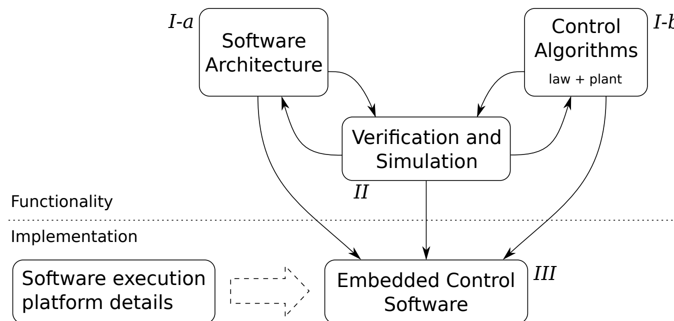


Figure 3.1: Overview of required steps for embedded control software development of cyber-physical systems

The embedded control software is created by combining the software architecture and control law algorithm models (*III*). Additional implementation details, like details of the computing platform, sensors and actuators, are added to make the software executable on the cyber-physical system.

More detailed steps to design control software are shown in Figure 3.2. The figure contains 5 starting points (of the 5 available design branches), labelled (a) to (e). Each starting point handles a particular domain of the design. Efficient design of control software is full exploited when certain software aspects are developed simultaneously. This simultaneous development makes sure that problems are implemented in the most suitable domain.

The *electrical domain* (a) and the *mechanical domain* (e) are out of scope, when looking at control software development, and are not further discussed. The software and controller design branches ((b), (c) and (d)) correspond with the two starting points (*I-a* and *I-b*) that are shown in Figure 3.1.

Depending on the focus and interest of the developer, branch (b) or (d) are the most likely starting points, as branch (c) requires information of branch (d) that is not yet available. Both branches (b) and (d) can be simultaneously used to start the (control) software development, if a large enough development team is available

At certain steps there is interaction between the different development branches, depicted in the figure by the horizontal, bidirectional arrows. At these points, the designs in the different domains are ‘synchronised’; the features of both models are compared. As each domain has its unique properties, design problems can be solved or simplifications can be obtained by choosing a correct domain, during the implementation of a certain software aspect. For example, unwanted dynamic behaviour can be solved in both the controller and mechanics domain, the interaction point at the top-right of the figure provides means to move the problem to the domain that is most suitable to solve it.

Branch (b) is most interesting for software design for cyber-physical systems, therefore this branch will have the main focus of the following discussion. Furthermore, it is

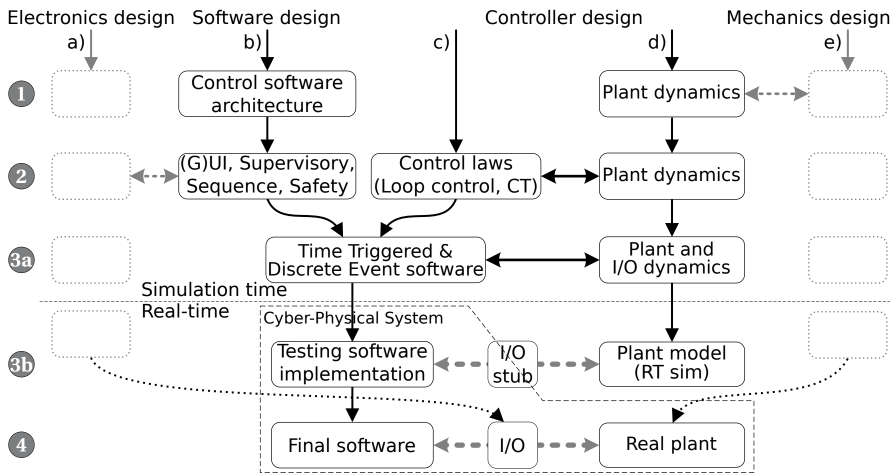


Figure 3.2: Steps of the way of working to design control software for cyber-physical systems, further improvement of Broenink et al. (2010b)

assumed that the plant and its models are already available or being developed by other members of the team.

The way of working consists of the following steps, represented by the grey circled numbers in the figure. The steps are optional, if one is not required for the cyber-physical system software design, it can be skipped, but in order to design first-time-right software it is adviced to make use of all the steps:

1. The software branch (b) starts with designing an architectural overview of all loop control components, required to steer all parts of the cyber-physical system.

2. Additional components are added to the architectural overview to handle high-level tasks, such as the sequence control, supervisory control or user interface components. The design is now looking similar to the embedded control software part of Figure 1.2. At this point branch (c) requires that the low level (Loop) controllers are designed, which is typically done using the dynamic plant model of branch (d).

3. The implementations of the loop control components of the architectural model are provided by the control law designs of branch (c). The implementations of the other modelled components, also need to be filled in by now. At this point the modelled functionality of the control software of the cyber-physical system is complete. The dynamical plant model now includes I/O dynamics to mimic the actual sensor and actuator signals that are available in the cyber-physical system.

   3a In this sub-step the resulting software (models) are *(co-)simulated*, in collaboration with the dynamic plant model, to check whether the software behaves as intended. Depending on the simulation results, the models can be fine-tuned to get a better behaviour.

(3b) When the software fine-tuning is finished and showing the intended behaviour, real-time constraints are included by testing the control software on the target computing platform. Again, the simulation and test results can be used to further fine-tune the software components.

(4) When the mechanical, electronic and software designs are finished and properly functioning according to the simulations, the software can be deployed and tested on the cyber-physical system.

The modelling phase (Figure 3.1, step *I*) is most interesting for control software design. According to Brugali and Scandurra (2009), separating the component specification and implementation leads to better reusable component parts. The way of working also practices this separation: The control component specification is taken care of by step (1), this results in an architectural model. The high-level components are added to this network in step (2). Now it provides information on the requirements of the components that are used in the system and shows the network of these components. The next part of step (2) and the first part of step (3) provide the implementation of the components that were defined in the architectural model.

The generic overview of the way of working, incorporating all development domains and their interactions is depicted in Figure 3.2. In the following sections, the focus is placed on the specific aspects of the steps of the software design branch of the modelling phase, explaining the details required to design the cyber-physical system software following the way of working.

3.1.1 Software Architecture Modelling

The software architecture needs to be modelled according to the CPC methodology, described in Section 2.5. Separate parts of the system are modelled as components with ports and connections for their communication. These components implement the controller algorithms required to control the separate parts of the system. Additional supervisory and sequence control components need to be added to the model to take care of the cooperation of the controller components, so the system is able to complete complexer tasks.

An example architecture overview of a cyber-physical system is shown in Figure 3.3, it is mapped to the overview of Figure 1.2. At the right an example of a plant is depicted, in this case a youBot arm (Bischoff et al., 2011), it has 5 joints, a gripper and is mounted on a base. In the middle the hardware interface is available, depicted in a simple manner with a bi-direction arrow as its actual implementation is out of scope for this thesis. The left side of the figure shows all separate joint control components with some additional components of the supervisory and sequential tasks.

The architecture model of the control software of the cyber-physical system is shown in Figure 3.4. All defined units of the architectural overview have their own component containing their control law implementations. These loop controllers require the sensor data of their component from the hardware interface and provide the control signals to the hardware interface, resulting in the typical control loop structure. The

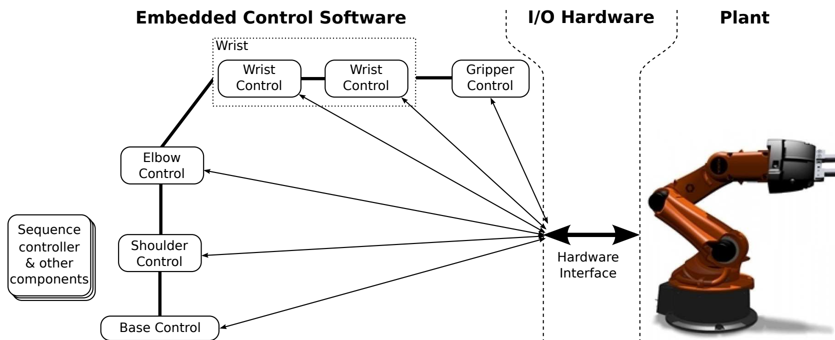


Figure 3.3: Example overview of a cyber-physical system

sequence controller component is used to manage the loop controllers. Connections between the sequence controller and the loop controllers are used to steer and synchronise the loop controllers in order to perform the actual required tasks with the arm. The implementation of the hardware interface component makes use of available frameworks or drivers that are able to communicate with the particular physical hardware units.

As mentioned, each of the joints has its own component in the architecture model. Assuming that the joints in this example cyber-physical system are controlled in a similar way, the implementations of (some of) the joint components can be reused. The shoulder, elbow and wrist tilt joints are all angular rotating joints with a limited range. Therefore, their loop control component should use the same implementations, provided with their initial parameters for obtain the required dynamical behaviour for the part of the robotic arm they need to steer. The same goes for the base and wrist roll joints, these joints could also share their controller implementation.

Reusability can be taken one step further as all components require a basic set of functionalities. The basic functionality set is required to make sure that the components are able to run next to each other, to combine them into larger, more complex components and to be able to let them interact with each other and their environment. All components also require some form of safety support, to react on (un)foreseen safety problems, and configuration support, so configuration parameters can be provided to the controller to provide the specific behaviour of a more generic, reusable controller implementation. These kind of default component functionalities should be provided by a template component, preventing reinventing the wheel for each component and prevents repetitive development efforts. Section 3.3 provides more information of such a generic component.

3.1.2 Software Testing

After the architecture modelling phases are finished, steps ① and ② of Section 3.1, the results need to be tested more thoroughly as indicted by step ③. During the design of the models, some functional tests are likely to be performed already, this

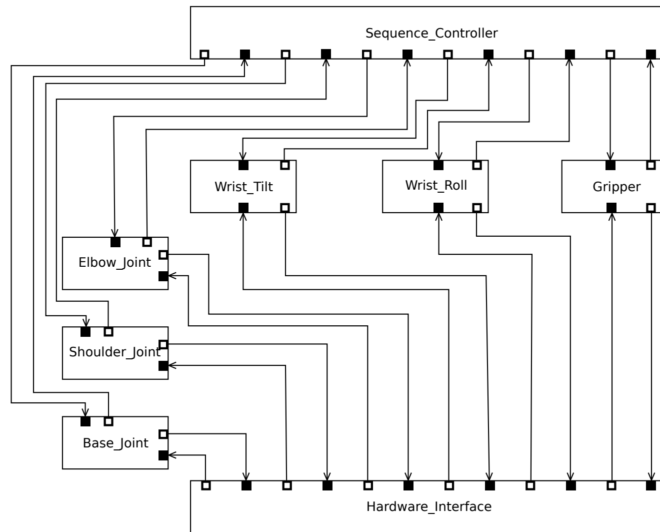


Figure 3.4: Architecture model for the cyber-physical system example

step includes behavioural tests to test the component implementations as well. As mentioned earlier in the first-time-right objective discussion (Section 1.3.3), thorough testing of the models prevents problems during execution, which then takes more effort to solve or might result in broken (physical) parts of the system.

The testing step is split into two parts:

1. Step 3a tests the models against a plant on a more architectural level, i.e. to test whether the software is behaving as expected.
2. Step 3b focuses on the timing requirements to check whether the required deadlines are met or not.

The actual testing methodologies depend on the type of the model that requires testing, for example a state-machine needs to be checked whether all states, that are part of the regular execution, are reachable and a CSP model needs to be checked using formal verification methods to check for scheduling problems, like deadlocks, causing the application to become unresponsive. Furthermore, testing methodologies can either test the functional correctness, for example the formal checks, or whether the application behaves as expected. The latter mostly is performed using *visual inspection* by the developer as it is hard to provide behavioural tests that check the complete behaviour of the system. Visual inspection is possible using co-simulation and the dynamic plant models, this prevents breaking the actual system when the behavioural tests failed and the developer did not turn off the system in time.

3.1.3 Software Deployment

The software can be executed on the target platform, after software testing passed. The models need to be converted into an executable application that can be deployed to the *computing platform*. Converting and deploying software on the target platform requires the steps shown in Figure 3.5, together these steps form step 4.

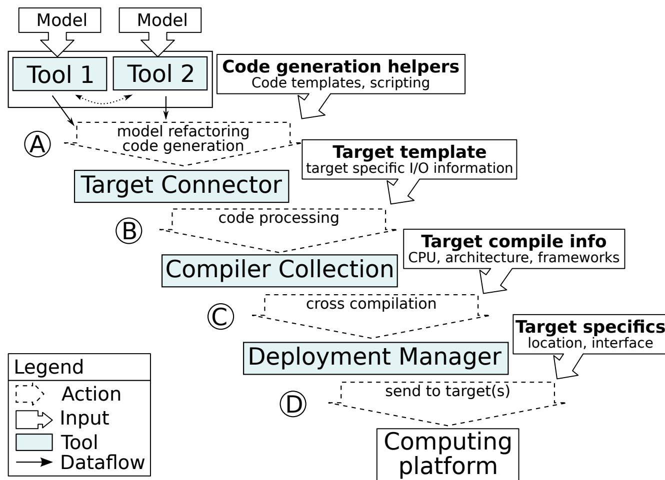


Figure 3.5: Overview of required tools in embedded control software development, based on Bezemer et al. (2011a)

The resulting models of step 3 of the way of working are shown at the top of the figure. One of more tools might be used to design these models and they might interact which each other. The exact details depend the tools that were used in the design process and the requirements of the control software. The conversion from models to executable code on the target platform is done through these steps:

Ⓐ These models first need to be converted to source code. Before this conversion model refactoring using model-to-model transformations might take place for *model optimisation* reasons. The *code generation* is done by the modelling tool itself, or a closely integrated part of a tool suite or toolchain of which the modelling tool is also part of.

Ⓑ Target-specific information is added to the source code by the *target connector*. This information typically contains details about the target cyber-physical system that hosts the control software, like I/O capabilities and other (peripheral) hardware information. Whether this information is added by the modelling tool suite, e.g. during the model refactoring, to the models before code generation, or directly to the source code afterwards, does not really matter. Just as before, this probably depends on the modelling tool and whether it is part of a larger tool suite.

Ⓒ The code now needs to be *compiled* to create the actual executable application that is going to be executed on the target. During compilation one or more *frameworks* are typically added to provide execution engines and hardware abstraction layers.

Ⓓ The last step is to send the executable to the computing platform. For this the *deployment manager* requires target specifics, like the location of the actual target so it is able to connect to it or the required communication interface. Typical features of a deployment manager also include: Remotely starting the execution of the software on the computing platform or sending logging information back to the development platform.

Steps Ⓑ until Ⓓ are usually handled by a single deployment tool or technically a tool suite as deployment is probably done by multiple specialised tools. The tools have access to the information about the resources and capabilities of the intended target platform and is able to provide these details to the software. Cross-compiling the software is also target dependent, so the deployment tool suite provides the correct cross-compilation tools during step Ⓒ. Sending the compiled executable to the target requires support on the target itself, as the executable needs to be accepted, stored and executed on the target. This on-target support must match with the expected communication protocol. It therefore provided by the deployment tool suite as well and typically needs to be installed on the target once. The deployment tool suite in combination with the on-target support, also provides means for sending log and debug information back to the development platform.

An example of such a deployment tool suite is 20-sim 4C (Posthumus, 2007), it includes the described steps and requirements. Its intended use-case is to provide this support for 20-sim (Controllab Products, 2012), but it is possible to use 20-sim 4C with other modelling tool suites as well. In such situations the (generated) source code needs to be accompanied with additional configuration files. The configuration details provided by these files are used by 20-sim 4C to cross-compile and deploy the software.

Simulink Coder (The MathWorks, 2012) is another example of a combined code generation and deployment tool. It uses MATLAB and Simulink files and models as input to generate code and execute it on a provided execution environment.

3.2 Tool Coverage

The previous section explained the different methodologies that are part of the way of working to design the software models and how to convert and deploy them. This section maps these methodologies on actual tools of any tool suite, e.g. the TERRA tool suite that is described in Chapter 6, and shows what features the tools need to provide in order to support the described way of working. Additional ideas and possibilities of the tooling as discussed as well.

3.2.1 Graphical Modelling

The most important and central part of the way of working are the models, as all design steps revolve around them. Graphical models and modelling tools further increase the understanding and maintainability of the models and the way of working. Modern graphical modelling tools, called *editors*, are likely to interactively guide the user to use the correct model syntax and semantics, point out errors and help in various other ways during the design process. Thereby increasing the productivity of the designer, as the designer is able to directly correct a problem and learn from it, so this type of problem can be prevented on beforehand the next time.

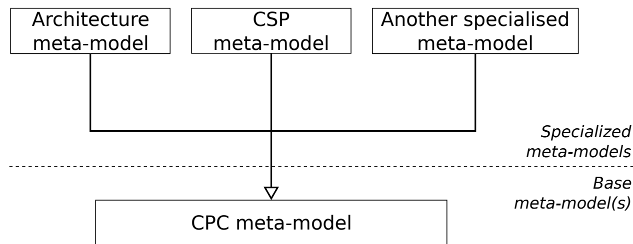


Figure 3.6: Inheritance diagram showing the shared modularity of the meta-models

The first step in the way of working towards the embedded software for a cyber-physical system, is to design the architectural model. Such a model provides the overall overview and should be based on the CPC methodology, as an architecture model typically requires components, ports and connections.

The CPC meta-model is shown in Figure 3.6, acting as the base implementation of the other meta-models. These other meta-models extend the CPC meta-model, as shown by the open arrows in the figure, to add more, specific model semantics. By using the same base implementation, the models become (more) compatible to each other, which is one of the reasons for using the CPC methodologies. Furthermore it saves development resources as this base and its support only needs to implemented once and then can be reused for all other specialised meta-models and their tools.

Optionally, but not shown in the figure, it is possible to let a meta-model reuse multiple base or specialised meta-models in a modular way. For example, the architecture meta-model could reuse the CSP meta-model as it defines definitions to specify the relation between processes, components in the architectural meta-model, which could improve the flexibility (and complexity) of the modelling capabilities of the architectural models. In the case that the architecture meta-model actually does implement the CSP meta-model, it would make the CSP-related features, like formal checking, available to the architectural models and tools without much effort.

The architectural modelling tool, or *architecture editor*, provides means to model the software architecture of the system by defining the (control) components (step ➊ of Section 3.1). During steps ➋ and ➌, these software components are provided with their actual implementation. Their implementations must be based on a meta-model that extends the CPC meta-model. Furthermore, the *component interface* and the

component *implementation interface* must be matching. Both interface types have their own role to synchronise the architecture component and their implementation:

- The architecture model defines the interface of each components which specifies what ports are available, or required.
- The component implementation needs to adhere to this interface by providing the same ports (with reversed direction as described in Section 2.5), so the communication flow is able to go from the architectural model to the correct location in the implementation models.

Furthermore, as these implementation models need to be easily connectable to the architectural models, their meta-models must also support the CPC methodology.

A specialised meta-model is not required in situations where the architectural components do not require an implementation that is designed with the modelling tool suite. This is the case, for example, when the implementation is directly designed in a programming language or when the implementation is provided by an external tool. The architecture modelling tool needs to provide direct support for these cases. This might for example include providing a textual editor or means to modify the model parameters to fine-tune the external model.

3.2.2 Code Generation

Code generation tool support takes care of transforming the models into *source code*, using *model-to-code transformations*, also known as *code generation*. This source code, together with *framework(s)*, can be *compiled* into the actual executable application, as shown in Figure 3.7. In a sense, code generation tools add information that is required to execute the model.

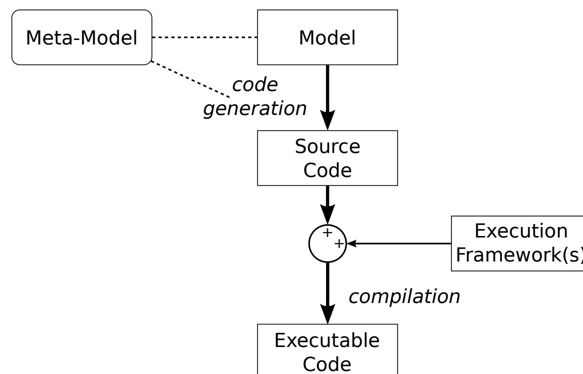


Figure 3.7: Overview of the required steps to convert a model into an executable application

One or more frameworks are used to properly compile the source code into the executable code. These frameworks contain the *static code* that is model independent to provide specific functionality, like a model execution engine. Code generation tools

make use of these frameworks to reduce the generation of static code, which increases the maintainability of the source code and the code generation tool itself. Additional requirements for execution frameworks are provided in Section 3.4.

LUNA, described in Chapter 5, is an example of such an execution framework. Each meta-model is typically backed up by an accompanying execution engine to support the generated code. LUNA provides, for example, a CSP execution engine, which contains the static code to support CSP constructs, schedulers and rendezvous channels.

Other types of frameworks, containing for example advanced or complex sensors, are typically targeted when generating code for cyber-physical systems. These additional frameworks are then used to provide the static code, or drivers, for these complex hardware components. ROS (Quigley et al., 2009) is for example a typical additional framework, it supports all kinds of complex sensors like range finders, cameras, force, torque and touch sensors (ROS Sensors, 2012).

A framework could also provide support for specific, complex controller related tasks, like cartesian or kinematic controllers. Code generation of such frameworks requires meta-model specialisations that is able to address these complex tasks, which needs specialised tool support. Orocos is a framework that provides support for these kinds of tasks as shown by Buys et al. (2011).

Code generation tools are also required when an external tool does not provide an explicit meta-model. Instead of using the target meta-model to transform the source model, the code generation tool needs to provide its own version of the target semantics in order to generate the target code. This is usually more error-prone, as the target semantics are not completely interpreted correctly, or are unknown and needed to be reverse-engineered (guessed), or have been changed over time and the semantics used by tool has not been updated accordingly. For example, the formal verification tool FDR2 uses a plain text file as input using the machine-readable CSP syntax. Converting a CSP model into such a file, requires model-to-text transformations, so the CSP model can read by the external tool in order to formally check the original CSP model.

3.2.3 Model-to-Model Transformations

Model-to-model transformations are a more reliable transformation method, compared to the model-to-code transformations described in the previous section. They use the strictly defined model structure of the meta-models of the source and destination models. If the source or target meta-model changes, the tool can be recompiled and uses the update meta-model as well. In situations where the meta-model underwent some drastic changes, recompilation fails and the tool developer needs to address these changes. In all situations, the updated tool makes use of the most up-to-date model semantics, or simply does not support the newest meta-model version anymore. In the end, the tool will (should) never try to use outdated semantics for its transformations.

Additionally, it is possible for model-to-model transformations to use the same meta-model for both the source and destination model. This is for example the case when

the transformation is used to optimise the model for a specific requirements, like optimal model execution speeds or low resource usage. In such cases all redundant model information, as defined by optimisation goals, is stripped without influencing the intended behaviour of the model. The resulting model might look completely different compared to original model and might not match the *modelling point of view* of the designer anymore.

As the optimisation result still uses the same destination model, all other tools are still usable to perform their tasks on the optimised model, so model optimisation can be seen and used as an intermediate step in the modelling toolchain. This type of model transformation is typically be done before model-to-code transformations to automatically generate an optimal implementation of the model to execute on the target platform (Bezemer et al., 2009).

3.2.4 Co-Simulation

As explained earlier, co-simulation requires a tool that is leading the simulation. This tool should have knowledge about the architectural model structure in order to be able to run the simulation.

An additional and desired feature of a co-simulation tool is to be able to choose a specific implementation for each defined component interface. For example, components that are providing a connection with the actual cyber-physical system should be simulated with a component implementation that simulates the hardware or that is able to connect to the dynamic plant model. This is depicted by the I/O stub component in Figure 3.2, which is replaced by the real I/O implementation in a later stage. This prevents that the actual system needs to be finished and available and solves safety and construction issues early stages of system development. In later stages of the system development, the chosen component implementations could provide support for hardware-in-the-loop simulations.

This feature can also be used to try out different component implementation ideas. Especially in combination with a *model management tool*, that is able to keep track of the different component implementations and complete coherent sets component implementations of an architecture model. These sets of component implementation can also be used to simulation specific *scenarios*.

3.3 Generic Architecture Components

It is concluded in Section 3.1.1 that a *Generic Architecture Component* (GAC) is desired. Such a component provides a generic component implementation in order to prevent wasting design resources due to reinventing such a component structure over and over. The GAC needs to be usable in a variety of situations in order to make it a usable modelling method with respect to the way of working. Therefore, it needs to provide features that are required by all (or at least most) component implementations, but in such a way that these features are optional and not restricting the designers in their preferred ways of working.

Most important is handling the *life cycle* of a component during its execution. All com-

ponents need to be initialized, configured, started, executed and stopped during the execution flow of the application. Support for such a life cycle could be provided with a Finite State Machine (FSM) implementation. A command interface would be required to provide means to externally influence the life-cycle states of the component.

Each component also requires some form of safety support. The GAC must provide mechanisms to check whether incoming and outgoing signals are correct. For example, it might be required that these signals stay within their corresponding, expected ranges. If this is not the case, an error signal needs to be propagated to the (local) error handlers. These error handlers are able to correct the behaviour of the cyber-physical system, or to (partially) shut it down in a safe way to prevent damage to itself or its environment.

The GAC needs to be reusable in two ways:

- It is used in a variety of component implementations and situations. The GAC must provide a basic implementation that is compatible with (most of) these possible use cases.
- The actual component implementation using the GAC, support a complex part of a cyber-physical system, otherwise it was not sensible to use a GAC. Designing and testing such complex components take time and effort, so afterwards they must be usable in different (parts of) cyber-physical systems.

The GAC requirements that are described in this section, result in a GAC *component interface* defining signals for the commands, sensor and actuator data, and so on. Interfaces of different GACs need to be connectable, so a network of component can be constructed, resulting in the software architecture for the cyber-physical system control software.

These requirements are taken into account during the development of the GAC implementation. More information about all requirements, the design and the implementation of the GAC is provided in Chapter 4.

3.4 Execution Framework

The execution *framework* sits in between the modelling tools and the target platform, as depicted in Figure 3.7. It provides an *application programming interface* (API) and a corresponding implementation, providing sufficient support to keep the amount of generated code to a minimum. As a result the code generation tools do not require to generate the required *static code* each time.

A suitable execution framework must also provide support for the target platform that is going to execute the control software. The execution framework should provide an *abstraction layer* between the provided interface and the actual hardware and its capabilities. It implements the requirements to properly use the hardware, like the attached peripherals, sensors and actuators, which can then be used the other components of the framework or by the applications themselves. Additionally, it is desired to support an implementation for the development platform as well. This would provide

means to execute and test the software more easily within the environment a designer is working, resulting in easy testing even when the target platform is not yet available.

The execution framework must be able to guarantee that deadlines are always met within the defined time, called hard real-time support. Hard real-time deadlines are required for proper functioning of the loop control law implementations.

