University of Twente

EEMCS / Electrical Engineering
Control Engineering

Design of a hard real-time, multi-threaded and CSP-capable execution framework

Robert Wilterdink

MSc report

Supervisors:
prof.dr.ir. S. Stramigioli
dr.ir. J.F. Broenink
ir. E. Molenkamp
ir. M.A. Groothuis
ir. M.M. Bezemer

June 2011

Report nr. 009CE2011
Control Engineering
EE-Math-CS
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Summary

Nowadays, embedded systems are designed for multiple and more demanding tasks, whereas previously they had a single use case and were less demanding. One can think of automated robot vacuum cleaners in the past versus complete household robots presently. This progress is possible due to improved hardware architectures, with multiple parallel cores/processors, and increased resources for embedded systems. The formal language Communicating Sequential Processes (CSP) is designed to aid development of such parallel computing software.

In the past, the CTC++ framework was developed to implement CSP models on embedded systems. This framework was designed for single core/processor architectures and its design now is outdated.

To design a new framework, which can also fully utilize future embedded architectures, state of the art frameworks and libraries were investigated. They are analyzed on their software architecture, platform independence, timing and real-time capabilities, and scalability and extensibility. A comparison on these subjects is included as well.

The LUNA framework described in this report is a hard real-time, multi-threaded, multi-platform, CSP capable and a component based framework. Many of its components can be enabled/disabled separately to achieve high scalability.

For the first framework implementation the hard real-time QNX operating system (OS) is chosen, since it includes CSP similar rendezvous channels. But, as it turned out, QNX' rendezvous channels are not usable for some of the rather advanced CSP rendezvous communication operations.

Additionally, a distinctive implemented feature is the real-time logger. It can be used to transfer log messages, (a)periodic (control) signals and CSP execution traces to a development PC for further processing, without destroying the real-time constraints.

To analyze LUNA's performance a simple robotic setup is implemented. The results show that the new CSP design is more efficient and faster than the CT library.

From the foregoing, it can be concluded that LUNA is suitable to develop real-time control applications, with or without CSP, for a multitude of hardware architectures and OSs. The main recommendations are to implement LUNA for other real-time OSs as well and to implement a graphical modelling tool, such as gCSP, to generate LUNA code from complex models for robotics applications. Besides this, the visualisation of CSP traces recorded with the RTLogger should be improved, as the data mining for these is still a lot of work. Perhaps, other setups in the lab should also be implemented with LUNA to verify its performance extensively.

Samenvatting

Tegenwoordig worden embedded systemen ontworpen voor zowel meerdere als veeleisendere taken. Vroeger hadden deze echter een enkele toepassing en waren ze minder veeleisend. Men kan denken aan geautomatiseerde robotstofzuigers in het verleden versus volledig geautomatiseerde huishoudrobots op dit moment. Deze vooruitgang is mogelijk gemaakt door verbeterde hardware-architecturen met meerdere parallele cores/processors en meer rekencapaciteit voor embedded systemen. De formele taal Communicating Sequential Processes (CSP) is ontworpen met als doel het vereenvoudigen van het ontwikkelen van dergelijke paralelle software structuren.

In het verleden is het CTC++ framework ontwikkeld om CSP modellen te implementeren op embedded systemen. Dit framework is echter ontworpen voor enkelvoudige core/processor architecturen. Het ontwerp is nu achterhaald.

Om tot een nieuw framework te komen, dat ook volledig benut kan worden op toekomstige embedded architecturen, zijn moderne frameworks en bibliotheken onderzocht. Ze zijn geanalyseerd op hun software architecture, platform onafhankelijkheid, timing en real-time eigenschappen en tot slot schaalbaarheid en mogelijkheid tot uitbreiding. Het verslag bevat tevens een uitgebreide vergelijking van deze onderwerpen.

Het LUNA framework zoals beschreven in dit rapport is een hard real-time, multi-threaded, multi-platform, CSP-capabel en component-gebaseerd framework. Een groot deel van de componenten kan apart in- en uitgeschakeld worden waarmee een hoge schaalbaarheid wordt bereikt.

Voor een eerste framework implementatie is het hard real-time QNX besturingssyteem gekozen, omdat deze CSP-vergelijkbare rendezvous communicatie kanalen biedt. Uiteindelijk bleken deze QNX rendezvous kanalen alsnog niet geschikt voor de redelijk geavanceerde CSP rendezvous communicatie.

Een interessante extra feature is de real-time logger. Deze kan log berichten, (a)periodieke (regelaar) signalen en de CSP executie volgorde doorsturen naar een ontwikkel PC voor verdere analyse zonder dat het real-time gedrag wordt beïnvloed.

Om de LUNA prestaties te kunnen analyseren is een eenvoudige robot setup geïmplementeerd. De verkregen resultaten tonen aan dat het nieuwe CSP ontwerp efficiënter en sneller is dan de CT-bibliotheek.

Uit het voorgaande wordt geconcludeerd dat LUNA geschikt is voor het implementeren van real-time control applicaties, met of zonder CSP en voor een groot aantal hardware-architecturen en besturingssystemen. De belangrijkste aanbevelingen voor LUNA zijn het implementeren van meer real-time besturingssystemen, maar ook het bieden van ondersteuning in een grafische modelleringstool zoals gCSP. Dit laatste heeft als doel de LUNA code te kunnen genereren van complexe software modellen voor robotica applicaties. Daarnaast moet de visualisatie van CSP executies worden verbeterd, omdat het verwerken van de CSP executies nog veel werk kost. Om de prestaties van LUNA onomstotelijk vast te stellen zouden eventueel ook andere opstellingen in het lab kunnen worden geïmplementeerd.

Preface

This thesis marks the end of my Computer Science and Electrical Engineering studies at the University of Twente. The choice for the additional Electrical Engineering master originates from my interest in mechatronics and robotics. The combination of both hopefully gives me an extra challenge.

Looking back, the experiences gained during both studies, among others my internship in England, and extracurricular activities made me the person who I am today. The extracurricular activity I have perceived as most educational and enjoyable, was being part of the Arashi board 2007-2008 (sports association). Over all, it was a pleasant period of my life in which a lot of good memories were made.

I would like to thank my graduation committee dr.ir. Jan Broenink, ir. Bert Molenkamp, ir. Marcel Groothuis and ir. Maarten Bezemer for their support on this assignment. Special thanks go out to my daily supervisor, Maarten Bezemer, for the many interesting discussions we had on the development of the software framework and my thesis. Writing a paper together and presenting the framework on the CPA 2011 conference was a nice bonus for me.

Thanks also go to Raymond, Youri and the other students for their tips, discussions and the good times during this project. I have enjoyed this very much and it has helped me to accomplish this thesis.

Most of all, I want to thank my parents and my girlfriend, Alette Beusink, for giving me the opportunity to do two studies and providing me with the their support and (extreme) patience during my study.

Robert Wilterdink

Hengelo, June 2011

Contents

1 Introduction

1.1 Context

Nowadays embedded systems are designed for multiple and more demanding tasks, whereas previously they usually had a single use case and were less demanding. One can think of automated robot vacuum cleaners in the past versus complete household robots presently. So, an embedded system is a complete device which not only contains electronics and mechanical parts, but also a (special-purpose) computer, which organizes the various tasks and execute their functions to control the hardware in a safe and efficient manner. These increased requirements obviously result in more complex control software, as the software is responsible for the 'smart' behavior of the device.

The Control Engineering (CE) group at the University of Twente deals with the realisation of complex (embedded) control structures for mechatronic setups. These mechatronic setups currently range from humanoid robots to miniature production plants to autonomous pipe inspection robots. On a software level these setups require, amongst others, hard real-time computing constraints, multiple parallel processes and sophisticated design patterns which facilitate safe and easy software development.

A proven approach, to coordinate several computational entities running at the same time (i.e. a concurrent system), is the process algebra Communicating Sequential Processes (CSP) developed by Hoare (1985) and Roscoe et al. (1997). CSP theory can also be used to prove correctness, deadlock freeness and lifelock freeness of a model.

In the past a graphical modelling tool, called gCSP (Jovanovic et al., 2004), has been developed at the CE group to aid modelling of concurrent systems with CSP. This tool is able to generate code from a model which can be executed on a PC or on a (real-time) hardware target. The generated code makes use of the Communicating Threads (CTC++) library (Hilderink et al., 1997; Orlic and Broenink, 2004), which provides a framework for the CSP language. During development one can also monitor the application in gCSP through an animation facility (Steen, van der et al., 2008). The complete overview is shown in Figure 1.1.

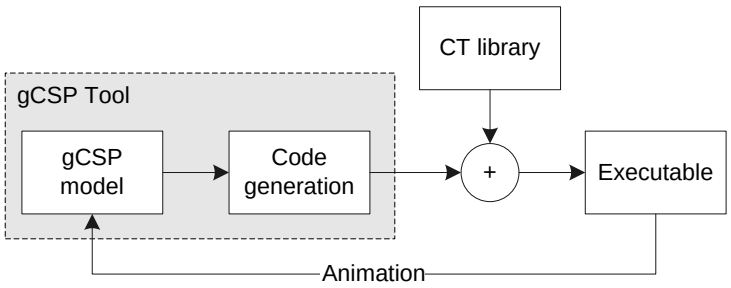

Figure 1.1: Overview of the gCSP and CT framework

The current CT library is a product of many years of research and development, but unfortunately the library has become outdated:

- The CT library was designed with single CPU/core embedded architectures in mind. Whereas nowadays embedded systems with multiple CPUs and cores are almost common practice. Unfortunately, the current CT library design cannot make (optimal) use of these new architectures, which is a pity since CSP was designed especially for this use case.

- The scheduler and all processes run in the same software thread. The Operating System (OS) does not cooperate with these internal processes. The CT library therefore intends to deliver any external events like timer interrupts to the appropriate process. The current scheduler cannot guarantee when this event is handled. This sometimes results in inconsistent and inadequate timing behavior.
- The years of ongoing development did not benefit the software quality and obfuscated the original design.

For modern frameworks it is advantageous to employ an OS, because common functionality, such as parallel processing and network facilities, is already available and therefore will considerably shorten development time. The use of an OS has become a viable choice, because computing resources have increased a lot since the development of the CT library. Unfortunately, not all OS' are hard real-time capable, but for example QNX (QNX Software Systems, 2011) is. In addition QNX offers rendezvous communication channels which are similar to CSP rendezvous channels.

1.2 Goals and approach of the project

The goal of this project is to design a new hard real-time, multi-threaded and CSP-capable execution framework. Furthermore, the framework should be platform independent.

The QNX operating system offers rendezvous communication channels by default and, therefore, seems to offer a good base for CSP implementations, which will hopefully lighten the task at hand. As a result, the main research question to be answered in this thesis: Is QNX a valid choice to implement CSP based communicating threads? The idea to use QNX is not new (Veldhuijzen, 2009). The difference is that a complete resigned is considered and not only a reimplementation, which limited the design choices considerably. Furthermore, a non real-time CSP framework (C++CSP2, 2009) is also investigated.

In the future the framework should be usable on different platforms to make it wider applicable. Multiple framework projects have solved this problem in different manners, without a clear solution yet. Therefore a subgoal of this thesis is to determine how a platform independent framework should be designed. This will be accomplished by analyzing state of the art frameworks on their software architecture, platform independence, timing and real-time capabilities, and scalability and extensibility. The gained knowledge is then applied for the new framework design.

The current implementation of the CT library 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. This is an obstacle to use the library from a generic robotics point of view and might result in ignoring the CT library altogether. Furthermore, a loose coupling between CSP processes and the rest of the framework will probably enable more research opportunities. Thus, the second subgoal of this thesis is to investigate whether the CSP language implementation and framework can be loosely coupled. The results from investigating the CT library and the non real-time C++CSP2 framework are used for this part of the design as well.

The approach of the project is then as follows. First, the requirements for the new framework are determined. Then, state of the art libraries are investigated to gain insight in framework design in general. Since the design should be platform independent, the implementation should balance QNX specific benefits and generic OS concepts. The CSP execution engine then extends these to achieve platform independence. Furthermore, the CSP execution engine should be designed such that it is only loosely coupled to the rest of the framework. At the end, a real robotic setup is implemented with the new framework to determine its usability, efficiency, performance and future improvements.

The main question to be answered in this thesis:
Is QNX a valid choice to implement CSP based communicating threads?

Minor questions:

a How should a platform independent framework be designed?
b How can the CSP execution engine be designed, such that the CSP implementation is loosely coupled to the rest of the framework?

1.3 Evaluation objectives

The project can be qualitatively evaluated with the research questions posed in the previous section.

The quantitative evaluation will be performed with a real robotic setup with two degrees of freedom. Using this setup hard real-time properties can be empirically proven and the new framework can be compared with the CT library, which also has been used with this setup in the past. The CSP community's (standard) Commstime test is used to compare the performance also with a non real-time CSP library.

1.4 Thesis outline

Chapter 2 presents background information, such as OS concepts, an introduction to the QNX OS and the CE group's embedded software design methodology. Chapter 3 then presents the requirements for LUNA. Chapter 4 gives the framework and library research results and the recommendations deduced from these. Next, the complete design and implementation of LUNA is discussed. Chapter 6 evaluates the LUNA design by reviewing its requirements and discussing performance tests. Furthermore, it concludes on the research questions mentioned in this chapter. The last chapter concludes to LUNA's overall design and the attained goals, and gives recommendations for future work. Furthermore, the paper (Bezemer et al., 2011b), on the combination of the LUNA framework and CSP, is included in Appendix I.

2 Background

This chapter provides background information. Section 2.1 explains the difference between frameworks and libraries. Next, the considered hardware architectures in this thesis are discussed. Section 2.3 explains general Operating System (OS) concepts. Then hard, soft and non real-time are explained in Section 2.4. Following, the hard real-time QNX OS will be shortly introduced. Section 2.6 presents common testing methods and levels in software engineering. Last, Section 2.7 discusses the design methodology used in the CE group.

2.1 Frameworks versus libraries

A library is essentially a set of reusable functions, usually organized in classes. Each function invocation performs some work and then returns control to the client. A library, therefore, is a set of components to develop an application with.

A framework embodies a reusable abstract design, which should be extended on specific points. The framework's code then calls this code at these points, to achieve the intended framework behavior. A framework therefore is a set of components and design patterns, which specify the behavior of an application.

2.2 Hardware

The CE group currently has multiple custom setups and demonstrators, which are controlled by a few different standard hardware architectures. For this thesis, these hardware architectures can be divided in two groups:

1. *x86/PowerPC/ARM architectures, with enough resources to run a real-time OS*
   These architectures in combination with enough resources (such as memory) are able to run general purpose OSs as well as real-time OSs. The x86 and PowerPC architectures are also employed in consumer computers. The ARM architecture is mostly used in embedded setups or handhelds.

   The OS capable hardware requirement facilitates framework design, since a lot of required functionality is already being taken care of by the OS. For example multi-threading and common user input/output facilities. Furthermore, functionality non-essential to most applications, such as debugging support, will also be present. Both will shorten development time considerably.
2. *Other architectures, such as FPGAs*
   This category comprises of all hardware architectures not capable of running a general purpose OS nor real-time OS, such as FPGAs and AVR micro controllers. These are mostly used for small setups or for dedicated functions of a setup, like PWM signal generation.

The PC/104-stack mentioned in Chapter 6 is an example of a small x86 form-factor embedded computer with additionally various I/O boards. One of the I/O boards comprises a FPGA to implement multiple encoder counters and PWM signal generators.

Although in this thesis explicitly an OS and OS capable hardware are used, the ideas discussed are in principle not limited to any architecture or OS.

2.3 Operating Systems

An Operating System (OS) is a collection of software, consisting of programs and data, that runs on computers, manages computer hardware resources, and provides common services for execution of various application software. The OS acts as an intermediate between application programs and the computer hardware, although the application code is usually executed directly by the hardware and will frequently call the OS for support or be interrupted by it. The latter, for instance, can be the case when multiple programs time-share a CPU.

2.3.1 Organization

Modern OSs use a kernel-based architecture. A kernel has facilities to receive resource requests and grant access to resources such as allocating space for a new file or creating a new process. So, the kernel is the gate-keeper to the computer's resources. Applications use system calls to interface with a kernel.

Except for single purpose OSs, all modern OSs are both multitasking and multi-user. Since computers are limited by the number of cores/CPUs available, the OS must perform a trick to actually perform more than one (or a few) tasks at a time. The kernel uses time sharing for each available core/CPU at a high speed, such that it appears that the computer is multitasking for multiple users. Because the system is sharing resources a gate-keeper is needed for two primary reasons; Facilitating time sharing and to make sure users do not violate other users' resources.

Time sharing is accomplished by context switching, which consists of the following steps:

1. Save the current processes context: registers and program execution (counter).
2. Load another process' context and perform its operations from the point it was interrupted.
3. After the allotted time is over[1], repeat from beginning.

The kernel also has three other resource management tasks, besides time sharing a core/CPU:

- *Servicing requests from applications*
  When any application requires resources, it must access the resource via the kernel indirectly. The kernel processes the request and returns the result. For example, on a memory allocation request the kernel allocates the memory to the process and makes sure that the allocated memory is not in use.
- *Servicing system interrupts*
  Computer systems have a set of hardware and software interrupts. An interrupt interrupts the current process because of a condition. An interrupt can be for instance a timer event, i.e. after a pre-specified time has elapsed, the OS is notified of this.
- *Managing system resources*
  Kernels also performs internal management, that is not directly related to services. The kernel has to keep track of what resources it has provided to applications and may collect information about other aspects of the system as well.

The OS has a variety of protection methods to prevent unauthorized access to system resources. The strongest protection is achieved in corporation with a CPU that has different privilege levels. Modern CPUs have at least two privilege levels: kernel level and user level.

If properly designed, only the OS kernel code is allowed to execute in an unrestricted mode (kernel level) on the processor. Everything else runs in a restricted mode (user level) and must use a system call to have the kernel perform any operation that could potentially damage or compromise the system on its behalf. This makes it impossible for untrusted programs to alter

[1] or when a higher priority process has become ready, as will be discussed later.

or damage the system. Mode protection may extend to resources beyond the CPU hardware itself, for example protecting process memory boundaries.

So, a system call changes the execution mode from user level to kernel level and back when done. As a side effect other operations might also take place, such as memory swapping. Thus, the disadvantage of multiple privilege levels is a (slight) performance drop.

Multiple types of kernels exist: nano-, micro- and monolithic kernels. The exact type of kernel does not matter for this thesis, since the OS Application Programmers Interface (API) hides this anyway.

2.3.2 Processes and threads

An application, in general, is instantiated by the OS as a process. Processes have their own memory space, which boundaries are guarded by the CPU. Furthermore, OS resources are often assigned to the whole process.

In OS terms, the process is a thread container. A thread is an execution entity, which is scheduled by a scheduling algorithm onto the CPU. Threads share, with other threads belonging to the same process, their application code, data and other OSs resources, such as open files and signals. Support for threads may be provided at either the user level (user threads) or by the kernel (OS threads).

OS threads, also called kernel threads, are supported directly by the OS. The kernel provides support for the OS to create, schedule and manage threads in kernel space.

User threads are supported above the kernel level and are implemented by a thread library at the user level. The library is responsible for creating, scheduling and management of the threads, without support from the kernel. A user thread is executed by an OS thread and at most one user thread can run simultaneously in one OS thread. Most likely, the OS is unaware of the presence of user threads and as a consequence does not support user thread preemption.

User threads are generally faster to create and manage than kernel threads, since no system calls are necessary. However, preemption (rescheduling) of user threads is only possible on explicit request. So, if a blocking system call is performed on a user thread, the system call stalls the underlying OS thread and thereby all user threads in the container. When a blocking system call is performed on an OS thread, the kernel is able to preempt the thread and schedule another thread for execution.

2.3.3 Multi-threading models

The different multi-threading models, that are possible for a combination of OS and user threads, are depicted in Figure 2.1. The one-to-one threading model discards the user threading advantages, since for every user thread a 'heavy weight' OS thread is needed.

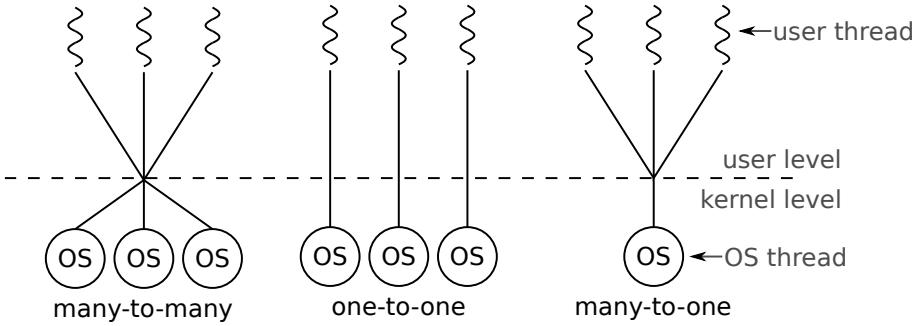

Figure 2.1: Multi-threading models (Silberschatz et al., 2004)

The many-to-many model can be used to prevent the 'one system call can block all' problem, but there are three potential problems:

- If more OS threads are used than cores/CPUs are available, expensive OS thread context switches are needed to run all user threads.
- Since the OS thread priorities take precedence over user thread priorities for scheduling (as will be explained later), a priority inversion problem might be created in the previous case as well. Because, a high priority's user thread running time is dependent on the underlying OS thread priority and its allotted time slot. This might result in non-deterministic timing.
- (Repeatedly) moving user threads over different cores/CPUs will result in non-deterministic CPU cache updates. This problem is very technical and its explanation is considered out of scope for this thesis.

The first two problems can probably be prevented with an elaborate coordination mechanism. However, this will cost computational resources and thereby discard the user thread advantages.

The many-to-one threading model can take full advantage of the lightweight user threads, but there are also drawbacks:

- Since only one OS thread is used to run the user threads, it will suffer from the 'one system call can block all' problem, as was experienced with the CT library. However, the problem can be prevented by using other solutions, such as assigning buddy threads for the OS function calls.
- The single OS thread cannot take advantage of multi core/CPU architectures. Nonetheless, multiple many-to-one user thread containers can take advantage of these architectures. To take full advantage, the user threads must be able to communicate and work together.

The many-to-one threading model is further discussed in Section 5.2.11.

2.3.4 Scheduling

The scheduling of OS threads is done by the OS scheduler. This scheduler determines the order of the ready processes on their thread priority level. Most OSs will also preempt lower priority OS threads in favour of any higher priority OS thread becoming ready. Most OSs contain priority inheritance algorithms to prevent priority inversion, in which a higher priority thread must wait for a lower priority thread to finish.

The scheduling of user threads must be done by the implementing thread library. Since in the OS there is no information on user threads, the user threads will get their running time when the underlying OS thread runs. Since preemption is not available for user threads, execution times should be kept short, otherwise a form of priority inversion will be created.

Most OSs contain multiple scheduling algorithms: First In First Out (FIFO) and Round-Robin (RR) are commonly available. The FIFO scheduler runs from the highest priority queue the longest waiting thread first. The RR algorithm assigns time slices to each thread in equal portions and in circular order.

2.3.5 Other OS features

Other OS features used in this thesis are:

- *Synchronisation primitives*
  To protect concurrently accessed data from becoming corrupt. The data access must be limited with, for instance, mutual exclusion objects. Furthermore, rendezvous channels can be used to safely synchronise program execution on data exchange points.

- *Timers*
  Parts of programs, for example control loops, need to be executed periodically. A timer can be used to accomplish this.
- *Sockets*
  Communication between two or more hosts via an ethernet link can be achieved with this.

A comprehensive explanation on OS concepts can be found in Silberschatz et al. (2004).

2.3.6 POSIX

The Portable Operating System Interface (POSIX) standard (IEEE Std 1003.1, 2004) defines a standard OS interface, functions and environment. It is being jointly developed by the IEEE and The Open Group.

An OS can be POSIX compliant, POSIX conformant or certified POSIX conformant. POSIX compliance means that it provides partial POSIX support, which functions and specifications it supports should be indicated in the documentation. Conformance means that the entire POSIX.1 standard is supported and certified means that it is accredited by an independent certification authority. Code which uses POSIX function calls can be compiled and run on any OS which is POSIX conformant, resulting in the same behavior.

2.4 Real-time

When controlling robotic setups, real-time is an important property. There are two levels of real-time: hard real-time and soft real-time. According to Kopetz (Kopetz, 1997): "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". For non real-time, deadlines do not apply at all.

The basic requirements according to Silberschatz et al. (2004) and Cooling (2000) for a real-time OS are:

- Preemptive, priority-based scheduling
- Preemptive kernel
- Fixed upper bound on latency (preferably deterministic timing)
- Task structuring of programs
- Parallelism (concurrency) of operations

Using a real-time OS does not automatically make the executed code real-time, see Section 5.1.5.

Some OSs provide different thread types for hard and non real-time, for instance the combination of Linux and RTAI (DIAPM, 2011).

The difference between soft and hard real-time is achieved by using different thread priority levels. Therefore, when hard real-time is discussed, soft real-time is also implicitly meant.

2.5 QNX

QNX is a commercial Unix-like hard real-time OS (QNX Software Systems, 2011) and is certified POSIX conformant. The OS is centered around a micro-kernel architecture. The idea is that most of the OS is running in multiple (small) tasks, known as servers, with only a small kernel containing essential functionality. So, any functionality not required, is simply not started. The QNX micro-kernel is fully preemptable and designed for hard real-time applications. The OS provides only one thread type, which is usable for hard and non real-time.

The benefit of QNX, for this thesis, is that it provides rendezvous communication channels by default. As was already pointed out in the introduction, these might be handy for the implementation of a CSP execution engine, since these are similar to the CSP rendezvous message passing paradigm.

The OS can be run on, among others, x86, ARM and PowerPC architectures and has multi core/CPU support. Due to the hard real-time design, the micro-kernel system architecture and the resulting small memory foot print, QNX is suitable for embedded control systems.

Additionally, QNX provides useful development features. Such as an Eclipse based IDE, an instrumented kernel (i.e. containing debugging symbols), remote debugging facilities and application profiling functionality.

A more detailed description on the QNX OS can be found on the website QNX Software Systems (2011). The first work performed with the QNX OS, within our group, is described in Molanus (2008).

2.6 Software testing

For framework development, a short description of commonly used testing methods and levels in software engineering is given next. A more detailed explanation can be found in Lethbridge and Laganiere (2001).

Testing methods
Testing methods in general can be classified as black-, grey- or white-box tests. For black-box testing, test cases are build around specifications and requirements to verify the (external) functionality of a component or function.

White-box testing is a method to verify internal structures or workings of a component, as opposed to its functionality. For white-box testing an internal perspective of the component, as well as programming skills, are required to design test cases.

Grey-box testing is a combination of the white- and black-box testing methods, but the verification is done at the black-box level.

Testing levels
Test levels can divided in unit, integration and system testing.

Unit tests, test the functionality of a specific component or its functions. This type of test is usually written by developers as they work on code to ensure that the specific building block is working as expected. Unit testing alone cannot verify the overall functionality of a framework, but rather is used to assure that the building blocks work correctly independent of each other.

Integration tests verify the interfaces between components for the benefit of a cooperative software design. The tests works iteratively, progressively larger groups of tested software components are integrated and tested until the software works as a system.

System testing tests a completely integrated system to verify that it meets its requirements.

2.7 Design methodology

At the CE group, embedded control software is developed using the design trajectory as defined in Broenink et al. (2007), Broenink et al. (2010b) and Bezemer et al. (2011a), see Figure 2.2. The dashed box shows the steps in the design trajectory where this project applies to.

The design trajectory promotes iteratively development, whereby each step should be verified by a simulation or a validation on the setup. When a step reaches the desired outcome, the following step can be undertaken. Of course, sometimes it is necessary to (partially) redo previous steps if it becomes clear that a step cannot reach the desired outcome.

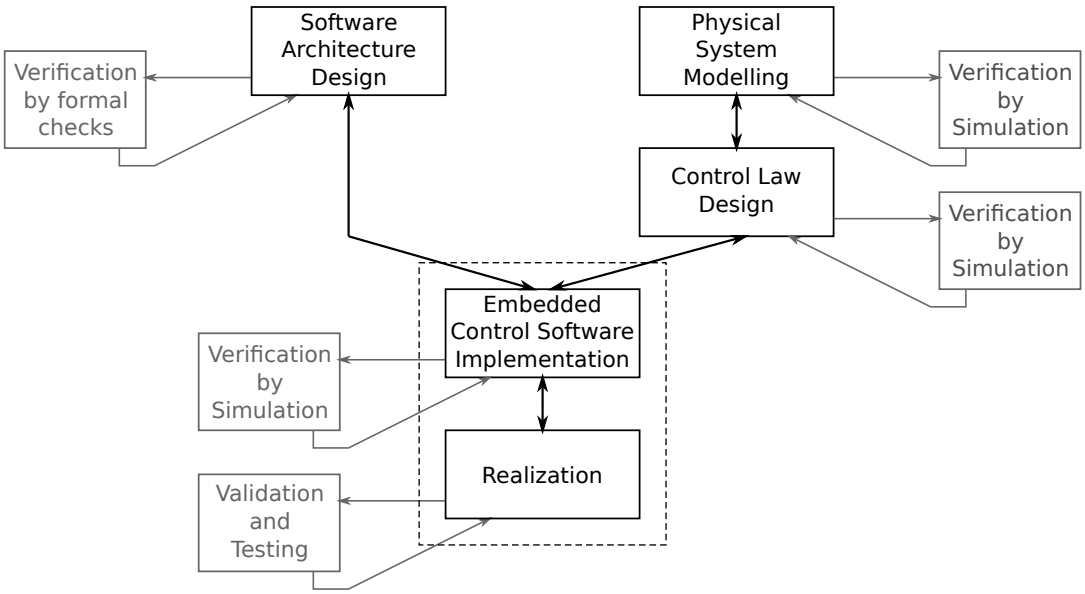

Figure 2.2: CE design methodology (Broenink et al., 2010b)

Short description of the design steps:

- *Software Architecture Design*
  A software architecture is created to add high-level behavior to the embedded system.
- *Physical System Modelling*
  The interesting dynamic behavior of the system is modelled.
- *Control Law Design*
  For the dynamic model, loop controllers can be designed.
- *Embedded Control System Implementation*
  The controllers are implemented using (software) algorithms and are combined with software architecture.
- *Realization*
  The complete collection of algorithms are converted into an executable to be run on a computing system.

The development route *Software Architecture Design* and the route *Physical System Modelling* and *Control Law Design* can be performed in parallel. All steps are clarified in the following sections.

2.7.1 Physical System Modelling and Control Law Design development route

For instance, bond graphs can be used to model a dynamic system and a simulator can be used to analyze the resulting system behavior.

20-sim (Controllab Products, 2011) is a graphical modeling and simulation tool for, among others, bond graphs. Using its Control Toolbox the controllers for the system can be designed and verified in its simulator.

2.7.2 Software Architecture Design development route

The software architecture defines how the different algorithms of the embedded system should work together, i.e. which processes run concurrently and when data exchange is required between processes. However, the development of complex concurrent software tends to become tedious and error-prone. Communicating Sequential Processes (CSP) can ease such a task, as explained next.

The CSP algebra, introduced by Hoare (1985) and Roscoe et al. (1997), can be used to reason about concurrent processes and patterns of communication between these processes. The algebra is based on a few simple constructs, such as denoting sequential or parallel execution of processes, and the rendezvous message passing paradigm. When two processes try to communicate the first arriving processes, at this synchronization primitive, will be stalled until the second is also ready for communication. Thereafter, both processes can freely continue. A good introduction to Add "CSP" to dictionary can be found in Nissanke (1997).

As was already stated in the introduction, the graphical modeling tool gCSP (Jovanovic et al., 2004) can, for instance, be used to ease the development of CSP models. Note, this tool is an interpretation of the CSP algebra. During development the application can be monitored in gCSP through an animation facility (Steen, van der, 2008). Furthermore, the gCSP tool can generate code for the Failures-Divergence Refinement tool (Formal Systems (Europe) Limited, 2008) to formally check a design on deadlocks and lifelocks. Without such tools, the developing of complex concurrent software tends to become tedious and error-prone.

2.7.3 Embedded Control Software Implementation step

Along both development routes a design is made in a (graphical) modelling tool. These designs need to converted into code, such that the software architecture and controllers can be combined.

The 20-sim Code Generation Toolbox can be used to generate software algorithms of the control laws.

The gCSP tool can generate code for the CTC++ library (Orlic and Broenink, 2004). The combination of the CT library and gCSP is depicted in Figure 1.1. However, as was already stated in the introduction, the library cannot make use of multi-core/CPU architectures and its design is outdated. Therefore, the framework designed in this thesis will replace the CT library in the future.

Veldhuijzen (2009) reimplemented the CT library for the QNX OS. The main difference with the original library is that OS threads are used instead of user threads for CSP processes, and thereby multi core/CPU architectures are supported. Unfortunately, the approach to only use OS threads makes the CSP model execution slow (see Appendix I) and the implementation is not complete (apart from the basic processes).

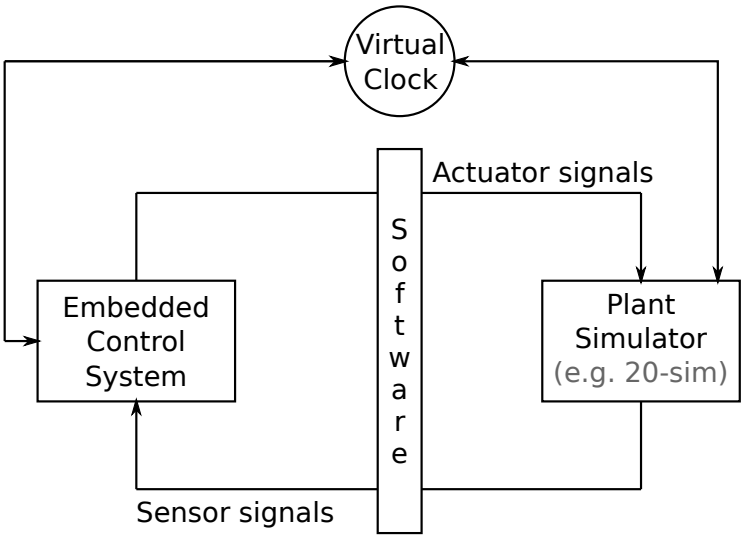

Figure 2.3: Co-simulation test bench

The combination of the software architecture and controllers can, for instance, be tested using co-simulation. Figure 2.3 schematically depicts such a test bench. The embedded control system hardware runs the combined software design, but generated actuator signals are routed to an off-target physical system simulator (e.g. 20-sim) instead of the actuators. The calculated physical states are send back to the embedded control software as sensory input. So, the test bench is created purely in software and the embedded control system does not use the actual physical system nor its actuator/sensor hardware. The virtual clock synchronizes the embedded control system and simulator execution, since simulating a complex physical system might take more time than running the application in hard real-time on its dedicated hardware.

2.7.4 Realization step

In the realization step, the generated software architecture and controller code is compiled into an executable for the target embedded system. This executable can then control the physical system. The complete system can be tested and validated by manipulating its inputs and verifying the generated output behavior or signals.

2.7.5 Embedded control software layers

Additionally for the *Software architecture design* step in the design trajectory, a generic design pattern has been developed at the CE group, see Figure 2.4.