Additional details on the requirements of an execution framework are provided by Chapter 5. The design and implementation details of the LUNA framework are discussed in the rest of that chapter.

3.5 Conclusions

A way of working that is suitable to design (embedded) control software for cyber-physical systems using model-driven design techniques, is proposed and discussed in this chapter. It is integrated and compatible with the development trajectories of the electrical and mechanical domains. Furthermore, the way of working is designed to be as scalable as possible. For example, it provides and describe simulation phases to it possible to use for complex cyber-physical system. On the other hand, all steps are optional so simple software designs do not require to implement all design steps.

The requirements of the way of working are provided and discusses as well. These requirements consist of the generic architecture component, tool support and execution frameworks. Without implementations of these requirements the way of working loses much of its value.

The next chapters provide further details on these requirements and show implementation details of the actual development of these way of working parts. First, in Chapter 4 the details of the GAC model design, implementation and use-case are provided. Next the LUNA design and implementation details are described in Chapter 5 and finally the TERRA tool suite is described in Chapter 6.

Together these three chapters provide the implementation details of largest part of the way of working. The only missing part of the way of working is the deployment stage and its accompanying tool-chain, as it is not part of the research of this thesis.

4

Generic Architecture Component

Information on the design and implementation of the *Generic Architecture Component* (GAC) is provided in this chapter. Such an architecture component provides a base template to design component implementations for cyber-physical systems. It is part of the way of working (Chapter 3), but the ideas of the GAC are also usable in other situations.

The generic architecture component has the following properties:

- *Genericity*
  The component is usable for all kinds of (cyber-physical) systems, ranging from low resource embedded systems to large industrial or medical systems. Furthermore, all kinds of tasks must be supported by the GAC, ranging from low-level loop controllers to supervising algorithms which plan the behaviour of the system. In order words: the component needs to be *generic*.
- *Architectural*
  The component is suitable to be used as part of the system architecture. The system architecture consists of the components which together are able to control the physical system. These components are connected to each other, in order to be able to work together on their required tasks. The GAC needs to be part of this *architecture* of components and provides support for this.
- *Component*
  The GAC behaves like any other software component, it provides base support for often required functionalities.

The GAC that is discussed in this chapter is a blue-print or template with base functionalities/support to construct real component implementations that are derived from this template GAC, instead of a real component implementation. The GAC blue-print is called *template GAC* in this chapter and the component implementation based on the GAC is called *specialised GAC*. Note that, by default template GAC is meant when neither the template GAC nor the specialised GAC are explicitly mentioned.

The GAC needs to support all kinds of components that are usable in cyber-physical systems ranging from humanoid to medical to industrial robots, because it needs to

be generic. This range of target systems requires specific support, which is discussed in the first section of this chapter. Available component models and accompanying frameworks that could be used to implement the GAC are discussed in the following section. Next, the GAC design approach and implementation are described. Followed by a use-case showing that the GAC is usable to implement control software for an actual cyber-physical system. The chapter ends with a discussion, looking back on the design, implementation and the use-case of the GAC.

4.1 Requirements

The requirements for the GAC, which are based on the properties described above and on the required GAC features that are discussed in Section 3.3, are elaborated in this section. The requirements of the way of working either focus on reusability and flexibility, i.e. the requirements that make sure that the GAC indeed is generic, or on the required features that define its behavior and functionality. The list is structured using the MoSCoW method (Clegg and Barker, 1994).

Requirement 1: *The GAC must be reusable*

The generic architecture component is supposed to be generic; ideally it needs to be usable for all kinds of cyber-physical system implementations. Therefore, both the template GAC and the specialised GAC must be reusable.

The template GAC needs to be applicable to multiple situations and design requirements, so the specialised GACs also become applicable to a wide range of situations.

On the other hand, the specialised GACs themselves also need to be reusable, so they can be used to implement multiple parts of the system. For example configuration support makes it possible to fine-tune a specialised loop-controller GAC, making it possible to control both the left-hand and the right-hand parts of a symmetrical system using the same specialised GAC.

Requirement 2: *The GAC must support the CPC methodology*

As indicated in Section 3.2.1, the CPC methodology must be used as a base meta-model type. By using the CPC meta-model for the template GAC model, it becomes possible to easily connect specialised GACs to other GACs (or other components) that are based on the CPC meta-model.

Requirement 3: *The GAC must be real-time capable*

The GAC needs to be able to operate at different real-time levels, in order to make it suitable as a component implementation of all kinds of use cases. Loop-controller GACs need to provide hard real-time support to implement the controller code. On the other hand, supervisory or sequence controllers do not require the strict hard real-time guarantees. However, communication between GACs with different real-time levels is required. For example, soft real-time decision-making GACs typically are used to assign tasks to the hard real-time control loop GACs.

Support for multiple real-time levels is also required within the GAC itself, this pre-

vents excessive, superfluous resource usage (and thus the unnecessary waste of these resources). Control related parts of a GAC, that must perform their calculations periodically to prevent unstable dynamic behaviour, need to run with hard real-time guarantees. Whereas command handlers for example suffice with soft real-time guarantees, as it does not matter if a command is interpreted slightly later than it was issued. Flexibility is required, so these real-time levels can be chosen when the specialised GAC is designed or when it is used to model the control software.

Requirement 4: *The GAC must provide means to support safety measures to handle undesired situations*

Even though the model-driven design techniques help with developing correct software, safety measures are still required to handle undesired situations. Therefore, the GAC must provide means to handle safety aspects, like detecting and handling unsafe situations, on both a local and a global level.

Requirement 5: *The GAC must provide means to extend it into a specialised component implementation*

The template GAC must provide means to actually specify its required behaviour, and thus converting it into a specialised GAC or specialised component. For example, a loop controller component needs to execute the actual loop controller algorithm, so the GAC needs to provide means to add this algorithm and to make sure it is executed periodically.

Therefore, the GAC must provide place holders called *hooks*. These hooks provide means to add the actual implementation content or payload of the component. Each hook provides support for a specific part of the implementation, which are used by designers to mold the template GAC into specialised GACs. By providing enough of these hooks, the template GAC can be used to create any desired component. The actual implementations of these hooks should be optional and allow to be specified in multiple formats, in order to keep the GAC as generic as possible.

Requirement 5.1: *The GAC hooks must support source code*

The hook payload needs to be accessible by the software, which is built from source code. Even when using MDD techniques, the models usually are transformed into source code when building the application. So the hooks must at least support source code as their payload.

Requirement 5.2: *The GAC hooks should support models*

It should be possible to use models as payload for a hook, so the specialised GAC can be constructed using MDD techniques as well. These models can be easily connected to the GAC if they are based on the CPC meta-model as well.

Requirement 6: *The template GAC must be formal correct*

The GACs are used in a variety of situations due to their reusable, flexible nature. They

must function properly in all these situations, so template GAC must be formal correct to prevent (unexpected) problems, like deadlocks, due to the situation it is used in.

Requirement 6.1: *The specialised GAC should be suitable for formal checking*

The specialised GAC should also be formal checkable. Implementing a template GAC into a specialised GAC might introduce new problems that can be detected with formal verification. Furthermore, if a specialised GAC is formal checkable, it becomes possible to formally check a complete network of specialised GACs or the complete control software. This helps finding design errors that occur due to wrongly added connections between the specialised GACs.

Requirement 7: *The GAC must be usable in hierarchical networked situations*

It must be possible to use the GAC in a network of other GACs, in which they together form the control software. Additionally, one or more GACs must be able to provide the payload of another GAC. With this requirement, a complex GAC can be split into multiple GACs which together provide the complex behaviour. These simple GACs could then be (re)used for multiple situations.

4.2 Existing Component Models

Several implementations of cyber-physical software frameworks are available, like Orocos and ROS, as indicated by Mallet et al. (2010). Most of them make use of some custom component model, interface or template, which needs to be implemented by the engineer.

Earlier work on defining, designing and implementing a generic component has been done by Wijbrans (1993) and Groothuis et al. (2008). As mentioned earlier, this work is not flexible and reusable enough to design control software for a wide range of cyber-physical systems, but their design ideas are kept in mind.

4.2.1 BRICS Component Model

The *BRICS Component Model* (BCM) provides the *5Cs* approach (Klotzbücher et al., 2013). It defines the following 5 concerns (5Cs): Computation, Communication, Coordination, Configuration and Composition. As mentioned earlier, separation of concerns is used throughout this thesis, hence an implementation in the form of the 5Cs is used for the GAC design. More details of the implementation of the 5Cs in the GAC design are provided in Section 4.3.1.

4.2.2 Orocos

Orocos (Bruyninckx, 2001) is a framework targeting cyber-physical development, especially focusing on more complex components like sequence and supervisory control components.

The Orocos *TaskContext*, shown in Figure 4.1, provides the basis for an Orocos component. A component offers *Services* through the *Operations* interface, and requests them through the *OperationCallers*. The *Flow Ports* are used as a data transport mechanism between components. At the center of the component, its *Execution Engine*

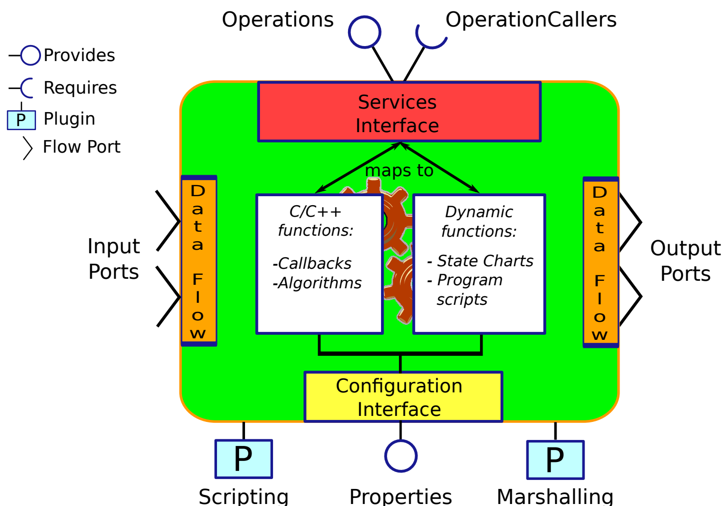


Figure 4.1: Orocos Component Model (Soetens, 2012)

handles the execution of the component, depicted by the gears in the background of the figure. Each component has a state machine handling the state flow of the component. The user is able to add functionality to hooks which are activated upon state transitions. Orocos provides a real-time scripting environment to program the component without modifying its source code, allowing for fast, iterative design cycles. This only summarizes the main features of the Orocos components, in reality it is a fairly complex component with lots of other features and possibilities, making it very flexible for all kinds of situations and requirements.

Using the Orocos component model results in partially real-time software only: The component itself is real-time, i.e. only deterministic operations are used during the actual execution. But, the components (basically their *OS threads*) are being scheduled by the *operating system* (OS), which makes it hard to provide hard real-time guarantees for scheduling the components. Unfortunately, the scheduling behaviour is strongly dependent on the implementation of the (real-time) operating system.

Over-dimensioned hardware results in proper execution of the soft real-time components, like sequence or supervisory control components. However, this solution is unsuitable for embedded computing platforms unsuitable, so a real-time OS is required.

4.2.3 ROS

ROS (Quigley et al., 2009) is another well-known and widely used robot-control software framework. The main focus of ROS is to support code reuse in robotic research and development.

It provides high-level (supervisory) solutions for message-passing between processes and package management by providing a publisher-subscribe mechanism. Unfortunately, ROS does not provide a specified component model, it cannot provide real-

time guarantees and ROS only runs on general purpose PCs and operating systems, it does not support embedded platforms yet.

Although ROS is not hard real-time capable, it is a good framework to stay compatible with, due to the amount of supported (specialistic) sensor and actuator hardware drivers. These drivers could be used with soft real-time guarantees, in order to provide the other components their data. This is especially possible for complex sensors, like cameras or laser range-finders, which are typically used by sequence or supervisory control components that are soft real-time themselves.

4.2.4 Conclusion

It can be concluded that there is not a suitable component model according to the requirements of Section 4.1. Therefore a new component model has been designed, called Generic Architecture Component (GAC), as discussed in the rest of this chapter.

Besides the lack of a suitable component model, there is also a lack of a suitable execution framework, so a newly designed framework is used for the GAC implementation. This new framework is called LUNA and its design and implementation details are described in Chapter 5.

4.3 Design

The template GAC design is kept simple and light-weight in order to keep it usable on embedded platforms. Due to the use of MDD techniques, the modelled GAC is platform independent and easy to connect to the rest of the control software models. These design choices are a first step to design a reusable GAC as required by Requirement 1.

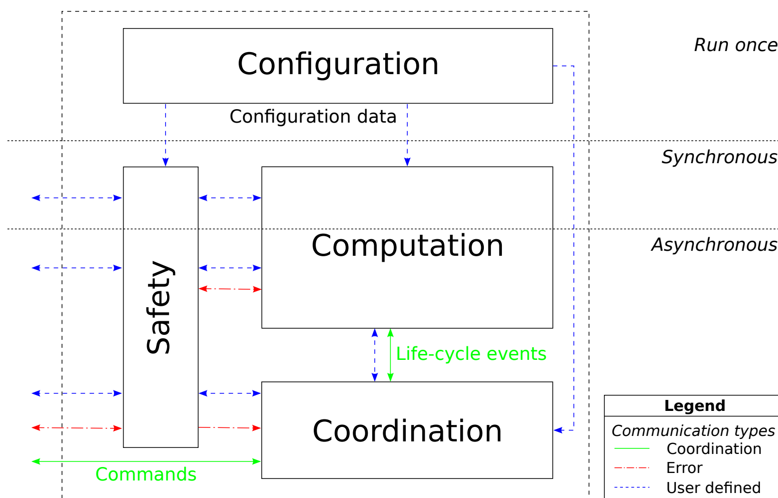


Figure 4.2: Design overview of the generic architecture component

An overview of the GAC design is shown in Figure 4.2. Its functionality is spread over 4 blocks that work together and provide the GAC implementation. Furthermore, the GAC is split into three execution layers with their own properties:

- *Run-once layer*
  The blocks in this layer are executed once, during the initialization of the GAC.
- *Synchronous layer*
  The blocks in this layer are periodically executed.
- *Asynchronous layer*
  The blocks in this layer are executed when an event is received and needs to be processed.

The blocks of the GAC are placed in one or more of these layers, depending on their requirements.

There are multiple communication flows between the blocks and between blocks within the GAC and from blocks to the outside of the GAC. These communication flows are split into three types. Two of these types are predefined by the template GAC, consisting of coordination and error communications. The third type represent optional user definable communication flows as required for specialised GACs, i.e. the I/O communication from/to the outside of the component.

Most incoming communication flows, from outside of the GAC, go through the *Safety block* to their destination. Incoming data from the outside of the GAC is considered to be unsafe, the safety block checks the data for errors, unexpected or undesired values and handles these signals as necessary. The data that leaves the safety block is considered to be safe. Data leaving the GAC also go through the safety block, in this case the block also checks if the data is correct, for example within an expected or allowed range. If this is not the case, it might indicate that the GAC is malfunctioning and requires attention to solve the problem.

Details on the separation of concerns within the GAC are provided in the next section. Each of these separations and how it is influencing the GAC design is discussed in the following sections.

4.3.1 Separation of Concerns

As depicted in Figure 4.2, the design of the GAC is split into 4 separate blocks, each of these blocks handles a separate, specific part of the GAC. Three of the blocks are directly derived from the 5Cs approach of the BRICS component model. Although the Communication and Composition concerns are not directly visible in the figure, they also influence the GAC.

The 5Cs are used in the GAC design in the following way:

- The *Computation block* of the GAC provides means for the execution of the (control) algorithm, i.e. the payload of the GAC.

- The *Coordination block* handles the life-cycle state machine to synchronise the activities, or states, of the component.
- *Configuration block* of the GAC provides the values of the specialised component parameters, increasing the reusability of the template GACs and the specialised GACs.
- The *Communication concern* takes care of the communication by adding ports to the component, thereby providing communication with the environment of the component.
- The *Composition concern* is implemented indirectly with the different possibilities of creating an architectural network of GACs.

Safety is not one of the original 5 concerns, but its role is important enough to have the it included in a prominent place in the GAC design. Because for this prominent place, a designer is likely to notice the block and make use of it, resulting in specialised GACs with a high(er) quality.

The roles within the GAC of each of these concerns (including safety) are further discussed in the following sections.

4.3.2 Computation

The computation block takes care of the computation of GAC payload. It needs to provide means to add user-defined computation hooks, one for each life-cycle state, as required by Requirement 5. The implementation of these computation hooks can be provided by the designer as CPC-based models or as direct source code, as specified by requirements 5.1 and 5.2. The implementation of the hook that belongs to the active state needs to be executed by the coordination block.

It is also possible to use other GACs as a (partial) implementation of the computation block, this is explained further when the composition concern is explained. The computation block is placed in both the synchronous and the asynchronous layers in order to make the computation both event based and periodic.

4.3.3 Coordination

The coordination block takes care of the *life-cycle* and user-defined state machines of the GAC. The life-cycle state machine is used to provide the regular life cycle of the GAC, whereas the user-defined state machine allows for custom, more fine-grained component states.

The block is able to receive commands, which are basically events, from other GACs to control the active life-cycle state of the GAC. For example, a task-planning GAC keeps track of the long-term goals of the system and controls the other GACs to obtain these goals using events. Changes of the active state are communicated to the computation block, which changes its behaviour accordingly by using the correct hook for the active state. The coordination block is placed in the asynchronous execution layer, as it is completely event based.

The life-cycle state machine that exists in the coordination block is shown in Figure 4.3. Event names are written in capitals and the state names are written bold, in the figure as well as in the text. The figure is kept simple and only has a few states and transitions.

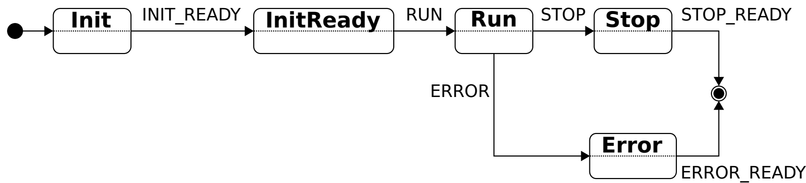


Figure 4.3: GAC life-cycle state machine diagram

The GAC starts in the ***Init*** *state*. This state is meant to perform initializations of the GAC and homing actions of the mechanical system. When the initialization is finished and the *INIT_READY event* is issued, the GAC enters the ***InitReady*** *state*. This state is used to synchronise all GACs: After all have finished their initializations, the *RUN event* is sent by a supervising component to enter the ***Run*** *state* of the GACs. In this state the actual functionality of the GAC is activated. The **Run** state can be exited either in the normal way, by stopping the GAC using the *STOP event*, or due to an unrecoverable error, which is signalled by the *ERROR event*. Depending on the event, either the ***Stop*** *state* or the ***Error*** *state* is entered respectively. Both states have a ready event: *STOP_READY* and *ERROR_READY*. When the finalization actions, that are required to properly halt the GAC, are finished the according event is sent.

This simple state machine does not provide means to restart the GAC when it is stopped or an unrecoverable error is encountered, the **Run** state is only exited when the software is shutting down. Restarting the component is not desirable as it does not seem to make sense to halt a component whilst it is (actively) steering the mechanical part of a cyber-physical system. It also does not make sense from a software point of view: The architectural network of components suddenly is (temporarily) incomplete, resulting in problems when interaction is required with the halted component. An obvious reason to halt the component though, is to change its behaviour, for example to swap it with another loop-control component due to a change of the active task of the cyber-physical system. It is suggested to implement such situations by implementing the user-defined state machine and to use its states to select the correct control law for a specific situation. This prevents stopping the component, and thus stopping to steer a mechanical component of the cyber-physical system. Additionally, it keeps the complexity of the GAC implementation as low as possible.

As mentioned already, the coordination block supports, besides the life-cycle state machine, an *user-defined state machine*. The user-defined state machine can be used by specialised GACs to add custom states for a more fine-grained control of the GAC life cycles.

4.3.4 Configuration

The configuration block provides means to support configuration parameters of the specialised GAC. Values for these parameters are provided when the specialised GAC is used as a component implementation. Currently, the parameters are just set upon initialization of the GAC, hence the block is placed in the run-once layer, this is done to keep the GAC as simple as possible.

Additional means to support run-time configuration, can be provided by moving the block to the asynchronous layer and adding command-based communication. This makes it possible to send commands to the block in order to reconfigure the GAC when required. It is even possible to embed the configuration parameter values into the command, so it is possible to dynamically change the parameters. This might be convenient for GACs that contain sub-GACs which need to be configured.

4.3.5 Communication

Generalized ports need to be introduced into the GAC to satisfy the communication concern. They are part of the CPC paradigm, which is therefore needed as modelling tool of the GAC design. The ports form the interface of each component and define a sort of communication protocol. This protocol 'defines' what kind of data is going to be available or required by each port. The designer makes use of this protocol by connecting compatible ports with each other.

The ports of different components need to be connected using channels, so data can be transferred from one port to another. These channels dictate the data flows between components ports. By using a suitable channel implementation for each pair of connected ports, the GAC is able to communicate with other components that are part of the system architecture. Examples of available communication implementations are rendezvous channels and buffered channels.

A MDD tool should be able to help selecting the correct channel implementation. Using the modelled system architecture information, it can decide the correct or optimal channel implementation.

4.3.6 Composition

The composition concern provides support to use the GAC in a network of GACs. The GACs can reside next to each other, providing the *architectural network* for the software, or they can be used in a *hierarchical network*, one or more sub-GACs provide the functionality of the parent GACs. More details on these possibilities are provided in this section.

An example of networked GACs is shown in Figure 4.4. The output of the motion profile GAC is a stream of periodically updated set-points, which is connected to the input of the PID GAC. The PID GAC uses these set-points to calculate its steering signal that is used to steer its actuator. Note that this example is too simple, in a real situation it makes no sense to divide this functionality into two GACs.

Usually, the motion profile generator does not require hard real-time guarantees: It

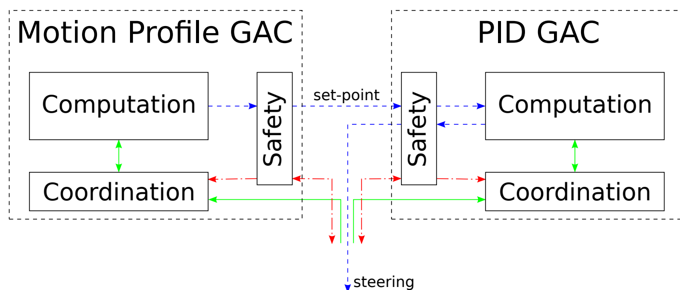


Figure 4.4: Example network of two GACs

does not matter whether the most recent set-point is available to the PID controller all the time. Although they do require to get updated frequently, so it is sufficient to provide the motion profile with soft real-time guarantees. The PID controller directly steers its actuator, these steering signals are required each period time to prevent problems with the dynamics of the physical system. Hence the PID controller requires hard real-time guarantees. Whether soft or hard real-time guarantees are required depends on the controllers and the availability of resources. The networked GAC setup does allow different real-time guarantees for each GAC.

Communication between two GACs with different real-time guarantees requires a buffered channel implementation. This is required to prevent *priority degradation* of the GAC with the more strict real-time guarantees. Priority degradation happens when more strict real-time guarantees are made less strict by another component, because the GAC with the strict real-time guarantees is forcefully waiting for data coming from the other component. The buffered channel always has data available, albeit it might be outdated, which is in contrast with a rendezvous channel that requires both components to be ready before communication takes place.

Both GACs of the networked GAC example provide together the control signal for an actuator. If there are multiple actuators in the cyber-physical system that requires such a controller, it makes sense to create a single GAC that provides the control signal. This GAC might reuse both separate GACs as its implementation for the communication block. Such a GAC is shown in Figure 4.5. The sub-GACs of the loop controller GAC have a grey background.

An hierarchical network of GACs is shown in the figure: The loop controller GAC contains two sub-GACs, the motion profile and the PID GAC. The implementation of both sub-GACs (with the grey backgrounds) is the same as in the previous example, but they are now connected to the coordination block of the loop controller GAC instead of a supervisory GAC. Together the sub-GACs provide the implementation of the computation block of their parent GAC. The result of the computation block, the control signal that is provided by the PID sub-GAC, is the output of the loop controller GAC.

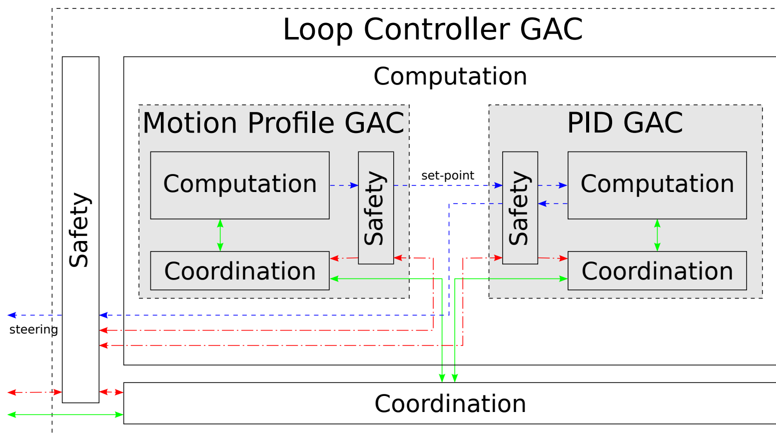


Figure 4.5: Example hierarchical GAC setup

4.3.7 Safety

The safety block provides means to monitor the data flows between the GAC and components outside the GAC. As mentioned before, safety is considered important, hence *all* data signals go through this safety block. A hook is provided to add user-defined checks depending on the requirements of the specialised GAC. For example it could check if a data value is within a certain range, or if a value is not changing too fast. If a problem is detected by these user-defined checks, the safety block is able to modify the data to correct it.

If correcting the data is not possible or allowed, the safety block is able to send the *ERROR event* to the coordination block requesting further action. This event then can be used to sort out the error on a *local level*. If this is also not possible, the supervisory control is notified and the error is handled on a *global level*, probably by shutting down the cyber-physical system completely.

The error signals leaving the GAC are also used when the supervisory component needs to notify the component that a global error occurred and the component needs to shutdown along with the rest of the cyber-physical system. It can either enter the STOP state or the ERROR state to shutdown. This depends on the current state of the component and whether it also has errors or not.

4.3.8 Discussion

Compared to the Orocos component shown in Figure 4.1, the GAC design has similar blocks: The execution flow is handled in its own block, the Orocos component has its execution engine and the GAC has the coordination block. Both handle coordination in separate blocks by providing several life-cycle states, which can be extended and filled in by the user. The configuration of the component is handled by both component models and both handle communication with other components using generalized ports.

Besides these similarities, the GAC design does not provide as many publicly available interfaces to configure the component or to execute component functionalities as the Orocos component does. The GAC only supports (custom) commands/events to change the active (life-cycle) state and user-defined data signals from the outside. This reduction of support limits the flexibility of the GAC a bit, but thereby improves the low resource usage and reduces the complexity of the GAC considerably. Additionally, the GAC provides intrinsic safety handling functionalities, making it possible to detect problems in the incoming and outgoing signals and it is able to handle these problems locally or globally. This will improve the likelihood of the component designer thinking about safety issues, as base support is ready to be filled in with the details.

Because the GAC blocks are strictly separated, it is possible to have multiple real-time levels within the components and/or its sub-GACs. When looking at the hierarchical example GAC setup, the PID sub-GAC needs to be hard real-time, as it directly controls the actuators of the cyber-physical system, while the motion profile sub-GAC does not require these guarantees. Therefore, it would be plausible to provide soft real-time guarantees to the motion profile sub-GAC while keeping the hard real-time guarantees for the main and PID GAC. As mentioned before, the example is too simple to actually split the GAC into two sub-GACs, especially with different real-time constraints as well, but for more complex situations this would make sense.

In conclusion, the GAC design is less complex than the Orocos component, mainly due to the reduced availability of (complex) features. The GAC being a simplification compared to the Orocos component, results in lower resource usage, which is particular convenient for (small) embedded computing platforms. The GAC is still complete enough to be usable as a generic template for cyber-physical system components, as is shown later on in this chapter. An additional advantage of the GAC is that the lower amount of available interfaces results in less complex designs and components, which results in less confusion for (new) engineers.

4.4 Implementation

A first actual implementation is developed by Hoogendijk (2013). It shows that the ideas behind the GAC are feasible, although more work is required to increase the quality and to reduce the resource usage before this implementation becomes fully usable. This GAC implementation is mainly developed using CSP models, so the template GAC and its specialised GACs are, for a large part, formal checkable as required by Requirement 6.

An overview of this GAC implementation is depicted in Figure 4.6. The figure is based on the design overview of Figure 4.2, but filled in with the implementation details, like the described hooks, state machines and signals.

The life-cycle state machine is added to the coordination block. Unfortunately, its implementation is provided in C++ and still needs to be converted to CSP to make the GAC implementation completely formal verifiable. For situations where additional states are required, the *user-defined FSM* hook can be implemented. The user defined

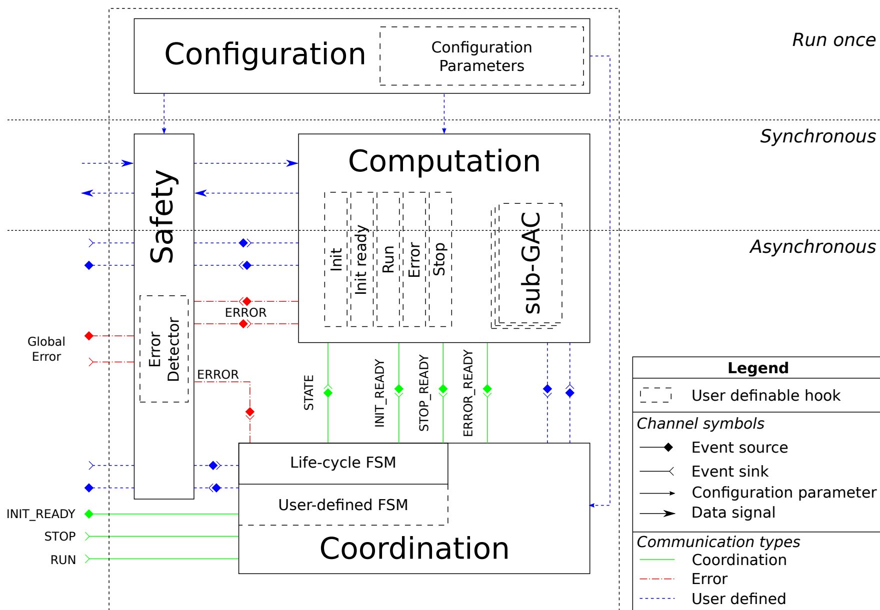


Figure 4.6: Implementation details of the generic architecture component

event channels can be used to send additional user-defined events to the user FSM. Compared to the design overview, a STATE event is added to notify the computation block about life-cycle state changes, so it is able to execute the hook belonging to the active state.

The life-cycle states are represented in the computation, each life-cycle state has its own hook. Changes in the active state are communicated to the computation block, which uses this information to periodically execute the correct life-cycle state hook. The INIT_READY, STOP_READY and ERROR_READY events are used by the computation block to notify the coordination block that the corresponding state has finished. For example, after the Init hook has finished its homing procedures it needs to issue the INIT_READY event.

The configuration block provides a hook to implement the user-defined configuration parameters. These parameters can be used to configure the other hooks, making the specialised GAC employable on a broad(er) range of situations. The coordination block typically reads the parameter values from a configuration file and provides them to the hook to be processed and sends them to the correct locations. As discussed before, it could be modified to accept configuration parameters to configure the GAC at run-time.

Compared to the GAC overview, the available user and error communication signals of the safety block are provided in more detail. The incoming user-defined signals are either forwarded to the coordination or to the computation block. The error-related

events are managed by the safety implementation and are sent when a problem arises that cannot be handled by the safety block alone. The ERROR event channels to and from the computation block are used to notify the sub-GACs, if present, of system-wide error situations. They are connected to the *Global Error* channels of these sub-GACs. The *Error Detector* hook is used to define how the user-defined signals need to be checked.

The GAC implementation provides all kinds of optional support, making the GAC scalable and flexible to be used in a wide range of situations. It does not provide a lot of nice, but complex, features as the Orocos component model though. This can be partially countered by cleverly reusing implemented features in the specialised GAC, as shown in the next section.

4.5 Usage of the Generic Architecture Component

The Production Cell setup (Groothuis et al., 2009), shown in Figure 4.7, is a scaled model of an injection molding machine. It is used in this section to demonstrate how the GAC can be used as a base to design the control software of an actual cyber-physical system.

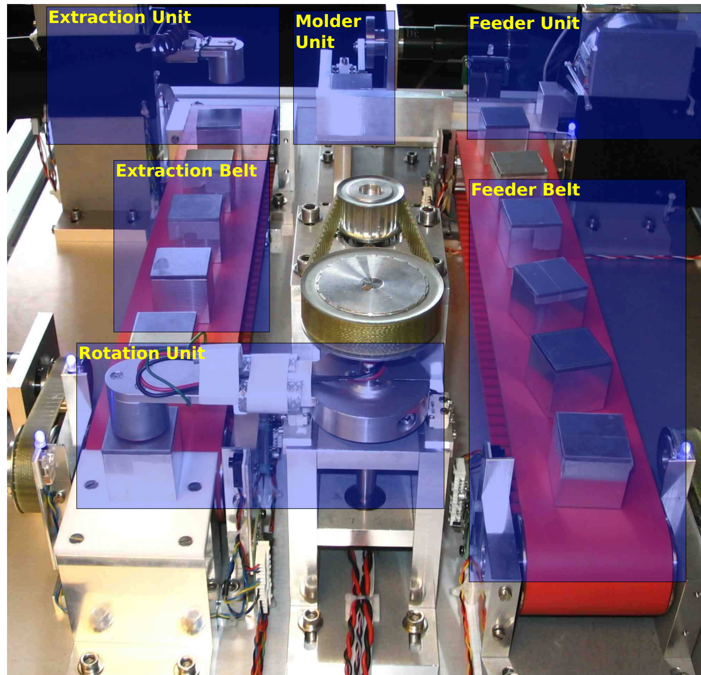
The metallic blocks on the belts represent the materials that are transported through the molding machine. These modelled materials have a piece of iron on their top, so they can be grabbed by the electromagnets of the setup.

The Production Cell consists of 6 separate Production Cell Units (PCUs). The feeder belt transports the materials to the molding area and the extraction belt transports the extracted materials away from the molding area. The molding area consists of three parts: the feeder unit, the molder unit and the extraction unit. The molder unit consists of a molding door, against which the material is pressed by the feeder unit, to simulate the molding process. Afterwards the door opens and the extraction unit grabs the molded material and places it on the extraction belt. For demonstration purposes the rotation unit is used to keep the material flow going. It prevents the materials from falling off the extraction belt, by moving them from the extraction belt onto the feeder belt again, closing the circle.

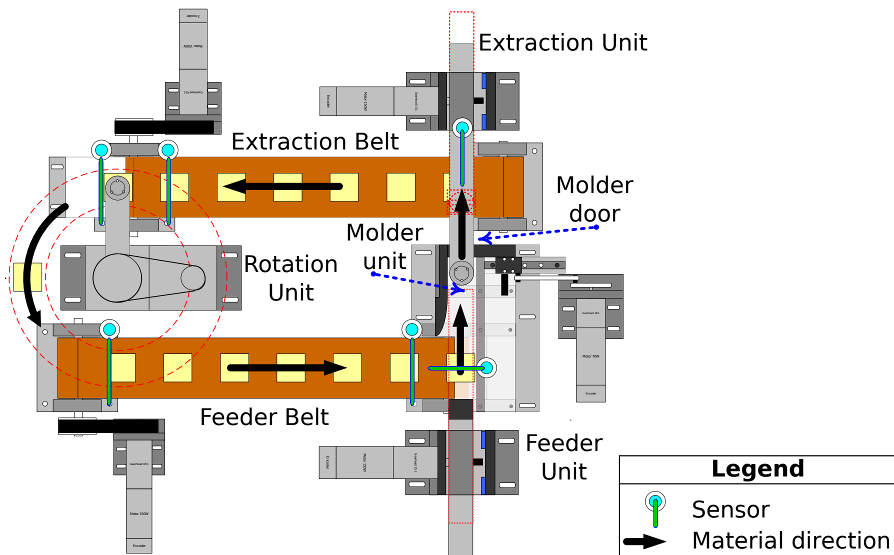
Each PCU has one or two actuators that need to be controlled. They all have a motor that drives the belt, moves the robotic arm or handles opening the molder door. The extraction unit and the rotation unit have a second actuator in the form of an electromagnet, making the PCU able to grab of the piece of material.

Besides the actuators, all PCUs have several of sensors. All motors have an accompanying rotary encoder to measure the rotation of the motor axis. Additionally, each PCU has one or more ‘material sensors’, consisting of optical switches, to detect whether a certain position in the area of the PCU is occupied by a piece of material.

Two neighbouring PCUs need to communicate whether it is possible to move the material to the next PCU or that there is another piece of material in the way.



(a) Photographic overview showing the Production Cell Units



(b) Schematic overview

Figure 4.7: Production Cell setup, a model of a molding machine

4.5.1 PCU GAC Design Considerations

Due to the similar control requirements of the PCUs, it makes sense to design a reusable PCU GAC. As the PCUs do not all have exactly the same amount of sensors and actuators, there is a trade-off between having one generic PCU GAC, or having a number of specialised PCU GACs with a specific amount of sensors and actuators. Besides the obvious advantages of having a single generic PCU GAC, there is a disadvantage of having unused parts when the GAC is used of a PCU with only one sensor and actuator, resulting in unnecessary overhead. On the other hand specialised PCU GACs require more design and maintenance resources.

The GAC PCU *component interface* design is shown in Figure 4.8. Note that this is just one of multiple possibilities for the interface. It has periodic input signals for the encoder and material sensors and periodic output signals for the control values. The GAC has a user-defined event-based input and output. The OUTGOING_FREE event is used to get information whether the next neighbour is accepting a piece of material. The INCOMING_FREE event is issued by the component if it is accepting an incoming piece of material from the previous neighbour.

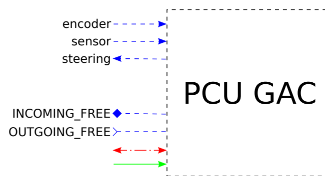


Figure 4.8: GAC interface for a Production Cell Unit

Each PCU control algorithm can be implemented by a PID controller that follows the set-point it received. The motion profile generator periodically provides the set-point from one of the available motion profiles. The motion profile that is active depends on the task of the PCU, for example it might need to start the actuator, keep it running or stop it. The use of a PID and motion profile component is the same as for the examples of Section 4.3.6.

The focus of the actual design implementation of the PCU GAC can vary between a *modelling point of view* or an *execution point of view*. A focus on a pure modelling point of view results in a detailed model of the PCU GAC, consisting of a motion profile GAC and a PID GAC for the computation implementation. Focusing on the execution point of view results in a minimalistic use of the GAC and the computation implementation consists of optimised source code.

Modelling Point of View

The PCU GAC implementation focusing on a modelling point of view is shown in Figure 4.9. It has two sub-GACs which are similar to the ones in Figure 4.4. Although, in this case the motion profile GAC has a collection of predefined motion profiles of which one is chosen depending on the required task of the PCU. The configuration

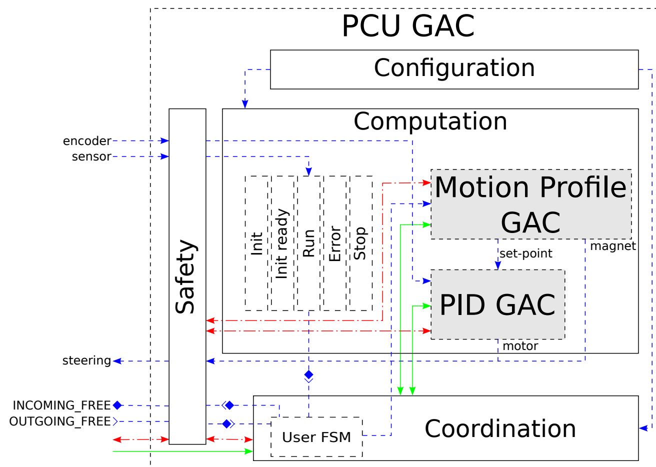


Figure 4.9: PCU GAC implementation from a modelling point of view

and error detector hooks are not included in the figure to keep it somewhat cleaner, but they are in use for their regular tasks.

A user-defined event-based signal is added between the computation and the coordination blocks to notify the coordination block when a piece of material passes a sensor. The event is generated in the Run hook, which uses changes of the sensor input signals to check whether a piece of material is passing the sensor. The event belonging to the sensor that is near the end of the material path of the PCU, is used to notify the user-defined state machine that a piece of material is almost leaving the PCU. If this is the case and the next neighbour accepts a new piece of material, nothing changes. Otherwise, the user-defined state-machine stops moving the material until the next PCU is accepting new materials again to prevent materials collisions. The sensor event representing the situation at the begin of the material path, is used by the user-defined state machine to generate the INCOMING_FREE event.

The motion profile generator GAC periodically provides its set-points using one of the available motion profiles, depending on the active task of the PCU GAC. The user-defined state machine provides the details about the active task, which is determined using the OUTGOING_FREE event and material positions within the GAC, as described above. If the GAC is also controlling an electromagnet, the chosen motion profile also provides the signal to turn the magnet on, before moving the arm, in order to actually pick up the piece of material.

The PID GAC uses the set-point and encoder signals to calculate the motor steering signal to control the actuator (motor) of the PCU. The signal of the electromagnet is not fed to the PID GAC, but directly used as output.

The quality of the design of the PCU GAC implementation is high when looking from a modelling point of view: All concerns are nicely separated, reusability of the sub-GACs is possible and the GAC is modelled in a clear way.

On the other hand, from an execution point of view this implementation has some drawbacks. For example: Both sub-GACs have all their concerns nicely separated, as they have the same implementations as the ones in Figure 4.5. Therefore, the steering signals are checked twice: once in the sub-GAC safety block and once in the PCU GAC safety block. The motion profile simply needs provide the set-points for the selected profile, so it is highly unlikely that this results in any errors, so the set-point signal does not even require checking at all. The motion profile GAC also does not require initialisation, therefore its life-cycle state machine contains some unused states, like the **Init** and **Error** states.

Execution Point of View

The PCU GAC implementation using an execution point of view, is shown in Figure 4.10.

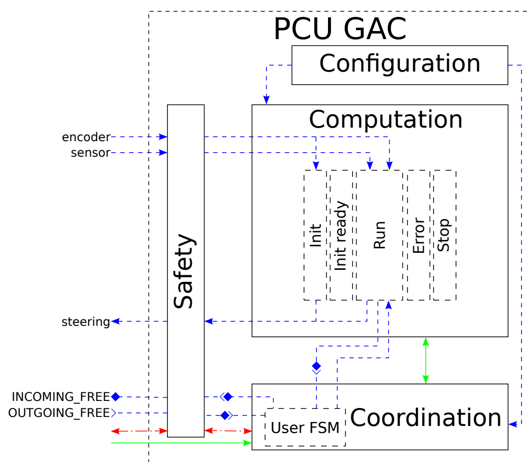


Figure 4.10: PCU GAC implementation from a execution point of view

The main difference between both GAC implementations is that the execution-point-of-view GAC does not have the sub-GACs anymore. Their functionality is merged with the hooks of the computation block, resulting in the loss of the separation of concerns and reusability. For example, the motion profile and PID functionality needs to be remodelled if another GAC requires such functionality as their separate implementations are merged and cannot used separately anymore. They are also not kept up-to-date anymore, for this simple case this is not a big problem, but when the GACs are more complex it is desired to keep them up-to-date with all fixes and improvements. Another disadvantage is that a separation between hard and soft real-time is not possible anymore, as the merged implementations are intertwined.

Looking from an execution point of view, this implementation indeed is more efficient. There is only one safety block, all life-cycle states are used and less internal communication is required, as for example the generated set-points are directly avail-

able by the PID part of the implementation, as both the motion profile and the PID implementation are placed in the Run hook of the PCU GAC.

Design Guidelines

The design guidelines have their main focus on determining the level of detail of the models. This depends on both the tasks that need to be performed for which the model is designed and the real-time requirements of the software.

The level of detail of the implementation is somewhere in between the modelling and execution points of view, depending on the specialised GAC that needs to be created. For example, the relatively simple PCU GAC can be implemented using more of an execution point of view than a modelling point of view. However, in a situation where optimised execution of the component does not play a role and a PID and/or motion profile GAC is available it saves design effort to reuse the GAC to implement the PCU GAC.

This guideline should be kept in mind to choose the level of detail:

- For illustrative or educational purposes the modelling point of view should be used.
- For highly optimised software, e.g. designed for embedded computing platforms, the execution point of view should be used.
- Otherwise it depends on the availability of partial solution for the component implementation, its complexity and the experience of the designer.

In the end, the closer the level of detail is to the modelling point of view the better, as this likely prevents problems at later stages.

Note that it is possible to change the level of detail, e.g. by optimising the design by merging components parts afterwards. Although, it is recommended to only do this using (automated) optimisation tools and only if it is required to prevent resource usage problems on (embedded) computing platforms.

The level of detail depends also on the requirement of real-time guarantees of the component. The execution-point-of-view of the PCU GAC implementation resulted in a complete hard real-time implementation. Due to merging most of its functionality into the hooks, it is not possible anymore to use different real-time guarantees, as the C++ implementation of a hook allows only one type of real-time guarantees. For example, if a component requires hard real-time guarantees for its control law and soft real-time guarantees for administrative purposes, it is not possible to merge these two activities.

4.5.2 Production Cell Architecture Implementation

Besides the 6 PCU GACs, 3 additional components are required for the complete control software implementation that is able to control the Production Cell:

- Supervisory controller to control the life cycle of the 6 PCU GACs.

- User Interface to show the Production Cell status to the user and show notifications when possible.
- Hardware interface to access the sensors and actuators from the software.

The architectural model of the control software, containing all GACs and their composition, is shown in Figure 4.11. The hardware interface is not explicitly shown to keep the figure clear, but it consists of one large I/O handling block to which all sensor and actuator channels are connected to, similar as the one used in Figure 3.4.

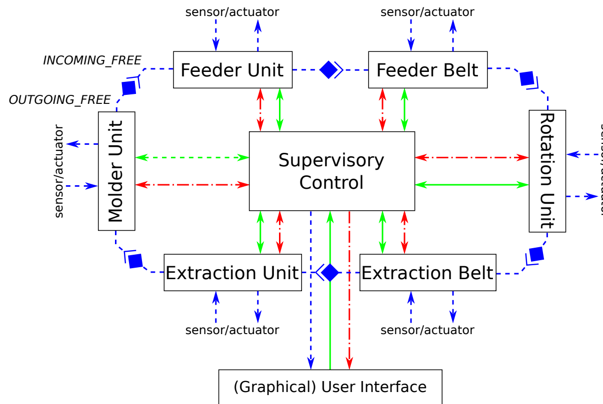


Figure 4.11: Production Cell architecture implementation using GACs

The figure shows a ring-like connection between the 6 PCU GACs composed of the connected INCOMING_FREE and OUTGOING_FREE event channels of the PCU GACs. Furthermore, each PCU GAC is connected to its own set of actuators and sensors, using the hardware interface that is left out.

The supervisory control is connected to the PCUs to send commands to keep their life cycles synchronised. The error signals are used to notify the supervisory control about global errors, so it can gracefully shutdown the system when an unrecoverable problem occurred.

The user interface is connected to the supervisory control. It has a command connection to send commands to the supervisory control GAC to influence the life cycle of the system, for example to start it up. The user-data connection is used to provide information about the state of the system. In this design, there are no user-data connections from the PCU GACs to the supervisory control GAC, so the data sent to the user interface cannot consist of actuator or sensor information. If this is required some additional signals need to be included between the PCU GACs and the supervisory control GAC.

The error signal is used to notify the user interface about unrecoverable errors, so the operator is able to handle them manually. Error handling within the control software is handled locally and globally, as suggested by the template GAC design.

An example of a local problem is when the material is stuck between the molder door and the wall: The safety block detects this as the accompanying encoder is not showing the expected movement of the door. The safety block of the molder unit puts the GAC in the Error state by sending the ERROR event, so it shuts down properly to prevent (further) damage. This is an example of a local problem as the rest of the system can continue to function properly, as it for example still makes sense to keep the extraction belt running to transport processed pieces of material to their storage space, or in this modelled system to the rotation unit.

Due to this error, the feeder unit will automatically come to a halt as well, as the OUTOING_FREE signal of the molder unit GAC is never indicating that a new piece of material is accepted. So a backlog arises at the feeder belt and the Production Cell will eventually come to a complete standstill in a safe way. For this example it is clear that supervisory control GAC does not need to shutdown the other PCU GACs, but in other situations this might be required though.

An example of a global problem is the situation where the extraction arm is struck in the molding area, blocking the open molding door. The molding unit cannot detect this problem, as it does not have sensor for this, but it must leave the door open to prevent damage to the door and extraction arm. The extraction arm is able to detect the problem by comparing its encoder values and the expected values. When these values do not match, the supervisory control needs to be notified so it can shutdown the molding unit. The rest of the system is still able to function properly, although the backlog will occur in this situation as well.

4.6 Discussion and Conclusions

All GAC requirements defined in Section 4.1 are met:

- It provides several methods, like hooks and configuration support, to provide means to reuse both the template GAC and its specialised GACs, Requirement 1.
- It makes use of generalized ports for the communication concern as provided by the CPC methodology, Requirement 2.
- It supports multiple real-time guarantees simultaneously and within a single component, Requirement 3.
- It has intrinsic safety support and mechanisms to handle safety related situations either on a local or a global level, Requirement 4.
- It is possible to design many different types of specialised GACs from the template GAC using the provided hooks, Requirement 5.
- Formal verification is partially possible as the GAC is developed using CSP models, Requirement 6. Unfortunately, the life-cycle FSM is not available in CSP, so complete formal verification is not yet possible.
- It can be used in a hierarchical networks, Requirement 7.

In comparison with the earlier implementation of a generic component (Groothuis et al., 2008), the current GAC design is more flexible and reusable as dictated by Requirement 1. It is usable for a wide range of different types of cyber-physical systems. The specialised GACs are reusable within a single system itself, or between different systems, due to the configuration possibilities.

Compared to the Orocos component, the GAC does provide less features and supposedly is thereby less reusable, as a logical result. This reduction of reusability is counteracted by providing:

- possibilities to add used-defined functionality at multiple locations in the GAC
- support for different real-time guarantees within the GAC
- an intrinsic safety block

These features all increase the (re)usability while keeping the unused functionality low, as they are likely to be used in all specialised GACs. Last but not least, the low resource footprint, due to the less complex design of the GAC, makes it more suitable to use it on embedded computing platforms.

Initial implementations and visual inspections show that the GAC is working as supposed. The Production Cell functions as expected and transports the pieces of material properly through the system. Therefore, it can be concluded that the template GAC is suitable to be used for specialised GACs and these specialised GACs are reusable to implement different parts of the system by only configuring the components to let them behave as required for their specific part of the system. Thus it can be concluded that the GAC indeed is a generic component that can be used to define the system architecture and provide an implementation of the control software of a cyber-physical system.

The design guidelines of Section 4.5.1 provide rules to help determining the level of detail of the implementation. On the one side is the modelling point of view and on the other side the execution point of view. It is also discussed that optimising the design is not recommended unless explicitly required due to the lack of sufficient resources of the computing platform. If model optimisation is still required, it is recommended to use tool support for this task as described and discussed by Bezemer et al. (2009).

Efficient usage of the GAC requires that a model-driven design tool is used to aid the engineer to use the template GAC when designing a specialised GAC. Currently, the template GAC has its hooks that can be used to add used-defined implementations for certain aspects of the specialised GAC. These user-defined aspects require additional data to process, otherwise they are not able to contribute to the overall system. Adding these user-defined aspects manually defeats the purpose of the GAC and the way of working. The MDD tool needs to automate these manual tasks to streamline the design process.

Such a MDD tool is described in Chapter 6, but it does not have integrated support for GACs yet. GAC support needs to provide means to add implementations to the GAC hooks, which are used by the code generation to LUNA source code. Such support

needs to be included in order to be able to actually try the way of working in combination with the GACs. So it can be tested whether their combination is indeed working nicely or adjustments needs either one need to be performed.

5

LUNA Universal Network Architecture

During the discussion of the way of working, it was stated that an execution framework is required to reduce the complexity of model-to-code transformations. Such an execution framework provides the static code forming an execution engine for the model execution in the application. The GAC is modelled using CSP, so the execution engine provided by the framework needs to be a CSP based execution engine.

An execution framework that is focusing on control software for cyber-physical systems requires some additional support. Cyber-physical system software typically requires hard real-time guarantees for its loop controllers in order to obtain correct dynamical behaviour. Modern systems have multiple computing platforms, CPUs and/or cores to distribute the control software, opening the possibilities for more complex control software consisting of more advanced algorithms. The execution framework needs to support these features as well, in order to make fully use of the hardware capabilities.

CTC++ (Orlic and Broenink, 2004) is a CSP based library, providing a hard real-time execution framework for CSP-based applications. Unfortunately, the CTC++ library has become outdated, it is not suitable for multi-core platforms, does not provide a proper hardware abstraction layer and has some other limitations as well. These features are implemented in the core of a framework, adding them in a later stage requires lots of effort, being similar to recreate the complete framework. The LUNA Universal Network Architecture (LUNA) framework is designed to overcome these problems and is built to support the required features from within its core.