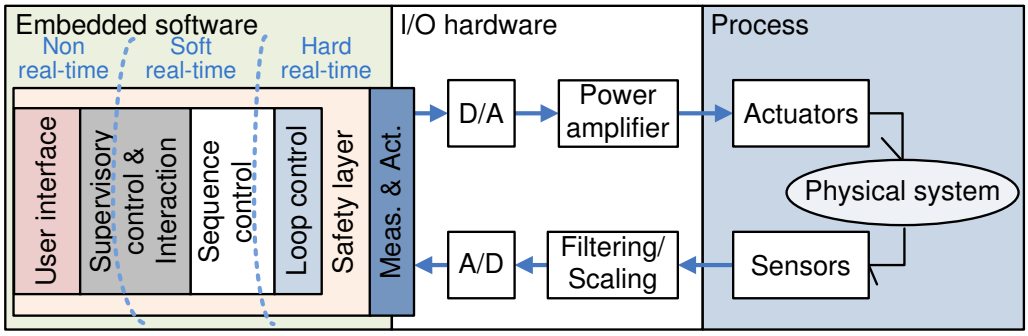

Figure 2.4: Software architecture for embedded systems (Broenink et al., 2010a)

Each layer supports a type of real-time, varying from non real-time to hard real-time. The following paragraphs discuss the common distribution for these.

The *Loop control* is the part of the application responsible for controlling the physical system and it is realised in hard real-time. If the loop controller fails to meet its deadlines for whatever reason, the system is considered unsafe and catastrophic accidents might happen with the physical system or its surroundings due to moving parts.

In the *Sequence control* the correct trajectory generator(s) and controller implementations are chosen based on the required action. The trajectory generator calculates the set-points for the *Loop control* to follow. Generally, this part of the application is more complex and requires more time to run its tasks than the *Loop control*. If a deadline is missed this is not immediately catastrophic, but some applications, like a wafer stage, might require hard real-time for this layer as well to guarantee smooth operation.

The *Supervisory control & interaction* contains algorithms which, for example, handle input events, map the environment, plan future tasks of the physical system or communicate with other systems. These might require soft or non real-time, depending on the application.

The *User interface* is non real-time, but some form of responsiveness to inputs is desirable.

The *Measurement & Actuation* interfaces with the hardware (directly). This usually means forwarding the signal values to and from the appropriate hardware registers.

The *Safety layer* is used throughout the application to prevent unwanted control signal values.

To conclude, the new framework should support such a layered design implicitly.

3 Analysis

Before one can start to design a new system an analysis should be conducted to identify the expected problems, requirements and use cases. The purpose is to bring the final design and the expectations of the future customers as close together as possible. First a domain analysis will be performed and hereafter the requirements can be deduced from this. Although a use case analysis might be interesting, it is considered out of scope for this project.

The rest of this thesis will be based on the outcome of this chapter.

3.1 Domain analysis

The domain analysis is conducted to select the general field of business for the new framework and to get a better understanding of the challenges commonly present in the domain. Therefore, a domain analysis can improve the development time, system integration and anticipation of future extensions. Furthermore, it can help select implementation priorities.

The domain analysis for the new framework has been performed with four corner applications selected from our laboratory. These are distinct set-ups commonly found in embedded (robotic) control research. The corner applications are:

- Jiwy - A small embedded control teaching aid.
- Production cell - An embedded distributed control application.
- Humanoid head - A control application focused on human-machine interaction.
- TUlip - A complex humanoid walking robot.

These corner applications combined represent the total set of applications which are supposed to be realisable with the new framework. Note, all corner applications have already been realised and that the selected implementations all required an underlying operating system. It is also believed that these setups will resemble, at least, the near future embedded control software requirements. The results can be found in Appendix A.

An analysis into competing software has also been performed (Chapter 4), however with a slightly different approach than one might expect from a domain analysis. This analysis intends to identify other frameworks strong points and weaknesses, and concludes with remarks for the new design.

3.2 Requirements

The purpose of the requirements analysis is to define all functional and non-functional requirements the new framework should fulfill. Using the results from the domain analysis, the requirements can be deduced.

In the domain analysis both threads/processes and layers are used, but in fact layers can be seen, from an OS point of view, as a group of threads. Unless explicitly denoted, both can be used interchangeably.

Furthermore, it was timely recognized that not all requirements can be fulfilled within one thesis project, therefore the requirements marked with a † are not further considered. Besides, these requirements are not deemed critical to the framework's first working implementation.

3.2.1 Functional requirements

3.2.1.1 Platform independence

The framework should ultimately *support different hardware platforms and operating systems*. For the first implementation the QNX OS on a x86 architecture was chosen. But, in the future also other target OSs (like RTAI and Xenomai) should be supported. If the chosen target OS supports them, different hardware architectures should be supported (like a PowerPC or ARM) as well. Furthermore, an OS-less version of the framework should in principle be possible in the future. Next to the mentioned target OSs, a few development OSs (like Windows and Linux) should be supported as well to facilitate development. So, *the framework should be designed as platform independent as possible to enable future OSs and architectures*.

3.2.1.2 Real-time constraints

The four corner applications all require *hard real-time constraints* for, at least parts of, their application. Because of the limited available computing resource in the more advanced setups, it is necessary to specify a *mixture of hard/soft/non real-time constraint* threads in order to fulfill the timing requirements.

3.2.1.3 Thread support

To take advantage of multi-core/CPU target systems, *OS thread support* is required. But, this should not cripple single core/CPU embedded systems.

3.2.1.4 Priorities

It is sometimes required to *subdivide a real-time constraint level into multiple smaller priority levels*. Such that optimal timing results can be achieved.

3.2.1.5 Periodicity

Certain parts of robotic applications are required to exert *periodic behavior*. In one application *multiple frequencies with each multiple threads* may be present. Furthermore, the component implementing the periodicity should adhere the priorities and real-time constraints assigned to the thread.

3.2.1.6 Communication

A common approach in software design is to divide and conquer, which requires parts of a design to communicate with each other to exchange data or active one another. All four corner applications are designed following this principle. Thus, the threads in an application will need to *communicate with each other, with other applications on the same embedded computer or even on different hosts*. Furthermore, the *communication should conform to the real-time constraints and priorities assigned to the communicating threads*.

3.2.1.7 Synchronization

Resources might be accessed concurrently. To ensure proper and safe operation the *threads and processes should be able to synchronize access to any software resource*, which in turn can be used to protect hardware resources.

3.2.1.8 (Link) drivers to/from hardware

Robotics inadherently need to interact with the environment, therefore *(link) drivers for sensors and actuators* should be included where possible.

3.2.1.9 External code and/or library integration

The new framework should not reinvent the wheel and therefore *provide mechanisms to include external code and libraries*. For example, one can think of the ROS (ROS, 2011) framework or the GSL (2011) library for performing advanced matrix calculations.

3.2.1.10 CSP execution support

For CSP models to run on the new framework *CSP execution support* should be added. Preferably, with the same functionality as the current CT library offers.

3.2.1.11 Safety layer

To prevent hazardous situations for the setup and environment a *safety layer* should be present in all robotic applications. This layer mostly provides *limits on actuator output signals*, *takes action on endstop events* and is sometimes used to *identify broken sensors*.

3.2.1.12 Debugging facilities

Although applications and frameworks are preferred to work out-of-the-box, it is a good idea to *supply (standard) debugging facilities*. These should also be used to generate bug reports for the framework developers, so that problems can be solved efficiently. Furthermore, it would be nice to have a *debugging utility which does not influence the real-time constraints nor timing*.

3.2.1.13 Self testing

The complete framework will most likely contain hundreds of classes and interfaces, whereby some only are available on specific platforms. Additionally, the framework might be used on platforms the developers even did not think of. Therefore it is wise to supply a *standard self-test*, which can be run on the intended platform *to see if all required features are operational*. This will also increase the confidence users have in the framework.

3.2.1.14 Reusable component specification and interconnection †

Parts of a robotic application might be reused in other parts of the application. In order to achieve this it would be beneficial if the framework could support a generic manner to perform:

- **Parameter configuration** - dynamically loading and replacing of settings for a component.
- **Component interaction specification** - stipulates how (repeated) components fit together in an application.
- **Flexible connections** - communication vectors might vary in size depending on physical properties they represent, and some components might be able to provide generic operations on these.

It is realized that the above definitions are very short, but they only serve to point out identified future work or, perhaps, these should be supported by a graphical tool.

3.2.1.15 Stepwise implementation support †

The divide and conquer approach in software design also applies well to implementing and testing the application (see Figure 2.2).

The proper working of the software architecture design can, for instance, be empirically verified by animating execution (of a special implementation version) in a graphical modelling tool. The software architecture implementation support provided by the new framework should be augmented with an *animation framework* and *missing algorithms need to be replaced with mock-up algorithms*.

The Embedded control software implementation step can, for instance, be tested with a *co-simulation test bench* (Section 2.7.3). The application is then tested using a simulation of the

physical robot, preventing possibly damaging the physical robot or the environment during application development.

In order to support co-simulated testing of complex physical systems, it should be possible to *seamlessly exchange the real-time clock with a virtual clock*, because the simulator will most likely not be able to keep up with the real-time constraints of the application itself. Generally it takes time before a simulator has calculated the next physical state, while the application is designed to handle all computations for the real physical plant in a relatively short time. The virtual clock will take care that the application will only progress whenever new simulation data was received. Furthermore, the *actuator output generated by the application should be (re)routed to the simulator* to update the simulated dynamic behavior and then the calculated *sensory outputs should be fed back to the application* for the next application cycle.

3.2.2 Non-functional requirements

The *software design and the employed programming style should reflect the real-time constraints*. The normal case should be to design and program for hard real-time and if this cannot be achieved it should be documented.

The *software design should be scalable*. All kinds of setups should be controlled: From the big robotic humanoids to small embedded platforms with limited computing resources.

Interfaces should be easy to use correctly and hard to use incorrectly. People using a well-designed interface almost always use the interface correctly, because that's the path of least resistance. Good interfaces anticipate mistakes people might make and make them difficult (ideally impossible) to commit.

Unify interfaces for components with similar operations as much as possible. For example, a simple communications protocol and a file writer could employ the same *write* function to 'send' their data to their respective destinations.

The new framework should preferably be *faster than the current CT library*. Furthermore, it should give *better or likewise timing results*. A lesser performance would undermine its future position as the to-be-used framework for embedded control applications in the laboratory.

Example and test programs should be supplied with the framework, to provide a show case, aid learning how to use the framework properly and increase confidence in the framework.

The *framework interfaces should not be dependent on features which are not commonly available*. For example, Boost (2011) tuple functions are only usable when the framework is compiled with Boost. Furthermore, such features might change overtime and render applications created with the new framework into legacy status.

3.3 Conclusions

The new framework will be able to control the aforementioned four corner applications and future applications, when the identified functional requirements are implemented. The corner applications represent the total set of embedded applications which are supposed to be realizable with the new framework. Furthermore, a subset of the functional requirements is also implemented by the current CT library.

Requirements marked with a † are not considered in this thesis, but are highly recommended for implementation in the future. These will simplify development and enable iterative development and testing.

The quality of the new framework is guarded by the non-functional requirements. These specify common (programming) practices to be employed in the new framework's design and implementation, and overall requirements for the new framework.

4 Framework and library research

The purpose of this chapter is twofold. First, state of the art embedded control libraries and frameworks will be investigated to determine if one can be found which already implements the requirements (Chapter 3). Secondly, which valuable lessons can be learned from these libraries and frameworks?

The introduction will first present all considered state of the art frameworks and libraries very shortly. Then a pre-selection will be made, which will narrow the investigative space to alluring frameworks and libraries regarding the purpose of this project. The approach section will then describe how the detailed research was conducted. Next, the selected frameworks and libraries will be discussed in detail. The last section will give a comparison between the discussed frameworks and libraries and concludes regarding the two-fold purposes of this research.

4.1 Introduction

The considered state of the art frameworks and libraries are ROS (2011), C++CSP2 (2009), Orocos (2011) and RoboFrame (2010). ROS is a framework for robot software development, providing operating system-like functionality on top of a heterogeneous computer cluster. C++CSP2 is a framework for CSP based algorithm execution. Orocos deals with the definition of components. RoboFrame is a message oriented middleware, which thus deals with the communication between modules or layers.

Framework	Hard real-time	Platform independence
ROS	-	-
C++CSP2	-	-
Orocos	+	++
RoboFrame	+/-	+
CT (CTC++)	+	+

Table 4.1: Generic framework characteristics

Table 4.1 presents the basic characteristics of the aforementioned frameworks. $++$ is the maximum achievable score and $--$ is the lowest achievable score. A framework is hard real-time when it adheres to real-time programming restrictions (for details see Section 5.1.5). A framework is platform independent when it provides methods to abstract the OS specific system calls. On the basis of these results a pre-selection will be made.

To be clear, when a framework provides POSIX compliance this does not mean that it will be hard real-time by definition on a hard real-time OS like QNX. Only if the real-time programming restrictions are followed it can be hard real-time. Furthermore, if the framework/library does not explicitly state it is designed for hard real-time nor provides an implementation for at least one hard real-time OS other than POSIX compliant ones, it is assumed to be soft real-time at best.

The ROS basic framework is not considered platform independent, although it works on most Linux/Unix operating systems and partially on Windows, because no special precautions can be found (in the file hierarchy) to establish platform independence, which one would expect for easy maintenance at least. The provided platform independence is supplied by the used Boost library (Boost, 2011). Furthermore, the Linux/Unix OSs do not include hard real-time additions like RTAI. Therefore, it was decided to not consider ROS any further.

The C++CSP2 library, which actually is a framework, provides a multi-threaded CSP engine. Unfortunately, it is not suitable for hard real-time applications controlling setups. Because, it actively makes use of C++ language exceptions to influence the execution flow, which make an application in general non deterministic (see Appendix B). Additionally, the framework does not have any means for platform independence. Therefore the C++CSP2 framework is not considered any further in this chapter. However, the C++CSP2 library was consulted during the CSP component development for the new framework.

Orocos is advertised as a hard real-time framework, but as will be shown in section 4.3 the framework is most likely not able to perform well enough for loop control nor implementing CSP constructs, due to its considerable software footprint. The framework supports multiple OSs, with among others RTAI.

The RoboFrame framework is not designed for real-time, but the extension explained in Section 4.4 can be used to perform loop control in robotic applications. Furthermore, its separation in modules and platform independence make it worthwhile to investigate.

As explained before, the CT library is build for hard real-time execution of gCSP generated code. However, due to its inaccurate timing, it does not get the full ++ score for hard real-time. Although the framework supports multiple OSs (among others RTAI), the tight integration between CSP and the platform abstraction is not desirable and results in a +.

The table shows that none of the considered frameworks is truly hard real-time (i.e. a ++ in this column), which is an integral requirement for the new framework. So, together with the fact that the current CT library is outdated, it can be concluded that the design of a new framework is a viable choice.

Because no suitable framework is found, one could choose to build a new framework with the aid of state of the art libraries, such as Boost (2011) or POCO (2011). The Boost C++ libraries are a collection of free libraries that extend the functionality of C++. It is aimed at a wide range of C++ users and application domains. POCO also is a collection of open source C++ class libraries and frameworks, but is targeted at building network and internet based applications that run on desktop, server and embedded systems.

Library	**Hard real-time**	**Platform independence**
Boost	+/-	+/-
POCO	+/-	+

Table 4.2: Generic library characteristics

Table 4.2 presents the basic characteristics of the aforementioned libraries. The scoring system is the same as for the framework comparison.

Any proposed library should also be C++ language exception free, to achieve the real-time constraints. Unfortunately, both libraries actively use them to handle exceptional circumstances.

Furthermore, it is important that the library functions are platform independent, without this the functions cannot be used directly in the new framework without considerable work. Nonetheless, the interfaces can still be reimplemented for the new framework. The last remark does not apply to exceptions as these are not a simple implementation issue, but affect the software design. The platform independence will be discussed in detail later. Both frameworks have a $+/-$ for hard real-time, because they only support POSIX compliant real-time OSs, which seems to be more of a coincidence for real-timeness than a design choice.

4.2 Approach

Most frameworks and libraries supply software manuals for the end user, which is considered good practice of course. Unfortunately, only Orocos has a manual for developers detailing its inner workings, which is what we are interested in. All frameworks also have papers published about them. But, even with all these resources some questions could not directly be answered. Therefore, to fully understand what makes the considered frameworks so good and what makes them tick, reverse engineering and code inspections were also performed.

Naturally all frameworks and libraries will have their own applications and boundary conditions. The introduction of each subsection presents these among others. Furthermore each framework and library will have the following subsections:

- *General software architecture* - presents the general idea behind the framework/library.
- *Platform independence* - discusses how platform independence is solved.
- *Timing and real-time capabilities* - deals with these specific aspects.
- *Scalability and extensibility* - comments on the scalability and extensibility, with respect to embedded system usage and building the new framework with it.

The introduction and general software architecture are fact based presentations on the specific framework/library. The other sections may also contain discussions on the subject at hand. Furthermore, these subsections roughly fit the requirements specified in Section 3.2.

4.3 Orocos

The Open RObot COntrol Software[1] (Orocos) is the most extensive robot and machine control software currently on the market. The Orocos project supports four C++ libraries: the Real-Time Toolkit (RTT), the Kinematics and Dynamics Library (KDL), the Bayesian Filtering Library (BFL) and the Orocos Component Library (OCL). The libraries are open source and can be freely downloaded, used and distributed (Orocos, 2011).

The Real-Time Toolkit (RTT) is not an application in itself, but provides the infrastructure and the functionality to build robotic applications in C++. The emphasis is on real-time, on-line interactive and component based applications. Only this library will be discussed in this chapter, but for completeness the other libraries will be introduced briefly. The Orocos Components Library provides some ready to use control components. The Orocos Kinematics and Dynamics Library is a library to calculate kinematic chains during run-time. The Orocos Bayesian Filtering Library (BFL) provides an application independent framework for inference in Dynamic Bayesian Networks, i.e., recursive information processing and estimation algorithms based on Bayes' rule, such as (Extended) Kalman Filters, Particle Filters and more.

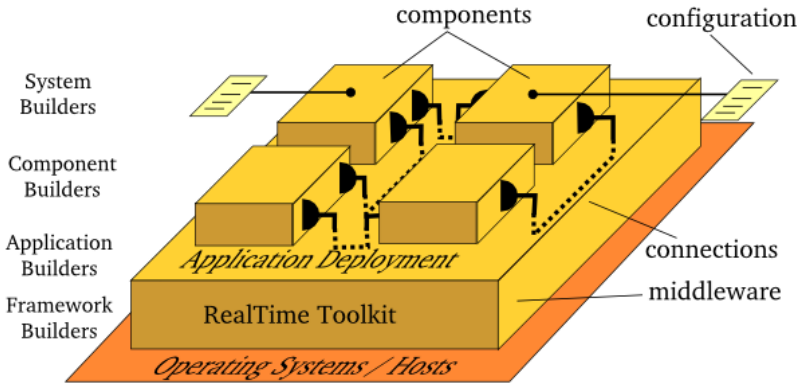

Figure 4.1: Orocos framework overview (source: Orocos, 2011)

[1] Version: 1.10

4.3.1 Software architecture

The Orocos project focuses on component based applications with fixed interfaces. For the component specification these interfaces are systematically separated in Computation, Communication, Configuration and Coordination (4C's). The components are build on top of the RTT framework (see Figure 4.1), which provides the infrastructure and the means for execution.

Each component is defined as a *TaskContext*, which defines the environment or 'context' in which an application specific task is executed. The *TaskContext* interface provides five public manipulation methods:

- *Attributes and properties*
  The attributes and properties in a *TaskContext* can be used to get and set configuration data. Attributes are plain variables which expose C++ class members to the scripting layer, additionally properties can be stored to and loaded from an XML file. The reading and writing of properties and attributes is not thread-safe.
- *Methods*
  Methods resemble normal C++ functions, i.e. they are directly executed by the caller, but they also can be called from a script or over a network connection. Calling methods is, as in C++, not thread-safe.
- *Commands*
  Commands are asynchronous function calls, with respect to the caller, and they are executed in the receiver's thread. Furthermore they might be queued and must provide a boolean value to indicate success or not. Commands are thread safe with respect to other RTT processors (discussed in the RTT Execution section) running in the receiving component.
- *Events*
  Events are designed according the classic publish-subscribe software pattern (Douglass, 2002), i.e. one ore more clients can register their callback functions for a specific event. When the event is raised, the connected functions are synchronously or asynchronously called one after the other. Publishing and reacting to an event is thread-safe only in asynchronous callbacks. A task may register its events in its interface in order to be used by its state machines and other tasks as well. Unfortunately, it cannot be predicted when the event is executed.
- *Data-Flow ports*
  A stream of data can be send between tasks using a data-flow port. This data can be passed buffered or unbuffered and a task may be woken up if data arrives or it can use a polling scheme to check for new data at one of its ports. The data-flow ports are connected using a modified Acceptor-Connector software pattern (Douglass, 2002), which decouples the connection and communication roles. The data-flow ports could for example be used to implement a classic control loop. Reading and writing data is thread-safe.

Furthermore, the *TaskContext* internals provide a build-in state machine, which can be extended through various C++ function hooks (e.g. *updateHook()*). The internal state machine will be explained in the RTT execution section.

The class *TaskContext* groups all these interfaces and serves as the basic building block for components. The *TaskContext* is schematically shown in Figure 4.2.

Dynamic program flow
An Orocos component (*TaskContext*) can have a dynamic program flow via:

- Real-time scripts
- User defined state machines
- Distributed computing, with CORBA (2011)

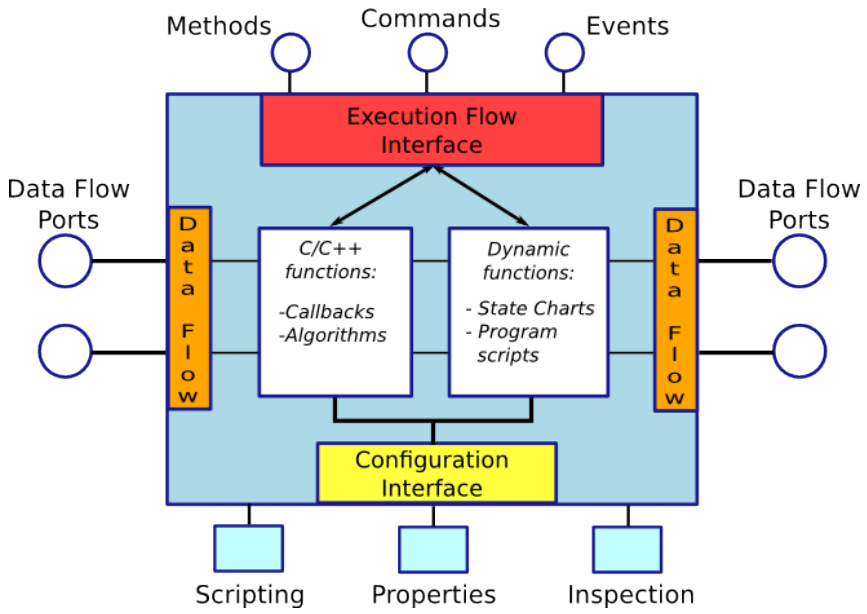

Figure 4.2: Orocos *TaskContext* (source: Orocos, 2011)

The real-time scripts and user defined state machines will be discussed in the RTT execution section. The CORBA package enables Orocos components in separate processes and possibly distributed over a network connection to communicate with each other. This part of the RTT is considered out of scope and will not be discussed any further.

RTT Execution

The execution of a *TaskContext* is performed by the *ExecutionEngine* (see Figure 4.3). An *ExecutionEngine* should be added to a *Periodic-*, *Nonperiodic-* or plain *Activity*, which ever is suited best according to its periodicity requirements. The different Activities map onto different operating system threads and these may hold multiple *ExecutionEngine*s with the same real-time constraints. The *ExecutionEngine*s contained in a *PeriodicActivity* are executed exactly once per period. Furthermore, if an overrun of the *PeriodicActivity* occurs this is recorded and can be dealt with accordingly.

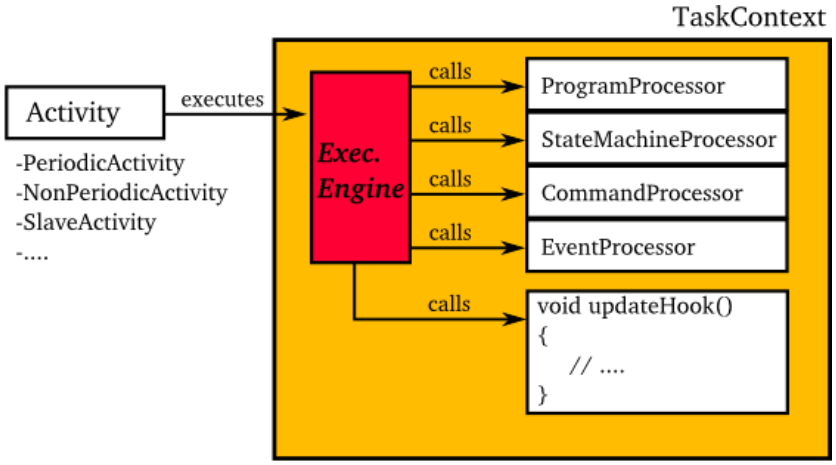

Figure 4.3: Orocos execution model (source: Orocos, 2011)

The commands, events and internal state machine of the *TaskContext* interface are executed by the *ExecutionEngine* using, so called, processors. Furthermore, the real-time scripts and user defined state machine are also executed by a processor. These five processors are executed in the order they are explained next:

- Real-time scripts - *ProgramProcessor*
  The Orocos scripting language enables users to write programs and state machines for controlling the system in a non hard-coded manner. The advantage of scripting therefore is that it does not need recompilation of the main program and, according to the Orocos project team, makes the Orocos components easily extensible. The disadvantage is that it decreases system performance, as scripts needs to be interpreted, and may violate hard real-time constraints as will be discussed in Section 4.3.3.

  The scripting language supports, among others: variables, constants, if-then-else, for-loops, while-loops, function declarations and interfaces to the other processors. Recursive function invocation is prohibited, since this results in non real-time behavior by definition.
- User defined state machines - *StateMachineProcessor*
  The component designer may add his own state machine to the *TaskContext*. This state machine has (conditional) edges and may emit commands to its own or other *TaskContext*s. A state machine can run in two modes: automatic mode and the reactive (also 'event' or 'request') mode, which can be switched at run-time. To be clear, this state machine is not part of the internal state machine which provides function hooks.

  The program and state machine scripts share the common syntax and semantics, additionally the state machine also provides extra functionality to specify states, preconditions for state entry and (conditional) edges. The script parser is strongly typed and if a run-time error occurs, i.e. if a statement fails, it will bring the *TaskContext*' state machine into the error state.
- Commands - *CommandProcessor*
  The (buffered) commands are sequentially executed by the CommandProcessor.
- Events - *EventProcessor*
  The asynchronous (buffered) events are executed sequentially by the receiving *TaskContext*'s EventProcessor.
- Internal state machine - by the *ExecutionEngine* itself via, for instance, *updateHook()*
  For implementing algorithms the *TaskContext* provides an internal fixed state machine, which should be extended in C++ function hooks to initialize, configure, start, update, stop and clean-up a component. The basic state machine without exceptions is shown in Figure 4.4.

Methods, data-flow ports, properties and attributes are not executed by any processor, because methods operate in the calling thread directly on a *TaskContext*'s internals. Data-flow ports may raise events or call the execution engine to process the newly arrived data. The attributes and properties can be manipulated from the *ProgramProcessor*, *CommandProcessor*, *EventProcessor* or internal state machine.

All processors perform one *step()* per *ExecutionEngine* step, but the number of internal steps defers per specific processor. For the internal state machine it depends on the actual code in the *TaskContext*. The other processors mostly process their complete internal queues (e.g. all received Commands) during one *ExecutionEngine* step. But, the number of internal steps the user state machine might perform can be configured, for example the user state machine might run until it blocks on a conditional edge or just follow one edge and perform the statements accompanying the new state.

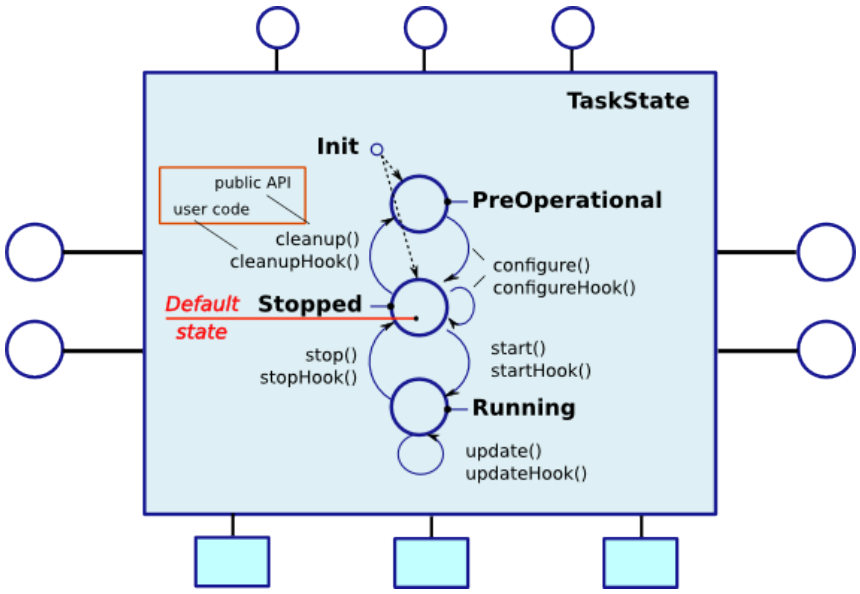

Figure 4.4: Orocos internal state machine (source: Orocos, 2011)

The *ExecutionEngine* processors for a *TaskContext* are thread safe with respect to each other, since they are run one after another in one *Activity*. Furthermore, the sequence of program actions is serializable, because all *ExecutionEngine* processors on an Activity run in a predefined order.

4.3.2 Platform independence

Components interact with the underlying operating system and other components through the Real-Time Toolkit, see Figure 4.5. The RTT is a platform independent layer that provides inter component communication, (a)periodic threading, mutexes & semaphores, data streams, time related operations (e.g. sleep), networking, property management through XML files and some general operating system calls. Furthermore the RTT specifies an interface for generic data acquisition and (actuator) drivers.

The platform independent RTT features have a generic interface and implementation. The implementation is composed of functions with a generic function signature, which is the same for all supported platforms. Macros then include the right OS dependent header, i.e. function skin, which contains the actual function implementation. The use of macros in general is not preferable, because one needs to actively keep track of them and problems with macros sometimes are hard to track in sources. For the currently maintained OSs the function skin approach worked out well, but these functions might not match well when building support for a new OS.

The Orocos RTT is currently maintained for RTAI (Linux) and Xenomai (Linux) real-time operating systems and the general purpose operating systems plain Linux, Mac OS X and Windows. Additionally, Orocos provides a device driver abstraction layer.

4.3.3 Timing & real-time capabilities

The Orocos RTT is advertised as a hard real-time framework. This is in principle true, but a programmer has to manually enforce and verify the real-time constraints of an application build with the framework. Due to Orocos' wide variety in manipulation methods, which make the framework inner working quite complex, the verification will be an elaborate, if not, impossible task. Thus it is advisable to use dedicated hardware for the control loops, which are fed by the

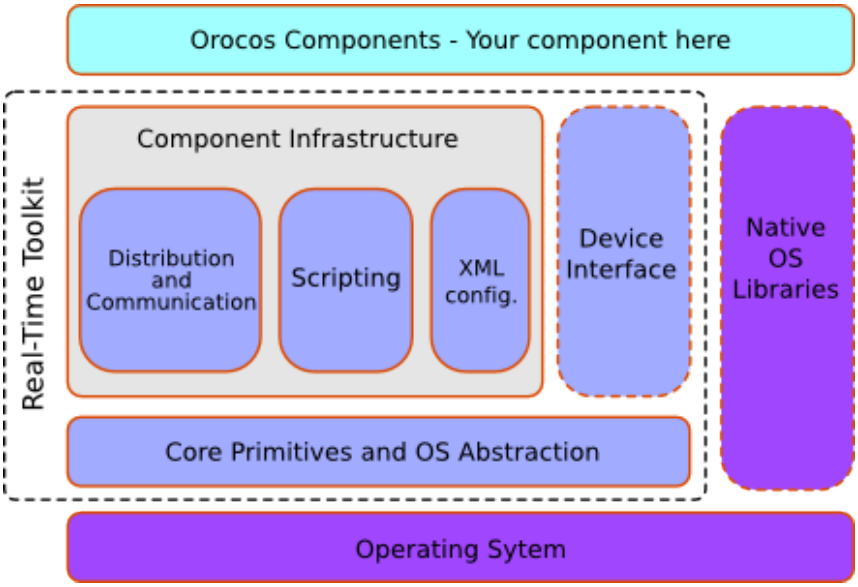

Figure 4.5: Orocos software design overview with respect to platform abstraction (source: Orocos, 2011)

framework with setpoints. Therefore, the framework basically is operating at a soft real-time level, since it does not matter if a setpoint arrives slightly late at the hardware control loop.

In our group we like to design the control loops ourselves and are not using such hardware control loop solutions. Furthermore, it is impossible to use formal methods to confirm that a complex application, build using Orocos, is deadlock or livelock free, because of the size and complexity of the framework (Dijkstra, 1972).

Fortunately, Orocos does provide means to determine, at run-time, if an overrun of a periodic task occurred and also provides some failsafe mechanisms for it. But, this is a run-time feature and therefore will not guarantee proper nor safe operation for a robotic application.

Real-time scripts

Real-time scripting, in general, should be looked at sceptically. Because, for a scripting system to be real-time it has to adhere to the same conditions as normal code, i.e. it should have deterministic timing. But, the commonly accepted method for parsing scripts is to incur multiple layers of abstraction which have to be passed through before, for example a statement is accepted. Note that different types of statements possibly have different layers of abstraction. Furthermore, certain commands in a script might want to allocate memory, which in most systems is not a deterministic operation.

The solution to these described problems, applied in Orocos, is to parse the script during non real-time operation into an internal format. This intermediate format should prevent non-deterministic timing during run-time by, for instance, allocating the required memory on beforehand. Furthermore, some optimizations may be identified and stored in the intermediate format. So, when the interpreter (which is either the ProgramProcessor for the real-time script or the StatemachineProcessor for the real-time state machine descriptions) receives the intermediate format it should be real-time. That is, if the command, method, et cetera it calls satisfies real-time constraints.

4.3.4 Scalability & extensibility

Orocos components can be used in both embedded or general purpose systems, depending on the application design. The placement of multiple components and the specification of their interconnections on a specific target is called component deployment in Orocos.

Three distribution levels of component deployment can be realized in Orocos:

- Not distributed
  In case the application will not use distributed components and a very small footprint (< 1 MB RAM) is required, the TaskCore should be used. The TaskCore is a base class of the previously explained *TaskContext* class, and provides the manipulation methods in a hard coded non-dynamic manner (i.e. no scripts and user defined state machines).
- Embedded distributed
  In case the application requires a small footprint and distributed components the *TaskContext* in combination with the distribution library, which does the network translation, can be used. The *TaskContext*, in this manner, can only handle a predefined set of build-in data types and requires modification if other data types should be supported.
- Fully distributed
  If the application footprint is of no concern and components should be distributed completely transparently, the *TaskContext* in combination with a remoting library, for network translation, can be used. The Orocos developers are working on a CORBA implementation for this, which can pick up user defined types without requiring modifications.

The components designed with the embedded and fully distributed deployment interfaces can be dynamically configured from an XML file.

Unfortunately the three distribution levels require different implementations because of their interfaces. A component, developed for one distribution level, therefore cannot be used on another without modification. Lootsma (2008) has also pointed out that the Orocos RTT framework can be quite resource consuming. This is, actually, not a surprise if one takes into account the large number of manipulation methods and execution flow solutions.

Due to its size the Orocos framework is not suitable to implement CSP constructs on a low level, but it would be possible to build a standard Orocos *TaskContext* for running or interfacing with a complete CSP model.

4.4 RoboFrame

RoboFrame[2] is a modular software framework for lightweight autonomous robots, created by the Technical University of Darmstadt. RoboFrame started as a master thesis project of Petters and Thomas (2005). It addresses the needs of heterogeneous teams of autonomous lightweight robots and it is used for example in the robot soccer competitions.