Besides providing these platform support features, LUNA is designed while keeping in mind that a new graphical tool suite (TERRA, Chapter 6) is planned to be developed to replace gCSP (Jovanović, 2006). Both the framework and the tool suite should be designed to improve each others functionality by complementing each other.

The next section describes the requirements for LUNA, followed by a discussion about existing frameworks and how they adhere to these requirements. After that the general idea behind LUNA and its implementation are discussed, like threading, the CSP approach, channels and alternative compositions. Next LUNA is compared with the other related CSP frameworks, by showing the results of two timing tests and the com-

parison with a cyber-physical system implementation, in Section 5.4. This chapter ends with conclusions on the features, requirements and the performance of LUNA.

5.1 Requirements

The following list contains the requirements to make LUNA a suitable framework to facilitate the way of working in Chapter 3. The list is structured using the MoSCoW method (Clegg and Barker, 1994).

Requirement 8: *The core functionality of LUNA must be hard real-time*

The core of LUNA, the part that is used for hard real-time processes, needs to be deterministic, so it is possible to guarantee that deadlines are always met and to determine the worst case execution times.

Requirement 8.1: *The framework should provide a layered real-time approach*

Each process should be able to define its real-time level depending on its own requirements.

Requirement 9: *LUNA must provide a hardware abstraction layer*

Such an abstraction layer provides support for core functionalities and makes it target independent. This requirement is needed to provide the '**U**niversal' in LUNA.

Requirement 9.1: *LUNA must support multiple hardware platforms*

Each cyber-physical system uses its own hardware platform with its own hardware functionalities. The framework must provide support to easily add a new platform for a new cyber-physical system that is being developed.

Requirement 9.2: *LUNA must support multiple operating systems*

Each platform might use a different *operating system* (OS), as each operating system has its own advantages and disadvantages. Each operating system defines things, like processes, mutexes or timers, differently. The operating system support must provide means to make these features universally available independent on the underlying operating system.

Requirement 9.3: *LUNA must support multi-thread environments*

As mentioned before, threading is especially useful for complex algorithms, like environment mapping or object recognition. Especially when these algorithms are executed on a multi-core or multi-CPU capable computing platform. Each operating system provides its own threading support, for example POSIX compatible systems have the pthreads library to provide this support. The hardware abstraction layer must provide a single interface for threading support and must provide implementations of this interface for each supported platform.

Requirement 10: *LUNA must provide support for execution engines*

However, LUNA should *not* force the use of the execution engines when the developer does not want or require it. Preferably, LUNA could even provide basic execution-

engine support, which can be (re)used by different engines, like a common interface with a base implementation.

Requirement 10.1: *LUNA must not be dependent on any execution engine*

It needs to be possible to disable the execution engines completely in situations where they are not required. In such situations LUNA is still usable, as it provides the hardware abstraction layer and a lot of other functional components.

Requirement 10.2: *LUNA must implement a CSP execution engine*

As suggested by the way of working, CSP is a good way to develop the architecture of a cyber-physical system. CSP support is also required by the GAC, as it mainly consists of CSP models. So LUNA must provide an execution engine that provides the static code requirements to execute CSP models.

Requirement 11: *LUNA should be scalable*

All kind of cyber-physical systems need be controlled, so LUNA should be flexible enough to support these kinds of systems: From big robotic humanoids with lots of processing resources to small embedded systems with limited computer resources.

Requirement 12: *LUNA should provide debugging and tracing support*

By providing good debugging and tracing functionality, the applications in development can be debugged easily. Additionally, it provides means to detect unexpected behaviour of LUNA in the initial stages of implementing a new component, and thereby helping to improve the quality.

Requirement 12.1: *LUNA should provide real-time logging support*

Normal logging support is not deterministic, as it uses memory allocations and/or disk activity to store the logging information. This is disastrous in a hard real-time execution environment. Real-time logging functionalities could provide logging support without violating the (hard) real-time guarantees.

5.2 Existing Solutions

The CTC++ library meets most requirements, as it was designed as an execution engine for control software. However as mentioned before, it does not have threading support for multi-core target systems, it does not have a complete hardware abstraction layer but lots of platform dependent support is intertwined with its other features. It also has a tight integration with the CSP execution engine, so it is not possible to use the library without being forced to use CSP as well. Therefore, other related frameworks which could replace the CTC++ library are discussed in this section.

A good candidate is the C++CSP2 library (Brown, 2007) as it already has a multi-threaded CSP engine available. Unfortunately it is not suitable for hard real-time applications controlling setups. It actively makes use of exceptions to influence the execution flow, which makes an application non deterministic. Exceptions are checked at run-time, by the C++ run-time engine. Because the C++ run-time engine has no notion of custom context switches, exceptions are considered to be unsafe for usage in hard

real-time setups. Additionally, exceptions cannot be implemented in a deterministic manner, as they might destroy the timing guarantees of the application. Exceptions in normal control flow also do not provide priorities which could be set for processes or groups of processes. This is essential to have hard, soft and non real-time layers in a design in order to meet the scheduled deadlines of control loops.

Since Java is not hard real-time, for example due to the garbage collector, we did not look into the Java based libraries, like JCSP (Welch et al., 2007). Although, there is a new Java virtual machine, called JamaicaVM (aicas, 2012), which claims to be hard real-time and to support multi-core targets. Nonetheless, JCSP was designed without hard real-time constraints in mind and it is highly improbable that it is hard real-time suitable.

There are also non-CSP-based frameworks which might be suitable to include a custom CSP layer. As described in Section 4.2 the Orocos framework is focused around its own component model, so lots of functionality of the Orocos framework is for supporting its model. The Orocos components and their execution engine are too resource heavy to function as a CSP process, so there is not much support for a CSP layer or execution engine in the Orocos framework besides the OS abstraction layer. Hence, it does not make much sense to use Orocos as a basis to implement the CSP execution engine.

ROS (Quigley et al., 2009) is developed as a message passing system between hardware and/or components. This could be useful for channels between CSP processes, but the topics and their subscriptions are too advanced for simple communication between two CSP processes. Furthermore, ROS does not guarantee that the messages arrive on time, so it is not able to fulfill real-time requirements.

Both also do not provide possibilities for formal verification methods, therefore it is impossible to confirm that a complex application, using one of these frameworks, is deadlock or livelock free according to Dijkstra (1972).

Other frameworks that do not provide real-time support are, RoboFrame, YARP, URBI and CLARAty as discussed by Lootsma (2008). Due to the lack of real-time support these frameworks have not be investigated further.

None of the discussed frameworks fulfil the requirements and is thereby suitable for the way of working and the GAC. Therefore the LUNA framework is developed, although keeping useful aspects of the other framework in mind. The rest of this chapter described the LUNA architecture and some of its design choices and a comparison with other related frameworks is provided afterwards.

5.3 LUNA Architecture

LUNA is developed in a modular way, it has components which can be disabled if they are not required. Furthermore, a component is able to have dependencies on other components, meaning that the implementation other components is required for the implementation of the component. Figure 5.1 shows the overview of the LUNA

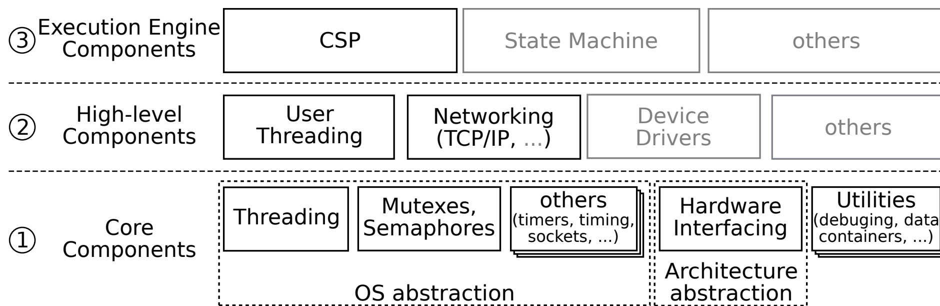


Figure 5.1: Overview of the LUNA architecture.

components, grouped by different levels. The grey components are not implemented yet, but are planned for future releases.

The *Core Components* ① level contains basic components, mostly consisting of platform supporting components, providing a generic interface for the platform specific features. *OS abstraction components* provide interfaces and implementation for operating-system specific features, like threading, mutexes, timers and timing. At the moment of writing QNX, Linux and partial Xenomai support is available. The *architecture abstraction components* provide support for features that are specific for an architecture (or hardware platform), like the support for (digital) input and output (I/O) possibilities. The other components must make use of these core components to make use of platform specific features without knowledge of the actual chosen platform or operating system. Another group of core components are not platform or operating system abstraction components, but provide low level features like debugging, generic interfaces and data containers. These are called *Utility components*.

The next level contains the *High-level Components* ②. These components are platform independent as they make use of the functionality of the core components. For example the Networking component, providing networking functionality and protocols, uses the socket component as platform-dependent glue and provides (high-level) protocol components.

The *Execution Engine Components* ③ provide the execution engine support. These components are used to determine the flow of the application. For example the CSP execution engine component provides constructs to have a CSP-based execution flow. The CSP component uses the core components for threading or mutex support. It uses high-level components like user threading to reduce the number of OS threads to improve the execution speed, and the networking component to implement rendezvous channels over a network.

Components can be enabled or disabled in the framework, depending on the type of application one would like to develop. The unused component can be turned off in order to save resources and use them in other aspects of the application, as is required by Requirement 11. Since building LUNA is complex due to the component based approach and the variety of supported platforms, a dedicated build system is provided.

The LUNA build system is heavily based on the OpenWrt build system (OpenWrt developer group, 2012; Fainelli, 2008).

OpenWrt provides custom firmware, consisting of the Linux kernel, operating system tools and applications, build from scratch for all kinds of routers. The tools and applications can be enabled by the users, depending on their needs. The supported routers have all kinds of different hardware, requiring corresponding driver support. Cross-compilation techniques are required to build the firmware, as it is not possible to build it on the routers themselves. This all combined, makes building the firmware a complex task, which is taken care of by the OpenWrt build system. The mentioned features of OpenWrt match with the needs of LUNA; it also requires support for different hardware, the modularity of components and a final firmware-like result.

The first operating system that is supported by LUNA is QNX (QNX Software Systems, 2012). This is a real-time micro-kernel OS and natively supports hard real-time and rendezvous operating system communication. Its basic set of features relieved the development load for the initial implementation of LUNA as it covered several of the requirements.

Additionally, QNX is *Portable Operating System Interface* (POSIX) compliant. This is a collection of standards to maintain compatibility between operating systems. Therefore, the QNX implementation for LUNA provides POSIX support to LUNA. A lot of other operating systems are also fully or mostly POSIX compliant, so adding support for these operating systems does not require much effort. Linux is such an operating system, it reuses most of the POSIX implementation of the QNX platform.

In a later stage of creating the LUNA implementation, the QNX rendezvous operating system communication mechanism is replaced with a custom code implementation as it is not possible to use QNX channels between two user threads that are running on the same OS thread. More information about QNX channels and their limitations is provided in Section 5.3.3.

5.3.1 Threading Implementation

LUNA supports *OS threads* (also called kernel threads) and *user threads* to be able to make optimal use of multi-core environments. OS threads are resource-heavy, but are able to run on different cores and CPUs. User threads on the other hand have a low resources usage, but must use an OS thread as their execution container. A big advantage of using OS threads is the preemptive capabilities of these type of threads: Their execution can be forcefully paused anywhere during their execution, for example when a thread with a higher priority becomes available. User threads can only be paused at specified moments, if such a moment is not reached, for example due to complex algorithm calculations, other user threads on the same OS thread will not get activated. Reducing the OS thread resource usage by combining them with multiple non-preemptive user threads, results in a hybrid solution. This adds the multi-core advantages of the OS threads to the user threads while keeping the resource usage reasonably low.

As the term already implies, the OS threads are provided and maintained by the op-

erating system. For example, the QNX implementation uses the POSIX thread implementation and for Windows LUNA would use the Windows Threads. Due to the different implementations, the behaviour of an OS thread might not be the exactly the same for each platform.

The user threads are implemented and managed by LUNA, using the same principles as Brown (2007) and Decho Corp. (2012). The main difference is that the LUNA user threads are not run-time portable to other OS threads, i.e. can be moved to another core or CPU at run-time. There is no need for it and this will break hard real-time constraints.

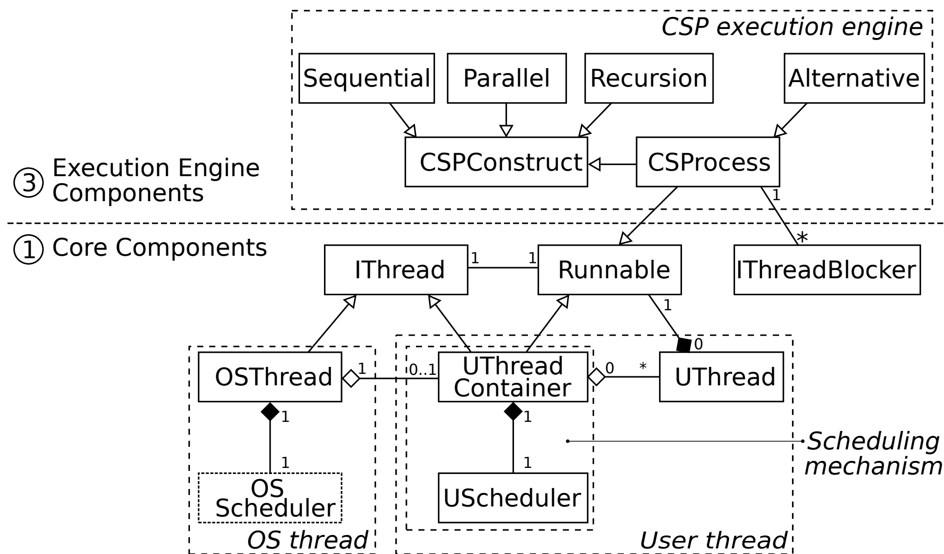


Figure 5.2: UML diagram of threads and their related parts.

The architecture of the *threading implementation* of LUNA is shown in Figure 5.2. The figure is simplified in order to keep it clear, fully detailed inheritance and usability figures are available in the generated LUNA documentation. Two of the components levels of Figure 5.1 are visible, showing the *separation of concerns* between the threading component implementation and the *CSP execution engine* implementation.

UThreadContainer (UTC) and OSThread are two of the available thread types, both implement the IThread interface. This IThread interface requires a Runnable, which acts as a container to hold the actual code which will be executed on the thread. The CSP functionality, described in more detail in the next section, makes use of the Runnable to provide the code for the actual CSP implementation.

To make the earlier mentioned hybrid solution work, each OS thread needs its own scheduler to schedule the user threads. This scheduling mechanism is divided into two objects:

1. the UTC which handles the actual context switching in order to activate or stop a user thread.
2. the UScheduler which contains the ready and blocked queues and decides which user thread is the next to become active.

The UTC also contains a list of UThreads, which are the objects that are contained by the UTC. The UThread object contains the context of a user thread: the stack, its size and other related information. Besides this context relation data, it also has a Runnable providing the code that is executed on the user thread.

The UTC implements the Runnable interface also, so it can be executed on an OS thread. When the UTC threading mechanism starts, it switches to the first user thread as a kickstart for the whole process. When the user thread is finished, yields or is explicitly blocked, the UTC code switches to the next user thread which is ready for execution. Due to this decision, the scheduling mechanism is not running on a separate thread, but makes use of the original OS thread, in between the execution of two user threads.

A separation-of-concerns approach is taken between the CSP implementation and the threading support they require to be executed on. The CSP processes are indifferent whether the underlying thread is an OS thread or a user thread, which is a major advantage when running on multi-core targets. This approach can be taken a step further in a distributed CSP environment where processes are activated on different nodes. This will also facilitate deployment, seen from a supervisory control node. Due to this separation, it is also possible to easily implement other execution engines.

The figure shows that the Sequential, Parallel and Recursion processes are not inheriting from CSProcess but from CSPConstruct. The CSPConstruct interface defines the activate, done and exit functions. CSProcess can be seen as a Runnable container for CSP: It defines the actual run functionality and context blocking mechanisms. Letting the processes inherit from CSPConstruct is an optimisation: This way they do not require context-switches because their functionality is placed in the activate and done functions, which is executed in the context of its parent respectively child threads. The Alternative implementation still is a CSProcess, because it might need to wait on one of its guards to become ready and therefore needs the context blocking functionality of the CSProcess.

During tests, the number of threads was increased to 10,000 without any problems. All threads got created initially and they performed their task: increase a number and print it. After executing its task, each thread was properly shutdown.

5.3.2 LUNA CSP

Since LUNA is component based, it is possible to add another layer on top of the threading support. Such a layer is the support layer for execution engines, on which the CSP execution engine is available. As shown in the previous section, it is completely separated from the threading model, so it will run on any platform that has an implementation for the threading component.

Figure 5.3 shows the execution flow of three CSProcess components, being part of this greater application:

```
P = Q || R || S
Q = T; U
```

Process **P** is a parallel process and has some child processes, **Q**, **R** and **S**. Process **Q** is a sequential process and has child processes **T** and **U**. Process **T** is one of these child processes and it does not have any child processes of its own. A graphical representation of this application is shown at the left of Figure 5.4.

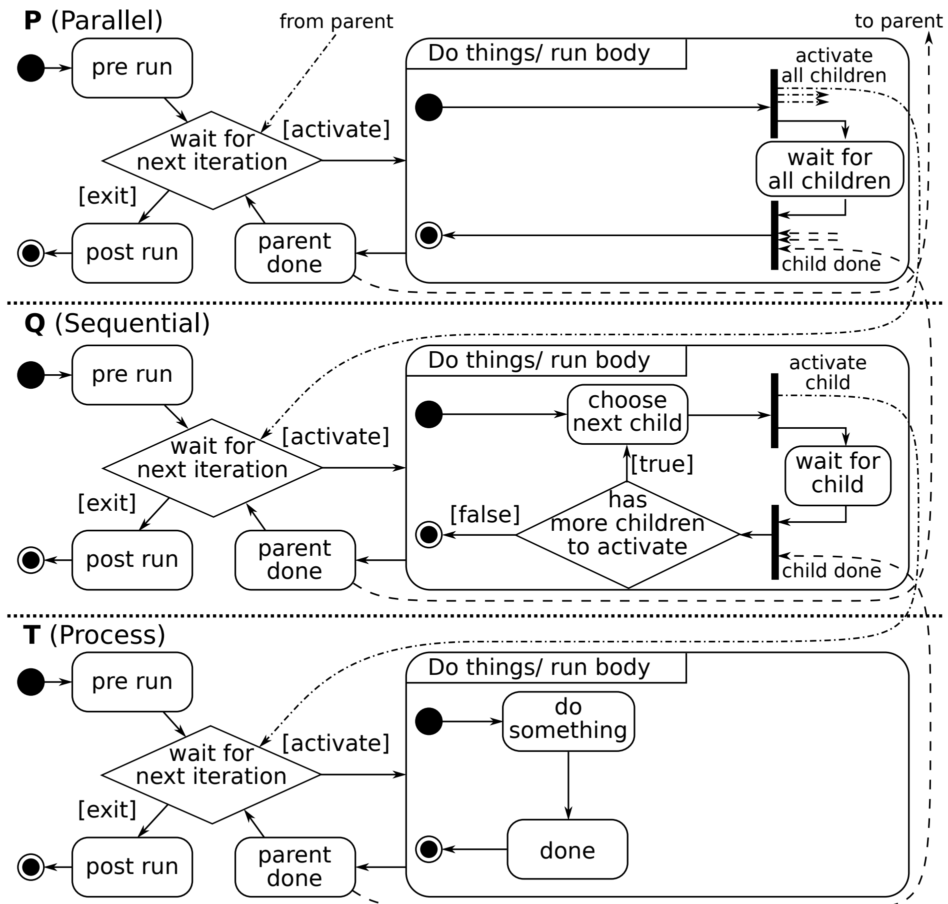


Figure 5.3: Flow diagram showing the conceptual execution flow of a CSProcess.

First, the *pre run* of all processes is executed, this can be used to initialize the process just before running the actual semantics of the CSProcess. Next the processes are waiting in *wait for next iteration* until they are allowed to start their *run body*. After all processes have executed their *pre run* the application itself is really started, so the *pre run*

does not have to be deterministic yet. The *post run* of each process is executed, when the process is shutdown, normally when the application itself is shutdown. It gives the processes a chance to clean up the things they initialized in their *pre run*.

In this example, **P** will start when it is activated by its parent. Due to the parallel nature of the process, all children are activated at once. The process will wait until all of its children are finished before signalling the parent that the process is finished. Signalling the parent that a process is finished is done by the *parent done* block.

Process **Q** is one of the processes that is activated by **P**. **Q** will activate only its first child process and waits for it until it is finished, as **Q** is a sequential process. If there are more children available, the next one is activated and so on. **T** is just a simple code blob which needs to be executed. At some point in time it is activated by **Q**, it executes its code and sends signal back to **Q** that it is finished. Same goes for **Q**, when all its child processes are finished, it sends back a signal to **P**, telling it is finished.

Due to this behaviour, the CSP constructs are implemented decentralised by the CSProcesses, instead of implemented by a central scheduler. This results in a simple generic scheduling mechanism, without any knowledge of the CSP constructs. Unlike CTC++, which has a scheduler implemented that has knowledge of all CSP constructs in order to implement them and run the processes in the correct order.

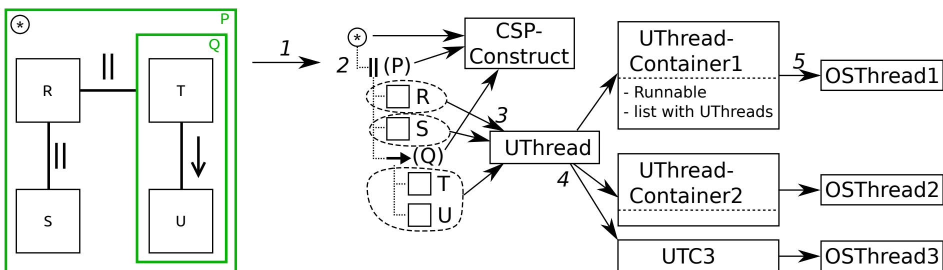


Figure 5.4: Steps to map a CSP model onto OS threads.

CSP processes are able to execute on either OS or user threads, so the designer, or probably the code generation tool, needs to map the threads on the user and OS threads. Figure 5.4 shows the required steps to map a CSP model to OS threads. The example application used earlier is depicted at the left of the figure. Its compositional hierarchy in a tree form is shown in the middle part of the figure. The MDD tool has to map the processes onto a mix of OS and user threads using the compositional information.

The following steps are required to map the CSP model onto OS threads, as depicted by the figure:

1. The model tree needs to be extracted from the model. This model-tree contains the compositional relations between all processes and is used to iterate over all processes during the code generation phase.
2. The user (or the modelling tool) needs to group processes that require to be

placed in the same OS thread. Example criteria for could be grouping processes that heavily rely on communication to each other or it could try to balance the execution load per thread.

3. Each process is mapped to a UThread object. Except for the compositional processes mentioned in the previous section, they are mapped onto CSPConstructs.
4. Groups of UThreads are put in a UThreadContainer (UTC).
5. Each UTC is mapped to an OS thread, so the groups of processes can actually run in parallel and have preemption capabilities.

It is clear that making good groups of processes will influence the efficiency of the application, so using an automated tool is recommended (Bezemer et al., 2009).

5.3.3 Channels

As mentioned in the chapter introduction, one of the initial reasons for supporting QNX was the availability of native rendezvous communication support between QNX threads. This indeed made it easy to implement channels for the OS threads, but unfortunately it is *not* for the user threads. Two distinct situations of using channels are shown by Figure 5.5.

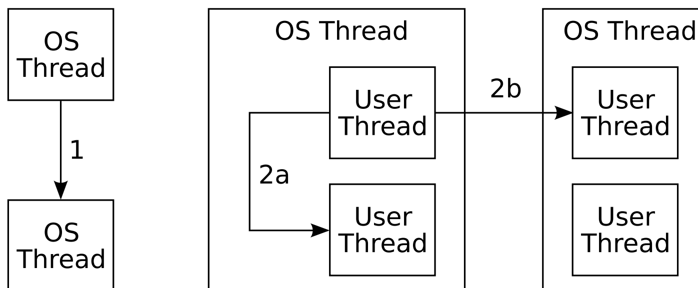


Figure 5.5: Overview of the different channel situations.

Channel 1 is a channel between two OS threads. The QNX rendezvous mechanism can be used for this channel without problems. Channels 2a and 2b are communication channels between two user threads.

If in situation 2a one user thread wants to communicate over a rendezvous channel with the other user thread and the other side is not ready, QNX channel blocks the thread that wants to communicate. QNX does not know about the LUNA implemented scheduler and its user threads, so the complete OS thread gets blocked. The other user thread which also just got blocked, along with the complete OS thread, never becomes ready anymore and a deadlock occurs. In situation 2b, the OS thread at the right of the figure could be blocked, resulting in the user thread at the bottom right getting blocked as well. This is undesired behaviour, which optionally might even lead to deadlocks in a worst-case scenario.

Communication between user threads on the *same* OS thread can never make use of the QNX rendezvous channels and for communication between user threads on two different OS threads it is undesired. This makes the choice to initially support QNX becomes less strong. An exception could be made for OS threads with one user thread, but such situations are undesired since it is more efficient to directly run code on the OS thread without the user thread, resulting in the channel becoming of type 1 again.

Note that guarded channels are also not supported by QNX, so for this type of channels a custom implementation is also required. Considering all limitations of the QNX channels, it is decided to only use custom-built channels in LUNA. This keeps things simple, as only one channel type needs to be supported.

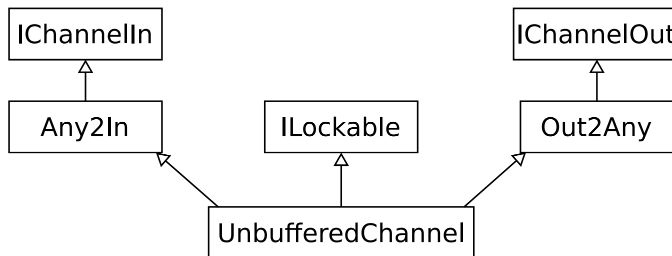


Figure 5.6: Diagram showing the channel architecture.

There are multiple channel configurations, it can be *buffered* or *unbuffered* (*rendezvous*), it can have one or many input connections and/or it can have one or many output connections. Figure 5.6 shows the architecture of an unbuffered many-to-many channel implementation. The UnbufferedChannel implementation is interchangeable with the BufferedChannel variant. The same goes for the Any2In and One2In implementations, and the Out2Any and Out2One implementations. Due to this approach, the actual channel configuration is hidden from the processes that use the channel for their communication. Therefore the processes are able to use the same interacting method for all channels.

Listing 5.1 shows the pseudocode for writing on a channel. The ILockable interface is used to gain exclusive access to the channel, in order to make it *thread safe*. Basically, there are two options: Either there is a reader (or buffer) waiting (ready) to communicate or not. If the reader is already waiting, the data transfer is performed and the reader is unblocked so it can be scheduled again by its scheduler when possible. In the situation that the reader is not available, the writer needs to be added to the *ready_list* of the channel, so the channel knows about the writers which are waiting to communicate. This list is ordered on process priority. And at last, the writer needs to be blocked until a reader is present. A similar set of actions is used for reading the channel, but the writers are replaced by the readers and vice verse.

The *findReadyReaderOrBuffer()* method checks if the availability of buffered data. If this is not the case, it calls the *findReadyReader()* method to search for a reader which is ready. The *isReaderReady()* and *findReadyReader()* methods are implemented by the Out2Any block or by a similar block that is used. Depending on the input type of the

```
write() {
  ILockable.lock()
  if (isReaderReady()) {
    IReader reader = findReadyReaderOrBuffer()
    transfer(writer, reader)
    reader.unblockContext()
    ILockable.unlock()
  } else {
    setWriterReady(writer)
    ready_list.add(writer)
    writer.blockContext(ILockable)
  }
}
```

Listing 5.1: Pseudocode showing the channel behaviour for a write action.

channel, the implementation is quite simple when there is only one reader allowed on the channel or more complex when multiple readers are allowed. The *transfer()* method is implemented by the (Un)bufferedChannel and therefore is able to read from a buffer or from an actual reader depending on the channel type.

LUNA supports communication between two user threads on the same OS thread by a custom developed rendezvous mechanism. When a thread tries to communicate over a channel and the other side is not ready, it gets blocked using the IThreadBlocker (see Figure 5.2). By using the IThreadBlocker interface, the thread type does not matter since the implementation of this interface is dependent on the thread type. For user threads, the scheduler puts the current thread on the blocked queue and activates a context-switch to another user thread which is ready. This way the OS thread is still running and the user thread is blocked till the channel becomes ready and the scheduler activates it. And for OS threads, it uses a semaphore to completely block the OS thread until the channel is ready.

5.3.4 Alternative

The Alternative is a compositional relation type between two or more processes. It makes sure that exactly one of its child processes can become active, its other child processes are skipped. Each child process provides a guard to evaluate whether it can becomes active. The first child process that is allowed to become active is chosen by the Alternative. Until then the Alternative is blocking.

As mentioned earlier, the Alternative implementation is based on a CSProcess, because it might need to wait on its child guards to become ready. This requires the context blocking functionality of the CSProcess. The compositional relation types 'borrow' the context of the current, active process to evaluate which child needs to be activated next, for optimisation reasons. Because of the Alternative needs the context blocking functionality, it cannot just borrow a context and block it. Instead it needs

its own context and therefore cannot get optimised in the same way as the other types are.

The Alternative architecture is shown in Figure 5.7 for a guarded Process, in this case a GuardedReader. A the guard of a GuardedReader checks whether communication over the channel can take place, i.e. the Writer at the other end of the channel needs to be activated and waiting (blocked) to communicate. A similar structure is used for the GuardedWriter, but not shown in the figure.

The Alternative is a CSProcess itself and it also has a list of other CSProcesses which should have their guard object set. The Alternative process uses this list when it is activated in order to try to find a process which meets the conditions of its guard, i.e. the process is allowed to get activated. Note that the guarded reader and writer processes implement the IGuard interface and have themselves set as their Guard objects. This is not necessarily required, a separate object is allowed to function as a guard as well.

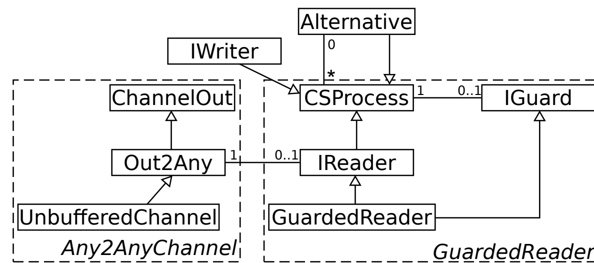


Figure 5.7: Diagram showing the relations for the Alternative architecture.

In the case of channel communication, the Alternative first checks if one of its readers or writers able to perform the communication without blocking. If this is the case, the Alternative makes sure the ability to communicate becomes guaranteed. Next, it performs the communication itself. The Alternative implements this sophisticated protocol in order to make sure the communication is guaranteed, even when different threads are part of the communication or some of the processes on the channel are not guarded.

First a communication scenario is discussed, where a guarded reader gains access on a channel, but gets blocked when it should actually perform the communication. This scenario is shown in the sequence diagram of Figure 5.8. The incoming and outgoing arrows from/to the edges of the figure indicate scheduling activities. Some of the objects of Figure 5.7 are grouped by the dashed boxes, they are shown in Figure 5.8 as a single object to keep the figure more clear.

Assume there is an any-to-any channel, which has a writer waiting to communicate (*1* in the figure). Next the Alternative is scheduled (*2*) and it checks if the GuardedReader is ready for communication, amongst the other of its guarded child processes. The GuardedReader is only ready if there is a writer or buffer waiting to communicate, so it checks with the channel. When the GuardedReader indeed is ready, it gets activated by

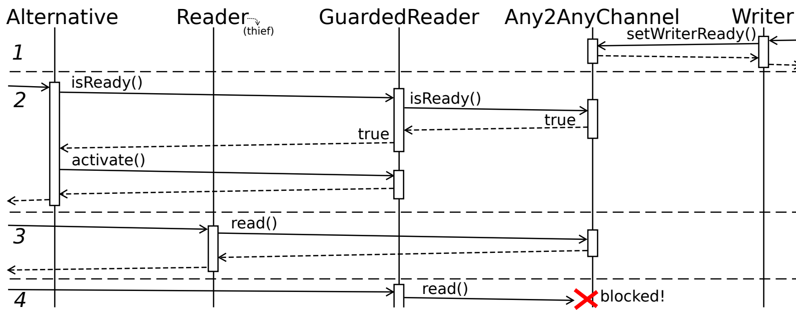


Figure 5.8: Sequence diagram showing a scenario were a guarded reader incorrectly blocks.

the Alternative so it can be scheduled to actually perform the communication. After this the Alternative is finished and a context switch takes place.

Unfortunately before the GuardedReader is scheduled, another Reader (thief) is scheduled and wants to communicate over the channel (*3*). As there is a writer present, communication between the Writer and the (thieving) Reader takes place.

Later, the GuardedReader gets scheduled and tries to perform the communication (*4*), but Writer is not available anymore. As a result the GuardedReader get blocked, even though it gained access to the channel through the Alternative. This behaviour is incorrect according to the CSP algebra, as the Alternative process can only finish when one of its guarded child processes gets activated and is able to perform its guarded task.

The correct and more complex protocol to prevent this erroneous behaviour is shown in Figure 5.9. In order to be able to describe the protocol completely, the scenario contains a channel of which both ends are connected to a guarded process, making it even more complex.

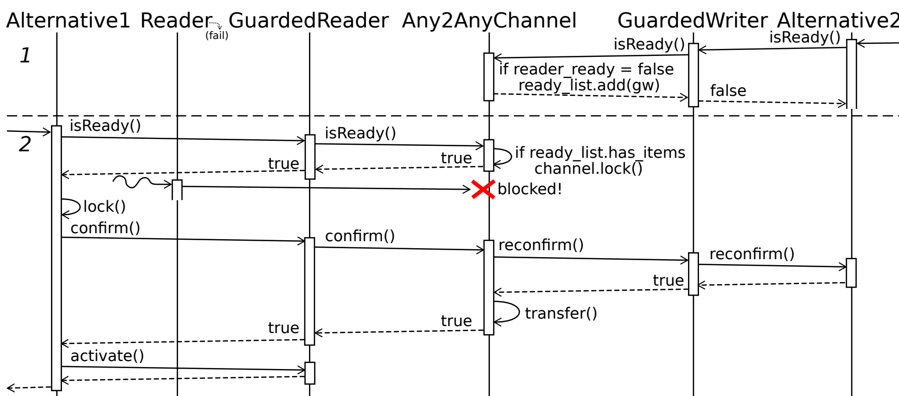


Figure 5.9: Sequence diagram showing the correct scenario.

Just as in the previous example, the writer registers at the channel, telling that it is ready to write data (*1*). There is no reader available yet, so the writer is put in the *ready_list* and gets `false` back as result. Since Alternate2 did not yet find a process that could be activated, it gets blocked and will try to find a suitable process next time it gets scheduled again. This is not interesting for the current scenario and is thus further ignored.

When Alternative1 gets scheduled, it checks whether the GuardedReader is ready or not (*2*). Since the *ready_list* has items on it, the channel is ready for communication. The channel gets locked, to prevent that other parties are interfering with the communication protocol.

Due to this lock the Reader fails when it tries to read from the channel, as a result it gets blocked. This is in contrast with the previous scenario, as Reader was able to read and steal the promised data. Note that the only possibility where the failing Reader can get scheduled, is when it is placed on another OS thread than Alternative1. This is due to the preemptive capabilities of the OS threads.

The rest of the scenario makes sure that the communication between both guarded processes takes place. When the *isReady()* request has a positive result, indicating that the channel is locked, Alternative1 continues the communication. It checks whether another of its guarded processes has been activated in the meantime, used the same *reconfirm()* request that is described next. If this is the case, Alternative1 is finished as one of its processes got activated and it needs to clean up. If this is not the case, it *lock()* itself, preventing other guards taking over the current communication by starting a new communication sequence.

Before the actual communication takes place, Alternative1 needs to check whether the GuardedWriter is still ready to write. This is again required due to the preemptive capabilities of OS threads, as it might be possible that Alternate2 found another of its guarded processes to activate and the GuardedWriter is not available anymore. Alternative1 does not have a direct link to Alternative2, so the *confirm()* request is used to forward this check to the channel. The channel forwards the check to Alternate2 with the *reconfirm()* method via the GuardedWriter.

If the result is positive, the actual communication (data transfer) can take place. In this scenario, the channel directly performs the *transfer* of data. This is not necessary, but is more efficient as it reduces a number of context-switches. A positive response on the *reconfirm()* request by Alternative2 results in the communication, so it can activate GuardedWriter immediately and finish its execution without getting scheduled. The same goes for Alternative1, as the communication already took place it can finish without waiting until GuardedReader gets scheduled and finishes again.

After the data transfer took place, Alternative1 revokes the *isReady()* requests of its other guarded processes, since a process was chosen. For this scenario it is probably unnecessary to activate GuardedReader, since the data transfer is completed already, but for another (non reader/writer) guarded process it is necessary to actually execute its run body. Although, the GuardedReader might be used to activate a chain of other processes, when it is the first process of a Sequential group of processes, and also does

need to get activated. So to keep things simple, Alternative1 activates GuardedReader and finishes.

The described alternative sequence of Figure 5.9 has been tested for difference use cases. Although it is not formally proven, it is believed that this implementation will satisfy the CSP requirements of the alternative construction.

5.4 Results

This section shows some of the results of the tests performed with the LUNA framework. The tests compare LUNA with other CSP frameworks, to see how the LUNA implementation performs.

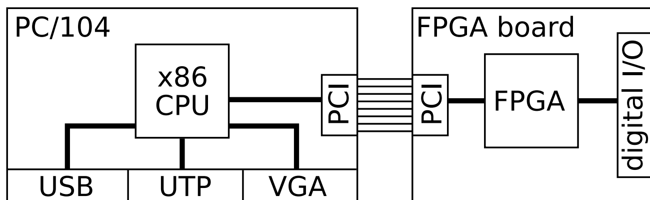


Figure 5.10: Overview of the used test setup.

All tests in this section are performed on an embedded PC/104 platform with 600 MHz x86 CPU as shown in Figure 5.10. It is equipped with an FPGA based digital I/O board to connect it with actual hardware when required for the test. The QNX results are obtained using QNX version 6.4.1 and the executable is built with qcc version 4.3.3. The other results are obtained using Linux version 2.6.23.17 patched with RTAI version 3.6 and gcc version 4.1.2. The same compiler optimisation flag `-O2` is used for both targets.

5.4.1 Context-Switch Speed

A context-switch speed test is performed to get an idea of the efficiency of the LUNA threading architecture and implementation. A test application is developed to measure the context-switching speed, consisting of two threads switching 10,000 times. The execution times were measured and the average time for a single context switch is calculated, as shown in Table 5.1.

Table 5.1: Context-switch speeds for different platforms.

Platform	OS thread (μs)	User thread(μs)
CTC++ (original)	-	4.275
C++CSP2	3.224	3.960
CTC++ QNX	3.213	-
LUNA QNX	3.226	1.569

The *CTC++ (original)* row shows the test results of the original CTC++ library compiled for QNX. It is not a complete QNX implementation, only the required parts for the test are changed to make them work:

- The Stack Pointer (SP) was changed to use the correct field for QNX.
- The `_setjmp`/`_longjmp` implementation used when switching to another user thread. Linux does not save a so-called signal mask by default when executing `setjmp` and `longjmp` for a context switch. QNX does and this additional activity considerably slows down the context switches. Therefore, the '_' versions of `setjmp` and `longjmp` are used for the QNX conversion, which do not store the signal mask.
- The compiler and its flags in order to use the QNX variants and thus making the comparison more acurate.
- The inclusions of the default Linux headers are replaced with their QNX counterparts.
- Some platform-dependent code did not compile and is not required to be able to run the tests, so it was removed.

Similar modifications are made to the C++CSP2 library to add QNX support. The SP modification is not required for C++CSP2. Furthermore as mentioned, the `_setjmp`/`_longjmp` are used for the quick conversion to QNX. The library already makes use of `_longjmp`, but it does not make no use of `_setjmp`. This might indicate that the author of the C++CSP2 library knew about this difference and intended different behaviour. The QNX implementation for the C++CSP2 library is also not complete and only the converted parts are tested.

CTC++ QNX (Veldhuijzen, 2009) is an initial attempt to recreate the CTC++ library for QNX. It was not completely finished, but all parts needed for the commstime benchmark are available, so context switching is also available.

LUNA QNX is the new LUNA framework compiled with the QNX platform support enabled. For other platforms the results will be different, but the same goes for the other libraries as well.

The *OS thread* column shows the time it takes to switch between two OS threads. The *user thread* column shows the time it takes to switch between two user threads placed on the same OS thread.

For LUNA it is clear that the OS thread context switches are slower than the user thread context switches, which is expected and the reason for the availability of user threads. All three OS thread implementations almost directly invoke the OS scheduler and therefore have roughly the same context-switch times.

A surprising result is found for C++CSP2: The OS thread context-switch time is similar with the user thread time. The user threads are switched by the custom scheduler, which seems to contain a lot overhead, probably for the CSP implementation. Expected behaviour is found in the next test, when CSP constructs are executed. In this test

the custom scheduler gets invoked for the OS threads as well, resulting in an increase of OS context switch time. In this situation the user threads become much faster than the OS threads, as expected.

The context-switch time for the LUNA user threads shows that the LUNA context switches are much faster compared to the others. The LUNA scheduler has a simple design and implementation, as the actual CSP constructs are in the CSProcess objects themselves. This approach pays off when purely looking at context-switch speeds. The next section performs a test that actually runs CSP constructs, showing whether it also pays off for such a situation as well.

5.4.2 Commstime Benchmark

The commstime benchmark, shown in Figure 5.11, is implemented to get an better idea of the scheduling overhead. This benchmark passes an imaginary token along a circular chain of processes. The Prefix process starts the sequence by creating the token and passing it to the Delta process. The Delta process passes it to the Successor process and to complete the circle it get passed to the Prefix process. The Delta process also signals the TimeAnalysis process when it receives the token, making it possible for TimeAnalysis to measure the time it took to pass the token around. The difference between this benchmark and the context-switch speed test, is that in this situation a scheduler is required to activate the correct CSP process depending on the position of the token.

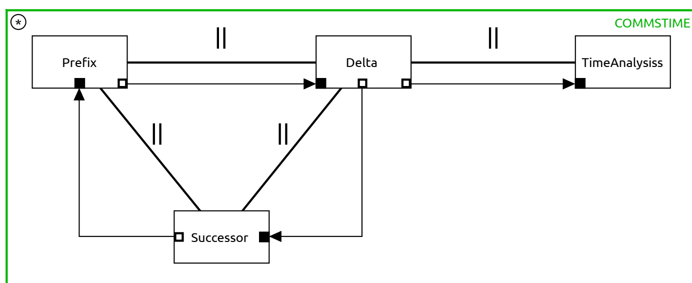


Figure 5.11: Model of the commstime benchmark.

Table 5.2 shows the cycle times for the commstime benchmark for different libraries. LUNA QNX has two values: the first is for the LUNA channel implementation and the second value for the QNX channel implementation. Note that the QNX channel test has been done using a proof of concept implementation, as this type of channel has not been included in LUNA after all. It is remarkable that the QNX channels are slower than the LUNA channels. This is probably due to the fact the QNX channels always have an any-to-any configuration and the used LUNA channels one-to-one, resulting in more overhead for the QNX channels. Again, a reason not to include the QNX channels in LUNA, since it does not even provide more efficient channels compared to the LUNA channel implementation, which was expected on beforehand. The amount of context-switches of OS threads is unknown, since the actual thread switching is

handled by the OS scheduler having preemption capabilities and there is no interface to retrieve this information.

Table 5.2: Commtime results of the frameworks/libraries for their supported thread types.

Platform	Thread type	Cycle time (μs)	# Context-switches	# Threads
CTC++ (original)	User	40.76	5	4
C++CSP2	OS	44.59	-	4
	User	18.60	4	4
CTC++ QNX	OS	57.06	-	4
LUNA QNX	OS	28.02 / 34.03	-	4
	User	9.34	4	4

The way of working advocates to use MMD tools for software design. Using MMD techniques would result in a different implementations of the commstime benchmark. For example, instead of using code blobs as process implementation, submodels are used containing readers, writers and other processes. In this situation the Successor is implemented using a sequential process having a reader, an increment and a writer process as its child processes. Note that the token consists of a number that is incremented each time it is passed around.

The result of the commstime benchmark implementation that is designed with a MDD tool, is shown in Table 5.3. The number of threads and context-switches clearly is increased, which is due to the additional processes in the models.

Table 5.3: Commstime results when using MDD tools to implement the benchmark.

Platform	Thread type	Cycle time (μs)	# Context-switches	# Threads
CTC++ (original)	User	88.89	10	6
C++CSP2	OS	12554.95	-	+15
	User	12896.22	19	+15
CTC++ QNX	OS	219.71	-	6
LUNA QNX	OS	93.23 / 99.62	-	10
	User	29.87	14	10

The implementation of the C++CSP2 test was somewhat different compared to the other implementations. The C++CSP2 threads are destroyed as soon as their processes are finished, which is after one cycle. With the optimal implementation this could be circumvented by adding clever code. With this modelled implementation this is not possible, due to the design ideas behind the library. Due to this limitation, the C+CSP2 implementation recreates 15 threads each cycle of the token, hence the +15 in the table. The construction and destruction of new threads generates a lot of overhead, resulting in the cycle times being about a factor 100 higher.

The table also shows that the close result of the original CTC++ and the CTC++ QNX libraries in the previous section were accidental. Now the difference is bigger, which is

expected since the CTC++ QNX library uses OS threads which have much more overhead compared to the user threads. The optimised channels of the QNX variant probably compensated for the slow OS threads, which is only possible for the result with the optimised implementation as it does not have much context switches.

The benchmark results of the user thread implementations of LUNA than the results of the CTC++ library. The LUNA results are also better than the C++CSP2 results for both the OS and user thread implementations, when looking at Table 5.2. This is due to the efficient context-switches, as described in the previous section. The distributed, simple LUNA scheduler, that is spread over the different CSPConstruct and CSProcess implementations, seems to pay off.

It is clear that using MDD tools and code generation results in slower code, when comparing both tables. This is due to the difference between the modelling and execution points of view. For simple applications with a few processes it is advisable to manually create the code, especially for low-resource embedded systems. When creating a complex application to control a large setup, like a humanoid robot, it saves a lot of development time to make use of the MMD tools.

5.4.3 Cyber-Physical System Use Case

A LUNA implementation for JIWY (Jovanović et al., 2002) has been developed, to see whether LUNA is usable for real cyber-physical systems. JIWY is a simple pan-tilt system, with 2 motors and 2 encoders to control both degrees of freedom. A joystick is used to provide set-points to the control algorithm. The control software of this system is designed from a modelling point of view and thus requires about 50 context switches for each period.

The CTC++ library already has an implementation for this setup available. A similar implementation, compared to the CTC++ one, was made for LUNA to keep the comparison fair. Real-time logging functionality was added in order to be able to measure timing information to be able to compare both implementation.

The timing results of LUNA and the CTC++ implementation are shown in Table 5.4. The experiments have been performed with $100Hz$ and $1kHz$ sample frequencies, so each control loop cycle time should be $10ms$ and $1ms$ respectively. As the measurements were performed for about 60 seconds, the $100Hz$ measurements resulted in about 6,000 samples and the 1 kHz resulted in about 60,000 samples. The processing time values are found by subtracting the idle time from the cycle time. The idle time is calculated by measuring the time between the point where the control code is finished and the point where the timer fires an event for the next cycle.

The hardware of the computing platform has circuitry to fire events at certain intervals, these hardware events are used to implement the available OS timers. Unfortunately, the period between two events is not exactly $1ms$, or whatever the resolution of the timer is. When enough hardware events are gathered the timer fires a software timing event and updates its waiting time to match the next period. The waiting time gets updated by increasing it with the configured software period, instead of the actual time between the current and the previous fired hardware events. This results in

Table 5.4: Timing results of the cyber-physical system implementation.

Platform	Frequency (Hz)	Cycle time (ms) Mean	Min	Max	Standard dev. (μs)	Processing time (μs)
CTC++ (original)	100	11.00	10.90	11.11	14.8	199.0
	1000	1.18	0.91	2.10	386.5	174.5
	1000.15	1.00	0.91	1.10	20.7	172.5
LUNA QNX user threads	100	10.00	9.93	11.00	39.6	111.6
	1000	1.00	0.80	2.01	35.8	89.3
	1000.15	1.00	0.79	1.21	33.2	87.3
LUNA QNX OS threads	100	10.00	9.97	11.00	39.1	214.3
	1000	1.00	0.96	2.00	14.4	185.6
	1000.15	1.00	0.95	1.05	8.3	190.8

a missing timer event every now and then, as the actual period using the hardware events is different compared to the configured software period (Charest and Stecher, 2011).

For the $100Hz$ measurements this is neglectable, as it has $10ms$ periods and once in a while a $11ms$ period, which is an error of 10%. But, for the $1kHz$ measurements the incorrect period is suddenly $2ms$, which is an error of 100%. Both errors are found in the max cycle time column of the $100Hz$ and $1kHz$ measurements. Therefore, the exact period between two events is measured for the testing hardware, and the $1kHz$ period is adjusted accordingly. This results in the $1000.15Hz$ rows in the table, which show much better results compared to the $1000Hz$ rows.

When looking at the original CTC++ implementation, it is seen from the mean time that the obtained frequency is $90.9Hz$ instead of $100Hz$. The standard deviation is low, suggesting that the library is being constant in providing the wrong frequency. Same goes for the $1kHz$ measurement of the CTC++ implementation, where a $847.5Hz$ frequency was obtained. For a frequency of $1kHz$ the standard deviation becomes very large as well, but this is due to the missed events as described earlier. The $1000.15Hz$ results show that the missed event problem is indeed solved.

The results show that LUNA performs well within hard real-time boundaries. The mean values are a good match compared to the used frequencies and a low standard deviation value shows that the amount of missed deadlines is negligible. The frequency of $1000.15Hz$ indeed solves the maximum cycle times of LUNA being two periods long. The CTC++ results for the $1000.15Hz$ frequency are slightly better though. On the other hand, the LUNA results of the adjusted and the $1kHz$ are not differing much, besides the high max cycle time value, showing that LUNA is more robust for all frequencies than the CTC++ library and adjusting frequency is not required to get reasonable hard real-time properties. But, for setups which needs to be *extremely* accurate it is important to use adjusted frequencies as: "this can make the difference between an industrial robot moving smoothly or scratching your car".

It is also noticeable that the processing times for the LUNA user threads are lower compared to the CTC++ processing times. Suggesting that the overhead is much lower and

that more CPU time are available for the control algorithms or other tasks. Even the LUNA OS threads processing times are comparable with the CTC++ user thread processing times.

5.5 Conclusions

LUNA has fast context-switches and the commstime benchmark is faster than the C++CSP2 and CTC++ implementations. These benchmark results are good but the main requirement (8), the real-time behaviour of the library, is much more important when controlling cyber-physical systems. The JIWY implementation shows that LUNA indeed performs as required. The maximum and minimum cycle time values are close to the requested cycle time and the standard deviation values are low, showing that the hard real-time properties of LUNA are good as well. Visual inspection also shows that JIWY reacts smoothly on the joystick commands.

The choice for QNX is not that obvious anymore when the provided rendezvous channels are only usable between OS threads. Nonetheless, QNX provides a good platform to build a real-time framework, there is enough support from the OS to keep implementation tasks maintainable.

All requirements mentioned in the introduction are met. Requirements 8, 9 and its sub-requirements are obvious: LUNA is a hard real-time, multi-platform, multi-threaded framework.

Requirement 11: Scalability, is also met, as early scalability tests showed that having 10,000 processes poses no problem. Furthermore, LUNA is used in Section 4.5 for a Production Cell implementation, so it is usable to control complexer systems than the JIWY setup as well.

The CSP execution engine (Requirement 10.2) is the only implemented execution engine at the moment. But Requirement 10.1, to not be dependent on any execution engine, it is met as it is possible to turn it off. In this situation it is possible to still use the core components and hardware abstraction provided by LUNA. Using the provided interface it is also possible to add other execution engines like a state machine execution engine (Requirement 10).

Requirement 12, debugging and tracing support is fulfilled. Even though it is not discussed explicitly, it was used to obtain results like timing properties and the number of context switches. Like any LUNA component, it is possible to enable the debugging and/or (real-time) logging components as well. These components provide means for debugging and tracing the other components as well as the application that is being developed. It is also possible to send the debug and trace information over a (local) network to the development PC, in order to have run-time analysis or to store it for off-line analysis. Especially logging the activation of processes is interesting, as this provides valuable timing information, like the cycle time of a control loop or the jitter during execution. It is also usable to follow the execution of the application by monitoring the states (running, ready, blocked, finished) of the processes.

The logger is able to fulfil hard real-time guarantees (Requirement 12.1) if required. It

has predefined buffers to store the debug information and only when there is idle CPU time available, it sends the buffered content over the network freeing up the buffer for new data.