The framework has been designed to facilitate reuse of its architecture in many different sets of boundary conditions. Each new application should extend RoboFrame at predefined extension points, more precisely by means of extending (abstract) base classes. RoboFrame consists of two parts: RoboApp is the base for any high level software running on the robot, while RoboGui is the base for a graphical user interface. The latter will not be discussed any further as this is not the target of this research.

Additionally, the RoboFrame framework by default provides data logging, debugging and configuration file interpretation services. The framework can also be extended with a finite state machine engine called XABSL (Risler and Stryk, von, 2008).

[2] The investigated package does not specify a version.

4.4.1 Software architecture

RoboFrame focuses on partitioning functional parts in loosely coupled modules by providing flexible and reliable communication mechanisms for data exchange between modules, which are based on messages (ring buffers) or shared memory (black boards).

The ring-buffers implement one-to-one or one-to-many location independent communication between modules. The modules themselves specify the demanded and provided type of data and are therefore not dependent on each other. The data exchange mechanism is part of the architecture and predefined. However, the data types should be specified in the form of classes, which will be serialized into a byte stream and later deserialized at the destination.

Via a blackboard an asynchronous many-to-many channel can be created. The data to be exchanged is placed in shared memory, which is more suitable if large data structures are modified incrementally (for example map processing tasks). Unfortunately this form of communication will most of the time, besides the communication primitives, need some form of custom arbitration (e.g. read and/or write locks) in contrast to message passing. Additionally, the blackboards are limited to their particular RoboFrame instance.

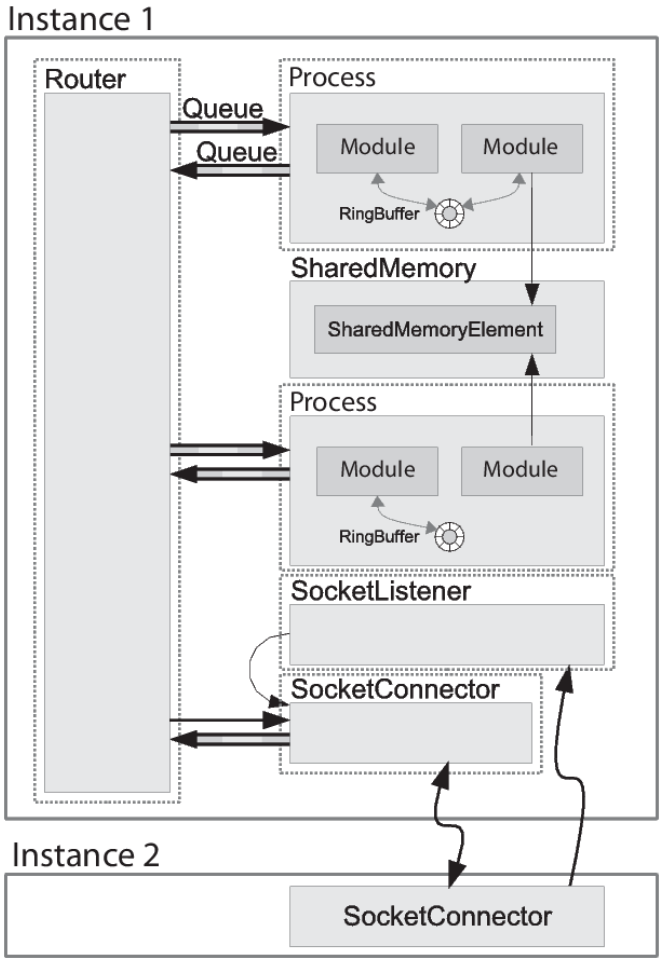

Figure 4.6: Example instantiation of the RoboFrame framework with router (Petters and Thomas, 2005)

The router is at the heart of the architecture by managing the data exchange and execution order. Figure 4.6 shows an example implementation. Modules have been assigned to threads, which can hold multiple modules. These modules can be executed periodically, with a delay or when data was received. Multiple modules inside one thread will be executed sequentially, except when a receive trigger is used.

4.4.2 Platform independence

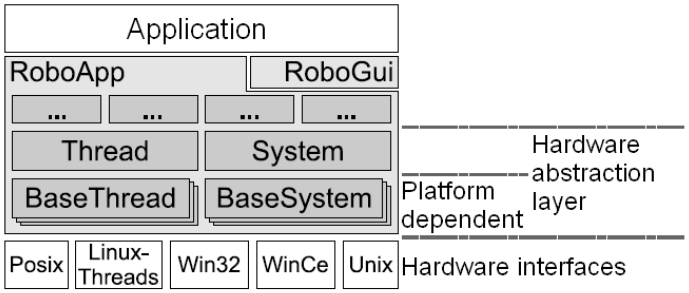

Figure 4.7: RoboFrame hardware abstraction layer (Petters and Thomas, 2005)

RoboApp has a platform abstraction layer which encapsulates all platform specific calls made to the operating system, see Figure 4.7. Currently a variety of operating systems are supported, like Linux, FreeBSD and Windows 2000/XP/CE 5, but unfortunately no hard real-time OS.

The platform independence has been achieved by applying the proxy design pattern, whereby the platform independent object is just a front. Function calls made to the platform independent object are redirected towards a separate platform specific implementation object. Thus incurring another function call as overhead.

The sources for the different platforms are clearly separated, which is nice from a design and maintenance point of view. However, macros are needed to join the platform specific implementation with the platform independent proxy-object.

4.4.3 Timing & real-time capabilities

As reported earlier, the framework itself does not provide any means of real-time support. The recommended approach for creating real-time behavior is using a hard real-time operating system (e.g. RTAI) and adding an additional hard real-time thread outside the RoboApp in the operating system. The RoboApp and hard real-time thread can then communicate via, for example, shared memory (see Figure 4.8). The hard real-time thread would then be suitable to implement control loops for hardware actuation and the RoboApp can handle the rest of the robotic application in soft real-time. This approach has been successfully implemented (Lootsma, 2008) in TUlip, the University of Twente's soccer robot.

Another solution would be to implement the platform specific objects for a hard real-time OS. But, the provided dynamic communication channels are not usable, because Run-Time Type Information (RTTI) is required for this (which is not deterministic). Furthermore, multiple modules on one thread are not advisable anymore, because a blocking system call will stall all modules of the particular thread.

4.4.4 Scalability & extensibility

The described router structure, with ring buffers and blackboards, enables the robot programmer to create a modular application. This approach, to split the application in multiple loosely coupled functional parts, provides a straight forward design pattern and is easy to comprehend. However, besides this and the platform independence to implement it, the framework does not offer any other functionality.

RoboApp has been designed to run with low overhead to allow deployment on systems with low computational capacity. This is certainly a desirable feature, especially if one intends to extend the framework.

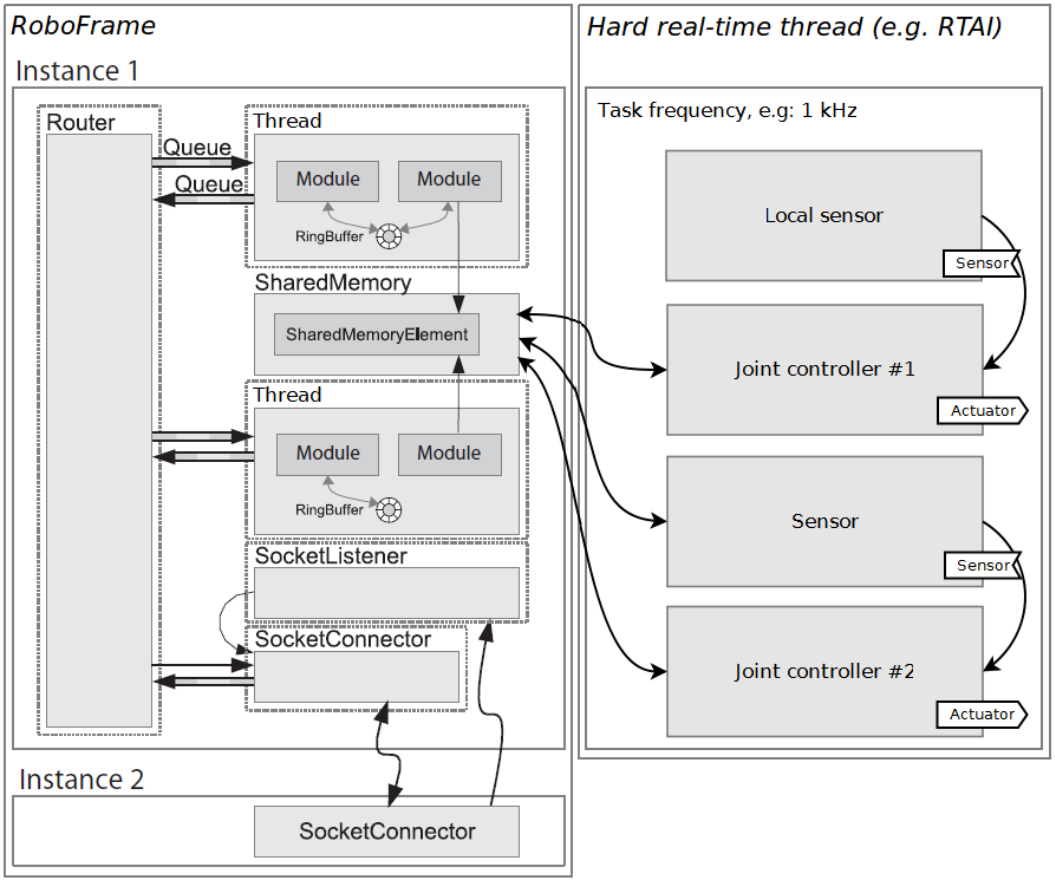

Figure 4.8: Example RoboFrame application with hard real-time support (based on: Petters and Thomas, 2005)

The framework can be used to implement CSP, but not on a low level. The blocking processes will require an own module, which will be very resource consuming. So, as in Orocos' case, an interface with a complete CSP model implementation will only be feasible.

4.5 CT library

The Communicating Threads[3] (CT) library was developed to bring the Occam constructs, and inherently the CSP constructs, to other platforms than transputers. It was first developed in the Java programming language by Hilderink et al. (1999) and therefore was not real-time at all. Hereafter, versions in C and C++ were created. Later the Java and C versions were abandoned in favor of the C++ version. The library has been restructured a couple of times. The current software version is still, more or less, based on the design by Orlic and Broenink (2004).

To aid the development process of complex concurrent systems a graphical modelling tool, called gCSP (Jovanović et al., 2004), was developed. An animation framework (Steen, van der et al., 2008) was later added to monitor gCSP applications and facilitate debugging. However, both are considered not of interest to this research and will not be discussed any further.

Because the author and the CE group are most familiar with the CT library and because the CT library matches best with the new frameworks requirements, it will be inspected in greater detail than the other frameworks. This might sometimes seem unfair, but it is done to get an 'improved CT library' in the end. Furthermore, this section will also highlight a few peculiarities where the motivation given in Chapter 1 is based on.

[3]Version: CTC++ / RestructuredCT

4.5.1 Software architecture

The current CT library has been designed to execute CSP constructions in hard real-time. Although its commonly used name is 'CT library', it is actually a framework as the function call direction in the software is from the CT framework to extension points (*Construct*s).

The initial requirements for the design included complete platform independence, meaning that it should be able to run by itself on MS-DOS and DSP processors without an OS. Additionally, at that time computing resources were very limited and mostly single core/CPU architectures. Therefore it was chosen to run the main CSP model always in one OS thread. Consequently, the CT library now cannot make optimal use of newer multiple core/CPU architectures.

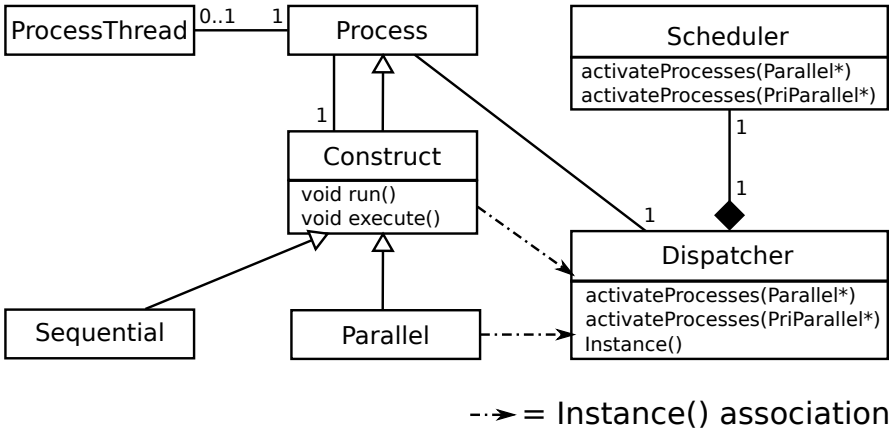

Figure 4.9: Partial UML diagram of the CT library

The requirement for complete platform independence also inspired the use of a kernel based design. The CT kernel is responsible for scheduling and dispatching CSP processes. The scheduler contains a prioritized FIFO scheduler and is non preemptive. Unfortunately, the CT scheduler is made completely aware of CSP, which is shown in Figure 4.9. Whenever a *Parallel* CSP process is run it will call the *activateProcesses(this)*, which will forward the function call to the *Scheduler*. The *Scheduler* must then either known how to place the *Parallel*'s CSP processes on its *PriorityQueue* (which is not shown) or how to run them immediately. The rest of the *Scheduler* interface, which is not shown, is targeted at CSP process activations. Thus, it is not easy to use the CT library for anything else than CSP.

In the *Construct* interface the main run method has been split. The *execute()* function contains code to manage the framework itself and the *run()* function should be overloaded with a particular implementation, for example the *Sequential* semantics. The advantage is that any particular CSP process can immediately call another CSP process' *run()* function or request it to be scheduled as a separate thread. This reduces the number of context switches between CSP processes to only the ones between parallel processes.

Unfortunately, the design is now considered outdated because of a number of reasons:

1. As already was mentioned, the design cannot easily be changed to take advantage of multi core/CPU architectures.
2. The design contains some strange constructions from a software engineering point of view. These are probably accidentally created as a consequence of its many year construction period and non-existing documentation. A few will be pointed out using a partial UML diagram of the CT library, which is shown in Figure 4.9. A strange association relationship can be seen between the *Process* and *Construct*, which is actually not necessary. The dashed arrows indicate Singleton function calls, which are overkill because

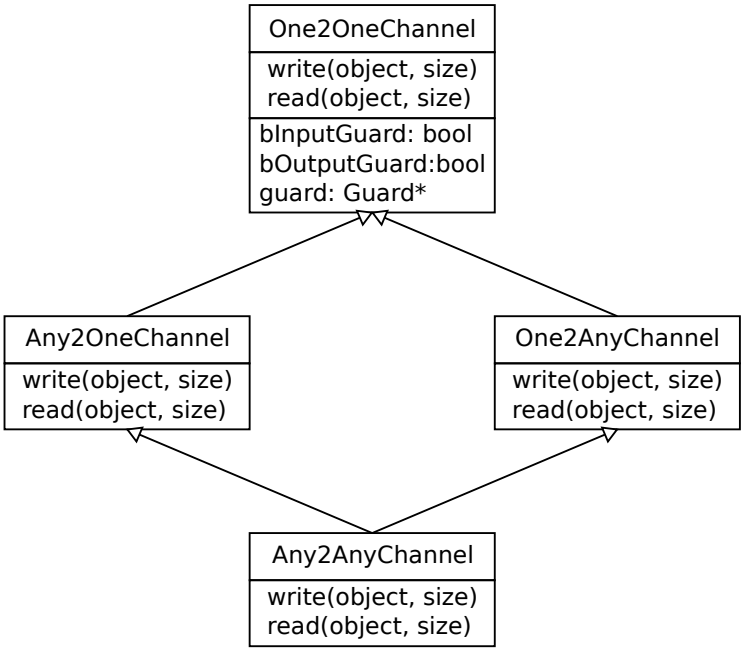

Figure 4.10: Class hierarchy of channels, source: Orlic and Broenink (2004)

the parent *Process* also contains an association relationship with the Dispatcher. A minor point and just an example, the design is considered wasteful since only *Parallel* constructs use a *ProcessThread*, but every *Process* has the association relationship and therefore an extra data field in their object data.

3. The channel design (shown in Figure 4.10) is undesirable since a, so called, 'dreaded diamond' class structure is created. The *One2OneChannel* class appears twice in the *Any2AnyChannel*, which can lead to ambiguities and therefore special and wasteful precautions have to be taken.

Summing all reasons given above, which is not a complete list, it was concluded that a new design is appropriate. Nonetheless, lessons can be learned from its advantages and 'mistakes', and these will be taken into account implicitly for the new design.

4.5.2 Platform independence

The CT library currently supports Linux, RTAI, Xenomai and Windows. In theory DOS and ADSP are also supported, but these have not been used lately and are therefore not further considered.

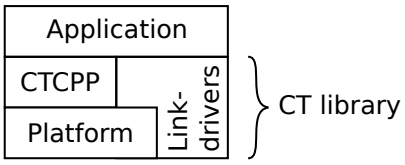

Figure 4.11: Top-level view of the CT library

The CT library is divided on the top-level into three categories on which an application can be build, see Figure 4.11. The Platform contains platform specific implementations, such as for threads and locking. The CTCPP holds the CSP language implementation and is highly integrated with the platform abstraction and the other way around. The CSP Link-drivers are partially build on top of the platform abstraction and partially interface with the underlying hard-

ware directly. The CTCPP uses these Link-drivers to interface with external hardware, such as sensors and actuators. The CTCPP, Platform and Link-drivers all are placed in separate folders, but are still highly dependent on each other.

The framework defines a few generic interfaces and classes for platform abstraction, but these seem to be designed for CSP-based applications only. Macros are used to select the correct platform (abstraction) headerfiles and the framework developer specifies the correct sources in the build system for compilation.

4.5.3 Timing & real-time capabilities

The library does not supply methods to specify real-time constraints for (sub-parts of) a design. So, one cannot choose nor mix hard real-time and non real-time threads themselves. Prioritized alternative and parallel CSP processes can be used to specify priorities in a CSP design.

CSP algebra has no notion of timing. However, the library has time support added by writing to external linkdrivers which use an OS timer. These timer interrupts and other external events have to be delivered to the appropriate CSP process. But, due to the single OS thread design, the scheduler cannot guarantee when this event is handled. Regrettably, this sometimes results in inadequate timing behavior.

The other way around, the library has explicit means to let hard real-time CSP constructs interface with a few pre-identified non real-time OS functions.

The gCSP modelling tool has the ability to generate CSP algebra from a model and this can be verified against deadlocks and live-locks by formal model checkers, for instance the Failure Divergences Refinement (FDR) (Formal Systems (Europe) Limited, 2008) tool.

4.5.4 Scalability & extensibility

One of the main requirements for the library was complete platform independence, which also included embedded architectures unable to run an OS. Resultingly, the library size was kept small and therefore is considered to scale well.

The CT library was specifically designed to enable the execution of CSP constructs. In its current form, other use cases than executing CSP models are virtually impossible.

4.6 Boost

Boost[4] is a free peer-reviewed portable set of C++ libraries (Boost, 2011). The libraries are intended to work well with the C++ standard library. Furthermore, they are intended to be widely useful and usable across a broad spectrum of applications. According to its website, its license encourages both commercial and non-commercial use.

The aim of the project is to establish 'existing practice' and provide reference implementations so that Boost libraries are suitable for eventual standardization. Ten Boost libraries are already included in the C++ Standards Committee's Library Technical Report (TR1) and will be in the new C++0x Standard, now being finalized. Additional Boost libraries are proposed for TR2.

4.6.1 Software architecture

The Boost eco-system contains over 100 tightly integrated C++ libraries, with among others functionality for threading, locking and math operations. Some of these are suitable for hard real-time applications as they adhere to the hard real-time programming restrictions (for details see Section 5.1.5).

[4]Version: 1.44

The libraries intended for extension, such as the iterators library, use template meta programming (TMP) concepts. The use of templates in general can be thought of as compile-time execution, because the requested functionality will be hard coded into the executable by the compiler. The advantage of using templates therefore is great compile time flexibility while maintaining execution speed. Most common uses of templates are to change the class policy or to specify the input/output types for the resulting template class(es). The consequence of using TMP is that its functionality must be implement in the headers. This manifests in the whole Boost library as most functional code is in header files. The downside is that programming faults only become visible when its specific part of code is used, since only then the code in the headers is compiled, which could make the library more error prone. To prevent these and other errors, the Boost distribution includes a test suite. Strangely, the current version of the test reports some errors, but the community users suggest that this is not a problem.

As stated before, the aim of Boost is to establish 'existing practice'. This includes prototyping new language extensions, such as lambda functions and expressions for the new C++0x standard. To test their use cases they, of course, have to be implemented. Unfortunately macros were used. As already explained, macros are difficult to maintain, but there is not a better alternative for this specific problem yet. Nevertheless, experimental functions should not be part of any framework intended to be production quality at some point.

For libraries to be included into Boost, they first are subjected to a peer reviewing system. The Boost community will be asked to give feedback (on use cases, et cetera) and suggest improvements. After several successful review rounds the library is added to the Boost distribution.

4.6.2 Platform independence

Boost consists of multiple libraries which themselves have separate folders for specific platforms and compilers. Their main headers use macro redirections toward the real headers in the sub folders, which contain the actual functionality. Boost.Build, a text-based system for developing, testing and installing software, will take care that the right macros are set and source files are compiled (if present), to achieve platform independence.

4.6.3 Timing & real-time capabilities

As stated before, some libraries are usable for hard real-time applications. It is not feasible to investigate them all. Therefore only the threading library will be discussed in this section, as this is one of the main features for the new framework. The other libraries should be inspected manually, when they seem of interest, to determine if they adhere to the real-time programming restrictions (Section 5.1.5).

Boost.Thread (as it is called in Boost) enables the use of multiple threads of execution with shared data in portable C++ code. It provides classes and functions for managing the threads themselves, along with other functions to synchronize data between the threads or providing separate copies of data specific to individual threads. Only POSIX and Windows threads are supported at the moment. The POSIX skin in Xenomai/RTAI can be used to achieve hard real-time other than with POSIX compatible real-time OSs. However, the application can then only run in hard real-time. So, requirement 3.2.1.2 cannot be satisfied with Boost.

4.6.4 Scalability & extensibility

The Boost libraries seem scalable and easy to extend, but code inspections have shown that the libraries are tightly integrated. For example, the Date&Time library, among others, depends on the Serialization library which is dependent on the Preprocessor library and others. This means that one cannot pull-out a specific library to use in the new framework. Though, due to the library mainly consisting of headers, the headers and functions not used will not incur any overhead.

The Boost libraries do not provide any means to interface with hardware nor implement any safety features.

Boost does not contain a rendezvous communication library. But semaphores are included, which can be used to implement rendezvous communication for CSP. Furthermore, the thread library provides support for parallel CSP constructs in either hard or non real-time.

4.7 POCO

POCO[5] is a collection of C++ class libraries (POCO, 2011), similar in concept to the Java Class Library or the .NET Framework. Its focus is on the development of network-centric, portable applications in C++. The libraries integrate well with the C++ Standard Library and fill many of the functional gaps left open by it, according to its authors (POCO, 2010). Furthermore, POCO is open source, licensed under the Boost software license, and thus also completely free for both commercial and non-commercial use.

4.7.1 Software architecture

POCO is aimed at building network and internet based applications which is also reflected in the partitioning of the components, see Figure 4.12. The heart of the libraries is the Foundation layer. It provides numerous platform independent classes for cross-platform programming.

Other libraries, providing higher-level functions, are build upon the Foundation library. These libraries include, among others, various network protocols, servers (like HTTP), XML parsers & writers and configuration file processors. These other libraries are not in the scope of this research and therefore not further considered.

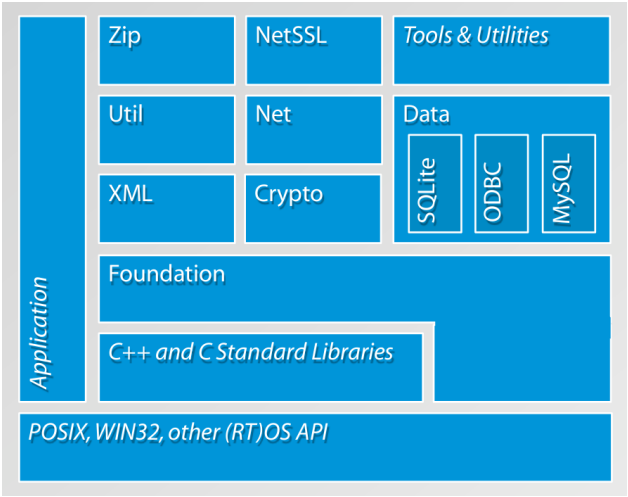

Figure 4.12: POCO software layers, source: POCO (2011)

4.7.2 Platform independence

The Foundation library is a thin platform abstraction layer, that encapsulates OS functionality in objects. These objects consist of two parts joined by inheritance. The derived class specifies the platform independent interface and the base class provides the platform specific implementation for this. Thus, an inverted inheritance relation is created, which violates general programming concepts. The OS specific implementations contain a suffix in their file name to denote the intended platform. The correct files are selected and included by macros, and this principle is also used for source files.

[5] Version: 1.3.6

POCO has a wide range of supported OSs: Microsoft Windows, Linux, Mac OS X, Embedded Linux (uClibc, glibc), iOS, Windows Embedded CE and QNX. Additionally, the library can be patched to support the HP-UX, Solaris and AIX OSs. Without considering the patches OSs can be divided into two categories: POSIX compliant and Windows environments.

4.7.3 Timing & real-time capabilities

Code inspections have shown that the Foundation library is a thin layer around OS functionality and therefore considered real-time capable. The other libraries should be inspected separately, but most of them are intended for non real-time purposes.

The Foundation library contains threading support, including the specification of several priority levels and timers. Besides Windows Embedded CE and the POSIX conformant QNX OS, no other real-time OSs are mentioned. As explained before (Section 4.6.3), the POSIX skin in Xenomai/RTAI can also be used to achieve hard real-time. Although, then requirement 3.2.1.2 cannot be satisfied with POCO as well.

4.7.4 Scalability & extensibility

The POCO project favors simplicity over complexity, by which is meant 'as simple as possible, but not simpler'. So, POCO is considered less configurable as Boost, but this also has its advantages. The POCO libraries seem much less intertwined than the Boost libraries. Due to the lower configurability demand and thereby non templated classes, far more functionality is placed in source files and these are immediately checked by the compiler for simple programming mistakes. Furthermore, the POCO classes are considered easy to comprehend and are therefore easier to extend.

The libraries do not provide any means to interface with hardware nor implement any safety features.

POCO's build system can be used to selectively use only those libraries needed for a particular application. There is no need to always embed the overhead of the entire package in the application unless it is really needed.

POCO does not contain rendezvous communication, but semaphores are included. Therefore, the same applies to POCO as for Boost with respect to CSP.

4.8 Conclusions

This section combines the results of the framework and library investigations in the form of a comparison. Also, some potential features for the new framework will be highlighted.

4.8.1 Frameworks

Table 4.3 shows the comparison between the frameworks. The scoring system is the same as in the introduction of this chapter, reasons for the individual scoring will be explained next.

Feature	Orocos	RoboFrame	CT
Platform independence	++	+	+
Hard real-time	+	+/-	+
Scalability and extensibility	+/-	+	-
Usability with respect to embedded systems	+/-	+/-	-
Usability with respect to CSP	-	-	++

Table 4.3: Frameworks compared in characteristics

All three frameworks use some form of platform abstraction and build classes on top of it. As mentioned before, only Orocos has specific implementations for other than POSIX compliant real-time OSs, which is more difficult to represent platform independently.

None of the frameworks satisfies requirement 3.2.1.2, which states that it should be possible to specify a mixture of hard/soft/non real-time constraint threads. Additionally, RoboFrame must be extended to provide hard real-time features at all.

The scalability and extensibility of Orocos is doubtful, because it already has multiple extension points. To clarify, this is an advantage because extension points are clear and predefined, but on the other hand a disadvantage because one is also limited by them. Furthermore, it is dubious if it will scale well for big projects as there is no formal verification method available. RoboFrame, on the other hand, only has one lightweight design pattern to offer. The advantage is one can extend the framework in all kinds of ways, but at the same time not much support is given by the framework. Because the framework is so small the 'not a formal verification method available' does not apply here, because one can still venture to any particular tool. The CT library is not extensible at all, because it has been designed for CSP only. But, there is tooling available to formally verify models created for it. Considering, the CT library can only use one thread it does not scale well on a software level.

The usability of Orocos for embedded systems is considered doubtful, although it has different deployment methods which should be suitable for embedded system, because one can argue that in this form it is not Orocos anymore (i.e. its advantages are discarded). Since, the RoboFrame framework is considered lightweight but does not offer functionality to interface with hardware, its use for embedded system is considered questionable. For the CT library an embedded and an OS-less embedded version exist, but the status of the latter is unknown. Furthermore, as it cannot make us of multi-core/CPU architectures, its usability is poor for modern embedded architectures.

The usability for CSP of Orocos and RoboFrame is considered less probable, because only complete (sub)models can be efficiently interfaced with their respective components/modules. The CT library, on the other hand, was build for a single purpose: CSP.

Thus, considering both Table 4.3 and the requirements from Chapter 3, it can be concluded that designing a new framework is a viable choice.

4.8.2 Libraries

Table 4.4 shows the comparison between the libraries. Again, the scoring system is the same as in the introduction of this chapter, but the reasons for the individual scoring will be explained next.

Feature	Boost	POCO
Platform independence	+	+
Hard real-time	+/-	+/-
Scalability & extensibility	+	++
Usability with respect to embedded systems	+/-	+/-
Usability with respect to CSP	+/-	+/-

Table 4.4: Libraries compared in characteristics

Both, Boost and POCO, provide platform independence via a small abstraction layer and only support POSIX compliant and Windows OSs, therefore their platform independence is a +.

Neither Boost nor POCO is designed for hard real-time, but some libraries may be usable in this context.

The scalability and extensibility of Boost is considered positive, due to its smart use of TMP. However, it is also experienced that excessive use of TMP makes the code harder to understand. POCO also uses TMP, but to a lesser extend. Furthermore, the POCO libraries seem less intertwined and dependent on each other. This is also favorable if one tries to determine if a library adheres to real-time constraints.

The usability of Boost with respect to embedded systems is doubtful, because of the coherence of the separate libraries. This makes it difficult to understand what is really going on and libraries cannot be removed in Boost without breaking the implementation. POCO's design is focused more on separate libraries, though all are dependent on the platform abstraction layer. Libraries can be disabled in the POCO build system. Neither provides any means to interface with hardware, so their usefulness for embedded systems is doubtful.

The Boost and POCO libraries both do not offer rendezvous communication by default. But, this can be implemented manually by using other features from the respective libraries.

Thus, the POCO and Boost libraries cannot directly be used for the new framework. Primarily, because they both use C++ language exceptions, which is not allowed for hard real-time software. Nonetheless, they can serve as good examples for the design.

4.8.3 Concluding remarks with respect to the new design

Separating the new framework in functional components or modules, as is proposed by RoboFrame and Orocos, is a good design choice. In this manner the component name can specify what it does and the functionality can be easily grouped by it. Furthermore, components can be reusable if properly designed.

Component inter-dependencies should preferably be vertical in a layered approach to prevent circular dependencies and intertwined code. Another disadvantage of circular dependencies is that components cannot be changed anymore without a great deal of maintenance to other components as well. So, in Figure 4.13, CSP may dependent on Threading but not on State Machine. A desired feature for the build system is to make inter-dependencies explicit and detect incorrect use of them.

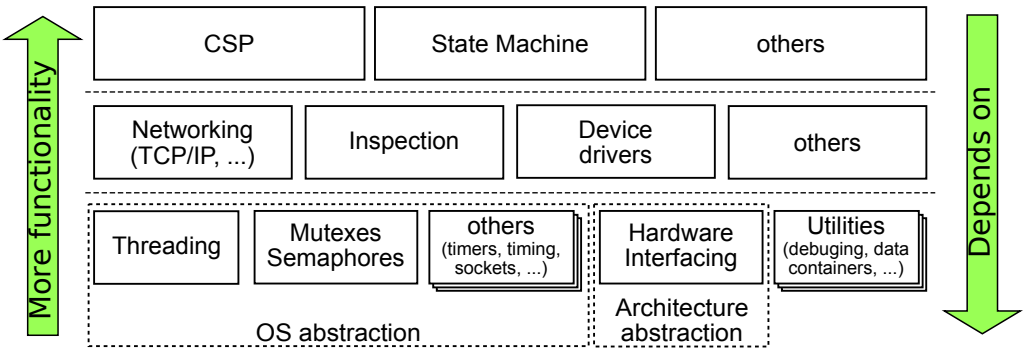

Figure 4.13: Example: Layered component approach

A platform abstraction layer can be designed with negligible overhead, as was reported for instance for POCO. Besides, object encapsulation in platform independent objects looks promising, because all investigated frameworks and libraries used this approach more or less.

Four different programming styles can identified for designing platform independent classes:

- *Function skins - Orocos*
  Abstract function names (not equal to OS function names) make this approach platform independent. The abstract function names are implemented for each platform.
- *Generic class + file overloading - Boost / CT library*[6]
  A generic class/interface is defined, whereby macros or its build system are used to select the appropriate implementation file.
- *Inverted inheritance - Poco*
  The derived class specifies a more generic interface than the base class. So, the base class provides the platform specific implementation.
- *Proxy design patterns - RoboFrame*
  Function calls to the platform independent object are redirected towards a platform dependent implementation object.

The last two methods are not advisable, because the inverted inheritance violates general programming concepts and proxy-objects incur overhead.

Clear interfaces should be designed for platform independent classes, where platform dependent classes must conform to. Otherwise, implementations might drift apart.

The new framework should not contain any C++ language exceptions and therefore Boost and POCO cannot be used. Furthermore, some C++ Standard Library (STL) use exceptions as well and other C++ language implementation might not be guaranteed to be deterministic, so one must be careful in using either of them.

From the past framework and library discussions it can be concluded that, generally speaking, perfect frameworks or libraries do not exist, because at some point trade-offs have to be made. These trade-offs might have to do with, for example, limited development time or features versus limited space. So, the requirements from Chapter 3 should be targeted specifically, trade-offs should be listed and the framework should be marketed accordingly.

[6] The other frameworks use this as well, but less explicit.

5 Design & implementation

The new framework is called LUNA, which stands for 'LUNA is a Universal Networking Architecture'. This chapter discusses its design and implementation. First, the general architecture of the framework and approach which led to the new design is explained. Next, the components which are interesting from a design point of view are discussed in detail. The other components section explains the remaining components, whose designs are considered straightforward. Last, a conclusion is given discussing the overall design.

5.1 Architecture and approach

The LUNA framework requirements (Section 3.2.1) stipulate a number of functions the framework should fulfill as well as supporting multiple target platforms. Some requirements are fairly simple to implement as most operating systems already provide this functionality. Others might be complex. Therefore, the design is based on the divide and conquer principle, meaning to divide the framework in smaller functional components and basing more complex components on top of the simpler ones. This way a complex and demanding framework can be systematically build from the ground on up.

The advantage of this approach is that functional units can be identified, which have a clear purpose and themselves provide the platform independence. Since not all applications will require the same functionality, the components can be enabled/disabled depending on the intended application.

5.1.1 Component based design

The functional components can basically be divided into three categories, see Figure 5.1: core, high-level and execution engine components. The grey components are not yet implemented in LUNA.