As mentioned earlier, both Orocos and ROS provide interesting features for supervisory and sequential controller solutions, so it would be a waste of development resources to (re)implement such features for the LUNA framework. Therefore an integrating method between LUNA and these frameworks is desired in order to be able to use the nice features of all frameworks. As a starting point, it might be possible to reuse and extend earlier integration work (Smits and Bruyninckx, 2011).

6 Twente Embedded Real-time Robotic Application

The way of working suggests that understanding and maintainability of models is increased by using a graphical editor and accompanying tools. Such a collection of related tools is also known as *tool suite*. The design and implementation of the *Twente Embedded Real-time Robotic Application* (TERRA) tool suite is described in this chapter. Most of its features are based on the feature descriptions for tools to support the way of working as discussed in Section 3.2.

TERRA is designed around the CPC and derived meta-models. The TERRA tool suite supports designing architecture and CSP models and it integrates external, non-TERRA related, tools to access other model types as well. Additionally, the TERRA tools provide means to use the models for formal verification, source code generation and (co-)simulation.

TERRA is developed because the existing gCSP tool (Jovanović et al., 2004) has not the quality that is needed to follow the way of working or implement the GAC model. The gCSP models are not based on an explicit meta-model, which should be the case according to the way of working. Furthermore, its stability get lower when designed models are getting larger: Models of the size of the GAC are nearly impossible to draw without problems.

This chapter starts with a discussion on related work of meta-models followed by a discussion of the available tools that provide support for CSP models. Use-cases of meta-models within the TERRA tool suite are shown next, followed by the implementation details of these meta-models. Next, the use-cases and the TERRA tools that provide their support are discussed further. After that an evaluation of the TERRA tool suite is provided, showing some insights of actual uses of the tool suite and pointing out the problems that were found. The chapter ends with conclusions on the meta-models and the TERRA tool suite.

6.1 Related Work

First related work on meta-model is explored, followed with a discussion why Eclipse in combination with EMF and GEF chosen as a basis for TERRA. Related work on tools

and frameworks supporting CSP or custom definable meta-models is discussed afterwards.

6.1.1 Meta-Models

UML diagrams (Object Management Group, 2011) are the de-facto standard for describing software designs. However, the standards are purely declarative and do not provide formal semantics. Formalising UML diagrams with CSP (i.e. adding formal semantics in general) is desired, such that livelock and termination checks can be done, to guarantee the quality of the components.

Varró et al. (2008) discuss a case study on model-to-model transformation from UML activity diagrams to CSP, whereby multiple transformation solutions and tools are evaluated. The authors divide the solutions in three categories: pure graph transformations, solutions with control structures and solutions based on a host framework / language. The presented (target) CSP meta-model resembles an Abstract Syntax Tree (AST) for CSP grammar, where programming language concepts (e.g. *ProcessAssignment*) are mixed with CSP concepts. Since the meta-model is more concerned with storing a CSP document than with modelling the process composition and communication, this meta-model is considered not suitable for the use cases presented in this chapter. A rule-based model transformation solution with control is discussed by Küster (2006), whereby the transformation from UML state charts to CSP is taken as a case study. However, no explicit CSP meta-model is given.

The BRICS project has defined a new generalised component meta-model, called the *BRICS Component Model* (BCM) (Klotzbücher et al., 2013). BCM is loosely based on the CORBA Component Model (CCM) (Object Management Group, 2006), whereby components exchange information through ports over connections (i.e. channels). However using the BCM, concurrency cannot be expressed as explicitly as in CSP (e.g. sequential execution cannot be specified).

In the DESTECS project, work is being done to define a Structural Operational Semantics (SOS) of co-simulation of discrete-event (DE) controller schemes and continuous-time (CT) behaviour of the machine to be controlled (Lausdahl et al., 2011). An SOS description consists of type definitions describing the static structure and transition relations for the behaviour of the model (Plotkin, 2004), whereas a meta-model defined the (strict) semantics of the model itself. The DESTECS SOS description is not a meta-model, but it serves more or less as a meta-model defining the behaviour of the model and its possible transitions.

Eclipse and the *Eclipse Model Framework* (EMF) (Steinberg et al., 2009) are used for the development of TERRA. Both are used by TERRA as implementation frameworks as well. Eclipse is mainly written in Java and has therefore, among some other reasons, excellent multi-platform support, so basing TERRA on Eclipse makes this platform support available to TERRA as well. EMF is used to design the meta-models that are present in TERRA and is also used by TERRA itself to add meta-model support to its tools.

Another important reason to use Eclipse is the tight integration of the *Graphical Ec-*

lipse Framework (GEF) (Rubel et al., 2011) with EMF. Using GEF makes it easy to develop a graphical editor based on the meta-models and modelling support of EMF. There are other Eclipse plug-ins that provide similar support, like GMF and Epsilon, but GEF is chosen for this support as it provides most freedom to design the graphical editors. Disadvantage of GEF, compared to some of these other plug-ins, is that lots of things need to be implemented manually as they are not provided by the framework.

Note that Eclipse, EMF and GMF are also used for the development of the BRICS Integrated Development Environment (BRICS Consortium, 2013). It is a similar development tool as TERRA, also towards architectural models with components. However, it is mainly focused on Orocos components and their deployment.

6.1.2 Tooling

These tools are available to design or use CSP models:

- The FDR2 tool can be used to formally verify CSP models.
- The gCSP tool (Jovanović et al., 2004) can be used to design CSP models.

However, none of these tools provides an explicit meta-model and are therefore not used for the given reasons in Section 2.4.

Ptolemy II (Ptolemy, 2012) is a heterogeneous modelling and simulation tool that allows to create multi-domain models using different models of computation (e.g. Finite State Machines or CSP), consisting of actors and directors. Actors are comparable to sub-models or CSP processes. The directors determine the domain and the model of computation that is used by the simulator for executing an actor. The interaction between actors with different models of computation is based on well defined interfaces and flow of control rather than model-to-model transformations.

Verhaar (2008) has performed a usability study of Ptolemy II and concludes that a single general design tool is not able to satisfy all specialistic needs for all disciplines. Furthermore, it is noted that the CSP domain support of Ptolemy II is insufficient to be usable. Reinventing the wheel for the sake of using a single design tool, is a waste of development resources. A development tool should provide means to reuse the specialistic design features for a certain discipline that are supported of external, third-party tools instead. Therefore, there does not seem to be a point to use Ptolemy to implement the required tool suite to support the way of working.

6.2 Meta-Model Usage

Models are at the center of the TERRA tool suite, as shown by Figure 6.1. The figure uses a CSP model as an example to show the meta-model usage within the TERRA tool suite, but the same goes for any other type of model. The large open arrows in the figure indicate interaction of the tools with the model, resulting in all kinds of actions or transformations.

All TERRA models are conforming to their meta-models, which are in the end derived from the CPC meta-model. The CPC meta-model is the basis of all other meta-models

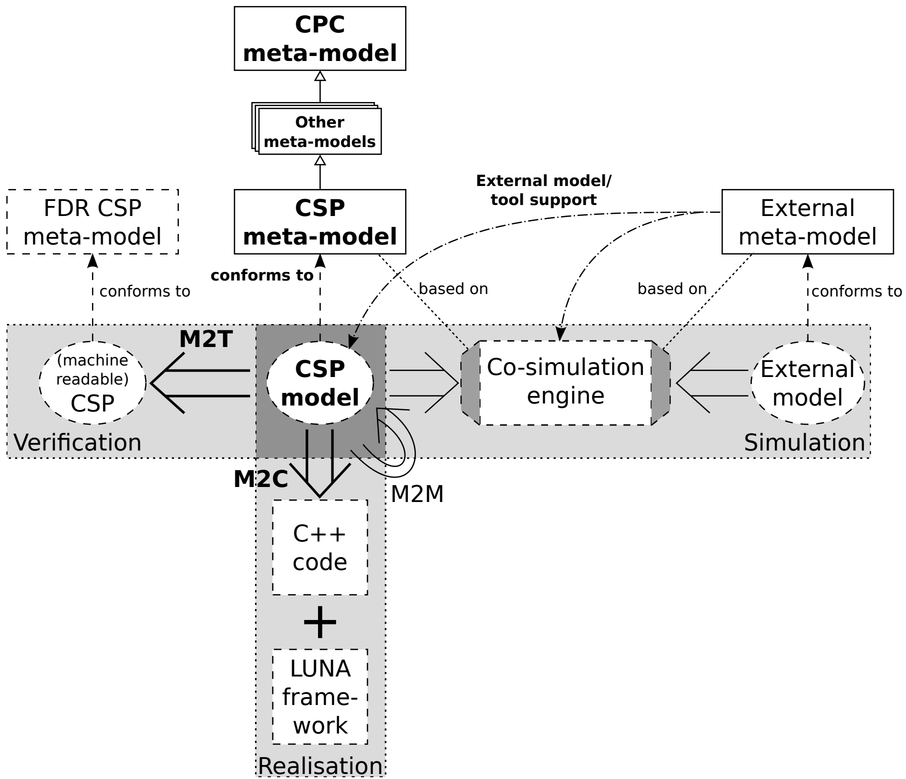


Figure 6.1: TERRA tool suite overview. The bold parts are already available in the current version.

as specified by the way of working, with interoperability as the main reason. One or more other meta-models might serve as a base as well, but still they all are derived from the CPC meta-model in the end. Whether a meta-model is derived from one or more other meta-models depends on the situation and whether is functionality is shared by other meta-models.

As mentioned earlier, the meta-model provides the semantics of the model, which are used by the tools to interact with the model. Due to the strict usage of the meta-model semantics by all tools, the models can get processed by these tools without a problem.

Different model transformation paths are shown in the figure. They are have their own uses and have different results, more information about model transformations is provided in Section 2.7.

At the right of the figure an external model and its accompanying meta-model, are depicted. This model is managed by an external tool, for which TERRA provides support as is shown in the figure. This support consists of plug-ins, see Figure 6.2, that extend the meta-models to provide means to store information about an external model

and add support the tools. Same goes for the co-simulation engine, it does not need to know how to simulate the external model, only how to interact with the external tool to let it handle the simulation of the external model. For this to work the external meta-model needs to conform to the CPC methodology.

As implicitly mentioned above, TERRA is a collection of Eclipse *plug-ins*, shown in Figure 6.2. Plug-ins can be added or removed, making TERRA modular. Of course, if a plug-in which is required by other plug-ins is removed, the other plug-ins become inactive as well. Each TERRA tool consists of one or more plug-ins. Together all plug-ins form the TERRA tool suite, providing support to design models and to use them for various purposes.

Most plug-ins are dependent on one of more other plug-ins, shown by the arrows in the figure. Dependencies on non-TERRA plug-ins are left-out to prevent the figure getting even more chaotic. Upon examining the figure, it becomes apparent by the dependency lines in the figure that TERRA indeed revolves around the meta-models. The meta-model plug-ins have relatively many dependencies pointing towards them, especially the `cpc.model` and `csp.model` plug-ins.

6.3 Meta-Model Implementation

An EMF meta-model in general consists of elements, as shown in Figure 6.3, to capture domain concepts (model objects). Each element might contain one or more attributes and/or operations. Attributes hold data associated with the element, like a string to store the name of the object, or references to store a relation between this object and the referenced object. Element operations provide additional functionalities or ways to use the element. All elements, attributes and operations have a set of properties to refine them. These properties include a name, type, default value, and so on.

The meta-models are implemented using the Eclipse Model Framework, as explained in the previous sections. The actual (Java) implementation of the designed meta-model is obtained through code generation. EMF code generation is able to keep modified code in place when regenerating the implementation, so non-modelled customizations of the implementation are not lost. The framework also provides additional services that are required for the implemented meta-models, like notification on model changes or model (de-)serialisation to load/store models.

The meta-model presentation in Eclipse/EMF, see Figure 6.3, is similar to the UML class diagram presentation. Both methods can be used to create an object oriented ontology to capture the domain concepts. Concepts like the described elements, attributes and operations can be found in the UML class diagrams, however they are named classes, variables and methods respectively.

6.3.1 CPC Meta-Model

The CPC meta-model is implemented by the `cpc.model` plug-in and provides means to design CPC-like models. The meta-model provides elements like component (CPCComponent), port (CPCPort) and connection (CPCConnection) objects, shown in Figure 6.4. Note that some object names in the figure start with an 'I', this indicates

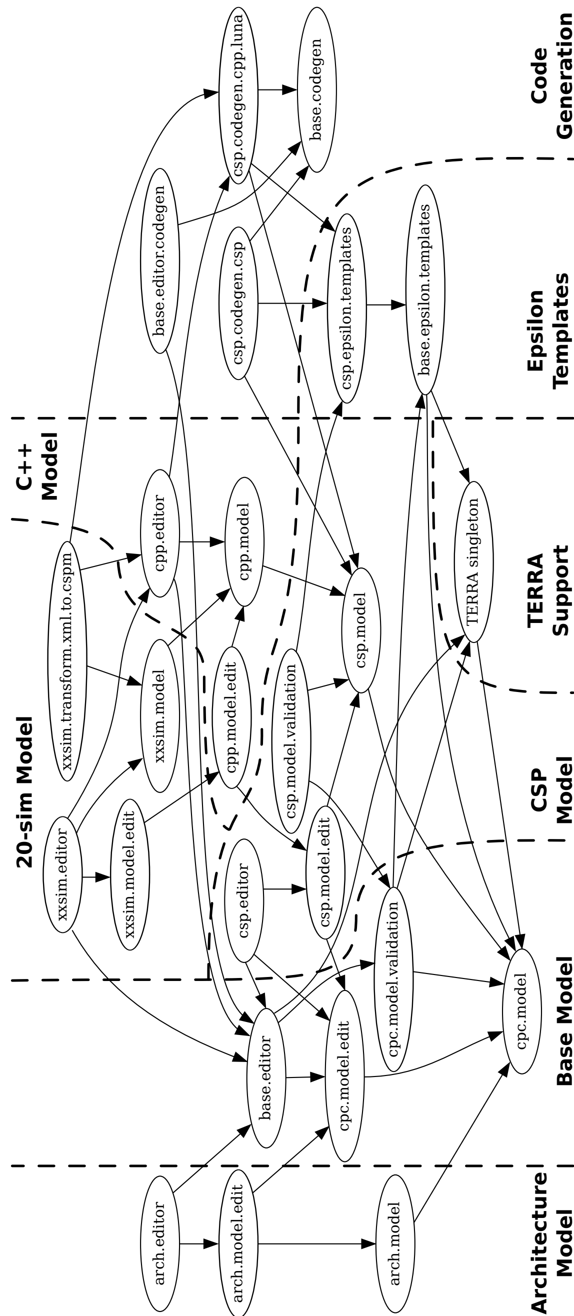


Figure 6.2: Overview of all TERRA plug-ins and their dependencies. (The figure is generated using the source code of TERRA, as it was on the moment of writing.)

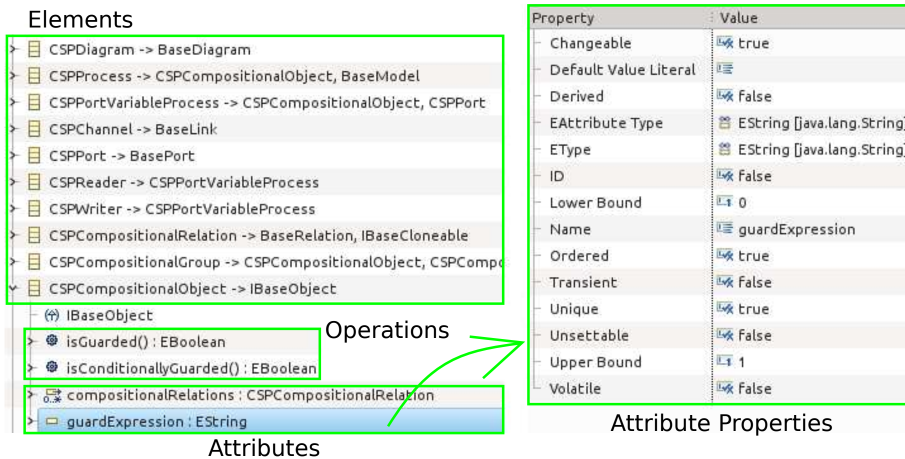


Figure 6.3: Meta-model terminology (applied on the CSP meta-model).

that the object is an interface that must be implemented by the object. There is no distinction made between regular and interface objects in the remained of this chapter.

The core CPC meta-model are also shown on the left side of Figure 6.5, Section 6.3.2, showing the inheriting relation between the CPC and CSP meta-models. All meta-models that use the CPC meta-model, directly or indirectly via other meta-models, add more domain specific features to the CPC elements. In BRICS the CPC meta-model is presented as another layer of abstraction, i.e. as a meta-meta-model (Klotzbücher et al., 2013). The inheritance relation between the TERRA meta-models is stronger than the regular *conforms to* relation between meta-model and its meta-meta-model. This strong inheritance relation of the TERRA meta-models makes it easier to create tools that are compliant with multiple meta-models, as these tools can rely on the CPC basis of the other meta-models.

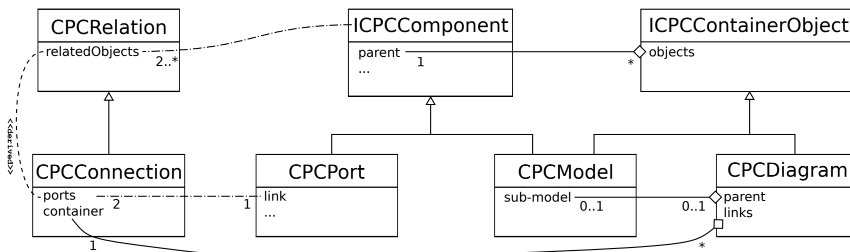


Figure 6.4: Class diagram showing the CPC meta-model details

The CPCComponent element provides the base for components. Most other component elements extend this CPCComponent element and add specific details. Even CPCPort implements ICPCComponent, as each port is seen as a component, which implements a generic port.

A similar construction is used for relations. The CPCRelation defines that two or more components are related. The actual type of relation that is shared between the objects, is defined by elements that extend the relation element. For example, the CPCConnection element defines a data exchanging relation between components.

Multiple nested levels of components need to be supported to create hierarchical models, e.g. components can contain sub-components. For this purpose the meta-model provides a specialised container element called ICPCContainerObject. This element is used by other elements to provide support for containing other objects. For example, the CPCDiagram element implements the ICPCContainerObject element to contain all of its components, ports and connections. The CPCModel implements this interface to store its contained ports. The CPCModel element has a sub-model attribute, which can be used to define a sub-model diagram that can be used to further specify the component in a hierarchical manner.

Note that Figure 6.4 and the explanation in this section are incomplete, only the CPC-related meta-model elements are discussed. The meta-model serves as the base for other meta-models, so additional elements, e.g. to store additional properties into, are included in this meta-model in order to make them available to all other TERRA meta-models. Their explanation is left out to keep the CPC meta-model explanation from becoming too long and too complex. Furthermore these elements are really interesting for the scope of this thesis.

6.3.2 CSP Meta-Model

The CSP meta-model, implemented by the `csp.model` plug-in, extends the CPC meta-model by adding CSP domain specific elements, shown by Figure 6.5. Together, these elements implement an explicit meta-model for Hoare's CSP algebra.

The CSP meta-model provides compositional meta-model elements, which are divided into a CompositionalObject, a CompositionalRelation and a CompositionalGroup element. The CompositionalObject (extended CPCComponent) is used to specify that the component has a compositional relation to another CompositionalObject. CompositionalRelations extend the CPCRelation element and are used to add details of this compositional relation. For example, the type attribute defines the type of the compositional relation, some of possibilities are parallel, sequential or alternative relation types. The CompositionalGroups are used to group the relation objects, with the same type, into so called compositional groups. These group elements are extending the CompositionalObject, so it is possible to hierarchically define compositional relations between compositional groups (and objects) as well.

Additionally, the CSP meta-model provides the CSPChannel element. It is an extended version of CPCConnection defining either a rendezvous or a buffered CSP channel. The additional attribute provides the means to optionally specify the channel buffer size.

The relation between the CSP constructs, in machine-readable form, and their representing meta-model elements is shown in Table 6.1. The CSP meta-model supports all regular CSP constructs required to define processes and the communication flow

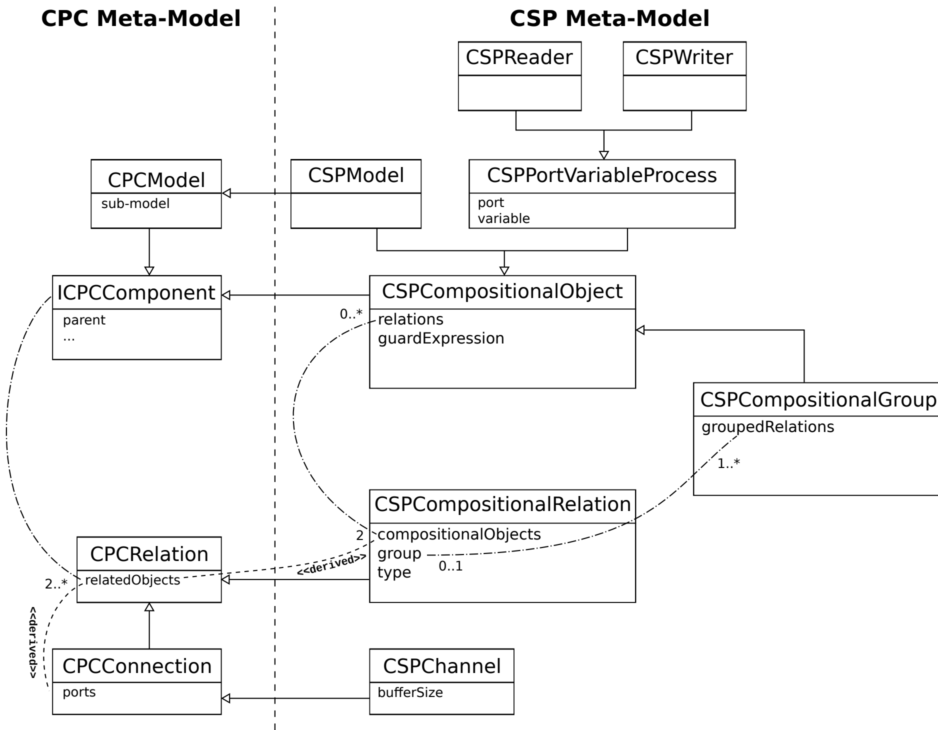


Figure 6.5: Partial CSP meta-model, showing the relation with the CPC meta-model elements and the CSP compositional elements.

between them. The elements of the meta-model are also depicted in Figure 6.5 showing their relations and interactions.

CSPModel and CSPCompositionalGroup are both used to define a CSP process. CSPModel is used to provide means to add sub-models to the design and CSPCompositionalGroup is used to define a group of processes sharing the same compositional relation type. In CSP there is no difference between them, as a 'sub-model' is also a group of processes since CSP does not have a notion of sub-models. CSPReader and CSPWriter are two specialised processes that interact with their given channel.

The compositional relation between two processes is defined by the CSPCompositionalRelation element. The type attribute defines the relation type of the processes. CSP processes that are part of an alternative compositional relation, require guard constructs. These are provided by a guard expression that is contained in the process object. The evaluation of the expression has either a positive or negative result. The guard expression is optional for the CSPReader and CSPWriter processes that are part of an alternative relation. If it is not provided a channel guard is used, which uses the channel to determine whether data can be read or written and this whether the pro-

Table 6.1: List of CSP constructs that have corresponding CSP meta-model elements.

CSP construct	meta-model element	attribute(s)
`p = ...`	CSPModel	
	CSPCompositionalGroup	*groupedRelations*: relations that are grouped
`channel c`	CSPChannel	*ports*: two connected ports/processes
`datatype <type>`	CSPVariableDescription	*name*: name of the variable
`= <name>`		*type*: boolean for `Bool`, integer for `Int`, etc.
`c !<variable>`	CSPWriter	*variable* contains data to write on the channel
		link: channel to write to
`c ?<variable>`	CSPReader	*variable* contains data read from the channel
		link: channel to read from
`p ; q`	CSPCompositionalRelation	*type* = SEQUENTIAL
		compositionalObjects: the two related objects
`p || q`	CSPCompositionalRelation	*type* = PARALLEL
		compositionalObjects: the two related objects
`p [] q`	CSPCompositionalRelation	*type* = ALTERNATIVE
		compositionalObjects: the two related objects
`if`-statement	CSPRecursionProperty	*expression*: true when another loop/iteration is required

cess can be activated, see the LUNA alternative explanation in Section 5.3.4 for more details.

Recursive constructions are also supported by the meta-model by the CSPRecursion-Property element. In CSP a recursion, or loop, is written as follows:

```
p = if (<expression>) then <process> ; p else SKIP
```

If `<expression>` evaluates to true, it activates the `<process> ; p` part. After `<process>` is executed, `p` is activated (again) and a looping behaviour, depending on `<expression>`, is implemented. In the CSP meta-model, recursions are implemented as properties, which can be attached to model objects supporting properties, like CSPCompositionalGroup elements or CSPModel elements.

More modern CSP extensions, like mobile channels (Welch and Barnes, 2008), are currently not supported by the meta-model. If the need rises, the meta-model can be extended to support such an additional feature, although the CSP execution engine needs to provide support for this as well.

6.3.3 Architecture Meta-Model

The architecture meta-model is implemented by the `arch.model` plug-in. The meta-model does not add any new elements to the CPC meta-model, as the architecture models also consist of components that have ports which are connected by channels for communication purposes. The architecture meta-model is only used to rename CPC meta-model elements and for future use.

It is developed to be able to design system architectural models, consisting of components and connections between them. It hides the underlying CSP paradigm that is used to be able to execute the architecture models by the code generation. The component implementations are typically provided by the GAC-based components. So

basically the model type enables the use of TERRA for the non-expert users, whereas the CSP models are meant for the expert users.

6.3.4 Other Meta-Models

TERRA provides support to add new meta-model elements to existing meta-models via the means of a new plug-in and to integrate support for the additional elements with the TERRA tools. The modular support of TERRA provides all kinds of hooks in its tools, which can be used by such a new plug-in to provide support for their additional meta-model elements. This is currently used to add optional support for C++ code blobs and external 20-sim models.

The 20-sim plug-ins depend on the C++ plug-ins as depicted in Figure 6.2, for example, the 20-sim meta-model plug-in `xxsim.model` depends on the C++ meta-model plug-in `cpp.model`. The C++ meta-model plug-in provides an additional elements to model C++ code blobs, which can be used to add custom code to the models. The disadvantage of the C++ model elements is that the user-defined C++ code is not recognized by the tools, obviously. Therefore, these elements cannot be used for verification, simulation or other operations.

The 20-sim meta-model is discussed in the next section as if its complete implementation is provided by the 20-sim plug-ins, ignoring the role of the C++ plug-ins to prevent confusion. This is also the case for the other 20-sim explanations later in this chapter.

20-sim Meta-Model

Support for the 20-sim meta-model elements, i.e. means to add custom, foreign model elements to a regular model, and how they integrate with the CPC meta-model is described in this section. The semantics of such a foreign element need to be provided by its own meta-model to comply to the requirements of the way of working.

A *20-sim configuration* element is provided to convert the regular CPCModel element into a 20-sim specific one and to provide a the graphical model figure. The editor should not allow to modify the model interface, as this is defined by the 20-sim model.

Another element that is provided by the 20-sim meta-model is a *20-sim code block configuration.* It provides a 20-sim specific code block, that is used to contain the glue code to connect the 20-sim model to the TERRA model. This element is only usable by the 20-sim model-to-model transformation, described in Section 6.6.2.

6.4 Graphical Model Editor

The architectural (`arch.editor` plug-in) and CSP (`csp.editor` plug-in) editors are both based on the base editor implementation that is provided by the `base.editor` plug-in. The editors form the central part TERRA, as they interact with all other tool types, basically gluing them together forming the TERRA tool suite. As the base functionality of all editors is provided by the base editor, it can be seen as the very heart of TERRA.

The base editor provides support for the user interface of the editors: the actual model

view, the preferences, the tool palette, the menu structure and many other parts that are shared between editors. Additionally, it also handles resource management, like opening, storing and sharing models or interaction with the other tools. This modular support that can be used by any plug-in, is provided by so called *extension points*. The extension point mechanism is provided by Eclipse and can be used by plug-ins to allow the registration of certain activities, i.e. extending a certain point of the application, hence its name extension point. This extension point mechanism is used throughout Eclipse and its frameworks, making it modular and extensible.

The base editor, for example, uses extension points to let other TERRA plug-ins register itself and the activity they provide. One of the extension points that is provided by the base editor is called `provideModelObjectEditor`. This extension point is used by plug-ins that want to register an editor component for a custom model object being backed by a custom meta-model, see Section 6.3.4 for an example use case. The plug-in must provide an implementation that is conforming to the interface defined by the extension point. The base editor is able to access the provided implementation via this interface and therefore able to interact with plug-ins that were unknown during the development of the editor. Both the C++ and 20-sim plug-ins use this extension point to integrate support for their custom meta-model elements into the editors, as described in Section 6.4.1.

The TERRA editors are tighter integrated with the base editor, due to an inheritance relationship, than the plug-ins and their extension points. Each editor needs to provide several implementations for abstract parts of the editor, as required by this inheritance relationship. This is, in contrast to the extension points, obligatory as the editor cannot function without those implementations. It results in a less flexible and modular means to extend TERRA compared to the extension points, but it provides means to add specialised editor support on a much deeper level, which is required to create the editors for the CSP and architectural models. The lower flexibility is not a real issue, as developing a new editors is already more complicated than a support plug-in.

The base editor makes use of the *Graphical Eclipse Framework* (GEF) (Rubel et al., 2011) to provide the editor user interface components. GEF provides means to design graphical user interfaces to Eclipse, with its focus on model-based editors that are backed by an EMF meta-model. Each EMF model element has a GEF counter-part, a so called edit-part, which provides the editor with a figure that graphically represents the model element. Upon interactions with the figures or other parts of the user interface, their edit-parts provide commands to modify the corresponding model element.

As mentioned, the editors work together with the other tools. The implementation and interaction with the models editors of the other TERRA tools are described in the following sections.

6.4.1 20-sim Editor Integration

20-sim support also requires editor support, besides a new meta-model element and transformation rules. The *20-sim configuration* elements convert a regular CPCModel

element into a 20-sim model element, shown by the rounded rectangles in Figure 6.6. The *Pan* and *Tilt* model elements use external 20-sim models for their control law implementations. Their required input values are provided by the *HWInput* writer elements, these read the signal values from the hardware and *writes* it to the channel. The steering values are sent to the hardware by the *HWOutput* readers.

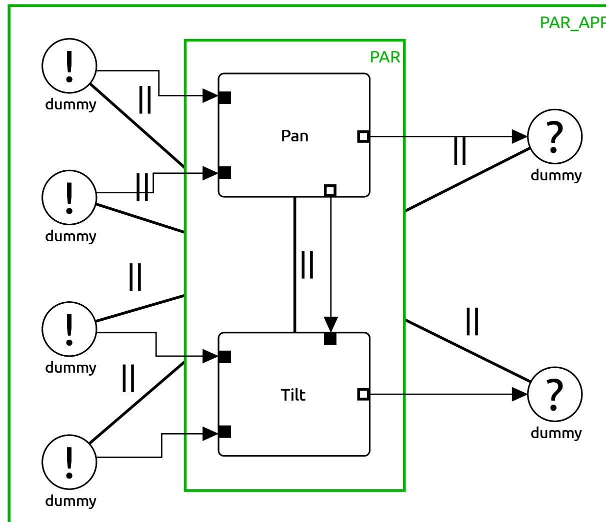


Figure 6.6: 20-sim model elements used in a software implementation of a simple cyber-physical system.

The base editor invokes a specialised 20-sim model editor to interact with these 20-sim model elements. This specialised editor is registered to the *20-sim configuration* element, using the `provideModelObjectEditor` extension point. The editor provides the custom figure, for example the rounded rectangles for 20-sim model elements. The 20-sim model interface is defined by the external 20-sim model. Therefore, the model interface must be locked by the TERRA 20-sim editor to prevent modifications to it. This is currently the only task of the 20-sim model editor, as the implementation of the element is provided by the 20-sim model-to-model transformation tool.

The 20-sim code block is handled in a similar way. It also has a custom editor that locks the element, as the implementation is completely provided by the 20-sim code generation plug-in.

6.5 Model Validation

The TERRA model validation and code generation tools use the Eclipse framework called Epsilon (Kolovos et al., 2012). It provides a language called Epsilon Object Language (EOL). EOL is used as a base to specify the Epsilon Validation Language (EVL) and the Epsilon Generation Language (EGL) and several other specialised languages.

The Epsilon framework provides means to used these language specifications for validation and code generation purposes.

The `cpc.model.validation` and `csp.model.validation` plug-ins register the meta-model, for which they provide validation rules, using an extension point provided by Epsilon validation plug-in. When a the model needs to be validated, the editor notifies the EMF validation framework. Epsilon is registered to this framework and when it gets notified, it activates the correct validation rules. This is a set of rules required to validate the meta-model, backing the model, and all of its inherited meta-models. For example a CSP model is validated using the CSP and CPC validation rules.

A model is restricted by the meta-model semantics. Hence, alien concepts are not representable in the meta-model, so the model does not contain related syntactical errors. The validation rules are needed to verify that the modelled elements are used in a correct way and that their attribute values are valid. For example, all objects require a valid name to identify themselves. These names need to be unique, otherwise it is ambiguous which object is meant by a name.

Depending on the purpose of the model, i.e. how the model is used by the designer, additional validation rules are required. These additional validation rules are not defined by the meta-models, as the purpose of the models depends on the available tools and the intentions of the designer. For example, the earlier example indicates that the element names need to be unique. When the model is created with the purpose to generate C++ code, the element also need to comply with the C++ syntax of class names, as the elements are converted into C++ classes. These additional rules can be checked separately before the tool is activated that requires the additional validation.

6.6 Model Transformations

Epsilon, described in previous section, is also used for model transformations. The Epsilon Generation Language (EGL) is used for model-to-text transformations (code generation) and the Epsilon Transformation Language (ETL) is used for model-to-model transformations. Currently, TERRA has two model-to-text transformation implementations: machine-readable CSP generation and LUNA C++ code generation. Model-to-model transformation is used by the 20-sim support plug-in to convert a generated, intermediate 20-sim model into a CSP model.

6.6.1 Model-to-Text Transformation

Both the code generation plug-ins (`csp.codegen.csp` and `csp.codegen.cpp.luna`) provide the model-to-text transformations for CSP models. Currently, there are no transformations for the architectural models available yet.

The plug-ins use an extension point to register the transformation and its details, like source model type and the transformation implementation, to the TERRA code generation framework (`base.codegen`). The framework adds a menu entry to the TERRA code generation menu and makes sure the entry is only available when the supported source model is active.

When the menu entry is activated, the framework uses the transformation implementation to start the transformation process. TERRA provides a base implementation for the code generation process. It invokes Epsilon and provides it with the EGL file that is required for the actual code generation. Within the EGL environment it is possible to generate multiple files, as desired by the transformation. In the case of C++ LUNA code generation, each CSP process has its own source and header file using a directory structure to make the file names unique.

It is likely that the generated C++ source code is not complete, since not all required meta-model elements are available yet, like model elements for hardware driver support. These missing parts still need to be implemented by hand in the C++ code, until support is offered by the TERRA tool suite. The Epsilon code generation engine simply overwrites all files, so manual changes will get lost each time code is generation. Therefore, the TERRA transformation tools generate so-called *protected regions* that can be used to add custom code into, which is kept intact by Epsilon when the file is regenerated. This technique is also used for the machine-readable CSP code generation.

The model-to-text transformation is also used by the 20-sim code generation plug-in. The code generation tool provides the `customModelCodeProvider` extension point, which can be used to provide custom code generators for foreign model elements.

The TERRA 20-sim code generation makes use of this extension point to provide glue code to implement the *20-sim code block configuration*. This glue code uses the variables of the readers to update the 20-sim model variables with the correct values. After all 20-sim variables are updated, the actual 20-sim generated code is executed, so all (control) algorithms perform their calculations. Their results are stored in the model variables, which are used to update the variables of the writers and are sent into the rest of the CSP model using the ports and channels.

6.6.2 Model-to-Model Transformation

Model-to-model transformation is only used by the `xxsim.transform.xml.to.cspm` plug-in. It converts an intermediate model file, which is generated by 20-sim from the source model, into a CSP sub-model implementation for the 20-sim model element. Together with the 20-sim generated C++ code it provides the implementation for a TERRA 20-sim model element. The intermediate model is also used by BRIDE for similar reasons (Brodskiy et al., 2013).

The conversion process iterates over all 20-sim model variables. For each variable it creates a CSP reader or writer containing the corresponding properties. A 20-sim C++ code element is created which implement the required calls to the 20-sim generated C++ code. After all variable are processes, all readers are placed in a parallel compositional relationship and all writers as well. Both parallel groups and the C++ code element are placed into a sequential compositional relationship.

The resulting structure is called an I/O Sequential (IOSEQ) pattern (Welch et al., 1993), which is shown in Figure 6.7 It uses a sequential order to first read the required values

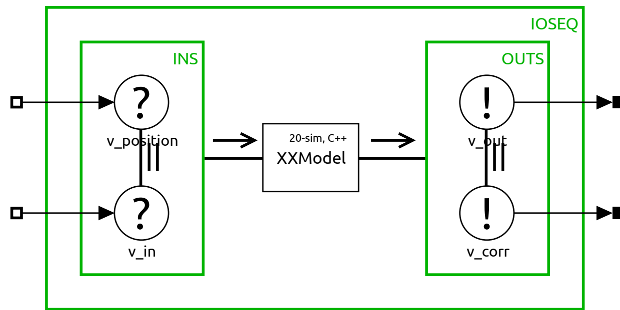


Figure 6.7: 20-sim model implementation, using the IOSEQ pattern.

from the incoming ports, shown at the left part of the figure. Then the 20-sim C++ code block uses the input variables to update the 20-sim model variables and executes the 20-sim C++ code. After the 20-sim code is finished, the results are stored in the variables at the right part of the figure and they are written the outgoing ports.

6.7 (Co-)Simulation

TERRA does not include a (co-)simulation engine yet, so future plans are described in this section.

It is possible to develop a modular simulation engine using the same techniques as described for the previous tools, like the Eclipse extension points. In order to simulate a certain model, the simulation engine needs to have the simulation rules for the corresponding meta-model registered. Simulation rules for a model are basically partial simulation engines that are invoked by the base simulation engine when required. If the model contains links to other models, their simulation rules needs to be present as well. For example, if an architecture model with component implementation provided by CSP models, the simulation engine needs have simulation rules for both the architecture and CSP meta-models.

The simulation rules need to define the order of components that needs to be simulated, using the domain specific details of the meta-model. Channels between the components and their data need to be updated each simulation step, so they can be used by the component implementations. Components are simulated, by simulating their implementations until primitive model elements are encountered that have their own simulation rules.

An example situation is encountered when an architecture model containing components with CSP implementations is simulated. When such a component needs to be simulated, the simulation engine invokes the CSP simulation rules on its CSP model implementation. Using the CSP algebra, the simulation order of the CSP processes can be determined and they are activated as required. This goes on for CSP processes that have a sub-model implementation, until primitive processes like readers or writers are encountered. In such cases their simulation rules determine to update a variable

with a value read from a channel or the write the value of a variable onto the channel, respectively.

The same goes for external models, although its simulation rules are defined in the corresponding external tool. Therefore, the TERRA simulation rules of the external model need to be able to interact with the external tool take let it perform the simulation of the external model. The interaction consists of providing updated data, a request to perform a single simulation step and the retrieval of the simulation results. This is the same approach as discussed by Ni and Broenink (2012).

In the case of an imported 20-sim model, its simulation rules need to grab the data from the correct CSP variable elements and update the model variables in 20-sim. After the simulation step, it needs to grab the updated model variables and write them back onto the corresponding CSP variables to feed the resulting data back into the model.

A workaround for the lack of a simulation and to prevent damaging the cyber-physical system due to software problems, is execute the software in a 'virtual test environment'. This is possible when software to simulate the plant, sensors, actuators and other hardware is available, e.g. by using the code generation of 20-sim when the required models are already designed. All software needs to be combined together with code to retrieve log information to create the virtual environment. After compilation of the virtual environment application, it can be executed to obtain the 'simulation' results. The quality of these results depends on the accuracy of the plant and other software parts, but this also goes for the regular simulation results.

6.8 Evaluation

The TERRA tool suite is evaluated in this section. First some usage examples and results are evaluated and discussed. These include different scenarios and use cases of the tool suite. After that a more general discussion of TERRA is provided together with recommendations to improve the tool suite further.

6.8.1 Usage of TERRA

TERRA has been developed to aid the development process of control software of cyber-physical systems as described by the way of working. It is shown in this section that TERRA indeed is usable for these intentions. Masters students have successfully been using TERRA for two years now to design control software for the JIWY setup (Jovanović et al., 2002) for the Real-Time Software Development course. It shows that TERRA is suitable for novice users to design relatively small models to control simple cyber-physical systems. TERRA had the first year several 'hiccups', ranging from platform (Windows) compatibility problems to missing features hindering the students to design their models as they wanted to. The second year when TERRA has been used, it has been improved considerably and no serious problems have been reported.

TERRA also has been used to design a first CSP implementation of the GAC (Hoogendijk, 2013), more details on this implementation are provided in Section 4.4. Again,

TERRA had several 'hiccups', this time mainly due to problems handling large models, like copy/pasting large parts of the design to reuse them elsewhere or in a new model version. These problems also have been solved and the initial GAC implementation is showing promising results. Therefore, it con be concluded that TERRA is suitable as a development tool suite for researching model structures and implementations.

The initial GAC CSP implementation has been used to actual control a large part of the Production Cell (Hoogendijk, 2013). This shows that TERRA is also suitable for larger models, compared to the JIWY setup, to control a cyber-physical system for research purposes.

The large amount of required CPU time to execute the GAC implementation, proved to be a problem for the software platform it had to be executed on, hence the partial implementation. The main reason is that the GAC implementation was a proof-of-principle, without any optimisations, to show that the ideas behind the GAC were actually sensible. Nonetheless, it shows that TERRA is a capable model-driven design tool suite to design the control software for cyber-physical systems.

6.8.2 Discussion

Code generation proved to be useful, especially the protected regions as TERRA does not yet provide a state-machine model editor to implemented the life-cycle states or support to interact with the hardware of the Production Cell from with the models. Same goes for the model validation tools, as such large models cannot be designed at once, so certain areas are left blank and these validation tools help to find the models parts that still need to be filled in.

The implementation of the Production Cell also showed that some tools were clearly missing. No simulation engine was available to test the (intermediate) models of the GAC and of the Production Cell, as prescribed by the way of working. As a workaround the generated code was tested on a virtual platform, but this proved to be quite a hassle and the resulting quality was lacking details to prove the control software should work on the actual cyber-physical system without breaking it.

Some problems, like dead-lock situations, occurred during the testing phase of the control software. Due to the size and complexity of the implementation models, it was for example nearly impossible to find out which parts suffered the dead-locks and what caused it. The integration of logging support tools to TERRA helps sorting out these problems, as the obtained information can be used to animate the models as described by van der Steen et al. (2008). Animating a model provides syntax highlighting and annotations for the models to reflect the current state of the running software. When the software is paused, due to user interaction or problems like dead-locks, this information helps showing the problematic areas of the model.

Furthermore, TERRA is missing GAC support. This is mainly due to the GAC implementation being immature and susceptible of changes. But this lack of support greatly complicated the implementation of the Production Cell as the GAC implementations needed to be copy/pasted from the master template and needed to be manually modified to meet the situations they were used for. An example of the manual modifications

is changing the number of custom ports and connecting them to the corresponding parts of the GAC. These manual changes are tedious and take a lot of time from the designer. GAC support would be able to handle the required changes depending on configuration parameters.

6.9 Conclusions

The presented CSP meta-model is suitable to design CSP models that conform to Hoare's CSP definition. Section 6.3 describes how the CSP meta-model is derived using a modular approach by extending the CPC meta-model. The CSP meta-model has all kinds of use cases, as described in Section 6.6. For example, model-to-text transformations are used to formally verify the CSP model with FDR or to generate code that can be executed on an embedded target.

The architecture meta-model is suitable to design system architecture models, consisting of the software control components and communication connections between them. Its amount of available elements is low, so it is easy to get used to. Complex implementations are 'hidden' by the specialised GACs and other (pre-build) component implementation.

The modular nature of the CPC and CSP meta-models makes it possible to support additional requirements. Therefore, the CSP meta-model is suitable as a standard for all kinds of CSP modelling related work. It is recommended to make use of this meta-model to standardise modelling within the CSP community. Hopefully a standard meta-model will emerge that is suitable for the needs of the community and helping to improve interaction between multiple disciplines within the community.

TERRA is an integrated collection of tools to support the way of working. The user is able to graphically construct a CSP model that conforms to the CSP meta-model. Model checking on livelock and deadlock conditions is supported by using FDR2. When satisfied, the CSP model can be transformed into LUNA based code using model-to-text transformations.

Several use-cases of TERRA are evaluated and discussed. Although it is not completely and thoroughly researched yet, TERRA is capable to provide a fully working tool suite that supports the provided way of working until the simulation phase. Future plans are to include co-simulation as described in sections 6.7, in order to support this phase of the way of working as well. The missing deployment phase is likely to be implemented by enabling the Eclipse C++ compilation support and by adding animation and logging. Including GAC support should improve the overall usability of the way of working and make the implementation of software for cyber-physical systems easier, as discussed in Section 6.8.2.

7

Conclusions and Recommendations

Designing control software for modern cyber-physical systems tends to become complex, due to the increasing complexity of these systems. The main objective of this thesis is to provide guidelines for managing the complexity of the software. These proposed guidelines provide a structural way of working, which is based on model-driven design techniques. The way of working aims to provide means to develop a first-time-right implementation for control software. The models at the basis of the control software designs, help in the understanding and comprehending the increasing complexity of the cyber-physical systems and their designs.

The way of working is based on a separation of concerns. At a system level this is provided by using separate components for each (control) part of the software. The generic architecture component provides a blue-print, or template, for these software components. Separation of concerns is also applied in the design of the template GAC, to keep it understandable and usable for specialised GACs.

The LUNA execution framework is designed for the execution of the modelled control software. Execution engines consist of the static parts to supporting meta-model definitions, like CSP constructs, a scheduler and hardware support. A tool suite, called TERRA, provides graphical ways of creating, manipulating and verifying the software models. Model-to-code transformations are used in the end to transform the models into software. The software makes use of the execution framework, thereby reducing the complexity of the transformed code.

7.1 Conclusions and Evaluation

The main objectives of this thesis are described in Section 1.3. Each of these objectives need to keep the sub-objectives in mind that are described in the sections 1.3.1 to 1.3.3.

7.1.1 Way of Working

The way of working provides steps for the complete software design trajectory. Its concerns are split according the Formalisms, Techniques, Methods and Tools approach, to make it as generically applicable as possible. For example, it uses meta-models to

define the semantics of the models instead of letting it handle by the tools. Also, the tool implementation is independent of the actual steps, i.e. any set of tools can be used with the way of working, assuming their functionality is sufficient.

The model design steps of the way of working provide a way to construct the models, using dynamical plant and control law models for the (loop control) component implementations. The way of working also consists of steps to verify and simulate the designed software models, thereby increasing the quality of the resulting software.

The way of working starts with designing the control software architecture. The actual implementation for the architectural components are filled in by later steps. This approach ensures that the system architecture is matching with the physical system and its separate concerns. Different component implementations can be used, as long as their interfaces match with the interface of the architectural model component, e.g. for design space explorations, simulations or debugging purposes.

The way of working is suitable for a wide range of cyber-physical systems, from complex medical systems to simple research setups. Scalability of the way of working is provided by making the steps partially optional, i.e. it is not required to perform each step to the letter. A certain step can be skipped completely, or several steps can be combined and performed at once, when allowed by the design complexity.

The software design trajectory combines and reuses information from the electronics, mechanics and controller design trajectories at several steps of the way of working. This increases the quality of the resulting control software further. Thereby, also improving the chances of a successful first-time-right design, as the control software is better integrated and matching with the other parts of the cyber-physical system.

7.1.2 Generic Architecture Component

The template GAC that is designed, tightly matches with the way of working and therefore increasing its value and usability. Due to its component-based nature, a GAC implementation is interchangeable with other implementations as long as their interfaces match, as described above. Although in the case of GACs, their functionality needs to match as well. For example, if a GAC expects certain user-defined commands to function properly, the replacement GAC needs to expect the same commands, otherwise the software will not execute properly.

The scalability and reusability of the GAC is obtained due to these three main reasons:

- The template GAC does not support all kinds of advanced functionality, like the Orocos component does. This makes it suitable for embedded applications requiring low-resource usage. On the other hand, advanced functionality is easily included using the provided hooks, which makes a specialised GAC suitable for more advanced applications if required. So the complete range of cyber-physical systems can be controlled using the generic architecture component.

- The variety of provided hooks allows the designer to add any required implementation to the specialised GAC without modifying the basics of the GAC. This makes the GAC suitable for any type of control application, ranging from real-

time loop controllers to complex tasks, like environment mapping, path finding or task planning.

- The configuration block increases the reusability of specialised GACs, as (small) differences between two controller components can be applied with the component configuration, without the need to designing separate components.

The separation of concerns approach that is applied to the GAC design, helps the designer to use the correct concern for the correct task. The coordination block for example contains all decision-making logic in the form of two state machines: a predefined lie-cycle state machine and a user-defined state machine to add more fine-grained used states to a specialised component. Support to validate incoming and outgoing signals is provided in the safety block, which has a hook so the user can easily add the required checks. This increases the probability of preventing unforeseen situations. As mentioned above, the configuration block increases the reusability of a components. An additional advantage is that it also helps in preventing design errors, as components are reused, they are likely to contain less errors that newly designed components.

All these separate advantages of using the GAC as basis for a component, increase the chances of success of the first-time-right approach of the way of working.

7.1.3 Framework and Tooling

The LUNA execution framework and the TERRA tool suite provide support to easily design, validate and test the software models.

The architecture model editor of TERRA tool suite provides support for design space exploration. Its resulting models typically describe the system architecture of the cyber-physical system and its software, completely separated from the actual implementation. Depending on the requirements, for example the model is used for a production implementation or used for testing or simulation purposes, the actual implementation can be chosen.

The component-based approach of LUNA is used to select the framework support that is required for the application. The LUNA hardware abstraction layer is used to provide specific support for the used hardware and operating system of different computing platforms. This all increases the scalability of the framework, making it suitable for the large range of systems.

The TERRA meta-models have helped a lot in the development of TERRA for various reasons. Using meta-models forces a developer to think about the model semantics before actually implementing the TERRA tool suite. Furthermore, due to the strict model definitions it is clear where and how a tool needs to obtain a required piece of information. The automatic model validation provided by EMF uses the semantics of the meta-model to determine the validation rules, so the model designer is indirectly helped by the meta-models and the meta-models help in preventing software errors.

7.1.4 Relevance

The way of working provides design steps that can be used by others to properly design the control software for a cyber-physical system. A practical tool implementation is provided by the combination of the GAC, LUNA and TERRA.

The way of working and accompanying support is supposed to decrease the design time of the control software and increase the quality of the resulting control software. Companies can use this work to improve their time-to-market requirement and keeping ahead of their competitors.

The academic community is able to embed the modelling point of view and its guidelines to teach student how to work structured. This help them understand and learn the concepts of modelling and its advantages, without struggling to solve all kinds of problems due to unclear models.

As Verhoef (2009) states

> "It seems that the well-known adage *'Price, Time, Quality - Pick Any Two for Success'* is still a fact of life"

This seems certainly true, but the way of working and the rest of this work tries to provide means to be able to '*pick any three*' though.

7.2 Recommendations

Even though it is concluded that the way of working, in combination with the GAC, LUNA and TERRA, helps to manage the complexity of the control software, improving the reusability and providing multiple other advantages, there are still possibilities to further improve their quality. The recommendations with the greatest impact are discussed here.

7.2.1 More Evaluation

The way of working completely covers the whole design trajectory and allows for different approaches depending on the requirements of the design of the cyber-physical system. Several different types of cyber-physical systems are described in Chapter 1, these are: medical, industrial, small embedded and research-related cyber-physical systems. The control software for all of these systems can be designed with the way of working. Nonetheless, this has not yet been explicitly tested for all these types of cyber-physical systems in different situations.

After the implementation of other use-cases and evaluation of the design process, it is likely that there are points found to further improve the way of working. When a more mature version of the way of working is obtained, its design steps also need to be tightened up in order to get a standardized structure which is better suited to be generically used.

This further improvement of the way of working results in higher software quality, further closing in to the first-time-right ideals. Additionally it also results in faster and

more efficient design of the control software, reducing the time-to-market period of new products.

7.2.2 Model Management

One of the reasons to use component interfaces in architecture models of the cyber-physical system, is to be able to change the actual implementation depending on the current usage of the model. With one or two components this is perfectly maintainable, but if the number of components is larger, switching all of the implementations becomes troublesome.

It is convenient to keep track of each set of implementations, so they can be switched as a whole, as described in Section 3.2.4. The DESTECS project resulted in a model-management system (Zhang and Broenink, 2013), which is used to be able to keep track of different versions or implementations of the same model consisting of multiple files. Such a model-management system can also be used to keep track of a set of component implementations, suitable for a specific implementation of an architecture model.

Extending the TERRA tool with support for a model-management system improves the usability of the way of working. It becomes easier to reuse the models for different use-cases and situations, thereby reducing the time it takes to perform simulations for example.