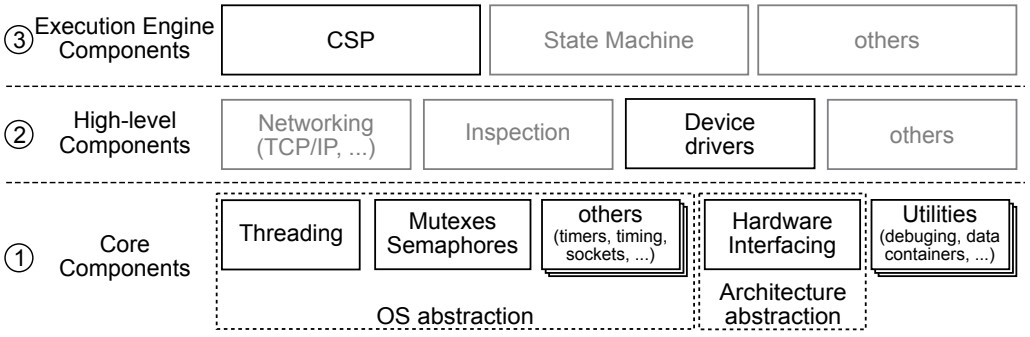

Figure 5.1: LUNA software layers

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 are available to support the target operating system (OS), like threading, mutexes, timers and timing. The architecture abstraction components provide support for features specific to an architecture (or hardware platform), like the support for (digital) input and output (I/O) possibilities. Other components can make use of these core components to make use of platform specific features without knowledge of the actual chosen platform. Another group of core components are the Utility components, implementing features like debugging, generic interfaces and data containers.

The next level contains the High-level Components (2). These are platform independent by implementing functionality using the core components. An example is the Networking component, providing networking functionality and protocols. This typically uses a socket component as platform-dependent glue and build (high-level) protocols upon these sockets.

The Execution Engine Components (3) implement (complex) execution engines, which are used to determine the flow of the application. For example a CSP component provides constructs to have a CSP-based execution flow. The CSP component typically uses the core components for threading, mutexes and so on and it uses high-level components like networking to implement networked rendezvous channels.

5.1.2 Generic OS abstraction

Some OS functions may be available on multiple OSs. The POSIX standards, already mentioned in Chapter 2, are the most well known. The considered POSIX standard (IEEE Std 1003.1, 2004) consists of the previously separate POSIX.1 (core), POSIX.1b (real-time extensions) and POSIX.1c (threads) standards and the amendments 1d (additional real-time extensions), 1g (protocol independent interfaces), 1j (advanced real-time extensions) and 1q (tracing). These standards only specify interfaces and semantics, not how they should be implemented. Other standards apply mostly to a subset of OSs, like for Unix systems the Unix98 mark (Open Group, The, 1998).

Table 5.1 lists the POSIX implementation for the intended target and development OSs. Unfortunately, both RTAI and Xenomai do not state which version they are exactly compliant to. The QNX OS is certified conformant (which is a stronger condition than compliance) to the PSE52 Realtime Profile standard (IEEE Std 1003.13-2003), which is even more elaborate and demanding than the compared POSIX.1 + amendments standards.

Operating System	**Hard real-time**	**POSIX.1 +b +c (IEEE Std 1003.1, 2004)**
Linux	No	Fully compliant
Xenomai	Yes	Partially compliant, via POSIX skin
RTAI	Yes	Partially compliant, via a special header file
QNX	Yes	Fully certified conformant
Windows	No	Only through Cygwin (2011) and others.
Mac OSX	No	Fully compliant

Table 5.1: Platform abstraction for the target and development OSs

Looking at Table 5.1 one can conclude that the new framework should use the POSIX standard, because then the required OSs will be automatically supported. This is for a large part true, but for the combination of Linux and RTAI or Xenomai this is non straightforward. Linux provides non real-time and RTAI or Xenomai will be responsible for hard real-time execution. Thus a method to specify hard real-time and non real-time threads will be required.

RTAI and Xenomai both advertise their hard real-time functionality via two interfaces. The first is a POSIX layer of which functions will overlap with the Linux POSIX API. This actually is desired to enable easy application development. One can then build and test a program on a development PC and later transfer it almost without changes to the target. The function overlap is then resolved by linking the application to the right implementation. Secondly, both also specify additional functions for use in hard real-time. The idea behind these functions is that these are faster if applied only in hard real-time mode due to their specialization.

Thus, the new framework should probably make use of the POSIX functionality for its first implementation as this the standard is generally accepted and therefore will partially enable a lot of OSs. Nonetheless, an abstraction layer around the POSIX functions in the form of a (lightweight) object is preferable to ensure that in the future other OSs and non-POSIX conforming functionality can be supported as well.

5.1.3 Platform independence and the build system

The divide and conquer approach has the following properties regarding the framework development:

- Components can be identified and designed as functional units, so that each component has a clear set of use cases and interfaces.
- Due to the specification in functional units, components can possibly be enabled/disabled individually. This is advantageous for example when a particular component is not available or implemented for a platform.
- The platform independence can be handled in layers. The core components will interface with the particular OS and hardware directly, and the other layers will access the platform through the abstraction layer. The advantage is that high-level and execution engine components can be based on the core components without considering the particular OS.
- Investigation of the multiple OSs in the previous section has shown that a lot of OS functions can be grouped in functional components and that these will probably not overlap with other components. Thus, functional units can be designed such that their platform abstraction is self contained.
- The divide and conquer approach results in unidirectional acyclic component dependencies. If a bidirectional relationship is required the components should be taken together, as they complement each other and can be considered one. For cyclic dependencies spanning multiple components the division in functional components has failed and should be reconsidered.

The build system used for LUNA is a modified version of OpenWrt (OpenWrt, 2011; Fainelli, 2008). The regular OpenWrt contains, among others, the following required functionalities:

- Separate package specification.
- Directed acyclic dependency specification in packages and automatically enforcing these.
- Import functionality for external software, like libraries and tools.
- Generalized menu structure for making settings and selecting packages.
- Multiple architecture support

Bezemer renamed packages into components and added, among others, the following required functionalities:

- Operating system dependency specification and thereby importing the correct component sources.
- Specification of component tests.
- Automated test execution depending on the selected components.
- Support for the QNX compiler.
- Documentation generation of the selected components
- Default processing operations for library components, which results in minimal Makefiles

Using the build systems editor ('make menuconfig' under Linux-like systems) an application developer can select components and set the intended target OS and other settings, for example if debugging output should be enabled. The values of these selections are also available as macros for use in the headers and sources if more demanding implementations are required than the use case explained below. Due to the dependency specification, not completely im-

plemented components and their descendants are automatically not shown in the build system menu. This will aid the application developer when selecting a platform and perhaps encourage to contribute to the project as it is clear what has to be done to enable functionality.

The rest of the build system capabilities are best explained with an example component's folder structure, see Figure 5.2. The components directory houses all LUNA components. In each component a 'Makefile' is used to define the complete component: headers, sources, dependencies, et cetera. Optionally a 'Config.in' can be provided to extend the build system menu for the component. The example component *comp1* provides a generic header, which will be used in most cases as will be explained later. Its platform directory can contain multiple OS specific header and source files. The files that matches the OS settings best are chosen for the application. The *posix* folder is used whenever no specific OS files are given, as the OS is POSIX compatible. The build system collecting the correct header and source files can be seen as overloading and is well suited to manage platform independence.

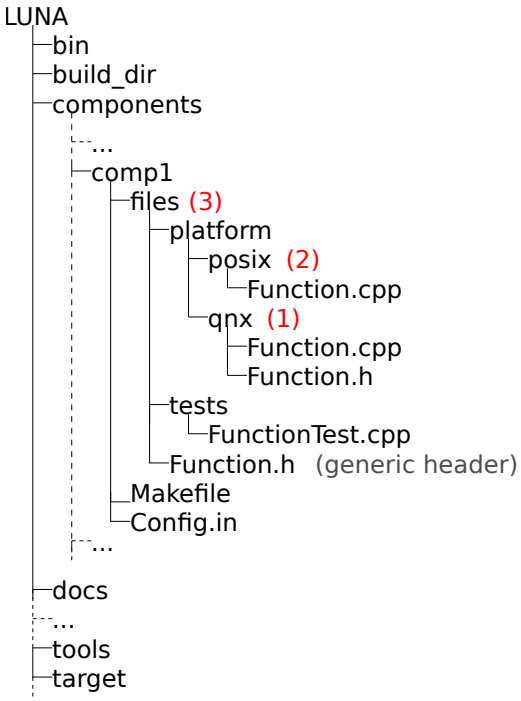

Figure 5.2: LUNA build system folders

For example, an application intended to run on the QNX will make the build system search for the files in the order: 1, 2 and 3 in Figure 5.2. Thereby, it will use the *Function.h* file in the *qnx* platform directory instead of the generic header. For POSIX compatible OSs (minus QNX) it only looks in folders 2 and 3. And for non-POSIX OSes only the generic header is found in folder 3.

Every component can have its own unit tests, specified in the *tests* folder. Whether a test was successful or not is specified by its return code, 0 means success and all others failure. The unit tests can be executed automatically or only be compiled. The latter is chosen automatically whenever the target OS is not the current OS type. Unit tests have a time limit by default, if the test takes too long a deadlock is presumed and the test is canceled (and has failed by definition).

The selected components and tests are compiled separately from the original files in the *build_dir*. One of the reasons is to prevent the programmer from accidentally deleting sources files while cleaning the source folder. External software can be downloaded from websites or

repositories to be included with LUNA. The *bin* directory contains any specifically build tests, the compiled LUNA framework and its collection of headers.

The build system currently uses Doxygen to generate documentation from header and source files. The *docs* folder can be used to extend this documentation with extra files (without any accompanying components).

To conclude, the build systems implements the functional requirement of external code and/or library integration (Section 3.2.1.9) and enables the functional requirement of self testing (Section 3.2.1.13). Most importantly, by properly using the build system the required platform independence (Section 3.2.1.1) can be achieved.

5.1.4 Identified framework components

The identified functional components are listed in Table 5.2. The core, high-level and execution engine columns show the component layer classification (Section 5.1.1).

Component	Core	High-level	Execution engine	Purpose	Documentation
AnyIO		+		Mesa Anything IO link drivers	Sec 5.3.7 and App G
Atomic	+			Atomic (integer) operations	App C
Barrier	+			Barrier synchronisation primitive	Sec 5.3.1
Clock	+			Time specification and wall clock	Sec 5.3.2
Communication	+			Rendezvous communication	Sec 5.2.4
CSP			+	CSP execution support	Sec 5.4
Debug	+			Debugging support	Sec 5.2.2
Device-manager		+		Device interface manager	Sec 5.3.6
Emergency-manager	+			Emergency handling functionality	Sec 5.2.9
Interfaces	+			Generic interfaces, e.g. to handle errors.	Sec 5.2.1
LockSync	+			Synchronisation primitives	Sec 5.2.3
RTLogger		+		Embedded real-time logging facility	Sec 5.2.12 and App H
Scheduler	+			Interface to the (OS) scheduler	Sec 5.2.8
Socket	+			TCP/UDP communication primitives	App 5.3.3
SystemInfo	+			System-specific information database	Sec 5.2.7
Threading	+			Thread support	Sec 5.2.10
ThreadPool		+		Pooled threads	Sec 5.3.4
Timer	+			Interface to the (OS) timer	Sec 5.2.6
Timing		+		Time measuring facilities	App 5.3.5
UThreading		+		User thread support	Sec 5.2.11
Utility	+			Generic utilities	Sec 5.2.5

Table 5.2: Component specification

5.1.5 Real-time programming

In Chapter 2 the concept of hard, soft and non real-time was explained. This section addresses how to program for hard real-time. Basically, for soft real-time the constraints given next do not apply anymore, if the resulting delays are acceptable. And for non real-time these do not apply at all, the results keep their value.

This text applies to the normal/full C++ standard (ISO/IEC 14882:2003), not the embedded version. The embedded C++ standard omits, in my opinion, the two most appreciated advantages over regular C: Templates, multiple inheritance and virtual inheritance.

The only criterion for hard real-time programming is that it must always result in deterministic timing. This cannot be empirically proven, because for 99.99% of the function invocations the code might return within a split second and other times it might for example stall indefinitely.

The following list, although probably not complete, is a guide for real-time programming:

- The use of a hard real-time OS does not magically make any code hard real-time.
- Particular code should preferably have O(1) or O(n) time complexity. Though, the latter should not be applied without careful consideration as the time increases linearly. Other time complexities should not be used or only when the operation is performed repeatedly with the same parameters and an equivalent data set. Thus, other time complexities as O(1), which are not inherently evident, should be documented and perhaps marked.
- Dynamic memory allocation tends to be non-deterministic. The time taken to allocate memory may not be predictable and the memory pool may become fragmented, resulting in unexpected allocation failures. The solution can be to pre-allocate all necessary memory before entering hard real-time or use TLSF: A dynamic memory allocator specifically designed to meet real-time requirements (Masmano et al., 2004).
- Thread/process synchronisation points (including rendezvous communication), should be designed with care. Concurrent processes will eventually need to work together or with the environment. In these cases, special precautions should be taken into account to prevent data corruption. A possible solution is to use synchronisation primitives, like a mutex or rendezvous communication, which guarantee exclusive access. However, these come at a price, because the thread will sometimes have to wait until another process finishes its exclusive action. In these cases one should prove that the overall solution is deadlock and lifelock free. An alternative could be to use lock-free implementations (Section 5.2.5.1) where possible.
- If a combination of a non real-time and hard real-time OS is chosen, the non real-time OS operations should be handled with care as they suffer from the same behavior as synchronisation, i.e. the OS might stall them. A standard solution it to assign a non real-time buddy thread, which reports the outcome via a real-time safe communication primitive when the action finished.
- C++ language exceptions should not be used, as explained in Appendix B. For this reason most C++ Standard Library (STL) implementations cannot be used.
- To extend on the previous point: Do not use C++ language facets which require dynamic run-time support, such as Run-Time Type Information (RTTI).
- Virtual functions and multiple inheritance are typically implemented efficiently (Meyers, 2010) and are suitable for hard real-time.
- Templates specify classes that will be created during compile time, thereafter they are statically available just as normal classes. Templates are therefore also suitable in hard real-time programming.
- Printing (debug) information to the screen can break real-time constraints for hard real-time applications, since it is quite slow on most OSs.

The new framework is designed according these guidelines and thereby satisfies the corresponding non-functional requirement of hard real-time programming constraints.

5.2 Detailed designs of components

This section discusses the components that are interesting from a design point of view. Section 5.3 contains the rest, which may still be integral to the framework overall design.

The general format for the detailed design sections is as follows: The introduction provides a reference to the requirements the component implements and describes the components features. Whenever interesting alternatives are considered, these will be given and discussed. The design section explains how the requirements and features have been achieved. As last, some implementation details are given, which mostly is a discussion on platform independence.

The order of the selected components is not alphabetically, but arranged to facilitate reading this thesis (see Table 5.3). Components required to understand other ones are presented first. Table 5.2 lists all components. The table shows their purpose and their absolute location in this thesis.

Detailed designs of components	**Other components**
Interfaces	Barrier
Debug	Clock
LockSync	Socket
Communication	ThreadPool
Utility	Timing
Timer	Device-manager
SystemInfo	AnyIO
Scheduler	
Emergency-manager	**5.4 CSP component**
Threading	
UThreading	**Appendix**
RTLogger	Atomic

Table 5.3: Order of appearance (top-down, left-right)

5.2.1 Interfaces

This component contains generically usable interfaces. Currently, it contains an interface for error code specification and an interface to prevent unwanted object copying.

The framework, the OS or both may at some point encounter a fault. Because the framework provides an abstraction layer for the OS, the system error codes are passed through LUNA.

5.2.1.1 Alternatives

For the error code specification, two alternatives can be thought of. Actually three, but C++ language Exceptions cannot be used to handle faults as is discussed in Appendix B.

The framework can provide a true abstraction layer and redefine the OS error codes in an error abstraction layer. In this case one has to document and match all error codes of all used OSs, which is a lot of work. Furthermore, the exact meaning of an error code might be dependent on the function returning it. Implementing this will be a lot of work.

The alternative is to return the OS error codes exactly as they are received and supplement the error code range with the framework's error codes. In this case the application developer must consult the OS or framework documentation in case an error is encountered. For this to work, OS and LUNA error codes must not overlap in the case that functions are nested, otherwise their origin might become unclear. The advantage is that the framework is always up to date. A minor disadvantage is that for all OSs the highest error code must be determined and maintained, to prevent error code overlapping.

5.2.1.2 Design

Error codes
The LUNA framework specifies additional error codes for function returns. The *LunaError* interface contains the OS' upper bound on error codes and defines its own error codes in the form of macros from this point on up. By looking at the range in which an error code falls the application developer can determine which documentation to consult.

Uncopyable
The *Uncopyable* interface is a dedicated interface to prevent copying of instantiated objects. The *Uncopyable* interface contains a private copy constructor and assignment operator which are not implemented, as explained in Meyers (2005a). Once a class definition inherits this interface its objects cannot be copied anymore as the compiler will give an error. This interface has for instance successfully been applied in Mutex objects (Section 5.2.3), which should remain unique.

5.2.1.3 Implementation

For the error codes the highest OS error code is determined manually. The POSIX standard also defines error return codes for functions, but its maximum depends on the implemented POSIX amendments.

The *Uncopyable* interface purely uses the C++ language and is therefore platform independent.

5.2.2 Debug

The Debug component provides the implementation for the debugging facilities requirement (Section 3.2.1.12). Debugging can be enabled/disabled for the whole framework in the build system. If debugging is disabled its statements do not add any overhead anymore. In combination with the RTLogger component (Section 5.2.12) the debug information can be rerouted to an off-target visualisation tool.

5.2.2.1 Alternatives

Standard C++ has two methods for printing information to any output: stdio or iostreams. The first originates from the C language and requires calling (special) output functions. The second uses I/O streams to route the information to the correct output. An example is given in Listing 5.1 where both should print 'Printing number 10.' to the screen.

```
// stdio:
printf("Printing number %d.\n", 10);

// iostreams:
cout << "Printing number " << 10 << "." << endl;
```

Listing 5.1: stdio versus iostreams sample

The requirements state that the debugging facilities should not result in overhead if disabled. The stdio functions can easily get wrapped in macros. If debugging is disabled and thereby the macros, the code is eliminated by the standard pre-processor before it enters the compiler. For the iostreams this will be difficult, perhaps one can reroute the iostream to some kind of sink. Unfortunately, this does not remove the overhead incurred by evaluating the code on the right hand side of *cout* in the example. So, the stdio alternative is chosen for debugging and logging in LUNA.

5.2.2.2 Design

The component design is split into C++ functions and a few macros. The former adds functionality to the standard stdio functions. The macros enable/disable the outputting of debug information based on a build system setting.

Three logging functions have been implemented: *log()*, *log_local()* and *log_if()*. The first two take a log level, formatted output string and the arguments to the string as parameters. The formatted output string and its arguments work similar to the well known *printf()* function. Multiple log levels have been defined for fine grained logging. The *log_if()* additionally takes a Boolean expression and depending on a successful outcome *log()* the rest of the parameters. The *log_local()* guarantees that any logged information will remain on the target, whereas the others might be rerouted to an off-target visualisation tool.

Macros with the same name in capital letters as the already presented functions are available. These will only call their lower capital letter equivalents if debugging is enabled during development. Otherwise they are transformed, by the pre-processor, into an empty line. So, when running the final version absolutely no overhead is incurred.

5.2.2.3 Implementation

The component is implemented with standard C++ functions and macros and therefore platform independent. Only for real-time operating systems the printing to the screen can possibly be real-time, but at least on QNX this is quite slow. An alternative, in this case, is to use the RT-Logger component to transfer any log messages to an off-target visualisation tool. In this way the real-time constraints on the target are fulfilled if a little overhead is allowed.

5.2.3 LockSync

This component specifies interfaces for the synchronization functional requirement (Section 3.2.1.7) and implements them for OS threads. Table 5.4 shows the POSIX synchronization methods implemented in LUNA. To facilitate locking, an automatic locking and unlocking object for Mutexes and Semaphores is added.

POSIX functionality	**LUNA class**
Mutual exclusion	*Mutex*
Condition variables	*Condition*
(Un)named semaphores	*Semaphore*

Table 5.4: POSIX synchronization functionality and its LUNA classes

5.2.3.1 Design

The UML class diagram in Figure 5.3 shows the component design. Parts of their POSIX function names are used for the object manipulation functions, for example *pthread_mutex_lock()* has become *Mutex::lock()*.

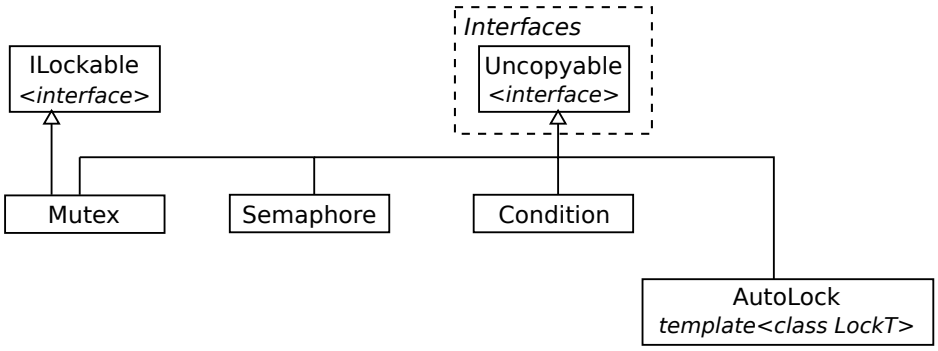

Figure 5.3: LUNA LockSync component UML diagram

A so called Resource Acquisition Is Initialisation (RAII) object manages the automatic locking and unlocking of Mutexes and Semaphores based on the scope. The example code given in Listing 5.2 is used to explain this technique.

```
Mutex m;
{
  AutoLock<Mutex> al(&m); // RAII object
  // Do things while holding the lock.
}
```

Listing 5.2: C++ RAII example code

The created *AutoLock* RAII object immediately requests the lock. The resulting behavior is that of a *Mutex* in this case. When the program has performed its actions and leaves the scope in which the *AutoLock* was created, the destructor of the RAII object will automatically unlock the *Mutex*. So, one does not have to deal with locking and unlocking of a *Mutex* anymore, which prevents possible deadlock situations due to forgotten locks.

To save computational resources the *AutoLock* class uses, so called traits, which are a programming technique in C++ to make information about types available during compilation. They are implemented using templates and template specializations. Together with overloading, traits classes make it possible to perform compile-time if-else tests on types. Basically, one has to specialize a structure, *lock_traits* in this case, which details the locking procedure for the specific class. The way traits are implemented enables the compiler to highly optimize the class (*AutoLock*) using the traits, because everything required is known at compile time. But, the traits parameter (*Mutex* in Listing 5.2) must be 'hard' coded and thereby the technique has a limited usability. The complete technique is well explained in Meyers (2005b).

5.2.3.2 Implementation

The implementation is completely done in POSIX.1 functions and should therefore be platform independent for most cases.

The hard real-time versions of the functions in RTAI and Xenomai were not considered, because these would require two different Mutex types: only hard and hard/soft real-time. This would hinder easy development with LUNA as an unusual concept is introduced which is not supported by other real-time OSs.

5.2.4 Communication

This component specifies interfaces for the communication functional requirement (see Section 3.2.1.6) and implements them for rendezvous communication between OS threads and processes.

5.2.4.1 Design

The UML class diagram for the design is shown in Figure 5.4 and is explained next. A communication channel always has an input and an output side. As, channel ends might be separated (networked or by a memory barrier), two interfaces have been developed, one for either side. The *IChannelIn* interface specifies input written to the channel and the *IChannelOut* interface specifies output read from the channel. The *IChannel* interface combines both interfaces and the *Uncopyable* interface into a full channel specification.

The *RendezvousChannel* specifies bidirectional inter thread rendezvous communication in the same OS process. The IPC in *IPCRendezvousChannel* stands for Inter Process Communication (IPC) and this interfaces specifies what the name already suggests: inter-process rendezvous communication and is unidirectional. The inter-host communication has not been defined yet, as this is considered out of scope for this project. Nonetheless, the *IChannelIn* and *IChannelOut* interfaces are considered generic enough to specify this case as well.

A *SharedRawMemory* class was designed to enable the exchange of data in shared memory, as is used for the 'blackboard' communication in RoboFrame (Section 4.4). This class supports bidirectional communication by definition.

Note, the design does not contain the 'dreaded diamond' structure sometimes found in communication channel designs (Section 4.5.1).

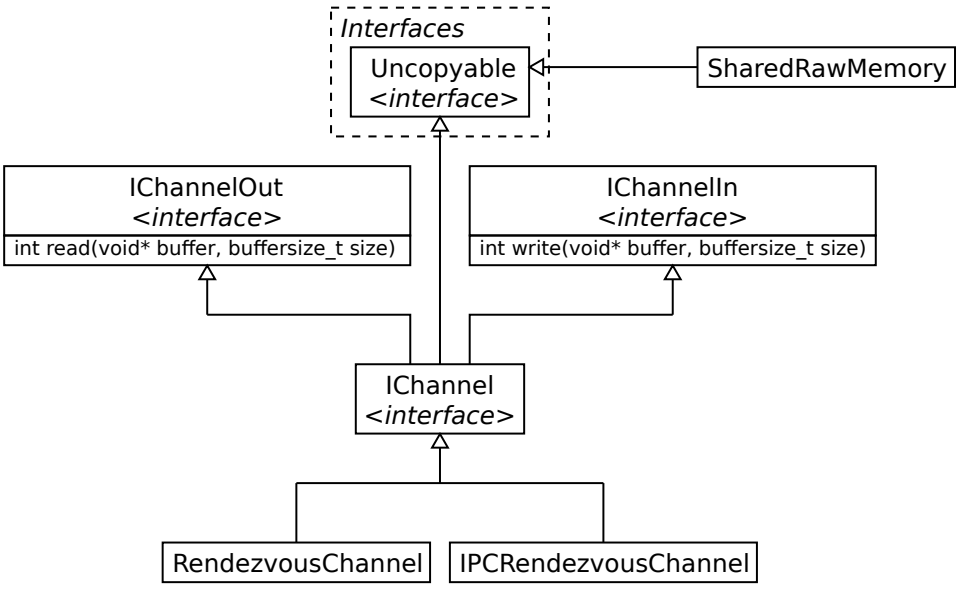

Figure 5.4: LUNA Communication component UML diagram

5.2.4.2 Implementation

The QNX OS provides any-to-any rendezvous communication between threads and processes, which also adheres the thread priority levels and, if pre-allocated, is real-time. The implementations for the *RendezvousChannel* and *IPCRendezvousChannel* were therefore pretty straight forward. The reports of Molanus (2008) and Veldhuijzen (2009) mention inter-host communication through Qnet, which extends rendezvous communication over an ethernet link, and this seems a viable choice for implementing this in the future.

The other OSs could provide rendezvous communication through POSIX message queues, although during timing tests these were quite slow compared to the rendezvous communication on QNX. So, it is believed that the interfaces can be implemented fairly easy for the other OSs.

The *SharedRawMemory* class is implemented with POSIX.1 functionality and therefore considerably platform independent.

5.2.5 Utility

The Utility component on itself does not implement any requirements, but provides the generic data containers and design patterns listed in Table 5.5.

Name	**Purpose**
Set	Generic set
Queue	First In First Out (FIFO) queue
TreeNode	Generic tree structure
LinkedList	Daisy chain of *ListItem*s
(Plain)LockFreeQueue	Lock-free FIFO queue
Singleton	Design pattern for unique globally accessible objects
GenericFactory	Abstract factory design pattern

Table 5.5: Utilities generic data containers and design patterns

5.2.5.1 Design

The UML class diagram for the Utility classes is shown in Figure 5.5, but it only shows per class a few of the available manipulation methods. For all data containers, applies that the container capacity only changes on explicit request by the application programmer, thereby possibly unwanted dynamic memory allocations are prevented. The API documentation furthermore specifies the (real-time) properties of classes and functions not discussed below. The template parameter *T* in the diagram indicates that the data container can hold any type of data, which can be pointers, values or objects.

An overview of the data container characteristics is shown in Table 5.6. Thread safe means that multiple threads can concurrently manipulate the data container without corrupting its data. For generic add, remove and search operations the performance is listed in big-O notation (Baase and Gelder, van, 2000). For some the Worst Case Execution Time (WCET) or Average Case Execution Time (ACET) is specified. If the execution time is not indicated, it exerts the indicated performance constantly. A question mark denotes that the execution time is unknown. The external data container column specifies if the structures, to hold the data, are inside the main class or not. The advantage of external data containers is that items can be transferred (real-time) between data containers of the same type.

An alternative for all data containers would have been to use the STL data containers, but these are not designed for real-time usage (see Section 5.1.5).

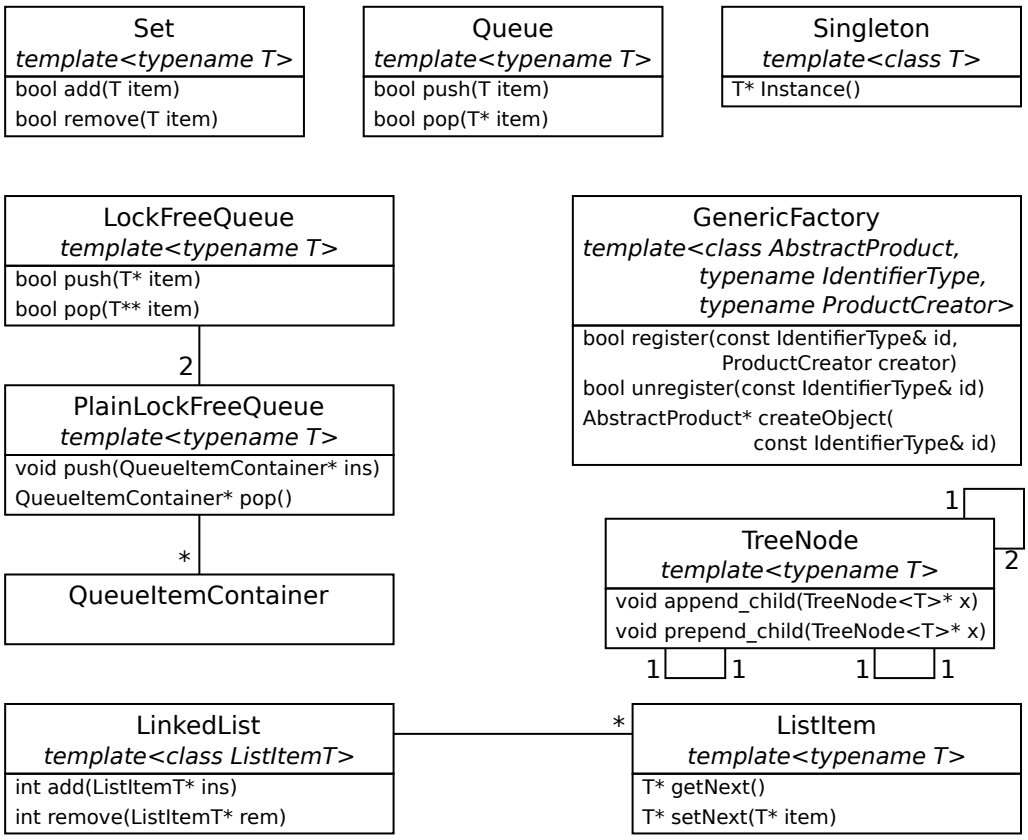

Figure 5.5: LUNA Utility component UML diagram

Data container	**Thread safe**	**Add**	**Remove**	**Search**	**External data containers**	**Summary**
Set	N	O(1)	O(n) WCET	O(n) WCET	N	Unordered with duplicates
Queue	N	O(1)	O(1)	-	N	FIFO
TreeNode	N	O(1)	O(n) WCET	O(n) WCET	Y	Unordered
LinkedList	N	O(n) WCET	O(n) WCET	O(n) WCET	Y	Ordered
PlainLockFreeQueue	Y	O(1) ACET O(?) WCET	O(1) ACET O(?) WCET	-	Y	FIFO, without locks
LockFreeQueue	Y	O(1) ACET O(?) WCET	O(1) ACET O(?) WCET	-	N	FIFO, without locks

Table 5.6: Data container characteristics

Set
This data container models an unordered *Set* which may contain duplicate items. A application is able to prevent duplicate items by using the *contains()* function before adding an item. Adding items is done in constant time (i.e. O(1)), but removing items may take linear time (i.e. O(n)) depending on the number of items contained in the set. Items in the set can be accessed directly for easy manipulation, but are not guaranteed to remain in the same internal place.

Queue
The *Queue* class is a First In First Out (FIFO) design. Adding and removing items is done in constant time.

TreeNode
Multiple *TreeNode*s can be put together to form a tree, see the example in Figure 5.6. The virtual root of the tree has 3 children of which the second child has 2 children. Any *TreeNode* can function as a virtual root, but there is always only one true root node in a tree. All *TreeNode*s also have a link with their parent, except for a true root node which does not have a parent. To traverse the tree horizontally a sibling iterator can be used. For repeated vertical traversal a depth first iterator is supplied. The design and implementation are taken from Peeters (2009). It was chosen to prune and modify the code, because the original code used C++ language Exceptions (Appendix B) and has a lot of undocumented extras.

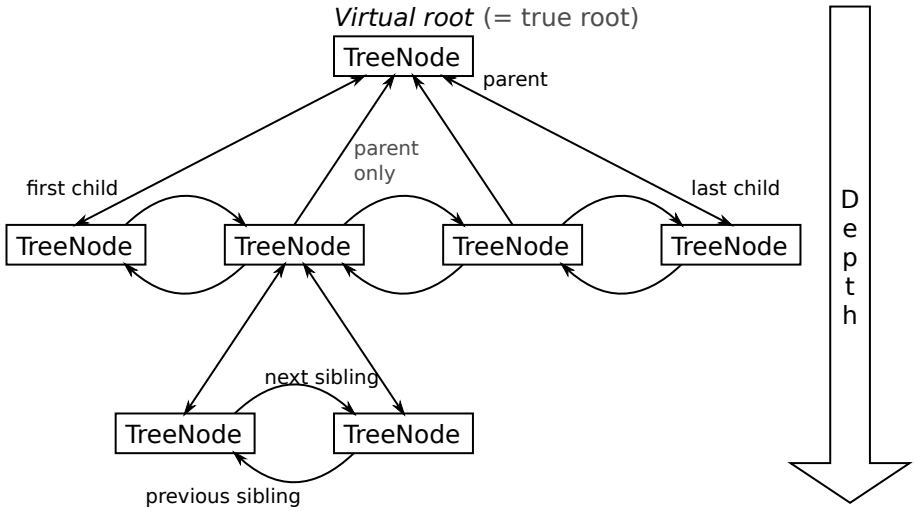

Figure 5.6: LUNA Tree design

Additionally, a non-member function has been added to collect all *TreeNode* instances from a virtual root into a *Set*. The classes requiring a tree structure can for instance extend the *TreeNode*, see the *CSPConstruct* in Section 5.4.

LinkedList
The *LinkedList* is an ordered unidirectional daisy-chain of *ListItem*s. Classes requiring such a structure should extend the *ListItem* and implement a compare function to achieve the appropriate ordering.

PlainLockFreeQueue
The *PlainLockFreeQueue* provides a lock-free FIFO queue. As the name suggests, the lock-free queue is distinguished from the previously discussed *Queue* by the fact that it is accessed without locking. Multiple papers have been published on this subject. Michael and Scott (1996) designed the first practical queue based on a singly linked list. Tsigas and Zhang (2001) implemented its queue using a finite array. They further state that the linked list implementation suffers from a memory management problem in the sense that nodes cannot be freed because

someone might still be using them. This not true, because the enqueue/dequeue semantics together with the dummy head will prevent this problem. Furthermore, an array-based implementation is impractical because memory array sizes cannot be adjusted dynamically. Shann et al. (2000) also used an array based queue, but to lower the memory requirements. Unfortunately, they thereby made their algorithm less robust, in the sense that the algorithm must be configured for the number of parallel readers/writers beforehand to guarantee correctness.

Michael and Scott (1996) also performed extensive performance tests, which showed that their implementation in general is faster than lock-based algorithms. Performance tests in (Shann et al., 2000) indicate that it is even faster compared to the latter. As mentioned before, this comes at a price, since it made the algorithm less robust. So, the current *PlainLockFreeQueue* design is based on Michael and Scott (1996) algorithm.

The discussion on lock-free shared data objects that will follow is generally applicable for any lock-free algorithm, but some points might be made targeted specifically at the current design.

The advantages of lock-free shared data objects are:

- Dead-locks cannot occur, because no locks are used.
- No priority inversion, because threads do not need to wait on locks.
- Less overhead than lock-based algorithms in a low concurrent environment, because less bookkeeping is needed. No list of blocked threads must be maintained nor any priority inversion has to be identified and solved.
- Potentially a higher degree of concurrency is possible, without resorting to methods like multiple locking levels within the data container to enable partial concurrent access.
- Often faster than lock-based algorithms in highly concurrent environments, which has been shown in the performance tests of Michael and Scott (1996).
- Can be employed in any design, even without knowledge of the data access sequences.

Unfortunately, employing lock-free algorithms also has its downsides for real-time applications. Its biggest advantage, using no locks, becomes its weakness. For this to become clear, the general operation of lock-free algorithms has be explained first.

Due to not using any mutual exclusion mechanisms (which would result in some kind of lock afterall) all threads can always concurrently access the internal data structures of the shared object. Furthermore, all possible permutations of the algorithm execution are possible, since no thread is being stopped at any point. So, there is no possibility to discover how far any thread has progressed and changed which data. For an algorithm to prevent the internal data structure from becoming corrupt, it must somehow incapacitate any other concurrent updates to the same data which have not been committed yet and might now be based on old data. These threads should start over and try again to commit their data before an other thread has changed something again, until they succeed. Or they must somehow be able to fix the apparent problem. Thus, it can be concluded that any lock-free algorithm at some point is non-deterministic.

In Section 5.1.5, it was stated that hard real-time threads must have code that results in deterministic timing. In Anderson et al. (1997) the non-determinism for periodic, hard real-time tasks that share lock-free objects on a uniprocessor was investigated. The authors concluded for this particular case the lock-free objects are likely to perform better than lock-based schemes, if the worst-case cost of a lock-free retry loop is at most half the worst-case cost of a lock-based access. Of course, this conclusion cannot directly be applied to multi core/CPU systems as well.

This paragraph will try to make it plausible that similar results are to be expected for multi-core/CPU applications which spend less than a certain percentage of their time on manipulating the shared data object. The lock-free algorithms in this setting will of course still have considerably less overhead, so redoing parts of the algorithm will not immediately result in low performance nor measurable non-determinism. Assuming that the percentage is low enough,

the concurrent access to the shared object is also low. So, the chance that parts of the algorithm will need multiple (re)tries is also very low. Besides, in the current implementation, for all contending threads one thread always succeeds and finish its update, so progress is guaranteed. Therefore, in the described setting, it is expected that the non-determinism will be unmeasurable and thereby allowing its use in real-time threads.

In contrast to lock-free algorithm, lock-based algorithms are only considered deterministic if it more or less can be predicted when possible blocking situations occur. In some complex situations a lock-free implementation might be indispensable, since it can be employed without any knowledge of the access patterns. For example, the scheduler in the UThreading component (Section 5.2.11) is manipulated from different threads with different settings. A user thread may, for example, unblock another user thread and at the same time the *PeriodicTimer* OS thread (Section 5.2.6) might unblock a user thread in the same scheduler. Since the scheduler updates cross thread boundaries, thread types, and possibly different real-time constraints, it is better to use a lock-free algorithm to guarantee progress for all threads and achieve the real-time constraints.

The *PlainLockFreeQueue* is based on a linked list and purposely does not manage its list items (called *QueueItemContainer*) other than when they are enlisted in the queue. Resultingly, the *QueueItemContainer* can be transferred freely between multiple *PlainLockFreeQueue*s. This turned out to be useful for user thread priority scheduling with a queue for each priority level; thread priorities can be changed in real-time.

Unfortunately, we discovered a problem with the Michael and Scott algorithm. The proposed algorithm contains a fault which, during very heavy concurrency, will result in loss of the queue internal structure. At the moment it is still being investigated how to resolve this error. However, since the concurrency is quite low in the evaluation setup (Chapter 6), the *PlainLockFreeQueue* is already used successfully.

LockFreeQueue
The *LockFreeQueue* manages its own *QueueItemContainer*s by using two *PlainLockFreeQueue*s, one to hold the empty containers and one for the actual queue.

GenericFactory
The factory design pattern in general handles polymorphic creation of objects. In some cases the creation of a specific object should be based on dynamic information and/or the specific object type should not be programmed statically into the code. For example, the type of mutual exclusion lock required for an object might depend on which kind of thread the object is executed on. This information might be present at compile time, but for flexibility reasons one might not want to specify this deep down in the objects code. So, an object factory can be used to create the proper lock object.

The *GenericFactory* design pattern implemented in LUNA is based on Alexandrescu (2001a). First, all factories need to be registered and then objects can be created with it at run-time by specifying an identifier. The factory should only be used during initialization, since a memory allocation is required to create the object.

Singleton
A singleton is a globally unique variable/object. Thus, a Singleton should be used when modelling objects that conceptually have a unique instance in the application, such as a system clock. Although the Singleton concept is simple, its implementation issues are complicated as described in Alexandrescu (2001b). The current implementation is also based on this text.

5.2.5.2 Implementation

Except for the *PlainLockFreeQueue*, all classes are build on top of the common C++ language. So, platform dependence is not an issue for these. The *PlainLockFreeQueue* uses a single and double size Compare And Swap (CAS) instruction available on most x86 systems. Other platforms might have different instructions for this particular functionality, but these can most likely be used to form a CAS equivalent instruction.

All classes were designed with real-time usage in mind. Nonetheless, creating or resizing any of the data containers requires memory allocation and should therefore be done at a suitable time (for example at start up).

5.2.6 Timer

This component specifies interfaces for a one-shot and periodic timer. The latter enables the periodicity functional requirement (Section 3.2.1.5). Furthermore, a timer manager has been added to efficiently manage the periodic timers and save OS resources. The timer implementations needs to interface with the OS to establish their functionality.

5.2.6.1 Design

The requirements state that an application might request a timer frequency multiple times. For instance, for the Production Cell six independent OS threads might be used all with the same periodicity. The solution could be to also request six timers with the same frequency from the OS. Or request one timer from the OS and handle the subsequent timer activations inside the framework. The latter is the solution to the following potential problems:

- The OS and hardware have a limited number of timers available.
- The OS will try to activate the thread when the timers fire, possibly resulting in multiple 'useless' interrupts shortly after each other.
- On some OSs the absolute time, when a periodic timer is requested, determines when the timer will fire in the future. This might create non-determinism for multiple timers in an application. Because application startup is commonly allowed to have non-deterministic operations and thereby the absolute timer request times changes every application invocation.

Both use cases are available within LUNA, but the latter is in most cases a smarter choice.

The UML class diagram of the Timer component is shown in Figure 5.7 and is explained next. The *TimerManager* can be used to manage multiple timers with one OS timer. But a *PeriodicTimer* may as well be created without the *TimerManager* in order to receive multiple OS timers. Unfortunately the inherited *PlainOSThread* then is overhead. The *TimerManager* is a *Singleton* to make it everywhere available with minimum ease.

To manage multiple timers, the *PeriodicTimer* requires its own thread to wait on the OS timer event. Furthermore, this thread requires the highest priority (the framework's internal *CRITICAL*, see Section 5.2.8), so that it can immediately resume when a timer event is received and service its list of *AbstractTimerHandler*s. The *timerCallback()* function of the *AbstractTimerHandler*s are called, which in turn either execute a callback function immediately (running in the *PeriodicTimer*'s thread) for the *CallbackTimerHandler* or unblock a periodic thread for the *PeriodicThreadTimerHandler*. So, the *CallbackTimerHandler* code must be kept short or an overrun of all timers will occur.

The *PeriodicTimer* services the *AbstractTimerHandler*s based on a prioritized list (*LinkedList*), such that a deterministic order of activations is created. *AbstractTimerHandler*s can be added and removed in hard real-time as long as they are pre-created.

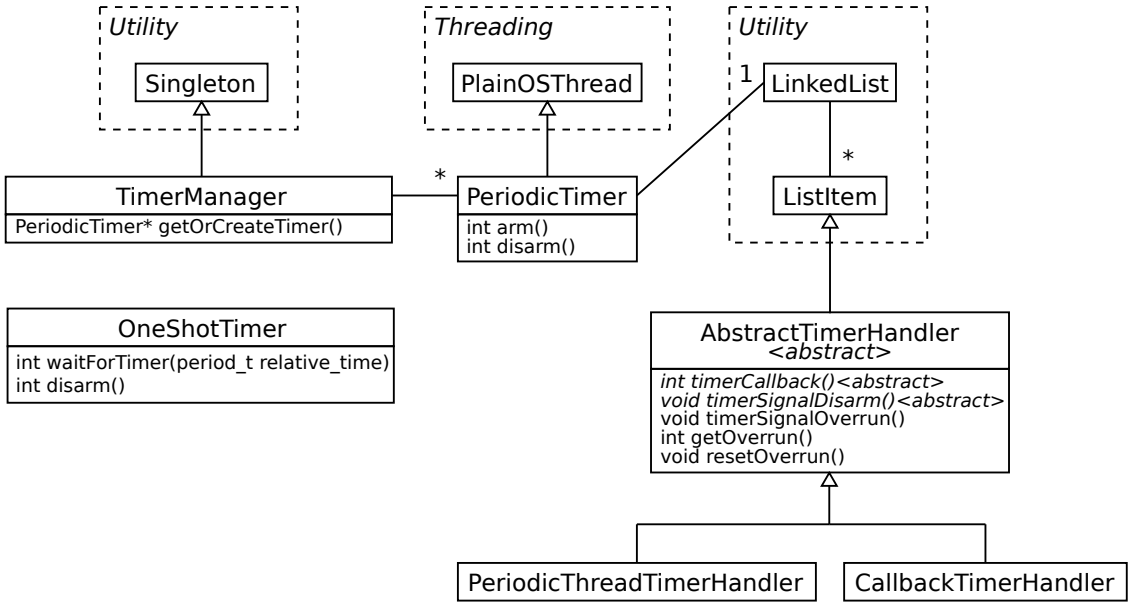

Figure 5.7: LUNA Timer component UML diagram

Both, the *PeriodicTimer* and *AbstractTimerHandler*s record if an overrun occurred. This functionality might be useful to determine if real-time code is obeying its timing specification and to take action otherwise. The first is notified by the OS that an overrun occurred. The *AbstractTimerHandler* notices an overrun when the handler reaches its blocking function (e.g. the *waitForNextPeriod()* in *PeriodicThreadTimerHandler*) after the *PeriodicTimer* has fired multiple times. The *AbstractTimerHandler* can be extend to change the latter behavior to notify the handler of this fact earlier.

The *OneShotTimer* provides a one time use only OS timer. Using a *LinkedList* with *AbstractTimerHandler*s is (probably) overkill for this timer in most cases.

5.2.6.2 Implementation

Appendix D gives a comparison between timers and counters commonly available on PC hardware architectures. The POSIX.1 real-time standard specifies interfaces for clock and timer functionality for the OS software. By inspecting the target and development OSs it was determined that most OSs use this standard for interfacing with at least one of the compared timers and counters. Furthermore, the POSIX standard also specifies how one can retrieve the OS timer and clock accuracy. So, for now it is probably best to use the POSIX.1 interfaces as these are relatively platform independent and if in the future higher resolution timers are required these can still be implemented and overloaded by the build system.

For timers it is worth noting that two possible problems might occur:

1. The OS might use a periodic timer itself to update its internal time and bases any timer requests on this. So, the internal time is not continuous as in the real world and may lag behind. When requesting a timer, one-shot in particular, the OS takes the current internal time plus one update period, adds the requested delay and sets any hardware timer to this value. The extra period must be added, because it is not accepted that a timer fires too early. Therefore it is not possible to get a timer with a higher accuracy than the OS internal clock update rate, although some other OSs might update the internal counter first. For periodic timers this inaccuracy only applies to the first timer event, thereafter the timer fires periodically at the specified constant rate. The whole process is visualized in Figure 5.8.

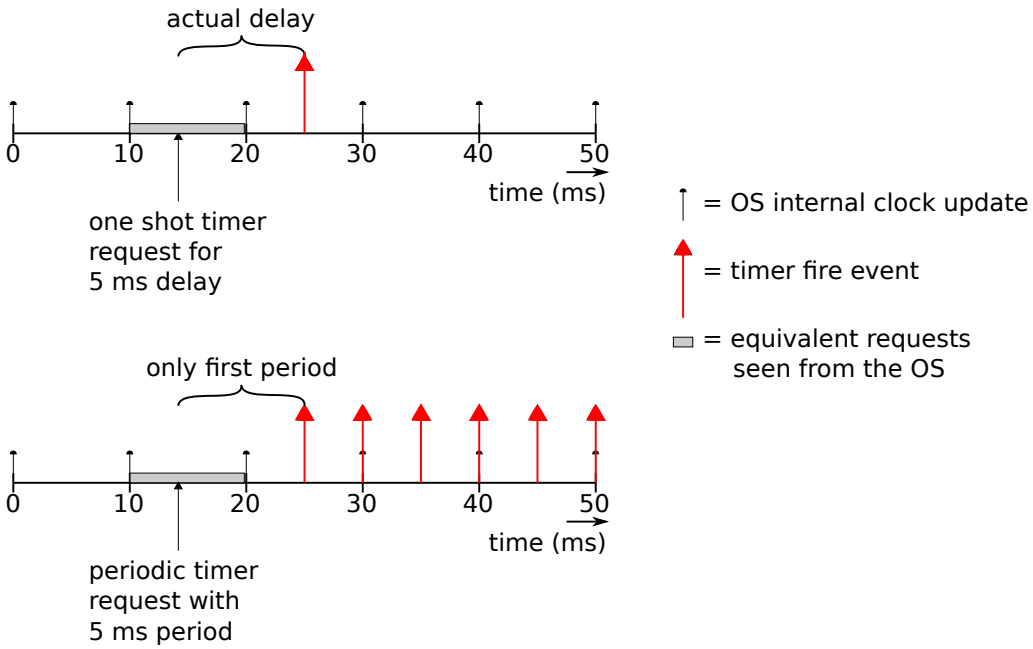

Figure 5.8: OS timer accuracy problem

By the way, QNX and most other OSs uses the same periodic timer for updating the internal clock and periodically rescheduling their processes and threads. Using the default QNX OS settings a one-shot timer with an accuracy between 1 and 2 ms can be achieved (see Section 5.2.8).

2. A timer quantization error might occur. The counters and timers in Appendix D all have a high-speed (MHz range) clock, produced by the circuitry in the PC. This high-speed clock must then be divided by a hardware counter to reduce the clock rate to the kHz or hundreds of Hz range (which the OS can handle). Resultingly, not all requested frequencies and delays can be matched well. Except from effectively getting a different frequency another phenomenon occurs due to this. Every so often the timer might not fire, because its counters appears not to have been incremented enough. This phenomenon is explained in detail in Charest and Stecher (2011) and its existence is shown in the results section of the CSP paper (5.4.1). The effect can be minimalized by matching the requested frequency with the hardware clock divisor, which is also shown in the mentioned section.

The QNX implementation for the *PeriodicTimer* and *OneShotTimer* uses POSIX.1 functionality to create the requested OS timer, but thereafter the implementation is QNX specific. This functionality is based on the general QNX philosophy of a nano kernel with multiple little servers. When the requested timer fires an event (pulse) it is sent over the supplied rendezvous channel, which the timer thread (client) can catch at any time it wishes by performing the read from the channel. The *PeriodicTimer* thread then handles the rest as explained. So, for other OSs the implementation still needs to be created.

5.2.7 SystemInfo

The SystemInfo component provides a system-specific information database. There is no requirement directly linked to this component, but during the design of the framework it was realized that system-specific information is required by multiple components.

The SystemInfo component currently retrieves the following information from the system:

- The number of CPUs.
- The number of cores per CPU.
- The frequency of each CPU in MHz.
- The number of ticks per second.
- The number of ticks for the measurement correction per time measurement.

For the last two items; computers measure time in discrete intervals (denoted as ticks), as is explained in the Timing component (Section 5.3.5). Since measuring the time also costs time, it is needed to correct for this.

Additionally, the component provides a platform abstraction to retrieve the current number of elapsed ticks. This is currently being used by the Timing and RTLogger components (respectively Sections 5.3.5 and 5.2.12).

5.2.7.1 Design

The UML class diagram of the SystemInfo component is shown in Figure 5.9. The *SystemInfo* class is a *Singleton* (see Section 5.2.5). Furthermore, in the current *Singleton* design the class instance will be created only once, which is a nice feature because accessing the OS and processing its system information is generally non-deterministic. So, the gathered system information is cached in the *SystemInfo Singleton* and can be safely accessed from any real-time thread.

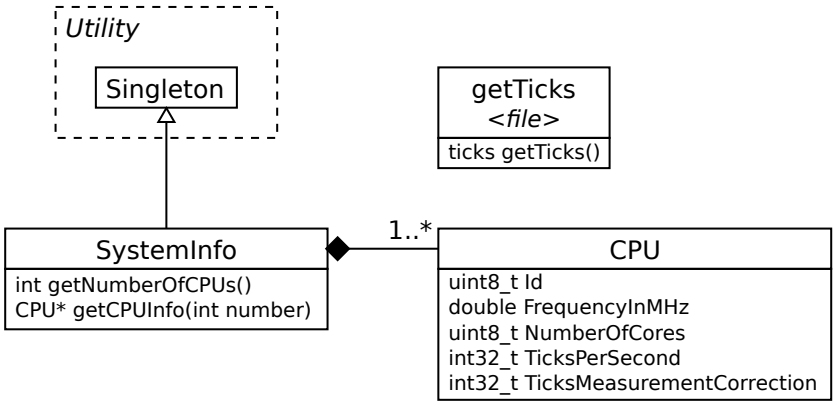

Figure 5.9: LUNA SystemInfo component UML diagram

5.2.7.2 Implementation

The retrieval and processing of system information from the OS is highly platform dependent. For example, Linux provides the */proc/cpu* file to store CPU information and QNX uses a global system variable (*SYSPAGE_ENTRY*) to store the same information.

The *getTicks()* function is dependent on the available counters and possibly on the OS drivers as well. Appendix D lists timers and counters commonly available on PC hardware architectures. The Time Stamp Counter (TSC) has been chosen for the current implementation, because it is a high-resolution counter and available on most PC hardware architectures, and it only incurs low-overhead for accessing the ticks counter.

5.2.8 Scheduler

This component specifies a generic scheduler interface for threads and thereby extends the thread support requirement (Section 3.2.1.3). The OS scheduler is responsible for ordering the ready OS threads into a real-time feasible run order. Unfortunately the OS scheduler, in all investigated OSs, cannot be manipulated directly other than changing the scheduling policy.

The component provides a generic scheduler interface and implements this for OS threads. Furthermore, it provides platform indepedent priority levels for both real-time and non real-time threads.

5.2.8.1 Design

The UML diagram for the Scheduler component is shown in Figure 5.10. The *IScheduler* is the basic interface for all schedulers. The shown *OSScheduler*'s functions are the only possible manipulation methods for the OS scheduler.

The *OSSchedulerPriorities* header file provides a platform independent priority abstraction. Both hard and non real-time threads have ten freely usable priority levels, which are more than enough for the corner applications. There is also another hard real-time priority level, denoted *CRITICAL*, which is for the framework internal use. A real-time OS will always preempt any running threads with a lower priority in favor of the higher priority thread becoming ready.

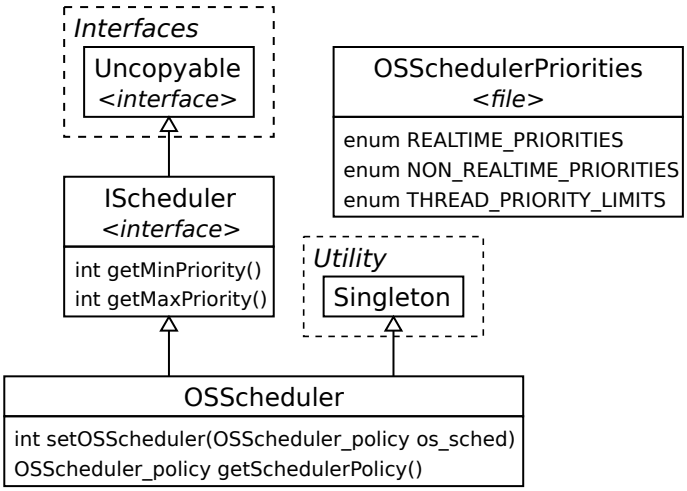

Figure 5.10: LUNA Scheduler component design

5.2.8.2 Implementation

The *OSScheduler* uses POSIX.1 functions and should therefore be platform independent for POSIX compliant OSs. The QNX OS supports round-robin, FIFO and sporadic scheduling policies.

The scheduler priorities unfortunately have to be determined manually for each OS. It is worth noting that on QNX the highest available priority, for non root processes, is 63. Root processes can have a maximum priority of 251. Most system processes and applications started with default settings run with priority 10 and are scheduled according the RR scheduler policy.

The default scheduling period for the scheduling algorithms is 1 ms for CPUs with $\geq$ 40 MHz and 10 ms for slower systems. This frequency can easily be changed with the QNX *ClockPeriod()* function. Experimenting with this function has shown that for a 600 MHz CPU a frequency of 1 kHz can be achieved, but the system then spends roughly 11% of its computing power on scheduling alone.

5.2.9 Emergency-manager

The safety layer requirement (Section 3.2.1.11) states that hazardous situations for the setup and environment should be prevented. A true one size fits all safety layer is probably not possible due to the diversity of setups which might be implemented with LUNA. So, multiple components and link-drivers can implement their own safety functions. To coordinate emergency signaling among these components and provide common safety practices and implementations the Emergency-manager component has been designed. This component partially implements the safety layer requirement while preventing dependencies.

Additionally, the Emergency-manager provides a way to handle exceptional circumstances. Note that it was already determined that C++ language exceptions cannot be used for this (see Appendix B). Exceptional circumstances are for instance a memory allocation or device driver failure, which can be non recoverable and therefore the application should shutdown. When such a situation occurs, the Emergency-manager should be used.

5.2.9.1 Design

The *EmergencyManager* is designed to coordinate an emergency shutdown among multiple components and the application. Classes requiring a safe shutdown should register a callback function, which will be called on an emergency shutdown in the order they were registered. These callback functions should contain functionality to prevent or resolve a harmful situation by for example setting the actuators to a neutral output value.

The design of this component is shown in Figure 5.11. A *Singleton* is required to make the *EmergencyManager* globally available and unique. A *LockFreeQueue* is used to manage a FIFO list of emergency shutdown functions. A lock-free implementation is used, because multiple threads can simultaneously identify and issue an emergency. By using this implementation the threads can work together safely and the *EmergencyManager* never suffers from a dead-lock.

The thread calling the *emergencyShutdown()* receives the highest priority (i.e. *CRITICAL*) to perform the shutdown immediately and without being interrupted.

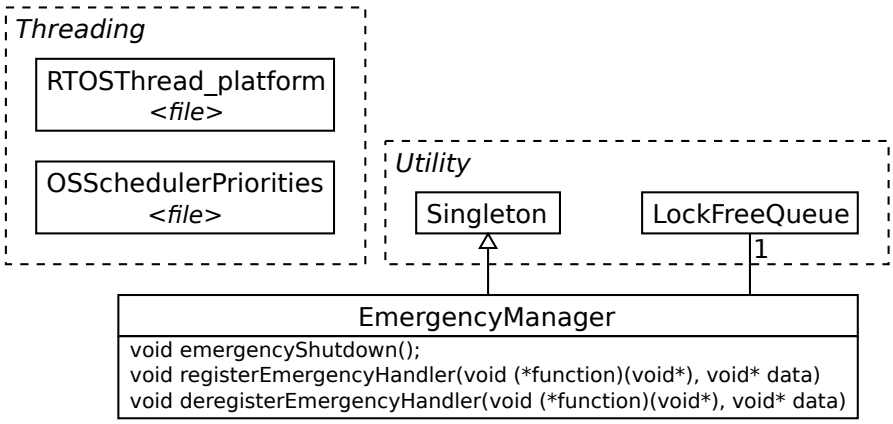

Figure 5.11: LUNA Emergency-manager component design

The current implementation does not identify emergencies itself yet. For instance, when the user presses *ctrl+c* most operating systems kill the application automatically. If the application designer did not anticipate this the last outputted actuator value might keep a motor running even though there is no control-loop running anymore. If the programmer did not handle such a case himself, the *EmergencyManager* should call the emergency shutdown functions.

Furthermore, the Emergency-manager should provide some good practices and their implementations. For instance, a simple algorithm to identify blocked actuators could be supplied.

Based on the combination of actuator output and collocated sensor input values it could be determined if an actuator is blocked. Furthermore, these functions could also be placed in a CSP library.

5.2.9.2 Implementation

The component is fully build on top of the Threading and Utility components and is therefore platform independent. The component is classified as a core component, because it provides basic and indispensable safety features for robotic applications.

5.2.10 Threading

The Threading component specifies the interfaces for the thread support requirement (Section 3.2.1.3). Since, priorities are a thread property, the priorities requirement 3.2.1.4 is also satisfied in this component. Additionally, it was recognized that threads might need periodic behavior (Section 3.2.1.5). The component implements these requirements platform independently for OS threads.

First, the requirements are worked out in detail to clarify the problems and features that this component is required to solve and implement respectively.

- *Periodicity* - Threads may be periodic.
- *Real-time and non real-time threads* - Provide a solution to specify both thread types with the same API. Some OSs provide different system calls for each type.
- *Generic thread interface* - Generic thread functionality should be identified and a unified interface should be designed for this. Applications using this unified interface can then manipulate the own running thread indifferent of its backing implementation.
- *Generic runnable object interface* - A generic interface for runnable objects, i.e. objects that can be executed by a thread, should be designed.
- *Thread configuration* - Threads should be configurable at creation for: priority, period, processor mask and stack size. The processor mask specifies the cores/processors a thread may be executed on. The available stack size should be configurable per thread.
- *Temporarily stop threads* - Provide a solution to temporarily stop a thread, thread type independently. For example, rendezvous communication is required to stall a thread while waiting for the other side to become ready.
- *Priorities* - Threads should adhere the set priorities.

5.2.10.1 Design

The UML class diagram for the Threading component design is shown in Figure 5.15.

Periodicity
As stated in the requirements, threads may require periodic execution. But, this functionality is not specified in the POSIX.1 standard as a thread property. Furthermore, none of the OSs supports the creation of a periodic thread. Still, it would be beneficial to have such functionality standard available in a class. The Timer component (Section 5.2.6) has a timer interface with the OS which is usable to implement this. Since the *PeriodicTimer* requires a thread, a plain thread should be supported as well (to prevent a circular dependency). The plain thread can then also be used by classes that do not have a use case for periodic behavior nor other advanced features, such as the pooled threads in the ThreadPool component (Section 5.3.4).

Real-time and non real-time threads
The method of choosing the thread type, real-time or non real-time should be straightforward. One OSs might also use different implementations for both (as is the case for the combination of RTAI and Linux) it was chosen to separate the thread types in separate classes. The investigated frameworks and libraries in Chapter 4 do not make a difference between the thread types in terms of specification and mostly specify one thread type for the whole application.

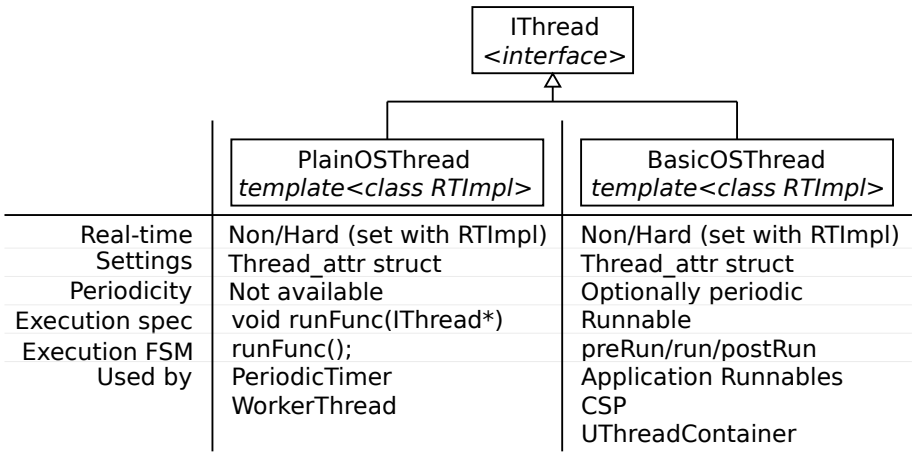

Figure 5.12: LUNA Threading approach and overview

Combining periodicity and real-time and non real-time threads
When the proposals from the last two paragraphs are combined naively, four classes would need to be designed and maintained. This is considered labour intensive and error prone, whilst function implementations are generally the same. Therefore it was chosen to create a plain and (a)periodic thread class. Both take their (non) real-time implementation as a template parameter (see Figure 5.13), which is a policy based design and similar to Aspect Oriented Programming (ASP) techniques. The periodicity for a periodic thread is supplied as a parameter at creation. The overhead by combining the periodic and non-periodic thread functionality is negligible, for non-periodic threads. The complete idea and overview of the class properties is shown in Figure 5.12.

Generic thread interface
Functionality commonly found in all thread types and on all OSs has been grouped in the *IThread* interface. All (OS) thread implementations must at least fulfill this interface.

Thread configuration
The *PlainOSThread* and *BasicOSThread* both can be configured at creation with the *Thread_attr* structure, which takes priority, period, processor mask and stack size settings. But, the *PlainOSThread* ignores the period setting.

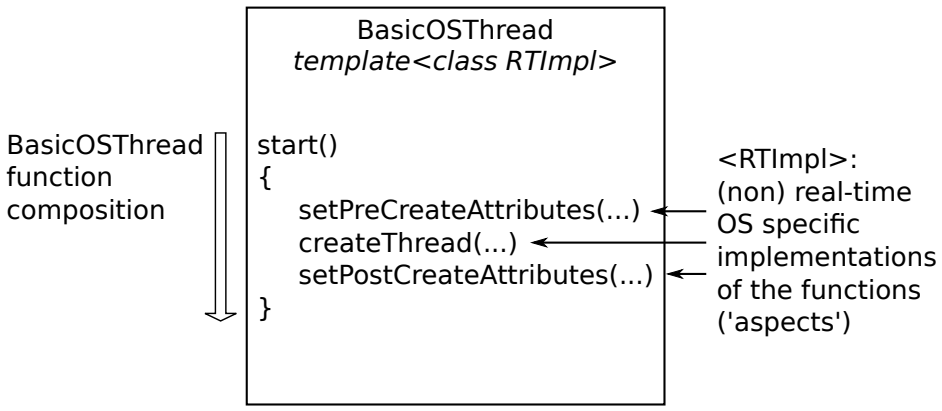

Figure 5.13: LUNA BasicOSThread policy based (RTImpl) design

Generic runnable object interface
A generic *Runnable* interface has been designed for executable objects. The interface specifies three functions (*preRun()*, *run()* and *postRun()*) which can be overloaded by virtual inheritance and are executed by the *BasicOSThread*, as explained in the next section. Only the *run()* function is mandatory.

The *getIThread()* function can be used to access the *IThread* interface for the *Runnable*'s OS thread. An alternative would have been to use Thread Local Storage (TLS), to retrieve the thread local *IThread* interface. It is expected that deeply embedded targets do not support this function. Since this is not an easy function to emulate it was chosen to use the *Runnable* interface to hold the association with its *IThread*.

Combining the generic runnable object interface and the BasicOSThread
The combination of the *Runnable* and *BasicOSThread* class also supports an advanced construction and destruction procedure which is best explained by the Finite State Machine (FSM) in Figure 5.14. This functionality is also explained in the LUNA paper (Section 5.4.1), but more specifically for CSP. The *BasicOSThread* thread is started by calling its *start()* function. This will setup an OS thread with the given *Thread_attr* settings. While the new thread is being setup the starting thread is blocked, namely until the *preRun()* function has finished. Then the *run()* function is executed. After the *run()* has finished and it is a periodic thread, it will wait for the next periodic activation after which the *run()* is executed again. If it is an aperiodic thread or the thread has been requested to stop, the *postRun()* function is executed. The OS thread will seize to exist after the *postRun()* has finished, but the OS thread object remains until it is deleted. The *BasicOSThread* may thereafter be restarted by calling the *start()* function again.

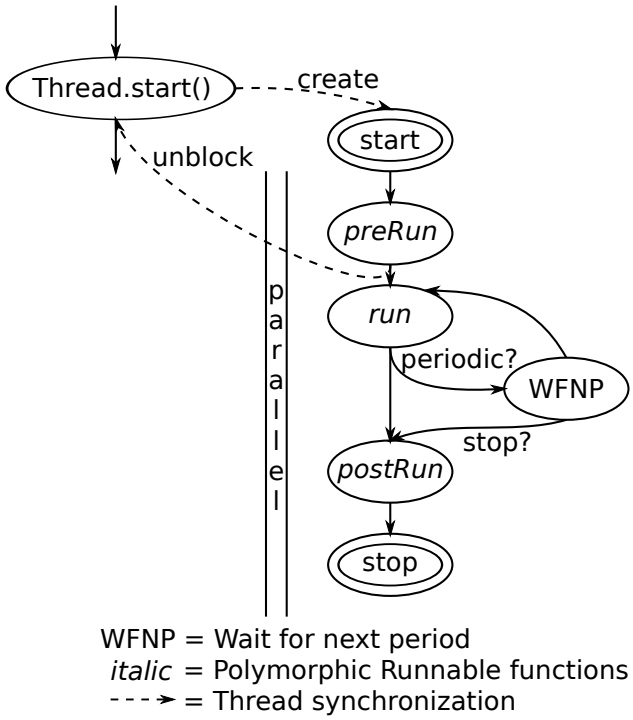

Figure 5.14: LUNA BasicOSThread FSM

The advantage of the *preRun()* function is that its function body is guaranteed to be fully executed before the starting thread can continue. Shared objects created in the *preRun()* function, which are necessary for the proper operation of the application, are guaranteed to have been fully instantiated, before the starting thread can 'run of' to perform other actions (and possibly use the shared object). Furthermore, this keeps the necessary housekeeping functions sepa-

rate from the *Runnable*'s *run()* implementation, which is nice from a design point of view. The *postRun()* function can be used the other way around in order to destruct the created objects.

For convenience reasons a hard real-time OS thread (*RTOSThread*) and a non real-time OS thread (*OSThread*) class have been added to the component. These classes already have the right template parameters for the *BasicOSThread* regarding the target OS and its real-time constraints. The *BasisOSThread* only uses a *PeriodicThreadTimerHandler* when the thread has been created with a period in the *Thread_attr* structure.