7.2.3 Model Optimisation

Even though design guidelines are provided to choose an appropriate level of detail for the control software, it is still hard to pick the correct level. Using a modelling point of view is still seen as a sort of 'utopia', it is supposed to make the model understandable with just a single glance. In practice these models are unsuitable for daily usage, due to the complexity and amount of used model elements.

Model optimisation techniques provide a way to change the level of detail from the modelling point of view to the execution point of view, or at least much more near this point of view. This makes the models more usable for practical situations.

Therefore, it is recommended to include these optimisation techniques in a model-to-model transformation tool (Bezemer et al., 2009; Boode et al., 2013). The model optimisation tool should be invoked automatically when certain tasks, like code generation of simulations, are started.

Such a tool could even adapt a third point of view, namely a simulation point of view, which defines a certain maximum hierarchical depth in the models. This is likely to improve the simulation quality as the complexity is reduces and the user is not bothered with the simulation results of the basic (component) implementations.

7.2.4 Simulation Support

The way of working states the importance of simulations and co-simulations. Unfortunately TERRA does not provide support for (co-)simulations, so control software

needs to be tested on the physical system. This has the disadvantages and danger that are discussed in sections 2.8, 3.1.2 and 3.2.4. A work-around for the lack of simulation support is provided in Section 6.7, but it is 'far from ideal'. Therefore, it is highly recommended to include simulation support into the TERRA tool suite.

Dankwoord

Als eerste wil ik Jan Broenink bedanken voor zijn begeleiding tijdens mijn promotie. Ik heb mijn promotie als een geweldige tijd met veel vrijheden ervaren, dit komt zeker door jouw aanpak in mijn begeleiding. Verder heb ik heel veel gehad aan je frisse blik tijdens onze discussies, dit gaf veelal een nog beter en aangescherpt resultaat. Naast jouw formele rol was er ook vaak tijd om te 'bomen' over allerlei zaken. Dit leverde een hoop leuke momenten op, zeker ook omdat onze humor goed op elkaar aansluit.

Stefano Stramigioli wil ik bedanken voor zijn rol als promotor en vakgroepvoorzitter. Maar ook voor jouw begrip en medeleven tijdens de moeilijke en zware periode thuis. Het was enorm fijn dat ik toen bij Marieke kon en mocht zijn zonder dat ik onder druk werd gezet om zo snel mogelijk weer terug aan het werk te moeten zijn!

En natuurlijk kan de rest van de vakgroep niet achter blijven: Mijn leuke tijd is grotendeels mogelijk gemaakt door de fantastische sfeer binnen de groep en tussen de collega's. Met name wil ik mijn kamergenoten Rob, Jeroen en Jitendra bedanken voor de gezelligheid en interessante gespreksonderwerpen. Robert wil ik bedanken voor de goede samenwerking en al je werk dat je in LUNA gestoken hebt: zonder jou zouden mijn resultaten er heel anders uitgezien hebben! Ook Carla, Jolanda, Marcel en Gerben bedankt voor jullie ondersteuning, zonder jullie zouden er een hoop dingen niet of veel later pas gebeuren! Marcel, sorry dat ik telkens maar weer langs kwam om te vragen of je de nieuwe software op de servers kon installeren of updaten...

Graag wil ik van deze mogelijkheid gebruik maken om ook mijn ouders Aad en Riet te bedanken voor jullie steun en vertrouwen in mij. Dit begon al op de basisschool waar ik geadviseerd werd om MAVO, misschien HAVO, te gaan doen en jullie vonden dat ik prima naar het VWO kon (wat ook zo bleek te zijn). Tijdens mijn studie hielden de adviezen aan om er de vaart in te houden en tijdens mijn promotie om te zien of ik wel op schema lag. Al was het in dit laatste geval wellicht wat overbodig ;)

Ook hebben jullie mij al vroeg gestimuleerd om verder na te denken over allerhande zaken. Zo heb ik het 'kurkentrekker boek' (*Over de werking van de kurketrekker en andere machines*, David Macaulay) al op jonge leeftijd gekregen en talloze keren gelezen. Dit heeft mijn academische blik op de wereld en mijn drang om te willen weten hoe de vork in de steel zit zeker geholpen!

En tot slot wil ik mijn lieve vrouw Marieke bedanken. Het is fijn dat ik ook thuis kon kletsen over de inhoudelijke zaken om zo geregeld nieuwe ideeën over mijn onderzoek te krijgen. Aan het einde van mijn promotie, toen ik druk met dit boekwerk bezig was, was het fijn dat ik hier mijn tijd in kon stoppen terwijl jij het overgrote deel van de huishoudelijke taken en onze verhuizing wilde regelen. Maar voornamelijk ben ik heel blij met onze liefde, de LOL die we vaak hebben (erg fijn is ook dat je mijn nerd-grapjes begrijpt) en dat je er altijd voor mij bent.

Bibliography

aicas (2012), JamaicaVM website.

Bezemer, M. M., M. A. Groothuis and J. F. Broenink (2009), Analysing gCSP Models Using Runtime and Model Analysis Algorithms, in *Communicating Process Architectures 2009*, volume 67, Eds. P. H. Welch, H. W. Roebbers, J. F. Broenink, F. R. M. Barnes, C. G. Ritson, A. T. Sampson, D. Stiles and B. Vinter, pp. 67 – 88, ISBN 978-1-60750-065-0, ISSN 1383-7575, doi: 10.3233/978-1-60750-065-0-67.

Bezemer, M. M., M. A. Groothuis and J. F. Broenink (2011a), Way of Working for Embedded Control Software using Model-Driven Development Techniques, in *IEEE ICRA Workshop on Software Development and Integration in Robotics (SDIR VI)*, Eds. D. Brugali, C. Schlegel and J. F. Broenink, IEEE, IEEE, pp. 6 – 11.

Bezemer, M. M., R. J. W. Wilterdink and J. F. Broenink (2011b), LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework, in *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, Eds. P. H. Welch, A. T. Sampson, J. B. Pedersen, J. M. Kerridge, J. F. Broenink and F. R. M. Barnes, IOS Press, Amsterdam, pp. 157 – 175, ISBN 978-1-60750-773-4, ISSN 1383-7575, doi: 10.3233/978-1-60750-774-1-157.

Bezemer, M. M., R. J. W. Wilterdink and J. F. Broenink (2012), Design and Use of CSP Meta-Model for Embedded Control Software Development, in *Communicating Process Architectures 2012, Dundee*, volume 69 of *Concurrent System Engineering Series*, Eds. P. H. Welch, F. R. M. Barnes, K. Chalmers, J. B. Pedersen and A. T. Sampson, Open Channel Publishing, pp. 185 – 199, ISBN 978-0-9565409-5-9.

Bischoff, R., U. Huggenberger and E. Prassler (2011), KUKA youBot - a mobile manipulator for research and education, in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 1 – 4, ISSN 1050-4729, doi: 10.1109/ICRA.2011.5980575.

Boode, A. H., H. J. Broersma and J. F. Broenink (2013), Improving the Performance of Periodic Real-time Processes: a Graph Theoretical Approach, in *Communicating Process Architectures 2013*, Eds. P. H. Welch, F. Barnes, J. F. Broenink, K. Chalmers, J. B. Pedersen and A. T. Sampson.

van Breemen, A. J. N. (2001), *Agent-Based Multi-Controller Systems - A design framework for complex control problems*, Ph.D. thesis, University of Twente, Enschede, The Netherlands.

BRICS Consortium (2013), BRIDE - the BRIcs Development Environment, website.

Brodskiy, Y., R. J. W. Wilterdink, S. Stramigioli and J. F. Broenink (2013), Collection of methods for achieving robust autonomy, Deliverable FP7 BRICS Project (231940) D6.2, University of Twente.

Broenink, J. F., M. A. Groothuis, P. M. Visser and M. M. Bezemer (2010a), Model-Driven Robot-Software Design Using Template-Based Target Descriptions, in *ICRA 2010 workshop on Innovative Robot Control Architectures for Demanding (Research) Ap-*

plications, Eds. D. Kubus, K. Nilsson and R. S. Johansson, IEEE, IEEE, pp. 73 – 77.

Broenink, J. F., Y. Ni and M. A. Groothuis (2010b), On Model-driven Design of Robot Software using Co-simulation, in *SIMPAR, Workshop on Simulation Technologies in the Robot Development Process*, Ed. E. Menegatti, ISBN 978-3-00-032863.

Brooks, R. (1986), A robust layered control system for a mobile robot, *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14 – 23, ISSN 0882-4967, doi: 10.1109/jra.1986.1087032.

Brown, N. C. C. (2007), C++CSP2: A Many-to-Many Threading Model for Multicore Architectures, in *Communicating Process Architectures 2007*, Eds. A. A. McEwan, W. Ifill and P. H. Welch, pp. 183 – 205, ISBN 978-1586037673.

Brugali, D. and P. Scandurra (2009), Component-based robotic engineering (Part I): Reusable building blocks, *Robotics Automation Magazine, IEEE*, vol. 16, no. 4, pp. 84 – 96, ISSN 1070-9932, doi: 10.1109/MRA.2009.934837.

Bruyninckx, H. (2001), Open robot control software: the OROCOS project, in *Robotics and Automation (ICRA), 2001. IEEE International Conference on*, volume 3, IEEE, pp. 2523 – 2528, ISBN 0-7803-6578-X, ISSN 1050-4729, doi: 10.1109/ROBOT.2001.933002.

Buys, K., S. Bellens, W. Decre, R. Smits, E. Scioni, T. de Laet, J. de Schutter and H. Bruyninckx (2011), Haptic coupling with augmented feedback between two KUKA Light-Weight Robots and the PR2 robot arms, in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pp. 3031 – 3038, ISSN 2153-0858, doi: 10.1109/iros.2011.6094925.

Charest, M. and B. Stecher (2011), Tick-tock - Understanding the Neutrino micro kernel's concept of time, Part II.

Clegg, D. and R. Barker (1994), *Case Method Fast-Track: A Rad Approach*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 020162432X.

Controllab Products (2012), 20-sim - Graphical modeling and simulation tool.

Cooling, J. (2003), *Software Engineering for Real-Time Systems*, Pearson Education Ltd, Essex, England, ISBN 0-201-59620-2.

Dao, P. B. (2011), *Safe-guarded multi-agent control for mechatronic systems: implementation framework and design patterns*, Ph.D. thesis, University of Twente, Enschede.

Decho Corp. (2012), Mordor website.

Dijkstra, E. W. (1972), urls on structured programming, in *Structured programming*, Eds. O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Academic Press Ltd., London, UK, chapter 1, pp. 1 – 82, ISBN 0-12-200550-3.

Fainelli, F. (2008), *The OpenWrt embedded development framework*, Free and Open source Software Developers' European Meeting (FOSDEM).

Formal Systems (Europe) Limited (2012), FDR2.

France, R. and B. Rumpe (2007), Model-driven Development of Complex Software: A Research Roadmap, in *2007 Future of Software Engineering*, IEEE Computer Society,

Washington, DC, USA, FOSE '07, pp. 37–54, ISBN 0-7695-2829-5, doi: 10.1109/fose.2007.14.

Groothuis, M. A., R. M. W. Frijns, J. P. M. Voeten and J. F. Broenink (2009), Concurrent Design of Embedded Control Software, in *Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling (MPM2009)*, volume 21 of *Electronic Communications of the EASST journal*, Eds. T. Margaria, J. Padberg, G. Taentzer, T. Levendovszky, L. Lengyel, G. Karsai and C. Hardebolle, EASST, ECEASST, ISSN 1863-2122.

Groothuis, M. A., J. J. P. van Zuijlen and J. F. Broenink (2008), FPGA based Control of a Production Cell System, in *Communicating Process Architectures 2008,*, volume 66 of *Concurrent Systems Engineering Series*, IOS Press, Amsterdam, pp. 135 – 148, ISBN 978-1-58603-907-3, ISSN 1383-7575, doi: 10.3233/978-1-58603-907-3-135.

Heemels, M. P. M. H. and G. Muller (2006), *Boderc: Model-based design of high-tech systems*, Embedded Systems Institute, Eindhoven, The Netherlands, ISBN 90-78679-01-8.

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice-Hall, London, ISBN 0-131-53271-5.

Hoogendijk, T. A. (2013), *Design of a generic software component for embedded control software using CSP*, MSc thesis 014RAM2013, Robotics and Mechatronics, University of Twente.

Isermann, R., J. Schaffnit and S. Sinsel (1999), Hardware-in-the-loop simulation for the design and testing of engine-control systems, *Control Engineering Practice*, vol. 7, no. 5, pp. 643 – 653, ISSN 0967-0661, doi: 10.1016/S0967-0661(98)00205-6.

Jovanović, D. S. (2006), *Designing dependable process-oriented software, a CSP approach*, Ph.D. thesis, University of Twente, Enschede, The Netherlands.

Jovanović, D. S., G. H. Hilderink and J. F. Broenink (2002), A Communicating Threads -CT- case study: JIWY, in *Communicating Process Architectures 2002*, Eds. P. W. J. Pascoe, P. H. Welch, R. Loader and V. Sunderman, IOS Press, Concurrent Systems Engineering 60, pp. 321 – 330.

Jovanović, D. S., B. Orlic, G. K. Liet and J. F. Broenink (2004), gCSP: A Graphical Tool for Designing CSP Systems, in *Communicating Process Architectures 2004*, volume 62, Eds. I. East, J. Martin, P. Welch, D. Duce and M. Green, IOS press, Amsterdam, pp. 233 – 252, ISBN 1-58603-458-8, ISSN 1383-7575.

Klotzbücher, M., N. Hochgeschwender, L. Gherardi, H. Bruyninckx, G. K. Kraetzschmar, D. Brugali, A. Shakhimardanov, J. Paulus, M. Reckhaus, H. Garcia, D. Faconti and P. Soetens (2013), The BRICS Component Model: a Model-Based Development Paradigm For Complex Robotics Software Systems, in *Symposium On Applied Computing*, ACM, 28.

Kolovos, D., L. Rose and R. Paige (2012), *The Epsilon Book*.

Kopetz, H. (1997), *Real-Time Systems - Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, ISBN 0-7923-9894-7.

Küster, J. M. (2006), Definition and validation of model transformations, *Software*

and Systems Modeling, vol. 5, pp. 233 – 259, ISSN 1619-1366, doi: 10.1007/s10270-006-0018-8.

Kranenburg-de Lange, D. J. B. A. (2012), *Dutch Robotics Strategic Agenda - Analysis, Roadmap & Outlook*, ISBN 978-94-6191-322-7.

Lausdahl, K. G., A. Ribeiro, P. M. Visser, F. N. J. Groen, Y. Ni, J. F. Broenink, A. H. Mader, J. W. Coleman and P. G. Larsen (2011), D3.3b — Co-simulation Foundations, Technical report, The DESTECS Project (INFSO-ICT-248134).

Lootsma, M. (2008), *Design of the global software structure and controller framework for the 3TU soccer robot*, MSc thesis 014CE2008, University of Twente.

Mallet, A., C. Pasteur, M. Herrb, S. Lemaignan and F. Ingrand (2010), GenoM3: Building middleware-independent robotic components, in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pp. 4627 – 4632, ISSN 1050-4729, doi: 10.1109/robot.2010.5509539.

Moody, D. and J. Hillegersberg (2009), Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams, in *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, Eds. D. Gašević, R. Lämmel and E. Wyk, Springer Berlin Heidelberg, pp. 16 – 34, ISBN 978-3-642-00433-9, doi: 10.1007/978-3-642-00434-6_3.

Ni, Y. and J. F. Broenink (2012), Hybrid systems modelling and simulation in DESTECS: A co-simulation approach, in *The 2012 European simulation and modelling conference*, Ed. M. Klumpp, ETI-The European Technology Insititue, pp. 32 – 36, ISBN 978-90-77381-73-1.

Object Management Group (2006), *Lightweight Corba Component Model (LCCM)*, OMG, chapter 13.

Object Management Group (2011), Unified Modeling Language (UML), Technical report, OMG.

OpenWrt developer group (2012), OpenWrt website.

Orlic, B. and J. F. Broenink (2004), Redesign of the C++ Communicating Threads Library for Embedded Control Systems, in *5th PROGRESS Symposium on Embedded Systems*, Ed. F. Karelse, STW, Nieuwegein, NL, pp. 141–156.

Petre, M. (1995), Why looking isn't always seeing: readership skills and graphical programming, *Communications of the ACM*, vol. 38, no. 6, pp. 33–44, ISSN 0001-0782, doi: 10.1145/203241.203251.

Pierce, K. G., C. J. Gamble, Y. Ni and J. F. Broenink (2012), Collaborative Modelling and Co-Simulation with DESTECS: A Pilot Study, in *3rd IEEE track on Collaborative Modelling and Simulation, in WETICE 2012*, IEEE-CS, pp. 1 – 6.

Plotkin, G. D. (2004), The origins of structural operational semantics, *Journal of Logic and Algebraic Programming*, vol. 60 – 61, pp. 3 – 15, ISSN 1567-8326, doi: 10.1016/j.jlap.2004.03.009.

Posthumus, R. (2007), *Data logging and monitoring for real-time systems*, MSc thesis 015CE2007, Control Laboratory, University of Twente.

Ptolemy (2012), Ptolemy II website.

QNX Software Systems (2012), QNX website.

Quigley, M., K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler and A. Y. Ng (2009), ROS: an open-source Robot Operating System, in *ICRA Workshop on Open Source Software*.

ROS Sensors (2012), ROS Wiki - Sensors.

Roscoe, A. W., C. A. R. Hoare and R. Bird (1997), *The Theory and Practice of Concurrency*, Prentice Hall PTR, Upper Saddle River, NJ, USA, ISBN 0136744095.

Rubel, D., J. Wren and E. Clayberg (2011), *The Eclipse Graphical Editing Framework (GEF)*, The Eclipse Series, Addison-Wesley Professional, ISBN 9780321718389.

Scattergood, B. (1998), *The Semantics and Implementation of Machine-Readable CSP*, Ph.D. thesis, University of Oxford.

Smits, R. and H. Bruyninckx (2011), Composition of Complex Robot Applications Via Data Flow Integration, in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 5576 – 5580, ISSN 1050-4729, doi: 10.1109/icra.2011.5979958.

Soetens, P. (2012), The Orocos Component Builder's Manual.

Sözer, H. (2009), *Architecting fault-tolerant software systems*, Ph.D. thesis, University of Twente, Enschede, iPA Dissertation 2009-05.

van der Steen, T. T. J., M. A. Groothuis and J. F. Broenink (2008), Designing Animation Facilities for gCSP, in *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering Series*, IOS Press, Amsterdam, p. 447, ISBN 978-1-58603-907-3, ISSN 1383-7575, doi: 10.3233/978-1-58603-907-3-447.

Steinberg, D., F. Budinsky, M. Paternostro and E. Merks (2009), *EMF: Eclipse Modeling Framework 2.0*, The Eclipse Series, Addison-Wesley Professional, 2nd edition, ISBN 0321331885.

Sung, G. T. and I. S. Gill (2001), Robotic laparoscopic surgery: a comparison of the da Vinci and Zeus systems., *Urology*, vol. 58, no. 6, pp. 893 – 898, ISSN 1527-9995.

The MathWorks (2012), Automatic Code Generation - Simulink Coder.

Varró, D., M. Asztalos, D. Bisztray, A. Boronat, D. Dang, R. Geiß, J. Greenyer, P. Van Gorp, O. Kniemeyer, A. Narayanan, E. Rencis and E. Weinell (2008), Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools, in *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture urls in Computer Science*, Eds. A. Schürr, M. Nagl and A. ZÃijndorf, Springer Berlin / Heidelberg, pp. 540 – 565, ISBN 978-3-540-89019-5, doi: 10.1007/978-3-540-89020-1_36.

Veldhuijzen, B. (2009), *Redesign of the CSP execution engine*, MSc thesis 036CE2008, Control Engineering, University of Twente.

Verhaar, C. A. (2008), *An integrated embedded control software design case study using Ptolemy II*, MSc thesis 11CE2008, University of Twente.

Verhoef, M. (2009), *Modeling and Validating Distributed Embedded Real-Time Control Systems*, Ph.D. thesis, Radboud University.

Verhoef, M., B. Bos, P. van Eijk, J. Remijnse, E. Visser, M. De Paepe, Y. De Witte, K. Rombaut and R. Van Lembergen (2012), Industrial Case Studies - Final Report, Technical report, DESTECS.

Welch, P., G. Justo and C. Willcock (1993), Higher-Level Paradigms for Deadlock-Free HighPerformance Systems, in *Transputer Applications and Systems ”93, Proceedings of the 1993 World Transputer Congress*, IOS Press, pp. 981 – 1004.

Welch, P. H. and F. R. M. Barnes (2008), A CSP Model for Mobile Channels, in *Communicating Process Architectures 2008, York*, volume 66 of *Concurrent Systems Engineering Series*, Eds. P. H. Welch, S. Stepney, F. Polack, F. R. M. Barnes, A. A. McEwan, G. S. Stiles, J. F. Broenink and A. T. Sampson, IOS Press, pp. 17 – 33, ISBN 978-1-58603-907-3, doi: 10.3233/978-1-58603-907-3-17.

Welch, P. H., N. C. C. Brown, J. Moores, K. Chalmers and B. Sputh (2007), Integrating and Extending JCSP, in *Communicating Process Architectures 2007*, Eds. A. A. McEwan, W. Ifill and P. H. Welch, pp. 349 – 369, ISBN 978-1586037673.

Wijbrans, K. C. J. (1993), *Twente Hierarchical Embedded Systems Implementation by Simulation*, Ph.D. thesis, University of Twente, The Netherlands.

Zhang, X. and J. F. Broenink (2013), A Concurrent Design Approach and Model Management Support to Prevent Inconsistencies in Multidisciplinary Modelling and Simulation, in *19th European Concurrent Engineering Conference*, Ed. P. Geril, EUROSIS, EUROSIS-ETI Publication, pp. 21–28, ISBN 978-90-77381-77-9.

Index

20-sim, **13**
5Cs, **40**
  Communication, 46
  Composition, 46
  Computation, 44
  Configuration, 46
  Coordination, 44

application programming interface, 21, 35
architecture abstraction components, 65
architecture editor, 31

BRICS Component Model, 23, **40**, 86
buffered communication, 16, 72

co-simulation, **19**, 25
code generation, 18, 29, 32
Communicating Sequential Processes, **16**
compilation, 20, 30, 32
component interface, **16**, 31, 35, 53
component life cycle, 34, 44
computing platform, **3**, 29
conforms to, 91
control algorithm design, 23
cross-compilation, 20
cyber domain, 3
cyber-physical system, 2, **2–3**

deadlock, 16
deployment manager, 30

Eclipse Model Framework, 86
editor, 31
electrical domain, 3, 24
embedded control software, 3
Error Detector, 51
ERROR event, 45, **48**
error handling
  global, 12, 48
  hybrid, 12
  local, 12, 48
Error state, 45
ERROR_READY event, 45
execution engine, **20**, 40
  CSP, 67
execution point of view, 53
extension points, 96

FDR2, **13**
first time right, 5, 23
formal verification, 16
Formalisms, Techniques, Methods and Tools, 6
framework, 21, 30, 32, 35

Generic Architecture Component, 34, **37**
  architectural network, 46
  hierarchical network, 46
  specialised, *see* specialised GAC
  template, *see* template GAC
Global Error, 51
Graphical Eclipse Framework, 87, **96**

hardware abstraction layer, 21, 35
hardware-in-the-loop simulation, 19
hooks, 39

I/O hardware, 3
implementation interface, **16**, 32
Init state, 45
INIT_READY event, 45
InitReady state, 45

life cycle, *see* component life cycle
livelock, 17
LUNA component levels
  Core Components, 65
  Execution Engine Components, 65
  High-level Components, 65

mechanical domain, 3, 24
mechatronic system, 2
meta-model, **14**
mission-critical, 10
model management tool, 34
model optimisation, 18, 29
model-driven development, 12
model-to-code transformation, 18, 32
model-to-model transformation, 18, 33
model-to-text transformation, 18
modelling point of view, 34, 53

operating system, 41, **62**
OperationCallers, 40
Operations, 40
OS abstraction components, 65
OS thread, 41, **66**

physical domain, 3
plant, 3
plug-in, **89**
POSIX, 66
priority degradation, 47
protected regions, 99

reader process, 16
rendezvous communication, 16, 72
robotic system, 2
RUN event, 45
Run state, 45

safety, 11
Safety concern, 48
separation of concerns, *see also* 5Cs, 67
Services, 40
simulation, **18**, 25
simulation scenarios, 34
software architecture design, 23, **26**
source code, 20, 32
specialised GAC, **37**
static code, 32, 35
STOP event, 45
Stop state, 45
STOP_READY event, 45

target connector, 29
TaskContext, 40
template GAC, **37**
text-to-model transformation, 18
thread safe, 72
threading implementation, 67
tool suite, **85**

unbuffered communication, *see* rendezvous communication
user thread, **66**
user-defined states, 45, 49
Utility components, 65

visual inspection, 28

way of working, 4, 23
writer process, 16

PROPOSITIONS

belonging to the thesis

Cyber-Physical Systems Software Development

way of working and tool suite

1. Multi-threaded work is only possible with a good way of working to tackle the synchronisation problems.
2. The time aspect of a real-time requirement only gets its real significance when the criticality aspect is known.
3. The choice between an essential but basic, and a feature-rich but complex component model is similar to the choice between Linux and Windows.
4. Models are very nice in theory, but one cannot do much with them in practice.
5. Modularity improves reusability both in engineering as in daily life, a practical example is a modular cabinet: After moving to our new home it has been reused all over the place.
6. Defining a universal way of working to design universal software for universal cyber-physical systems is quite an accomplishment.
7. The utopia of perfect software structure combined with the utopia of first-time-right designs, does make the dream more realistic.
8. Being able to let go of perfectionism is a virtue.
9. Taking shortcuts requires much more experience: Only people who feel old enough are qualified to take them.
10. This thesis would not have been necessary if people continued building simple robotic systems for single tasks.

Maarten Bezemer
14 november 2013

Designing embedded software for modern cyber-physical systems becomes more and more difficult due to the increasing amount and the complexity of their requirements. The regular requirements are extended by more complex requirements to get a cyber-physical system capable of fulfilling a variety of different, ad-hoc tasks and having interactions with humans. A typical example of cyber-physical systems, which have these requirements, are medical robotic systems used in surgeries.

The essential goal of this research is to provide a way of working for the design of the control software for cyber-physical systems. It uses model-driven design (MDD) techniques to reduce the complexity of the control software design. This way of working is supported by a component blue-print, an execution framework and tooling support.

The combination of the way of working and the additional support has been tried out in a laboratory and educational setting, and the experiences with it are promising.

ISBN 978-90-365-1879-6

UNIVERSITY OF TWENTE.