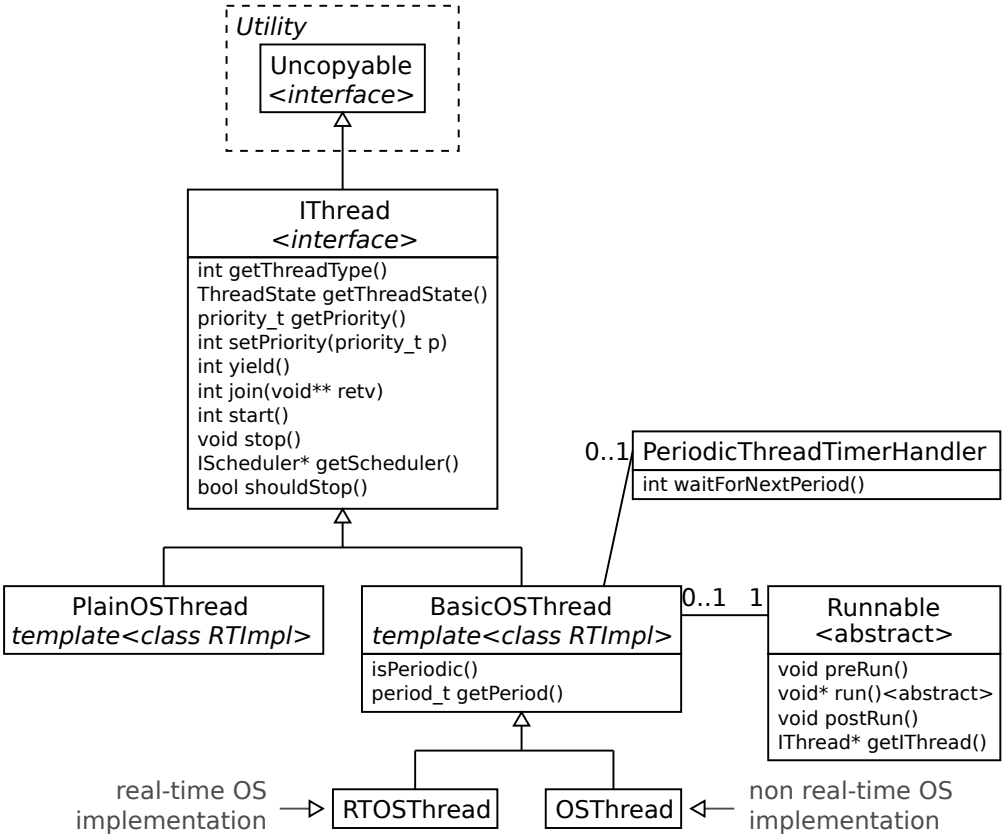

Figure 5.15: LUNA Threading UML class diagram

Temporarily stop threads

For threads and different thread types to collaborate it is necessary to have a generic interface which stops a thread temporarily and thread type independently. The *IThreadBlocker* is designed for this purpose. Separate implementations are required for each thread type, which might group a set of functions to efficiently achieve this.

The UML class diagram for the *IThreadBlocker* design is shown in Figure 5.16. An *ILockable* can be passed as an argument to the *waitOnEvent* or *signalEvent* function and will unlock it at the last possible moment, just before blocking the own thread or unblocking another thread respectively. This functionality is important if a critical section has been locked as a part of the algorithm. It then has to be freed at the last possible moment, because it may unblock a higher priority thread which might then preempt the own thread. See for an example the *blockContext* and *unblockContext* functions for the reader and writer CSP processes in the LUNA paper (section 5.4.1).

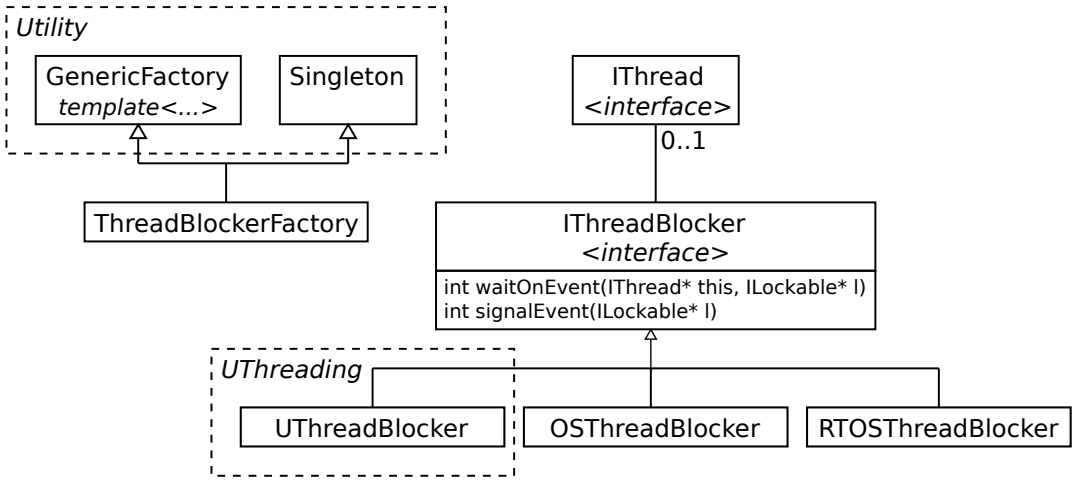

Figure 5.16: LUNA UML class diagram for the IThreadBlocker extension

Priorities
The OS threads on a real-time OS are scheduled according their priorities. This also means that a lower priority thread is immediately preempted when a higher priority thread becomes ready to run.

Factories
A *ThreadBlockFactory* and thread factory have been added for convenience reasons. The *ThreadBlockerFactory* can be used well in the discussed *Runnable::preRun()* function to create the appropriate thread blocker based on the thread type (see the *IThread::getThreadType()* function).

The thread factory does not use the *GenericFactory* design pattern, because the choice between thread types is not considered a run-time decision in a hard real-time context. Thus, the thread factory is established with function overloading, for instance the factory function *createThread(Set<Runnable*> input, Set<RTOSThread> output)* creates *RTOSThread*s for the *Runnable*s in the input set. A different thread factory can be selected on the basis of the input and output *Set* template parameters.

5.2.10.2 Use cases

Since the previously discussed functionality might be a bit abstract two examples are given in Appendix E. The first shows the specification of a periodic real-time OS thread and an aperiodic non real-time OS thread. Note, this is functionality not commonly found in other frameworks and libraries. The second example shows the implementation of a *Runnable* in combination with the creation of an *IThreadBlocker* by its *ThreadBlockerFactory*.

5.2.10.3 Implementation

The majority of the Threading component has been designed with POSIX.1 thread functionality in mind, because all target and development OSs have at least some form of implementation for this. But, optimized implementations are still possible for other OSs (e.g. RTAI). This will even be quite simple, since only the platform abstraction functions need to reimplemented (like *createThread()*). For more advanced implementations the *OSThread* and *RTOSThread* can be fully reimplemented, via the build system's file overloading feature. For the QNX OS, since it is fully POSIX.1 real-time conformant, the *PThreadImplementation* class (POSIX.1 thread implementation) is taken for both thread types as backing implementation.

The *PeriodicThreadTimerHandler* class provides the timer interface, which is not specified in the POSIX.1 standard. For this class the same reasoning can be applied; optimized implementations are possible. Note that the use case given in Appendix E only uses the generic interfaces and no platform dependent includes are necessary anywhere.

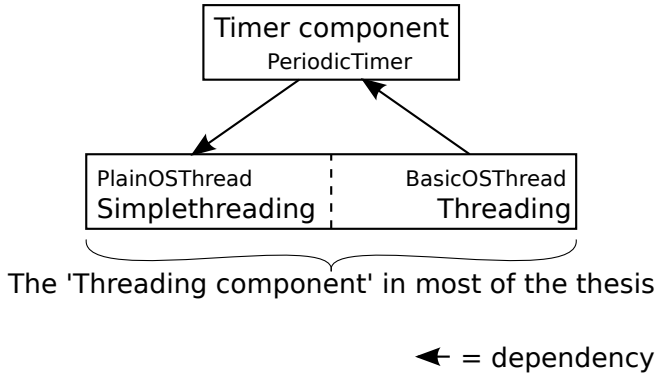

Figure 5.17: LUNA Threading and Timer dependency visualisation

Due to *BasicOSThread* being dependent on the Timer component and the *PeriodicTimer* being dependent on the *PlainOSThread* (see Figure 5.17), the Threading component in reality is split into two components: simplethreading and threading. The simplethreading contains the 'internal' *PlainOSThread* and the Threading components contains the (a)periodic *OSThread* and *RTOSThread* classes. So, application developers should use the Threading component.

5.2.11 UThreading

The UThreading component extends the thread support requirement (Section 3.2.1.3) and implements the *IThread* interface discussed in Section 5.2.10 for user threading support. The user threads also satisfy the priorities requirement (Section 3.2.1.4).

As explained in Chapter 2 multiple user threading models exist: one-to-one, many-to-one and many-to-many models. This component implements the IThread for user level threads (further abbreviated to: UThreads) in a many-to-one threading model, whereby multiple UThreads run in one OS thread.

UThreads are basically lightweight threads, without OS privileges. Modern OSs do not have any means to cooperate with UThreads and additionally have no clue about the UThreads running in parallel. So, the OS cannot preempt a UThread. This has the unwanted side effect that when a OS function call blocks in one of the UThreads the complete set of UThreads is blocked from executing. Furthermore, time sharing an OS thread is also not possible, because preemptions of UThreads can only happen on a voluntary basis, by for example calling the *yield()* function. Consequently, the code to be run on a UThread should be kept short in order to achieve the timing requirements. However, the non-preemptability can sometimes also be used to an advantages, as will be discussed in the design section.

The implementation section provides a small comparison between different UThread implementation methods and POSIX threads. A simple switching test, in which two threads during their turn yield, is used to determine its potential performance. It is shown that UThreads can potentially switch 7 times faster than OS threads on a uniprocessor Linux machine. But, overhead costs for UThread scheduling are not included in this measurement. Note, when threads with longer run-times are used, the ratio switching costs/thread running time becomes better and may even become negligible. So, the UThreads are favorable in case of light multi-threaded work.

The LUNA framework is certainly not the first to support UThreads, the CT library employed a user space threading model, the Kent C++CSP2 library provides a many-to-one threading model and the Mordor library (Mozy, 2011) provides a many-to-many threading model. The LUNA many-to-one threading model is therefore largely based on a combination of these implementations.

5.2.11.1 Design

The UML class diagram of the component shown is in Figure 5.18. To keep the diagram clear a few classes have been left out, which will be pointed out if necessary.

Runnable interface
The UThreading component implements a many-to-one threading model, whereby a set of *Runnable*s is executed on a single OS thread. The *UThreadContainer* must therefore implement the *Runnable* interface to able to be executed on any OS thread type.

IThread interface
The *UThread*s should be able to execute any *Runnable* object. Therefore, the UThreading component must re-implement the *IThread* interface. On the other hand, *UThread*s should be as lightweight as possible, otherwise they loose their competitive advantage. Since only one UThread can run simultaneously on one OS thread the *IThread* interface may as well be implemented by their set container (*UThreadContainer*) to lower the overhead.

The *Runnable::getIThread()* function returns a pointer to the *UThreadContainer* instance. Since only one UThread is running simultaneously the UThreadContainer always know which UThread called its function.

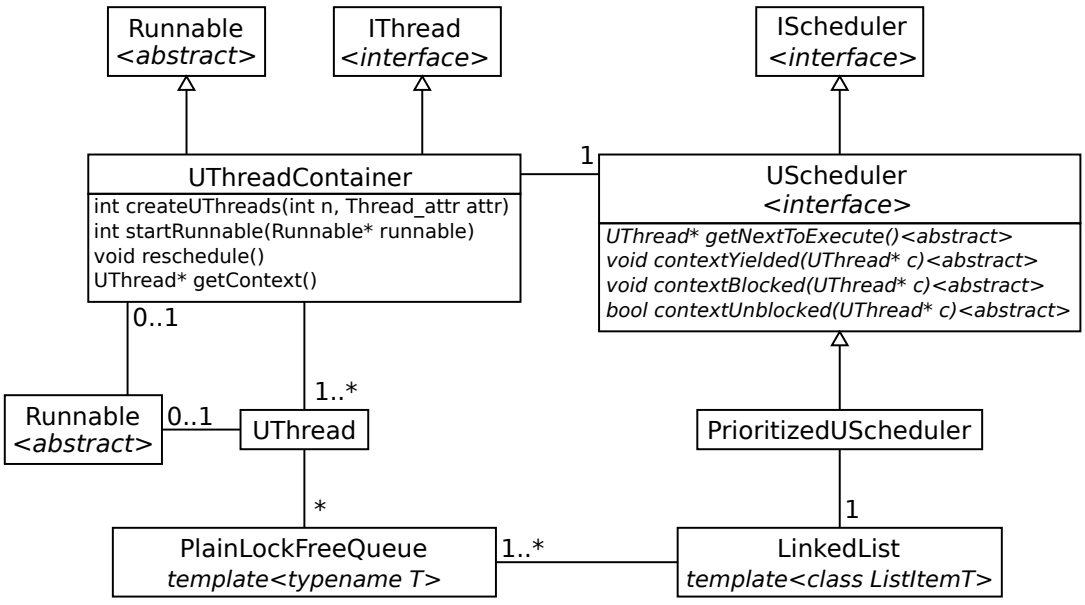

Figure 5.18: LUNA UThreading UML class diagram

UThread instantiation
*UThread*s can be created with the *createUThreads()* function, which saves the *UThread*s on a *Queue* (which is not shown, it only has an association with the *UThread*s directly). All of these *UThread*s will have received their own stack on which their thread will run in the future. With the *startRunnable()* function a *Runnable* can be assigned to a created *UThread*.

UThread scheduler and priorities
The user threading mechanism and the management of the ready and blocked queue have been split. The *UthreadContainer* is responsible for physically switching thread contexts. An *UScheduler* is in control of the ready and blocked queue. Whenever a *UThread* preempts or is finished, the *UThreadContainer* asks the *UScheduler* for a new *UThread* and reschedules.

Two implementations currently exist for the *UScheduler* interface: a prioritized and a simple equal priority scheduler. The latter is included for performance reasons and not shown in the diagram. The *PrioritizedUScheduler* maintains a prioritized ready (and blocked) queue. The idea is shown in Figure 5.19. The *LinkedList* maintains a FIFO *PlainLockFreeQueue* per priority (the mandatory *ListItem* is not shown in the class diagram). Whenever a new *UThread* is requested by the *UThreadContainer* it descends the *LinkedList* from the high priority to the low priority until a ready *UThread* is found. If no *UThread* is ready the *PrioritizedUScheduler* blocks on a *Semaphore* (not shown in the class diagram) until a *UThread* becomes ready. Priorities levels (i.e. the *ListItem*s) are currently not actively maintained by the *UThreadContainer*, meaning once added they stay throughout the application lifetime. The reason behind this is that currently no lock-free linked list implementation is available and otherwise some form of mutual exclusion must be applied, which is considered expensive and undesirable.

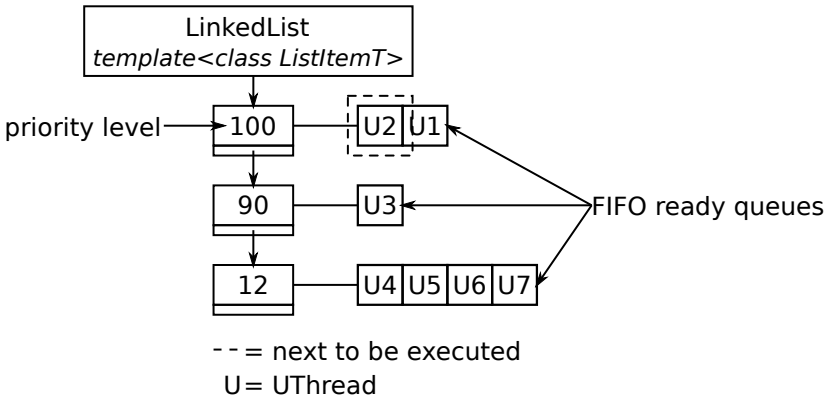

Figure 5.19: LUNA PrioritizedUScheduler

The *PlainLockFreeQueue* has been chosen for the FIFO queue for its fast performance in a concurrent setting. Furthermore, other threads (OS or UThread) may now place an *UThread* on the ready queue when required and without interfering with the *UThreadContainer* execution.

UThreadBlocker
The *IThreadBlocker* discussed in the Threading component (Section 5.2.10) is also implemented for the *UThread*s. Whenever a *UThread* must be blocked, for instance when waiting on the other side of the rendezvous channel, the *UThreadBlocker* can be used to preempt the own *UThread* and place it on the blocked queue. Actually, there is no need for a blocked queue, since the *UThreadBlocker* can hold the reference to the *UThread*. This enables rendezvous communication between UThreads and OS threads (see Section 5.4), since these will have to interact at some point.

Rendezvous communication
As already explained, *UThread*s cannot handle blocking system calls without stalling all the *UThread*s in a *UThreadContainer*. Unfortunately, the QNX rendezvous channels are also suffering from the problem. So, for the CSP execution engine a new and highly efficient *UThread* capable rendezvous communication channel, abbreviated to LUNA-U further on, has been designed and implemented (see Section 5.4). Although, this rendezvous channel has not yet been ported to the *UThread*s in general.

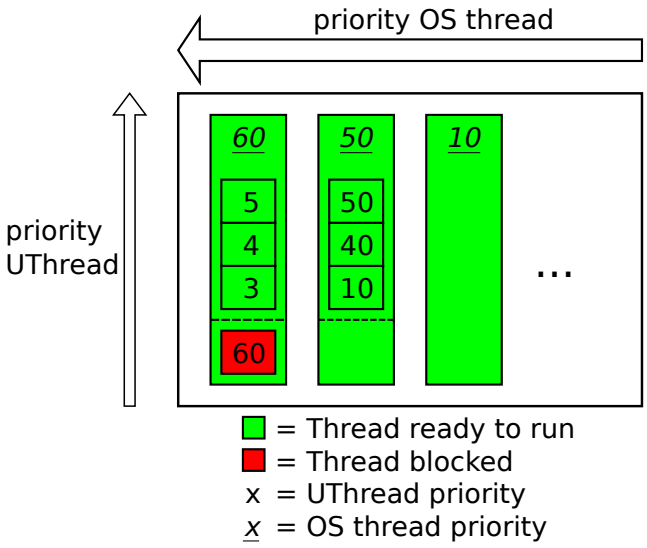

Figure 5.20: Mixture of OS threads and UThreads priorities

Combined OS threads and UThreads priority problem
The OS threads and *UThread*s are separately scheduled according their priority by the OS scheduler and the *PrioritizedUScheduler* respectively. The combination of OS threads and UThreads with different priorities might be problematic, since the OS does not know about the UThreads and does not respect the UThread priorities.

The OS threads are scheduled only according to their priorities and once the *UThreadContainer* is running it can arrange the scheduling of its own *UThread*s. Thus, this creates a form of priority inversion as can be seen in the example shown in Figure 5.20. The problem in this case is that the *UThread* with priority 50 is only allowed to run after the complete set of *UThread*s on the left OS thread (with a higher priority) have finished.

The simplest solution is to use only as much OS threads as the target platform has $cores * CPUs$. In this way all OS threads can always run and so can their *UThread*s. Nonetheless, also for this solution the *UThreads* need to be properly distributed over the OS threads, otherwise an OS thread might starve. More solutions are possible, but these are considered future work.

Synchronization and locking
The *IBarrier* interface discussed in Section 5.3.1 is also implemented for the *UThreading* component. The two-level barrier design of Brown (2007) is used for this and the idea is visualized in Figure 5.21. Basically *UThread*s requiring synchronization, first synchronize locally (*LocalBarrier*) and once all local *UThread*s have synchronized the synchronization is taken to OS thread level (*GlobalBarrier*). In the case that only local synchronization is required, the *LocalBarrier* does not need a *GlobalBarrier* and can skip this synchronization step. Regular OS threads can participate on a global level with the same algorithm and synchronize directly on the OS thread level.

Using the knowledge that *UThread*s can only be preempted on a voluntary basis in the OS thread, a smart design can be employed. Since, an object shared between *UThread*s on the same OS thread does not need any concurrency protection. Therefore the overhead costs for an *UThread*s implementation will be even lower than with a similar OS thread implementation. Of course, the whole set of *UThread*s may still be preempted when the OS thread it is running on, is forcefully preempted by the OS.

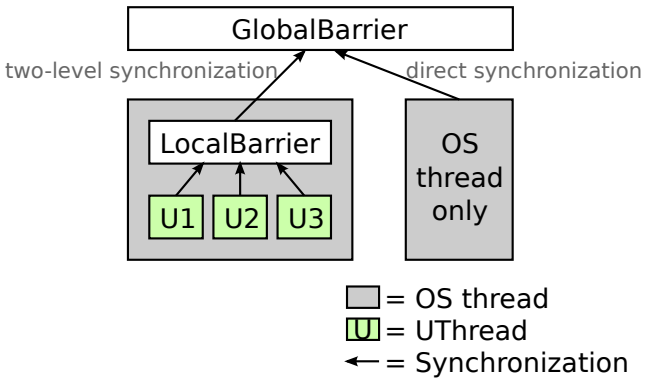

Figure 5.21: LUNA two-level barrier (based on Brown (2007))

5.2.11.2 Implementation

The component implementation is very platform dependent. For a *UThread* to run semi-concurrently to other *UThread*s three requirements should be met. First, it must have its own stack to run on, which should be 'saved' when the thread is preempted. Second, the preemption mechanism must save the current core/CPU registers and restore the registers of the thread that is activated. Last, some means to start a *Runnable* on the thread and cleanup the thread when finished are also necessary.

Creating a stack per *UThread* is easy; just allocate a fixed sized chunk of memory and divide this between the *UThread*s.

For implementing the second requirement multiple methods are available. (Jones, 2003) contains an extensive list of example implementations for Linux. The simple context switch performance test mentioned before and their availability on QNX determines which implementation is the best choice. The example implementations will not be explained as these are already well documented on the internet. The assembler implementation is just one of many possible assembler implementations. Table 5.7 lists the results of the comparison.

Method	**Mean context switch time (μs)**	**Available on QNX**
ucontext	6.99	No
assembler	0.99	Yes
setjmp/longjmp	1.68	Yes
POSIX threads	7.06	Yes

Table 5.7: Context switch speeds for different UThread implementations (Jones, 2003) on a 600MHz PC/104 running Linux (kernel 2.6.23.17)

The assembler implementation is the fastest. But, assembly code is notoriously difficult to develop and maintain, and it is by definition not portable. Therefore it was chosen to use the setjmp/longjmp implementation for switching between threads. The setjmp/longjmp functions save and reload the registers to and from a buffer respectively. Most OSs provide setjmp/-longjmp function implementations.

To get the *Runnable* to execute using its new stack (requirement 3) the stack pointer must be replaced. This is considered difficult, platform dependent and technical, and therefore the procedure is explained in the LUNA source code. However, this method cannot be used for Linux, since the stack pointer is obfuscated for security reasons there. So, the *UThreading* component interface is platform independent, but its implementation is not.

5.2.12 Real-Time Logger

The RTLogger provides an (a)periodic signal logger and extends the debugging facilities requirement, for which it makes the log output messages available off target. This is advantageous since logging to the screen or disk on a real-time OS is not always possible or limited and slow.

The RTLogger can be used on a mixture of non and hard real-time threads without breaking the real-time constraints. The application only pays for placing the logging information into a large buffer. The RTLogger thread then takes care of transferring the data whenever computational resources are unused. The buffer can be placed in ring-buffer mode, overwriting the old log values, or overflow mode, when the data cannot be transferred in time. The buffer size can be adjusted.

The 20-sim 4C package by Controllab Products (2011) also provides a real-time logger, but it is only capable of periodically logging signals. Whereas the RTLogger described here, can also aperiodically log any kind of information.

The RTLogger receiver (server) waits for a RTLogger to connect. Once connected, the received signal data is stored in a file and any log messages are printed to the screen. Optionally, 20-sim can be used to visualize the control signals during run-time. Since, the RTLogger server is not part of LUNA, but is build with it, it is discussed in Appendix H.

Furthermore, the RTLogger has been used to gather trace data of CSP processes and to determine the performance of Jiwy CSP models, see Chapter 6. However, correct CSP traces can only be generated with the number of OS threads less or equal to the number of cores/CPUs, since the OS may preempt an OS thread in the middle of a CSP process execution otherwise. The Matlab script to post process and visualize the trace data are presented in Appendix H.

5.2.12.1 Design

The UML class diagram is shown in Figure 5.22. The application and RTLogger thread are separated by a double *PlainLockFreeQueue* (Section 5.2.5.1). Any thread performing a log action (signal or output message) will first need to get an empty buffer from the 'empty' queue. If no empty buffers are left and the RTLogger is not set to ring-buffer mode, the log function will just return and thereby lose the signal value or log message. When an empty buffer was successfully acquired it is filled with the channel id, log data and current timestamp, and then placed in the appropriate transfer queue.

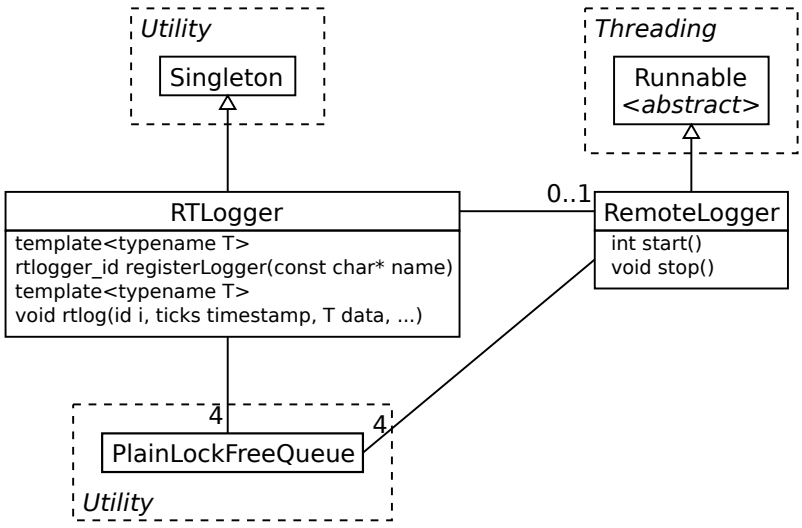

Figure 5.22: LUNA RTLogger UML class diagram

For the logging of signal data, the thread must first register the signal name and data type (*T* in the diagram). This information is then transferred to the visualisation tool, such that this information does not need to be transferred on each log action. The *rtlog()* function then performs the actions described in the previous paragraph.

The logging of output messages is handled automatically by the framework and does not require registering a channel.

The RemoteLogger uses a low priority non real-time thread for transferring the data to the visualisation tool. The low priority ensures that threads performing a function pertinent to the application will always have preference over logging. The RemoteLogger and its receiver communicate through the Socket component (Section 5.3.3).

Since signals are mostly fixed size data (e.g. integers) and log messages are variable sized data, it was chosen to maintain four buffers (two empty and two transfer queues). By default, much more buffers are available for signals than log messages, because signals mostly have a higher update rate and log messages are larger in size.

5.2.12.2 Implementation

The RTLogger component is build on top of the Threading and Socket component. The signal data types and the network packages are architecture dependent. Unfortunately, the visualisation tool does not have means to determine the correct format of both yet. So, currently only RTLoggers and visualisation tools having the same architecture work (e.g. both 32 bit systems).

5.3 Other components

The following components were also designed for this thesis project, but are less interesting from a design point of view and are therefore explained in less detail. The order of appearance of these components is shown in Table 5.3.

5.3.1 Barrier

The Barrier provides thread and process synchronisation primitives and thereby is an extension to the synchronization requirement in Section 3.2.1.7. A number of threads/processes can register a barrier object and then synchronize their program execution on it. Arriving threads will wait until all threads have arrived at the barrier, at that point all threads are jointly released. Although it is specified in POSIX amendment POSIX.1j it is found commonly available on all OSs.

The Barrier component provides a unified interface, called *IBarrier*, for OS and UThread barriers. Furthermore, it implements this interface for the OS threads.

5.3.2 Clock

The Clock component provides means to specify the date and time and time durations, and contains a wall clock interface with the OS.

The *DateTime* class consists of a time struct, available on all Unix based OSs, and a fractional seconds part. The fractional seconds are dependent on the requested resolution (*System_resolution*, a framework setting) and the used accuracy in determining the date and time itself, which is platform dependent.

The *TimeDuration* class resolution depends on the same framework setting as the *DateTime* class. Furthermore, a number of mathematical operations have been defined for this class, such as adding and subtracting time durations, to increase usability. This representation class is used in the Timing component for returning the results of time measurements.

The *WallClock* retrieves the current system time from the OS and therefore should be handled with care.

5.3.3 Socket

The Socket component provides a network socket abstraction from the OS, which perhaps in the future can be used to extend the communication requirement (Section 3.2.1.6) to inter-host communication.

Sockets can be specified to be either TCP or UDP and IPv4 or IPv6. Socket creation, setup and communication are of course not real-time. The send and receive functions of the socket can both be blocking and non-blocking depending on the used parameters.

5.3.4 ThreadPool

The ThreadPool component extends the thread support requirements (Section 3.2.1.3) and manages a pool of *PlainOSThread*s (Section 5.2.10). The threads in the pool can for example be used as buddy processes in hard real-time code.

In its current form the workers (pooled threads) take a *WorkItem* structure as a work specification. The *WorkItem* contains either a *Runnable* or function with the following signature: *void* (*func)(IThread*, void*)*. In the latter case, the *IThread* pointer is supplied by the *ThreadPool* as being the *IThread* the *WorkItem* is run on. Furthermore, a pointer to any data structure can be supplied and is fed by the *ThreadPool* as the second argument. Optionally, when the worker finishes it can execute a callback function.

This component is build on top of the Threading component.

5.3.5 Timing

The Timing component provides functionality to measure the computation performance of other components and classes. The *TimeMeasurement* class is responsible for this and return the measurement in the form of a *TimeDuration* object (see Section 5.3.2). A *TimeMeasurement* instance should be created for each required measurement.

Computers measure time generally with counters that increase on discrete intervals. A counter increase is commonly called a tick. Furthermore, these ticks are generally not in a SI unit for performance reasons.

The *TimeMeasurement* object retrieves the current tick counter value at the start of the measurement. When the measurement has finished (or an elapsed time is required) the object retrieves the new counter value. Subtracting the counter values gives the elapsed time in ticks. Because reading the tick counter takes time, the *TimeMeasurement* also corrects for the time measurement itself.

The *TimeMeasurement* class is build on top of the SystemInfo component (Section 5.2.7) and uses its platform abstraction. The SystemInfo component function *getTicks()* retrieves the current ticks counter value and furthermore contains the measurement time correction and a conversion factor to convert ticks into seconds.

5.3.6 Device-manager

The Device-manager component extends the link drivers requirement (Section 3.2.1.8).

The UML diagram for this component is shown in Figure 5.23. The main class acts as a resource counting object manager and registry. Multiple threads can share the same device driver without needing to create multiple objects for it nor destroying the device driver accidentally before all threads are finished using it. Furthermore, device driver initialization only has to occur once,

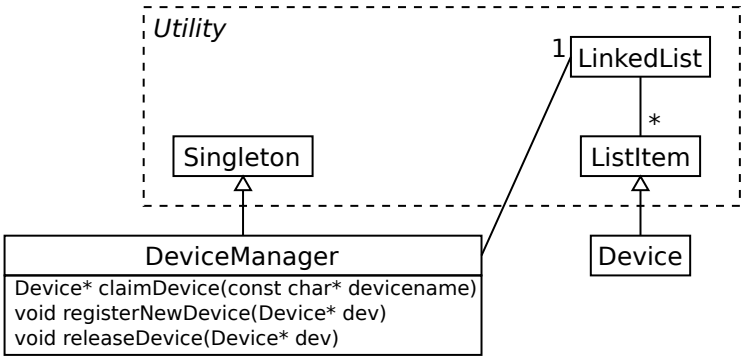

Figure 5.23: LUNA Device-manager component UML diagram

preferably at a safe non real-time initialization point. So, threads can look-up a *Device* in the registry when needed, then claim the device and lastly release it when they are done with it.

The *Device* interface contains only basic functionality. In the future it should be extended to enforce more good practices, for example automatic thread safety.

5.3.7 AnyIO

The AnyIO component contains CSP link drivers for the Mesa electronics Anything IO FPGA board (Mesa electronics, 2011). LUNA does not yet contain link drivers for all hardware available in the lab. However, this proves that link drivers in general can be included and therefore satisfies the link driver requirement (Section 3.2.1.8) for CSP.

The CT library already contained link drivers for CSP to interface with the encoders, actuators and digital IO on the FPGA. Therefore, these have been ported to the new LUNA CSP interfaces. But, the Linux driver, which was used for setting up the FPGA board, had to be modified for QNX (see Appendix G).

5.4 CSP component

This component implements the CSP execution support requirement (see Section 3.2.1.10). The paper (Bezemer et al., 2011b), partly written by the thesis author, serves as the basis for describing the design and implementation of the CSP component. A reading guide to the paper is given in Section 5.4.1 and the paper can be found in Appendix I. The standard implementation subsection is skipped, because the CSP component is only build on lower level components and thereby platform independent.

Section 5.4.2 extends the paper on some selected points, as not everything could be included into the paper.

5.4.1 CPA paper reading guide

Since this reports intends to describe the whole LUNA framework in detail and not just the combination of LUNA and CSP, some sections will contain information already presented in previous sections. The reading guide to the LUNA paper in Appendix I is given in Table 5.8. The reader is advised to *read the italic printed sections* now, as these present the CSP component's design.

Introduction	
Context	The context of LUNA is also discussed in Section 1.
Existing Solutions	Chapter 4 discussed these in more detail.
1. LUNA architecture	
Intro	Section 5.1.1 also presented the LUNA component architecture.
1.1 Threading Implementation	The threading mechanisms in LUNA are discussed in detail in Sections 5.2.10 and 5.2.11. *The two paragraphs between "For the CSP functionality ... of the CSProcess" describe what this means for CSP.*
1.2 LUNA CSP	Describes how CSP is executed conceptually.
1.3 Channels	Explains how the LUNA rendezvous channels are implemented.
1.4 Alternative	Discusses the Alternative CSP process and guarded channel implementation.
2. Results	
Intro	Presents the test setup.
2.1 Context-switch Speed	Discusses a pure context switch speed test.
2.2 Commstime Benchmark	Referred to from Section 6.2.1.
2.3 Real Robotic Setup	Referred to from Section 6.2.2.
3. Conclusions	
	Chapter 6 concludes to LUNA's usability, performance and requirements.

Table 5.8: Reading guide to Appendix I

5.4.2 Extensions to the paper

This section describes CSP details worth mentioning, but which did not make it into the paper. Table 5.9 provides an overview of all gCSP constructs and their current implementation status in LUNA. The interfaces have been updated with respect to the CT library and therefore these constructs will not work with the gCSP tool yet.

Item	Implemented	LUNA class name
Process	y	CSProcess
Reader	y	Reader
Writer	y	Writer

Construct	Implemented	LUNA class name
Process	y	CSProcess
Sequential	y	Sequential
Repetition	y	Recursion
Par	y	Parallel
PriPar	y	PriParallel
Alt	y	Alternative
PriAlt	y	PriAlternative
Input-guard	y	GuardedWriter
Output-guard	y	GuardedReader
SKIP-guard	y	AltElseOption
Watchdog	n	-
Exception	n	-

Channels	Implemented	LUNA class name
Channel	y	UnbufferedChannel or Any2AnyRendezvousChannel (OS only)
TimerChannel	y	PeriodicTimerChannel
BufferedChannel	y	BufferedChannel
ExternalChannel	n	-

Table 5.9: Available gCSP constructs in LUNA

5.4.2.1 Priorities

First of all, priorities are only mentioned twice in the paper, whilst it is fully implemented for CSP. Note, the CSP component is currently the only implemented execution engine and the implemented LUNA-U rendezvous channel for UThreads and OS threads is only available in the CSP component. Therefore it was chosen to discuss the combination of priorities and CSP processes and the effect that it has on rendezvous communication and threads in this section. Most of the discussion can be transferred to the UThreading component (Section 5.2.11) once the LUNA-U rendezvous channel has been adjusted for the general use case.

Threads
The priority of CSP processes is also set to their OS thread or UThread, such that these are scheduled accordingly (see Sections 5.2.10 and 5.2.11). The possible *UThreadContainer*'s OS thread priority problem is discussed in the UThreading component.

To set the priorities of the whole CSP compositional tree at once, the application developer sets the top level CSP process priority. Each parent CSP *TreeNode* will then set its children's priorities. For the *PriAlt* and *PriPar* processes, which are based on their non prioritized versions, this means that the priority of the first child is equal to parent (*PriAlt*/*PriPar*) priority and that the priority of the rest of the children decreases towards the last child.

To fully support CSP semantics the priorities should also be adhered during rendezvous communication.

Rendezvous communication
Table 5.10 gives an overview of the properties of the two rendezvous communication methods currently implemented. Most of these properties have already been explained in the paper and previous sections, but are presented here again for completeness. The QNX characteristics probably hold as well for the alternate implementations suggested for the other real-time OSs in the Communication component's section (5.2.4). Although only any-to-any communication is mentioned, this includes one-to-one communication and other well known combinations.

Property	**QNX rendezvous channels**	**LUNA-U rendezvous channels**
Communication between OS threads	y	y
Communication between OS threads and UThreads	n	y
Any-to-any communication	y	y
Guarded channels (for the Alt CSP process)	n	y
Priority scheduling	y	y
Priority inheritance algorithm	y	n
Buffered communication	n	y

Table 5.10: QNX versus LUNA-U rendezvous channels

The paper describes the implemented LUNA-U rendezvous channels in great detail (although it does not use this abbreviation explicitly). But a few details have been left out. Figure 8 in the paper does not show that *IReader* also implements the *ListItem* abstract class (Section 5.2.5). And in Figure 7 the *LinkedList*, which manages the prioritized waiting list for these *IReader*/*IWriter* processes, is not shown for either side of the any-to-any channel. So, the (un)buffered channel implements a prioritized any-to-any hard real-time capable (guarded) rendezvous channel.

5.4.2.2 CSP traces and timing verification

The RTLogger described in Section 5.2.12 can be used to generate traces of the CSP execution and verify timing. The build system has a menu setting to enable one or both. Once a trace or timing log has been obtained the Matlab scripts presented in Appendix H can be used to analyze them. These are also used in the next chapter for validating the real robotic setup.

5.4.2.3 Example code for CSP deployment

The paper already describes how CSP processes can be spread over OS threads, but to show the full potential of LUNA an example has been included in Appendix F.

5.5 Conclusions

The approach to divide the LUNA functionality in separate components, with clear purposes and interfaces, has lead to a design in which components are loosely coupled. The resulting components and their dependencies are clear and can be logically explained.

Grouping the components into layers gives additional meaning to their implementation. The core components layer, provides generic interfaces for platform specific features. Since roughly 70 % of the used OS functions are in the POSIX standard and this is well supported by the target and development OSs, using the POSIX standard as a basis was a good choice. The build system enables the framework developer to specify different files if non conforming functionality is required. No peculiar design patterns nor includes are necessary for this. The architecture

abstraction needed in the layer is limited to a few components and well documented, and also managed by the build system.

Other components can make use of the core component's interfaces to employ platform specific features without knowledge of the actual chosen platform. In this way a separation of concerns is achieved, for example, the CSP execution engine only depends on generic interfaces (like *IThread*) and therefore is able to run on any platform.

All functional requirements mentioned in the analysis chapter (3.2) have been met. To be specific, the framework is designed for hard real-time, with only a few well documented non real-time functions. The requirements each component fulfills are given at the beginning of each component's subsection. The next chapter reflects on the requirements in detail.

The addition of UThreads has had a considerable impact on the design of the other components. When inspecting their code it can be seen that the overhead inflicted by this is kept to a minimum:

- Only a few extra *IThread* pointers are necessary for some generic interfaces, to provide references to a particular *UThread*.
- The factory methods take care that the right OS thread or UThread implementation is chosen at initialization. Thus, no run-time overhead is incurred due to the introduction of multiple thread types.
- *UThread*s have great potential for small highly parallel threads, as is proven in the next chapter.

Finally, the LUNA framework has a complete new design. Unlike the CT and C++CSP2 library, the CSP implementation can be used multi-thread, hard real-time and with priorities. Furthermore, the separation of concerns between CSP and the Threading component enables new research as the design can be extended easily with new ideas (see Chapter 7).

6 Evaluation

This chapter gives a qualitative and quantitative evaluation on the LUNA framework design.

6.1 Qualitative evaluation

The quality of LUNA is reviewed by looking at the intended functional and non-functional requirements from Chapter 3.

6.1.1 Functional requirements

All functional requirements, given in Section 3.2, have been met. Nonetheless, each requirement is discussed in detail.

6.1.1.1 Platform independence

The LUNA framework has been divided into three component layers (see Figure 5.1). The platform abstraction 'core' layer takes care that any component build onto these components is platform independent. The platform independence is managed by the build system which, based on its configuration, selects the appropriate implementations for the target OS and architecture for each component separately. So, the other two layers are platform independent.

Most component designs are based on the POSIX.1 standard, because most of these functions are commonly available on the selected target and development OSs. The QNX OS, which was chosen for the first target OS implementation, is certified POSIX conformant. Only the Timer, UThreading and Utility components (respectively Sections 5.2.6, 5.2.11 and 5.2.5) required a (partially) QNX specific implementation. Although it is not mandatory, some generic implementations should be revised for RTAI and Xenomai to take full advantage of their OS.

Concluding, LUNA's first implementation is fairly platform independent due to its build system and using functions in the POSIX.1 standard.

6.1.1.2 Real-time constraints

LUNA is designed for hard real-time, with a few documented non real-time functions (e.g. for setup). Threads can have hard, soft and non real-time constraints, as explained next.

6.1.1.3 Thread support

The Threading component (Section 5.2.10) provides a generic thread interface and implements this for OS threads. Two OS thread classes were designed to clearly differentiate between hard and non real-time OS threads. On hard real-time OSs, like QNX, the real-time threads can be divided in hard and soft real-time by properly distributing the thread priorities.

The UThreading component (Section 5.2.11) provides a many-to-one hybrid threading model, whereby a set of user threads is run on one OS thread. User threads are lightweigth threads, can have priorities and are ideal for small parallel tasks. The disadvantage is that a blocking system call on a user thread blocks also all other user threads running on the same OS thread.

6.1.1.4 Priorities

Priorities can be assigned individually to OS and user threads. On real-time OSs higher priority OS threads preempt lower priority OS threads and the highest priority OS thread is run first.

User thread priorities are local to the user thread set and the OS thread they are run on. The OS decides which OS thread to run without considering the unknown user threads. User threads are not preempted by the OS, but are scheduled locally on their priority as well.

6.1.1.5 Periodicity

Timers can be used to periodically execute non and hard real-time threads or implement delays. For periodic timers a timer manager has been introduced. The manager can save scarce OS resources, by combining multiple OS timers into one OS timer and multiple internal timers. Additionally, its design enables application developers to specify in which order the timer events should be handled, in contrast to multiple OS timers.

6.1.1.6 Communication

The Communication component (Section 5.2.4) provides rendezvous communication for OS threads and processes. Real-time OSs schedule access to the rendezvous channel based on the thread priority, such that higher priority waiting threads are given preference when trying to acquire the communication channel.

The OS thread rendezvous communication is not suitable for user threads, since it uses a blocking system call. For CSP an any-to-any user and OS thread rendezvous channel was designed, but it has not yet been ported to the general use case. This channel also schedules access based on thread priorities.

6.1.1.7 Synchronization

The LockSync component (Section 5.2.3) provides synchronization primitives for OS threads. The Barrier component (Section 5.3.1) specifies a barrier interface and implements this for OS threads.

User threads do not require mutual exclusion primitives, as provided by the LockSync component, if a smart design is employed. For thread execution synchronization, the barrier interface is implemented in the UThreading component (Section 5.2.11) as a two-level barrier (Brown, 2007). First user threads are synchronized locally in their user thread set and then on a global level. OS threads can also participate in the same algorithm and synchronize directly on the OS thread level.

6.1.1.8 (Link) drivers

The Device-manager component (Section 5.3.6) can be used to maintain a globally unique and accessible device driver instantiation per physical device and arrange shutdown of the device when all threads are finished with it. The AnyIO component link-drivers use this and thereby shows that LUNA is capable of interfacing with hardware.

6.1.1.9 External code and/or library integration

The build system (Section 5.1.3) enables the integration of external software blocks.

6.1.1.10 CSP execution support

The CSP language interpretation followed by the CT library (Orlic and Broenink, 2004) has been fully implemented in LUNA, except watchdog processes, external channels and exceptions.

CSP processes can run on any LUNA thread type and a mixture of CSP processes running on different thread types can work together. Because, the CSP language implementation is not tightly integrated with the other LUNA components.

Since the QNX rendezvous implementation does not provide guarded channels (for the Alternative process) nor can communicate with user threads, the LUNA-U rendezvous channel is implemented to achieve this.

At the moment, the CSP execution engine is the only implemented execution engine. But, due to the component based approach other execution engines (like state machines) are also possible.

6.1.1.11 Safety layer

LUNA provides a solution for inter-component safety management in the form of an emergency manager (Section 5.2.9). This component will, when signaled of an emergency, call all the registered safety functions to properly shutdown the application/device. This should, for example, prevent the device from harming itself or its environment.

In the future, this component should be extended to 'catch' standard software failures, which the application developer did not anticipate, and provide standard safety blocks, such as blocked actuator detection algorithms.

The Anyio component also registers its device link-drivers with the emergency manager. This shows that LUNA is capable of interfacing with hardware and with added safety, compared to the other CSP libraries.

6.1.1.12 Debugging facilities

The standard available Debug component (Section 5.2.2) provides debugging, logging and tracing functions for other components and application development. The debugging functions can be enabled on request and when disabled they do not incur any overhead.

The real-time logger (Section 5.2.12) can send, among others, debug and trace information over a (local) network to a development PC, in order to perform run-time analysis or to store it for off-line analysis. The RTLogger can also be used to (a)periodically log integer or floating point signals. The use of the RTLogger does not break real-time constraints, but does cost some computation time to place the information in the appropriate buffers.

6.1.1.13 Self testing

The build system (Section 5.1.3) enables the self testing requirement and is discussed in Section 6.1.6.

6.1.2 Non-functional requirements

The LUNA framework is designed according to the hard real-time programming rules stated in Section 5.1.5. Non real-time functions have been documented in the LUNA API.

The design methodology to divide LUNA in multiple loosely coupled components makes it a scalable framework for any kind of OS capable architecture. Components not required can be disabled. Additionally, the abstraction in functional objects only adds a small overhead compared to using the bare OS functions.

The interfaces designed for the LUNA components have been designed according to the (comprehensive) POSIX.1 standard and by investigating other frameworks. Therefore, the interfaces are closely related to knowledge most application and framework developers already have about software engineering. Additionally, the framework components ease application development by implementing tedious setup and other routines in default behavior.

Threads have a unified interface for hard real-time, non real-time and user threads. Furthermore, the LockSync, Communication and Barrier components (respectively sections 5.2.3, 5.2.4 and 5.3.1) provide generic interfaces for various tasks.

LUNA has been carefully designed by investigating failures and shortcomings that are present in the CT library and by following the latest programming techniques. Section 6.2 shows that the new framework is also better than the current CT library from a quantitative point of view.

The LUNA framework has been tested with unit and integration tests, see Section 6.1.6.

Last, the new framework completely stands on its own, i.e. it is not dependent on any other library/framework nor uses any strange features not commonly supported by compilers.

6.1.3 Main research question: Is QNX a valid choice to implement CSP based communicating threads?

The QNX OS is hard real-time, certified POSIX.1 conformant, runs on multiple architectures (such as x86 and ARM) and provides an extensive development tool chain, which all are important for the LUNA framework in general.

The provided rendezvous communication channels are partially suitable for the CSP implementation, since these support unbuffered, any-to-any and CSP similar rendezvous communication for OS threads. However, the following disadvantages are recognized:

- Channel guards, required for the Alternative CSP construct, are not supported.
- Buffered communication is not supported. This form of communication is one of the possible solutions to prevent priority inversion by design, as opposed to the on-line and dynamic problem solving implemented by a priority inheritance algorithm.
- The QNX rendezvous channel cannot be used on user threads, but these are necessary in the current setting to achieve fast CSP implementations.

Thus, the choice for QNX as a first implementation is not obvious anymore, since its rendezvous communication channels are not usable except in certain settings.

6.1.4 Minor question a: How should a platform independent framework be designed?

From the investigated frameworks and libraries it can be learned that a framework should be separated in functional components, such that:

- A component's name clearly specifies its purpose.
- Components can be build on top of each other. Low level components provide platform abstraction and higher-level components can use this functionality platform independently.
- In the OS abstraction case, OS functionality belonging to each other can be grouped. This facilitates implementing the framework for multiple OSs.
- The framework becomes scalable, because components can be enabled/disabled on demand.

Clear interfaces should be designed for platform independent classes, where platform dependent classes must conform to.

Function skins (Section 4.8.3) can be used if OSs use different function names for similar behavior. A single function may also point to multiple functions, to form a kind of policy. Thus, the function skins approach promotes code reuse.

All investigated frameworks and libraries use file overloading to some extend, i.e. to select the appropriate OS specific implementation file. In the LUNA build system, the file overloading concept is extended. The file overloading mechanism matches each file separately based on the current platform configuration, whereby more specific matches get preference over generic implementations. So, the build system selects the highly optimized implementation for a target when available, and otherwise uses the generic implementation. This is achieved without macros in source nor header files, keeping the design clear for the developers.

For the OS abstraction components, clear definitions and groups can be deduced from the POSIX.1 standard.

So, for LUNA, the build system together with the POSIX.1 standard and smart programming techniques make it an efficient platform independent framework.

6.1.5 Minor question b: How can the CSP execution engine be designed, such that the CSP implementation is loosely coupled to the rest of the framework?

The CSP component is build on top of high-level and core components. These components provide platform independent functionality via generic interfaces, which are used by the CSP component. The following paragraphs try to make clear how this separation of concerns between CSP and the components it is build on has been achieved. The complete implementation has been discussed in detail in Section 5.4.

CSP processes implement the *Runnable* interface, such that they can be run on any thread type asynchronously. The CSP construct functions are run synchronously, i.e. directly by the invoking thread.

In the hard real-time LUNA setting the CSP processes and constructs are created beforehand and then wait for activation. The waiting action is implemented by *IThreadBlocker*, which can be used to block its own thread and thereafter any other type of thread can unblock the thread again. Thus, CSP processes and constructs can be activated by any parent or sibling CSP process or construct indifferent of their thread type.

The LUNA-U rendezvous channel also uses the *IThreadBlocker*. The CSP process arriving first at the rendezvous channel, blocks itself on its *IThreadBlocker*. When the other side of the channel arrives, it performs the data transfer and unblocks the earlier arrived process' *IThreadBlocker*. So, the particular thread types are not important for this mechanism as well.

The CSP component is loosely based on the Threading component, because all CSP constructs and processes use only the *IThreadBlocker* concept to influence each others execution. The other components used for the CSP implementation only provide a direct service, for instance exclusive access to the rendezvous channel's internal data structures.

6.1.6 Testing

LUNA is a component framework, its build system (Section 5.1.3) provides the ability to specify per component multiple tests. Particular components and their tests can be enabled/disabled on demand. In this sense the build system provides the ability to specify unit tests. Additionally, components can contain integration tests. To facilitate this, test component dependencies can be specified separately from the normal component dependencies inside a component definition.

The build system does not discriminate between black-, grey- or white-box test methods. The file overloading of the build system also works for tests, though slightly different. It is especially handy for white-box testing, since component implementations might differ per platform.

Currently, all components contain at least one black-box unit test to check basic functionality and some complex components have multiple grey-box unit tests to verify important details. The exact tests can be found in each components' test directory and are therefore not listed here.

An integration test has been performed with the Jiwy setup, discussed in Section 6.2.2. This extensively tested the CSP component and indirectly tested Communication, CSP, LockSync, Threading, Timer, UThreading and Utility components. But, this test is not reproducible without a similar setup.

In the future more software only tests should be added such that the LUNA framework can easily be checked on defects, perhaps even on unofficially supported platforms.

6.2 Quantitative evaluation

This section gives quantitative results obtained by running tests on an actual setup. The setup's embedded computer is described in the introduction of Section '2. Results' of the LUNA paper in Appendix I.

6.2.1 CommsTime test results

The CommsTime test is used by the OCCAM community to measure the context switch and communication time between CSP processes. The test is explained and a performance comparison between LUNA, the CT libraries and C++CSP2 is given in Section '2.2 Commstime Benchmark' of the LUNA paper in Appendix I.

To summarize, Table 2 in Appendix I compares optimized Commstime implementations and shows that LUNA is at least twice as fast as the other three frameworks. Table 3 in Appendix I compares Commstime implementations from simple code generation and shows that LUNA is again at least twice as fast as the other frameworks.

6.2.2 Jiwy demo

The Jiwy setup (discussed in Appendix A) is used to test LUNA in a practical setting, controlling a real setup. The test is described in Section '2.3 Real Robotic Setup' of the LUNA paper (Section 5.4.1) and gives the performance evaluation, comparing LUNA with the CT library.

To summarize, Table 4 in Appendix I shows that LUNA is suitable for embedded hard real-time applications. Furthermore, it shows that LUNA can handle non optimized frequencies better than the CT library.

The CSP implementation has been successfully verified with the RTLogger, which has an option to record CSP process execution while the CSP model is running and controlling a real setup. Appendix H describes how such a verification can be performed.

7 Conclusions & recommendations

7.1 Conclusions

The goal of this assignment was to design a new state of the art framework for platform independent embedded control application development, whereby CSP-based execution support should be available for optional use.

The new framework satisfies all functional and non-functional requirements, as shown in Chapter 6. A few distinctive implemented features, compared to other frameworks, are highlighted. The framework is designed for hard real-time. Multi-threading enables new embedded architectures with multiple cores/CPUs. The platform independence layer provides support for multi-platform development and thereby the framework should be able to run on all setups in our laboratory in the future. The optional CSP-based execution support makes that the framework is also usable in non Add "CSP" to dictionary related contexts.

Reflecting on the initial assumptions, the choice for QNX is not that obvious anymore, since the provided rendezvous channels are only usable between OS threads for non guarded and unbuffered communication. Nonetheless, QNX provides a good platform to build a real-time framework, there is enough support from the OS to keep implementation tasks maintainable.

Platform independence is achieved by the combination of a new build system, analysis and design. The build system provides OS, architecture and component configuration options. Based on these settings advanced file overloading mechanisms are applied to select the right implementation. The use of macros is kept to a minimum with this approach, preventing messy code constructions.

Analysis of state of the art frameworks and libraries has shown that clear interfaces, file overloading and function skins are preferable. LUNA uses these features as well. However, the LUNA build system file overloading is more advanced.

Furthermore, analysis of target and development OSs has shown that the POSIX.1 OS standard is available on most platforms by default and thereby a lot of common functionality is supported on all of them. Therefore, it is believed that implementing the other target OSs is not too much work.

Combining both the build system and analysis into the design, resulted in platform independence and code reuse where possible.

The framework implements standard OS threads and a many-to-one user threading model. The user threads are ideal for small highly parallel tasks which do not require frequent interaction with the OS. The OS threads make that multi-core/CPU architectures can be fully exploited now.

The CSP-based execution support uses generic concepts provided by high-level and core layer components, and thereby is loosely coupled to these. The core layer makes the CSP component also platform independent. Other execution engines, for instance a state machine execution engine, can also be implemented like this.

CSP processes can have priorities assigned, which are administered to threads and are uphold during rendezvous communication. The CSP processes in a model can be deployed on a mixture of OS threads and user threads, and still work together. The supplied factories and helper functions make that the application developer can easily distribute these processes differently, without changing its codes.

Besides standard debugging and tracing support, a real-time logger is available. This real-time logger can be used to (a)periodically send control signals or debug and trace information over a (local) network to a development PC, in order to have run-time analysis or to store it for off-line analysis.

The real-time logger does not influence the executing application noticeable as it is designed for mixed real-time constraints. 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. Though, data may be lost, when the logger is not able to transfer the buffers rapidly enough and a buffer overflow occurs.

Especially logging the activation of processes is interesting, as this could provide valuable timing information, like the cycle time of a control loop or the jitter during execution. So it is possible to influence the application with external events and directly see the results of such actions. It is also possible to follow the execution of the application by monitoring the states (running, ready, blocked, finished) of the processes. This information could also be fed back to a graphical modelling tool, in order to show these states in the designed model of the application.

The quantitative evaluation for a simple robotic setup (Section 6.2) shows that the new CSP design is more efficient and faster than the CT library.

7.2 Recommendations

To prove that the component designs are truly platform independent, not just theoretically, implement other target and development OSs as well. Furthermore, an OS-less architecture, like a NXP mbed, is also interesting.

To properly specify and handle expected faults, CSP exceptions and channel poisoning should be implemented. The CSP implementation is complete for the normal execution flow of CSP models.

The LUNA-U rendezvous channels, specifically designed for CSP, should be ported to the general user thread use case. Because, these are currently the only implemented user thread capable rendezvous channels.

More (corner) applications should be (re)developed with LUNA to check LUNA's performance and usability for a wider application range. Since only one corner application was used in this thesis to determine LUNA's efficiency and performance.

Create or modify a (graphical) modelling tool to implement LUNA code generation. Building the application for the simple robotic setup took some time. There are only about 51 processes to control this setup. Of course this could be less, but it already takes too much time to develop controller applications like this by hand, so code generation for LUNA is required. Furthermore, code generation is also required in order to attract users to start using LUNA, also for educational purposes.

Enhance the Matlab scripts discussed in Appendix H for better visualization. Tracing a CSP model execution with the real-time logger is useful to determine if it works correctly and where the model perhaps went wrong. But, (periodically) running embedded control applications generate a lot of data very quickly. To create better insight, the Matlab scripts should be extended, for instance to show only differing CSP execution traces.

Extend the framework's emergency manager to identify non-anticipated emergencies (e.g. *ctrl+c*), by the application developer, and deal with them accordingly. Furthermore, provide some good practices and their implementations. For instance, an algorithm to detect blocked actuators.

Implement a solution to handle blocking system calls from real-time user threads, since these cannot handle system calls themselves. For the combination of CSP and user threads this was not a problem, since CSP processes running on different thread types can work together. Whenever a CSP process requires a system call, it is simply placed on a OS thread itself and no buddy process is required. For other OSs, for instance RTAI with Linux, this might be problematic, since non real-time Linux threads are not equal to real-time RTAI threads. Furthermore, the problem is only solved for CSP. So, for user threads in general, a new solution should be implemented. For example, by letting a buddy thread, from the thread pool, perform the system call.

A Appendix - Domain analysis

The domain analysis has been performed with four corner applications, selected from our laboratory. First, each setup will be described in short to give the reader a broad idea of its purpose. Then the characteristics will be presented, with the interesting features highlighted. Note, this text is not intended to give a full description nor present the software solution of these setups, for this the reader is encouraged to consult the various references in the respective descriptions.

The results of the domain analysis are used in Chapter 3 to determine the requirements and anticipated problems of the new framework.

The discussed Jiwy setup next is used during software development and for evaluation purposes (Chapter 6) and as a first demonstrator for LUNA.

A.1 Jiwy

Jiwy is a mechatronic setup for holding a camera. The construction contains two joints that allow the camera to rotate on a horizontal axis and a vertical axis. The maximum swing is limited by mechanical end stops. These prevent full swings so that the wires cannot be twisted or damaged. Each joint is equipped with one DC motor and one incremental encoder. The mechanical setup is connected to a PC-104 embedded PC with an Anything I/O card (Appendix G), see Figure A.1. In the standard configuration, before LUNA, the stack has the Linux OS with RTAI installed. This setup has had multiple realisations over the years, among others, an ADSP and gCSP model implementation were used. The latter will be discussed for the domain analysis, but the design process itself is not part of the discussion.

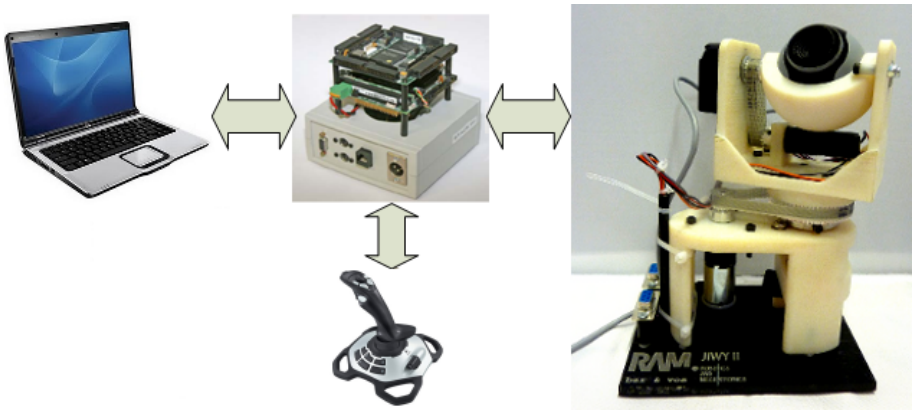

Figure A.1: Jiwy gCSP model

The gCSP model is shown in Figure A.2. Three main processes and multiple smaller reader/writer processes are shown. The Pan and Tilt processes generate the control signals for the separate degrees of freedom of the platform. The external readers and writers (in blue) interface with the environment, i.e. these are the device driver interfaces. The SanityCheck process checks the generated control values from the Tilt and Pan processes before passing them to the motor device drivers. The SanityCheck is therefore considered to be the safety layer of the setup. The required periodic repetition of all processes and setup calibration are not shown.

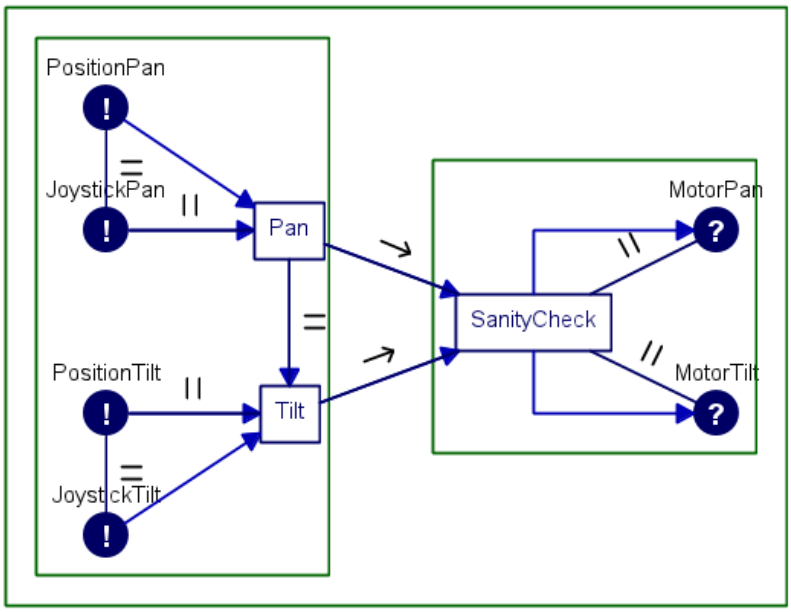

Figure A.2: Jiwy gCSP model

A.1.1 Characteristics

The system software has been designed in a **CSP model** with **multiple concurrent processes** and **rendezvous communication**. The execution flow is dynamically determined and managed by a **CSP scheduler**, which will take care that only (sub)processes with all their (execution and communication) dependencies fulfilled will be executed.

The implementation requires **hard real-time constraints** for the PID controllers inside the Pan and Tilt processes (hence the RTAI operating system). These controllers were designed with 20-sim and its, so called, 4C extension is used to generate C++ code for the controller implementation. This **externally generated code** has been plugged, without modification, into the model. The strict periodic timing required for the controllers is implemented by a **periodic timer** channel which synchronizes the (untimed) CSP model to a real-time clock by influencing the execution flow in the model.

To prevent harmful behavior a **safety layer** has been added. Furthermore **link/device drivers** have been developed which are suitable for hard real-time processes. The **strict separation between readers and writers** in gCSP models might introduce problems when device drivers are instantiated more than once.

A.2 Production Cell

The production cell setup designed by Berg, van den (2006) is a mock-up of an industrial production line system (in this case a plastics moulding machine). This particular production cell setup is a circular system that consists of 6 production cells that operate simultaneously and semi-independently, see Figure A.3(a). Each of these cells, called Production Cell Units (PCUs), executes a single action in the production process. Furthermore, the intermittent product transfer between PCUs may be organized decentralized, i.e. in the PCUs themselves. Hence, this particular setup is appropriate to investigate the specific implementation needs for distributed control using a framework.

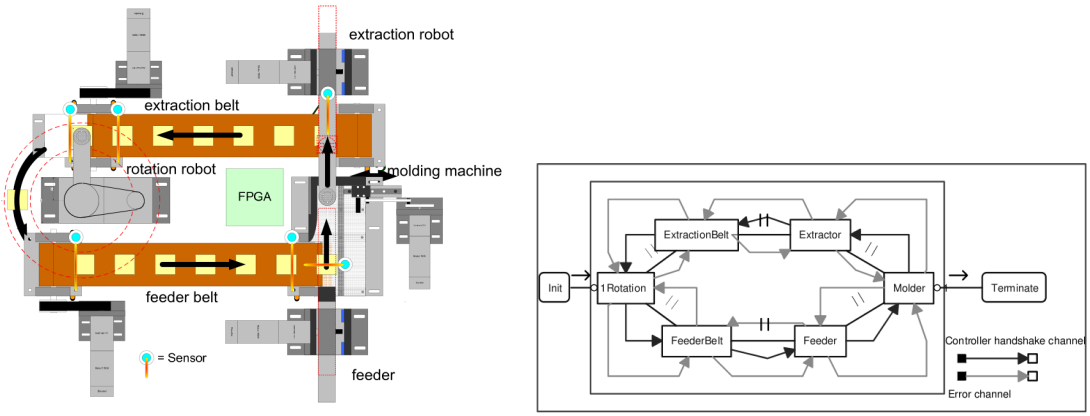

Figure A.3: Production cell setup overview (a) and top-level design (b)

In the master thesis of Zuijlen, van (2008) a general design for a distributed control structure is proposed, which will be discussed for the purpose of this domain analysis. Although the particular implementation is for a FPGA-based solution it is believed to be generally applicable for distributed control.

The top level design is shown in Figure A.3(b). This design is based on the preferred operating direction: feeder belt > feeder > molder-door > extractor > extraction belt > rotation robot > feeder belt. During normal communication and when a failure occurs the PCUs only need to communicate with their direct neighbors. The former mostly signals succeeding PCUs that its operation finished and therefore they should expect an input to their PCU. For the latter for instance, when the feeder is stuck the molder-door should be opened and the feeder belt should be halted in order to prevent cascading failures. The top level design thus couples the PCUs together and specifies their communication interfaces.

In order for the distributed PCUs to operate independently, a single PCU should maintain its own operating state, implement a stand alone controller and perform some checks to determine its operating health and prevent hazardous conditions. The proposed generic model for all PCUs is shown in Figure A.4, which also includes a user interface and low-level hardware interfaces.

A.2.1 Characteristics

This particular implementation of the production cell employs a **distributed control structure** with a **fixed repeated (software) pattern**. The distributed PCUs should be able to **run in different threads, operating system processes or on another embedded platform**. This means that a **flexible connector pattern** is required, with preferably **one communication interface** to enable seamless integration. Furthermore, some sort of **remote link driver** is required for spatial distributed PCUs.

The PCU design has a **controller, command and safety block**. A highly reactive implementation of the PCU requires **multiple threads with different real-time constraints** for one PCU. The controller will require **hard real-time periodic** behavior and therefore the safety block as well. But, the command block might only require **soft real-time** behavior and furthermore the user interface might even be **non real-time**. The soft and non real-time blocks should be able to **communicate with the hard real-time blocks without breaking its constraints**.

The **controller implementation might be different for each PCU**, where for some only the parameters differ and for others the complete implementation. Furthermore, the PCU might **switch the controller implementation during runtime**.

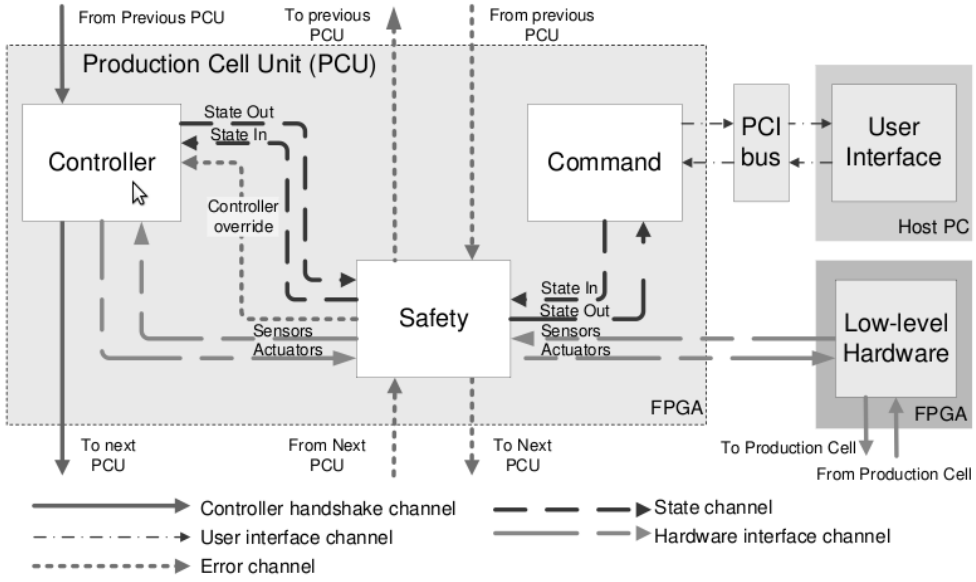

Figure A.4: Production Cell - Generic PCU design (Groothuis et al. (2008))

The complete communication (inter PCU, sensor inputs and actuator outputs) is routed through the **safety block**. The number of elements and the type of the **communication vector might be different per controller**, because for example the state vector is different per mechanical system controlled by a PCU. Furthermore, the safety block internally keeps a record of its own and neighbouring PCU states. For more sophisticated designs a **state machine** might be required.

For the production cell management an **user interface** is present which sends its commands to the command block, e.g. to start the production line. The production cell also requires **low-level hardware interfaces**.

A.3 Humanoid Head

The humanoid head (Visser (2008), Reilink (2008) and Bennik (2008)) consists of a neck with four degrees of freedom and two eyes (a stereo pair system) with one common and one independent degree of freedom. The mechanical design of the humanoid head is shown in Figure A.5(a). The intention of the project was to allow the head to focus on and follow a target, showing human like behavior.

A.3.1 Characteristics

The humanoid head consists of a **separate vision processing and motion control algorithm**. The vision processing is implemented on a **non real-time PC** and the motion control on a **hard real-time embedded platform**. The algorithms communicate through a **link driver with non-blocking I/O**, which therefore does **not break the real-time constraints** of the motion control algorithm.

The following text only discusses the proposed new software architecture.

The software implementation employs a **layered structure**. Top-down, see Figure A.5(b), these require **different task frequencies**, 60 Hz for the motion control algorithm and 1kHz for the PID loops, and real-time constraints. The layered structure uses a **fixed software pattern**, which facilitates development and enables multiple different behavioral implementations for components.

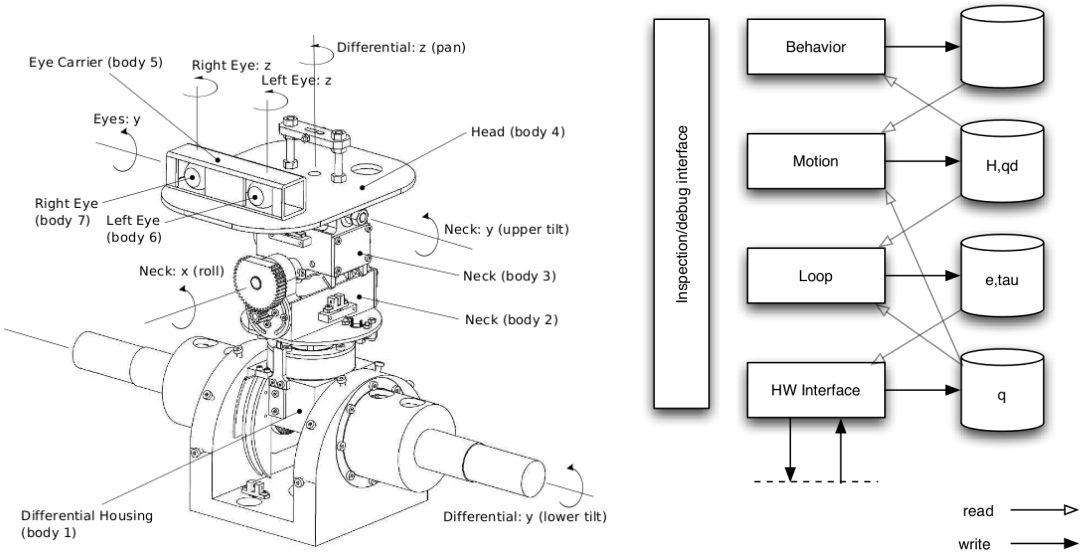

Figure A.5: Humanoid head - mechanical overview (a) and software design (b)

The motion control algorithm performs a considerate number of **matrix operations** to determine the set-points for the PID controllers. These matrix operations are implemented by an **external library**, the GNU Scientific Library.

The **PID controller** implementations are the same for all actuators, but **differ in the parameters**. The PID outputs flow through a separate safety layer (the software HW interface), which should prevent hazardous operation of the **hardware interface**. This hardware interface consists of a FPGA based PC/104+ Anything I/O card.

The embedded platform software will be implemented using a **stepwise implementation and testing strategy**. First only the motion control algorithm will be implemented whereby simulated target positions and low-level signals will be used to test the system. The **actuator and sensor values will need to be re-routed to the simulation environment**, which simulates the physical effects of the actuators and writes the resulting virtual sensors values back to the embedded platform. Because the simulations steps probably will be slow a **virtual real-time clock** will be needed, which only progresses time when simulation values are received.

A.4 TUlip

The TUlip soccer robot is a project of three technical universities in the Netherlands (3TU) to compete in the Robocup robot soccer tournament RoboCup (2011). The robot is teen size in height and has 14 actuated and 2 passive degrees of freedom. Each leg has 6 degrees of freedom, which allow the robot to move in a human like way. The mechanical configuration is designed for dynamical walking, i.e., it is unstable and should continuously have stabilizing control.

TUlip should be able to walk dynamically and kick the ball while maintaining its balance, process human-like sensor information, make tactical decisions and communicate among others. The information about the environment is acquired by sensors, such as:

- Pressure sensors in both feet
- Acceleration sensors
- Vision system

All functions are performed autonomously on the embedded platform in the robot, a PC/104 stack. The software design discussed in this domain analysis has been created by Lootsma (2008).

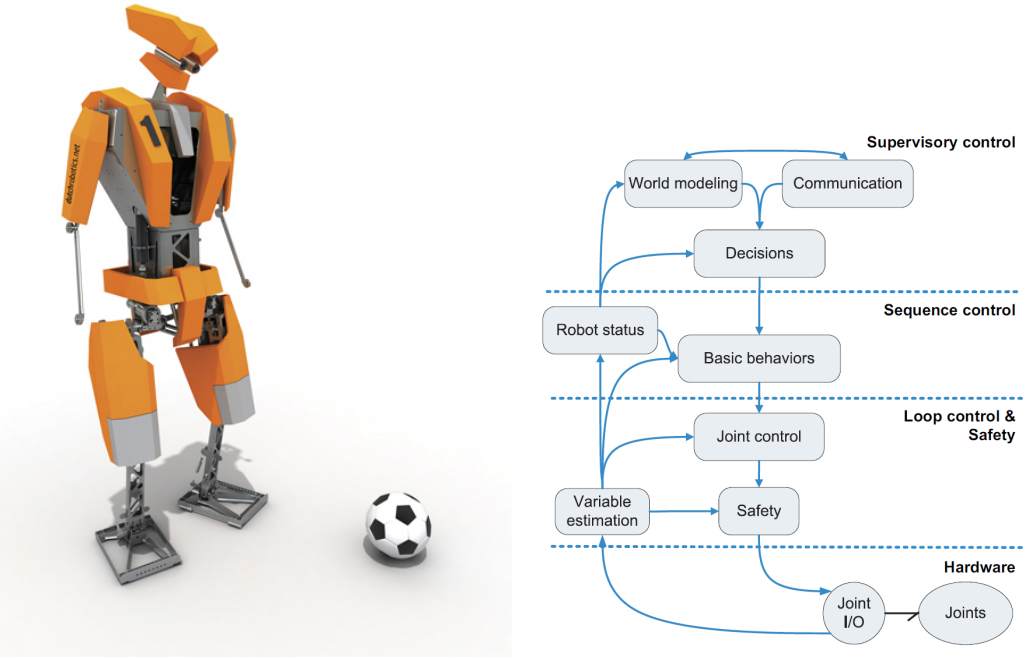

Figure A.6: TUlip - robot (a) and software design (b)

A.4.1 Characteristics

The TUlip software implementation has been split in multiple **loosely coupled modules**, see Figure A.6(b). These **modules are distributed over multiple layers, with different real-time constraints**. The **real-time constraints range from non to hard real-time**. Furthermore, the modules exchange data between each other, which might include **inter real-time communication** due to modules being in different layers.

The World modelling module, among others, requires a lot of **matrix manipulations**, which are provided by an **external library**. The Joint controller module manages the controllers for the joints. These **controllers have a unified interface**, which might have **different implementations or parameters**, and furthermore the module supports **online replacement of the controllers** depending on the requested behavior. All **actuator signals pass through the safety module** for inspection, because it is the last in line to prevent the robot from damaging itself or the environment.

The robot communicates through a **link driver** with the non real-time human-machine interface running on an external computer. This interface can also **show some preselected internal variable contents**, e.g. the current world model.

The TUlip software has been tested on a Model-In-the-Loop (MIL) set-up. In order to perform these tests the software had to **communicate with the simulation environment** of 20-sim, i.e. **reroute its sensor and actuator signals**. The simulation steps were time consuming and therefore the TUlip software also required a **virtual real-time clock** to perform these tests.

B Appendix - C++ language exceptions

Exceptions are part of the C++ language definition and provide a way to react to exceptional circumstances (like run-time errors) in a program by transferring control to special functions, called handlers. For non real-time applications this method is generally accepted and widely used. On the other hand, for real-time applications the run-time exception implementation might pose some problems and this will be investigated in detail next.

```
using namespace std;
class Object;

int main () {
  try
  {
    Object a;
    throw 20;
    Object b;
  }
  catch (int e)
  {
    // handle exception
  }
  return 0;
}
```

Listing B.1: C++ exception sample code

To illustrate the following discussion an example code is given in Listing B.1. When an exception is detected or thrown inside a try-block, the program execution is redirected to the exception handler, i.e. catch-block, that matches and handles the exception. So, when $throw$ 20 is executed the program will jump to the $catch(int\ e)$ line to handle the generated exception. Multiple catch-blocks might be specified for a single try-block and try-catch blocks may be nested. Thus, an exception might traverse from a deeply nested code to the main exception handler. Or, when no exception handler has been specified it will not be caught at all, resulting in a run-time exception which will halt the entire program.

Besides redirecting execution flow, when an exception is detected, the C++ run-time must also cleanup any local variables that were created and pushed onto the program stack. Because exceptions might occur anywhere during program execution the C++ language run-time must somehow keep a list of created local variables. Very specifically, the run-time must perform the correct number of stack unwounds. So, when $throw$ 20 is executed the run-time must only clean object *a*, otherwise another exception might be created in the form of a null-pointer for object *b*.

A side effect, due to exception (handling) semantics, is that the exception must first be resolved before the thread/program may continue. So, when exception handlers take a long time to resolve the exception at hand the rest of the thread/program cannot continue. Whenever there is a dependency between threads this might result in catastrophic consequences as the complete program may be stalled.

Unfortunately, the C++ standard does not specify how exceptions should be implemented nor a lower bound for performance. Different compiler vendors can therefore choose to implement it differently, which might cause C++ code compiled with different compilers to behave differently (Meyers, 2005a).

Discussion
The need for a mechanism to handle exceptional cases is clear. However, C++ language exceptions do not seem the proper solution. First, exceptional handling is not deterministic in time, because exceptions can be caught and emitted anywhere in the program and must then traverse to a handler somewhere up in the source tree. Additionally, clean up has to be done, depending on the state of the program, which again is not deterministic. Secondly, because exceptions might traverse outside the control/sequence loop as a handler is sought, they are considered unsafe if not handled properly. Last, the nature of exception handling might break real-time constraints due to handling the exception before any other code can progress.

An alternative is to program by specification. The documentation of the functions specifies what meaning the return values have. So, return values can be used to indicate if (and what kind of) a failure occurred. This also has its downsides. First, code specification is an elaborate task and faults might be made. Secondly, the executable size will grow due to its increased code size. To clarify, the paths the computer should take through the code have been programmed explicitly and therefore must be included into the compiled program. Whereas for exceptions this was handled dynamically by the C++ run-time engine. But, the biggest advantage over C++ exceptions is that they can be programmed deterministically.

Thus for real-time applications, which require deterministic timing, the use of C++ language exceptions is not a good idea and the new framework will therefore only use programming by specification. Perhaps, one would consider supporting both C++ exceptions and the latter as an option. But from a practical perspective this is not possible without using macros and other tricks extensively and therefore not further considered.

C Appendix - Atomic component

The Atomic framework component currently provides only an atomic integer, with a few different set and test operations. The atomicity discussed in this text is targeted at a multi-threaded and multi-core environment.

The implementation of an atomic type has three major (intertwined) restrictions:

A **Ordered atomics** - The order of atomic writes (and reads) will in most cases define the semantics of the program, but the compiler (and processor) cannot easily deduce this from the code.

B **Semantic-free variables for "unusual" memory semantics** - Although variables may be of a certain type, they need not act like them. Furthermore, their value might be updated from an arbitrary position in the code (e.g. due to an interrupt the main()-method variable error might be set), therefore the processor architecture should take care that values are not cached (at least not such that multiple versions exist in the system).

C **Unoptimizable variables and (not) optimization** - In ordinary code reads (and sometimes writes) may be moved due to optimization. Due to A) and B) the program might loose its semantics because of this, therefore this should be prohibited. For example, in a while(condition) reading a 'condition' variable, might be moved out of the loop and replaced by a while(true) for optimization.

For a complete explanation, see Sutter (2009).

Thus, compiler and architecture support is needed to guarantee the above restrictions. Currently, the C++0x specification is being finalized, but no compilers fully support the proposed atomic<T> template yet.

A few alternatives can be thought of:

1. Volatile inline assembly instruction insertion - with the proper settings it can achieve atomicity.
2. GCC intrinsics - actually the same as the above option, but platform independent is arranged by the compiler.
3. Locks - using OS locks the atomic updates of variables can be guaranteed. Though, under water the OS uses similar instructions as mentioned above.

So, except for the lock based alternative, it is impossible to implement an atomic variable that is larger than the maximum instruction data size. For instance, on a 32-bit platform atmost a 64-bit variable can be created. Therefore, it was chosen to only implement an atomic integer for LUNA.

For the current $AtomicInt$ all three alternatives have been implemented and tested:

1. Uses the same asm code as other projects (e.g. OROCOS and openpa).
2. Translated the assembly code (from 1) into similar GCC intrinsics. The GCC intrinsics assembler is not the same as the original assembly, but using an object dump it was proved to be a correct implementation.
3. OSMutex based locked version of the assembly code, which is much slower than the other alternatives.

D Appendix - Timers and counters

Table D.1 provides an overview of currently available timers and counters on most PC architectures.

D.1 Time Stamp Counter

Since the Time Stamp Counter (TSC) is applied in Section 5.3.5 a few technical side notes on the use of the TSC are appropriate.

The TSC is a 64-bit register which counts the number of processor clock ticks since reset. Since it uses the processor clock rate directly it is an excellent high-resolution and low-overhead way of measuring computation time. However, it potentially has issues, which will be discussed next.

The assembler instruction *RDTSC* reads the TSC. With the advent of multi-core/hyper-threaded CPUs, systems with multiple CPUs, 'hibernating' operating systems and frequency scaling, the TSC cannot be relied on to provide accurate results without special considerations. Firstly, the counters of multiple CPU cores might not be synchronised correctly. Secondly, the CPU speed may change due to power-saving measures taken by the OS or BIOS. Recent processors (P4+) fix this by using a constant rate TSC. Thirdly, starting with the Pentium Pro, Intel processors have supported out-of-order execution. Instructions are not necessarily performed in the order they appear in the executable. These reasons can cause RDTSC instruction to be executed earlier or later than expected, producing a misleading cycle count.

These problems can be solved by executing a serializing instruction, such as CPUID. The serializing instruction will force every preceding instruction to complete before allowing the program to continue. Newer hardware architectures (Core i7+) include the RDTSCP instruction, which is a serializing variant of the RDTSC instruction.

Name	**Features**	**Supported architectures**	**Frequency**	**Counter bit width**	**Operating characteristics**
Programmable Interval Timer (PIT) - 8253/8254	Timers and counter	IBM PC compatibles, but may be integrated with APIC.	1.193182 MHz	16 bits	3 Timers; timer 0 (IRQ0) is typically running on 18 Hz
Real Time Clock (RTC)	Timer only	IBM PC compatibles	Operating frequencies between 2 - 8192 Hz (power of 2)	-	Update/periodic/alarm functionality (on IRQ 8)
High Precision Event Timer (HPET)	Timers and counter	Incorporated in PC chipsets since circa 2005	10 MHz+	64 bits	3x 64 bits + 29x 32bits comparators per HPET chip
(Local) Advanced Programmable Interrupt Controllers (APIC)	Per CPU timer	Pentium Pro+	Operating frequencies between 1 Hz - system bus frequency	-	224 interrupt vectors
Time Stamp Counter (TSC)	Counter only	Pentium+	Counting on the processor frequency	64 bits	See Section D.1.

Table D.1: Timers and counters commonly available on must PC architectures

E Appendix - Threading use cases

The example given in Listing E.1 shows the specification of a periodic real-time OS thread and an aperiodic non real-time OS thread. Note how easy both thread types can be used next to each other in LUNA. Listing E.2 provides an example for the implementation of a *Runnable* with an *IThreadBlocker*. If no thread blocker can be created the emergency-manager component will shut down the application before any harm can be done.

```
#include "OSThread.h"
#include "RTOSThread.h"

using namespace LUNA::Threading;

void main(void)
{
  // Create the Runnable
  Runnable* rt_example = new Example1();

  // Take the RT OSThread default attributes
  Thread_attr rt_attr = RTOSThread_defaults;

  // Set a 1 millisecond period for 1 kHz
  rt_attr.period.seconds = 0;
  rt_attr.period.nanoseconds = 1000000;

  // Instantiate the periodic RTOSThread
  IThread* rt_thread = new RTOSThread(rt_example, rt_attr);

  // Create the Runnable
  Runnable* nrt_example = new Example1();

  // Instantiate the OSThread
  IThread* nrt_thread = new OSThread(nrt_example); // Will take the
      default OSThread attributes

  // Start both threads
  rt_thread->start();
  nrt_thread->start();
  // Both threads will run now.

  // Wait for both threads to finish.
  rt_thread->join(NULL);
  nrt_thread->join(NULL);
}
```

Listing E.1: Specification of a periodic real-time OS thread and an aperiodic non real-time OS thread

```
#include "Runnable.h"
#include "IThreadBlocker.h"
#include "ThreadBlockerFactory.h"
#include "EmergencyManager.h"

using namespace LUNA::Threading;
using namespace LUNA;

class Example1 : public Runnable
{
  public:
    Example1()
    {
      // The backing thread type is still unknown, so the IThreadBlocker
          cannot be instantiated here.
    }

    // Inherited from Runnable
    void preRun()
    {
      // Create the IThreadBlocker belonging to the current thread type.
      m_activate = ThreadBlockerFactory::Instance()->CreateObject(this->
          getThreadType());

      // If it could not be created => emergency shutdown!
      if(m_activate == NULL)
      {
        log(LOG_ERROR, "ThreadBlocker factory failed.\n");
        EmergencyManager::Instance()->emergencyShutdown();
      }
    }

    // Mandatory inherited function from Runnable
    void* run()
    {
      // Do example things.
    }

    // Inherited from Runnable
    void postRun()
    {
      // Cleanup the created IThreadBlocker
      delete m_activate;
    }

  private:
    IThreadBlocker* m_activate;
};
```

Listing E.2: Runnable implementation with an *IThreadBlocker*

F Appendix - CSP Threading use case

This example shows that CSP processes can easily be distributed over different thread types. The compositional CSP tree belonging to the sample code is shown in Figure F.1.

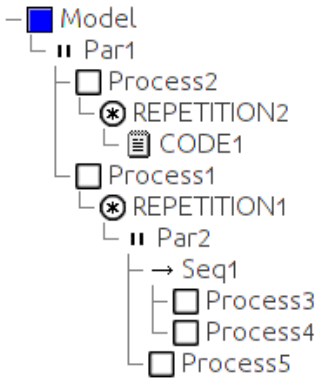

Figure F.1: Example gCSP compositional tree for LUNA deployment

The hypothetical target has two cores available and therefore it is desirable to run *Process1* and *Process2* fully in parallel. The code to achieve this is shown in Listing F.1. A few improvements can still be made, for instance, it would be beneficial to be able to search for a specific CSP process by name in the full (*Model*) tree. Currently a pointer is required (e.g. *process1* in the sample code) to collect the UThread set.

```
#include "Model.h"

void main(void)
{
        // Create the Model
        ...
        CSPConstruct* process1 = new Process1(...);
        CSPConstruct* process2 = new Process2(...);
        Model* model = new Model(new Parallel(p1, p2, NULL));

        // print the CSP tree for inspection
        printCSPTree(thread1);

        //Collect all CSP constructs and filter for the true CSP
            processes, since only these need threads
        Set<CSProcess*> full1 = filterCSProcesses( collectValues(
            TreeNode<CSPConstruct*>::depth_first_iterator(model) ) );

        //Collect all CSP processes for the first thread
        Set<CSProcess*> thread1 = filterCSProcesses( collectValues(
            TreeNode<CSPConstruct*>::depth_first_iterator(process1) ) );

        //Create the set of UThreads for the second thread
        Set<CSProcess*> thread2 = full1 - thread1;

        // Create both UThreadContainers
        UThreadContainer* thread_container1;
        createThreads(&thread1, &thread_container1);
        UThreadContainer* thread_container2;
        createThreads(&thread2, &thread_container2);

        //Put both containers into a set
        Set<LUNA::Runnable*> os_runn(2, thread_container1,
            thread_container2);

        //Create the OSThread objects
        Set<OSThread*> os_threads(2);
        createThreads(&os_runn, &os_threads);

        //Start the threads
        startThreads(&os_threads);

        //Wait for the threads to finish, which in this case will be
            never.
        joinThreads(&os_threads);
}
```

Listing F.1: LUNA deployment belonging to Figure F.1

G Appendix - QNX AnyIO driver

The Mesa electronics Anything I/O card (Mesa electronics, 2011) is build around a FPGA. The FPGA can be programmed for different purposes giving it a true Anything I/O nature. There are three different versions of the card all being used at the Control engineering laboratory, namely the Anything I/O 4I65, 4I68 and 5I20.

The current Anything I/O board driver has been developed in stages by various students: Groothuis (2004); Buit (2005); Molanus (2007). In Molanus (2008) the first QNX driver was written, unfortunately he did not implement this as a standalone driver.

During this thesis project the driver was rebuild by reusing certain parts of the first QNX driver, the Linux driver and following the QNX philosophy. The QNX resource managers (QNX, 2010), commonly known as drivers, do not require any special arrangements with the kernel and therefore the Linux driver must be changed accordingly. The resulting code has been merged with existing AnyIO code in the CE svn repository.

The standalone driver can now program the FPGA, erase it and read/write the FPGA registers/memory via direct or memory mapped IO.

H Appendix - RTLogger examples

Section 5.2.12 describes the RTLogger component design in LUNA. Section H.1 explains how to record information with this component. Visualizing CSP trace information is discussed in Section H.2 and visualization signals with 20-sim is shown in Section H.3.

H.1 Recording RTLogger information

The RTLogger component is enabled via *make menuconfig* (*Debug->RTLogger*). In the *configuration* the fixed and variable buffer sizes, transfer ratio variable/fixed buffer and the optional ring buffer behavior can be adjusted. These options and the available computing resources determine if and when data is transferred. This should be kept in mind for the 20-sim visualization especially, since the transfer may lag behind or show nothing at all if computing resources are scarce.

Note, the RTLogger and recorder are designed as a proof of concept.

H.1.1 Application side

The script in Listing H.1 can be used to start and stop the RTLogger in a (real-time) application.

```
#include "RTLogger.h"

// First argument is hostname, Second is the port
int main(int argc, char *argv[])
{
  using namespace LUNA::Logger;

  bool using_remote_logging = false;

  // If the argument count is incorrect, abort.
  if(argc != 3 && argc != 1)
  {
    log(LOG_ERROR, "Please use %s <hostname> <port>\n", argv[0]);
    return 0;
  }
  else
  {
    // If a hostname + port is specified => start the remote logger.
    if(argc == 3)
    {
      log_local(LUNA::LOG_INFO, "Using RTLogger.\n");
      using_remote_logging = true;
      if(!STARTREMOTELOGGER(argv[1], atoi(argv[2]), false))
      {
        log_local(LUNA::LOG_ERROR, "Could not start the remote logger.\n"
            );
        exit(EXIT_FAILURE);
      }
    }
    // Sink the data, without transferring or showing it.
    else
    {
```

```
      log_local(LUNA::LOG_INFO, "No address + port specified, using local
          logger.\n");
    }
  }

  // Do things

  if(using_remote_logging)
  {
    STOPREMOTELOGGER();
  }
}
```

Listing H.1: RTLogger start and shutdown example code

H.1.2 Recorder side

The recorder is a console application and it will show any incoming log messages on the console. Signal values are saved in a trace file with CSV format.

Type the following command on the console in Linux to start the recorder:
RTLOGGER_Server <TRACE CSV FILE> <TRACE LOG FILE> <PORT>

H.2 Visualizing CSP traces

The RTLogger component (Section 5.2.12) has an option to record CSP process execution while the CSP model is controlling a real setup, generating a so called trace. However, correct traces are only generated for CSP models realized with less or equal OS threads than cores/CPUs (Section 5.2.12).

The CSP constructs (*Seq*, *Par*, *PriPar* and *Rec*) emit an event when they are activating other CSP processes, blocked and when they are done. The CSP processes (*CSProcess*, *Reader*, *Writer*, *Alt* and *PriAlt*) emit an event when they are activated, running, blocked and done. In all of the above cases the done event is only emitted when all of its children have finished.

H.2.1 Recording

To record CSP trace information, the respective build system option (*Execution Engines->CSP->CSP forensics*) must be enabled. Events will now be transferred to the recorder, whenever computing resources are available.

H.2.2 Visualizing

The visualization of trace data is done with Matlab and a few custom scripts. These scripts, used to post process the CSP trace, can be found in the RTLogger component directory in LUNA. Unfortunately, correct visualization is currently only possible for single core systems with one OS thread and multiple UThreads. This section first explains the trace figures in general and then shows how such figures can be created with the scripts.

The Jiwy setup, presented in Appendix A.1, is used as a test and validation setup in this report. The complete Jiwy CSP model contains 51 processes and its visualization is much to large to show here. Therefore, the figures in this section only visualize the *Pan* sub-model of the Jiwy CSP model. The CSP compositional relationships for the *Pan* sub-model are shown in Figure H.1. Note, the process names are also used in the other figures.

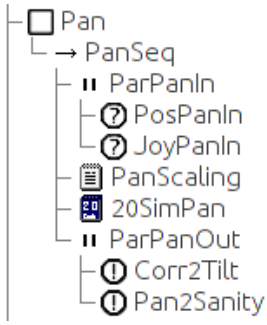

Figure H.1: Jiwy gCSP Pan submodel

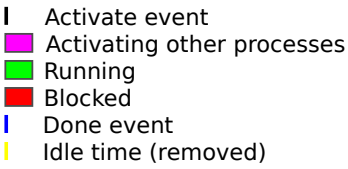

Figure H.2: CSP tracing legend

Figure H.3 shows for one CSP model iteration the UThread context switches and the *TimerWriter* process. The legend for the trace figures is shown in Figure H.2. The *TimerWriter* takes care of stalling the complete CSP model until the next period starts, since it is designed to operate at 100 Hz. It can be seen that the computation of the complete Jiwy model takes just under 1 ms, which is longer than stated in the LUNA paper (Section 5.4.1). But, the computation time in this case includes generating the CSP events and placing (logging) these in the RTLogger buffers.

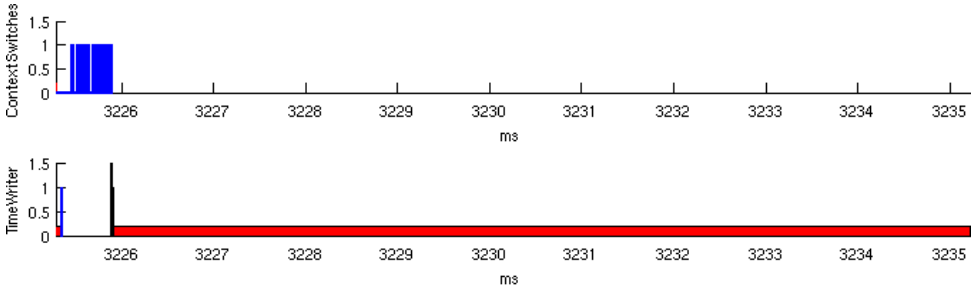

Figure H.3: One period of the Jiwy CSP model, showing context switches and the *TimerWriter* CSP trace.

Visualizing the complete period of a Jiwy CSP model cycle makes it hard to read. So, the idle time is filtered out and the same period is shown in Figure H.4. Now, single context switches and the *TimeWriter* running time can be identified. Only the blue events indicate a context switch, the smaller red bars are an artifact of using the same visualization as for CSP traces.

Figure H.5 shows the *Pan* sub-model trace. Constructs (*Pan*, *ParPanIn* and *ParPanOut*) are activated and during this activation activate at least one other CSP process or construct depending on its CSP semantics. For instance, the *ParPanIn* activates the *PosPanIn* and *JoyPanIn* processes. Processes are only activated, i.e. placed on the ready queue, and then the scheduler decides when it is actually run. Note, the gap between *PanScaling* activation and actually running is due to other CSP processes, which are not shown, being run first. When a process or construct is done it emits a done event.

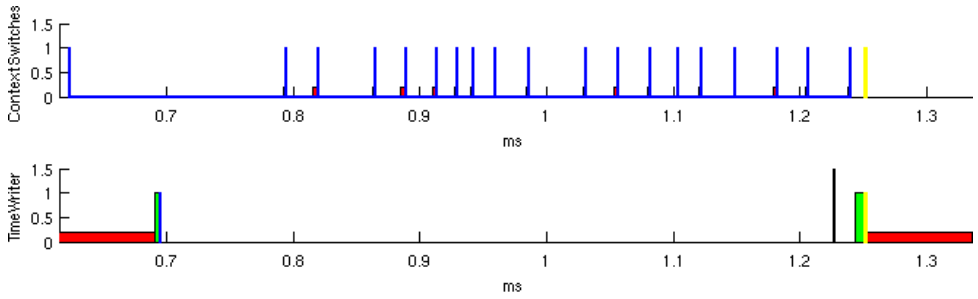

Figure H.4: The same period as shown in Figure H.3, but now with idle time filtering.

The procedure to create such figures from a trace CSV file is explained next. For most steps also example Matlab commands are given. When these are executed on the *CSP_visualization.csv* file (accompanying this report) they create exactly the same figures as mentioned before.

1. Start Matlab
2. Open the generated CSV file and remove the lines until the number of columns becomes steady. These may change during running-time, due to dynamically declaring them, but Matlab's import feature cannot handle this.
3. Import the CSV file into Matlab's workspace with: *File->Import data*.
   Name the variable for instance *trace*.
4. Open the LOG file belonging to this CSV file and copy the "names = { . . . }" line to Matlab. These are the column names.
5. Remove all non-CSP signals, i.e. the columns, except the time.
   For example: *trace = trace(:,[1:14,17:45]); names = names([1:13, 16:44]);*.
6. Select the interesting part of the CSP trace, i.e. the rows. If the trace is not downsized, the scripts will take a very long time to process and the visualization will get confusing.
   For example: *trace2 = trace(2000:3000, :)*.
7. *Optionally*, check the integrity of the CSP trace with the *csp_integrity* script. This verifies that only one process is running at the same time.
   For example: *csp_integrity(trace2, names)*.
8. *Optionally*, remove idle time from the plots with the *csp_remove_idle_time* script. The script will output a filtered trace and the indexes of idle times.
   For example: *[trace3, idle] = csp_filter_idle(trace2, 5)*, whereby the second argument is the minimum idle time in ms.
9. Select the interesting CSP processes, i.e. columns, and put them in the correct order for printing. For example: *print = [15,3,4,24:27,5,28,29];*.
10. Plot the CSP trace with the *csp_plot* script.
    For example: *csp_plot(trace3(148:300,:), print, names)*.

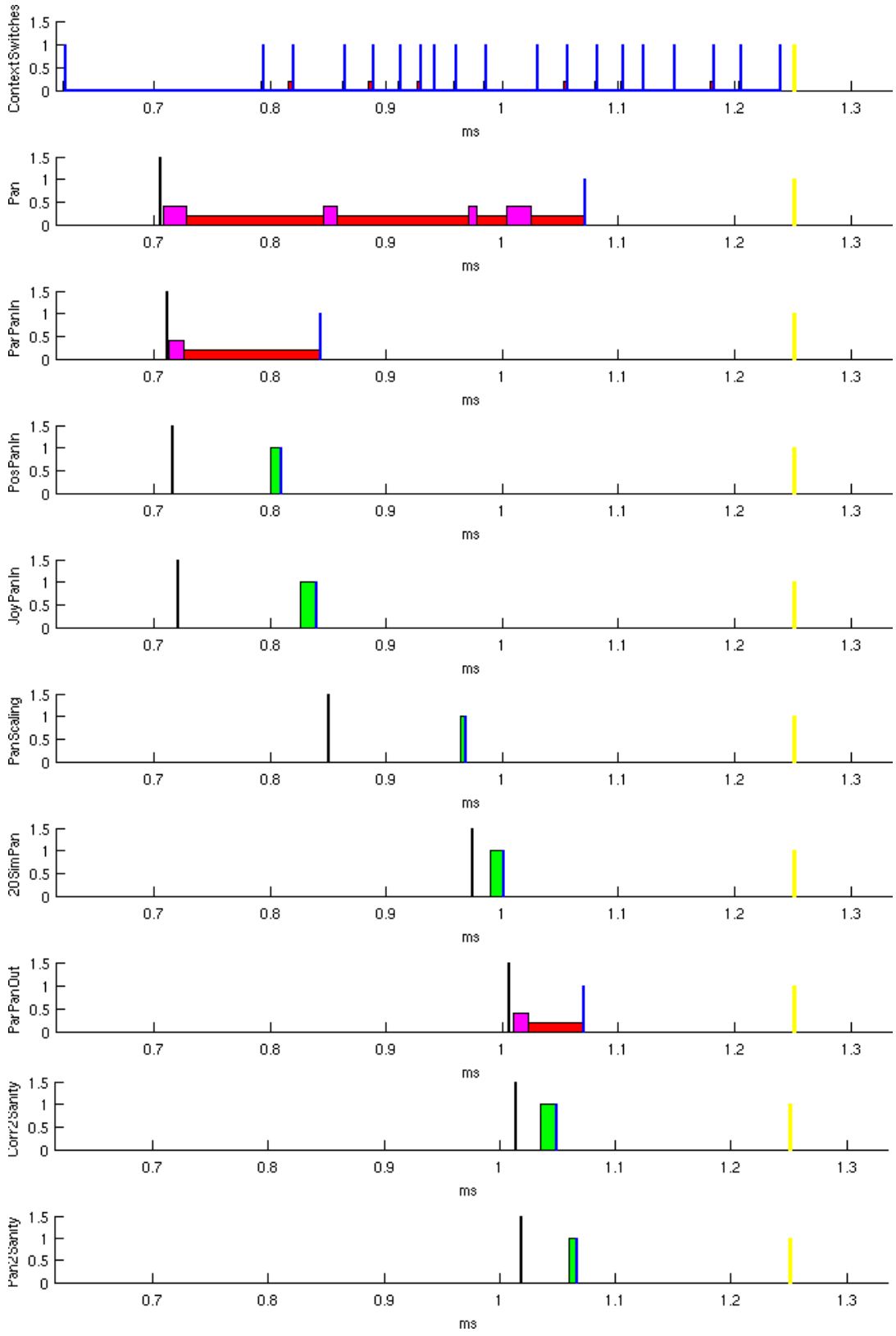

Figure H.5: Pan trace 1

H.3 Visualizing (control) signals with 20-sim

The RTLogger also has an option to log integer or floating point signals aperiodically and show these through the 20-sim experiment viewer. A logging statement does not influence real-time constraint levels, but adds some computation time.

The 20-sim to RTLogger recorder interface is developed for 20-sim running on Linux and is considered a proof of concept.

H.3.1 Recording

The RTLogger application should be set to ring-buffer behavior and only contain a small buffer, otherwise old data might be transferred on occasion when computing resources were temporarily insufficient.

Example code for the application is shown in Listing H.2.

```
// When the RTLogger is disabled in the menuconfig, the RTLOG* macros
   have no effect anymore. Note, the code does not need any adaption to
   remain valid.
#include "RTLogger.h"

class MyClass : public Runnable {
  public:
    void preRun()
    {
      // Signal types need to be pre-registered, so that type information
           is known.
      RTLOGGER_REGISTER(rt_signal, RTLOG_INT32, "Signal Name");
    }

    void* run()
    {
      // Logs the current Var value.
      RTLOG(rt_signal, Var);
    }

  private:
    int Var;
    RTLOGGER_ID(rt_signal); // The rt_signal is the channel id.
}
```

Listing H.2: RTLogger start and shutdown example code

H.3.2 Visualizing

An example run of the RTLogger and 20-sim visualization for the Pan sub-model in Figure H.1 is shown in Figure H.6. The figure shows the pan controller error, joystick input, position encoder input and the calculated PWM setpoint.

Between 3 and 8 seconds the joystick was moved in a sine-like motion. The 'Joystick Pan In' signal shows that the Jiwy setup almost immediately tries to follow this motion, but fails to exactly follow it since the output control signal is limited for safety reasons.

Between 14 and 18 seconds it was tried to keep Jiwy on a stable position other than the origin. But, the joystick created a lot of noise (see the 'Joystick Pan In' signal) and the controller followed this reference input, such that the setup was shaking a little bit. So, the 20-sim visualization is quite useful to determine the cause of some problems while manipulating the robotic setup.

The following steps need to be performed to create a similar plot:

1. Start the RTLogger recorder (see Section H.1)
2. Start 20-sim
3. Open the *Logger.emx* experiment
4. Click on the simulator and press start
5. Add signal names by hand in the 20-sim experiment, since these cannot be imported into 20-sim automatically.

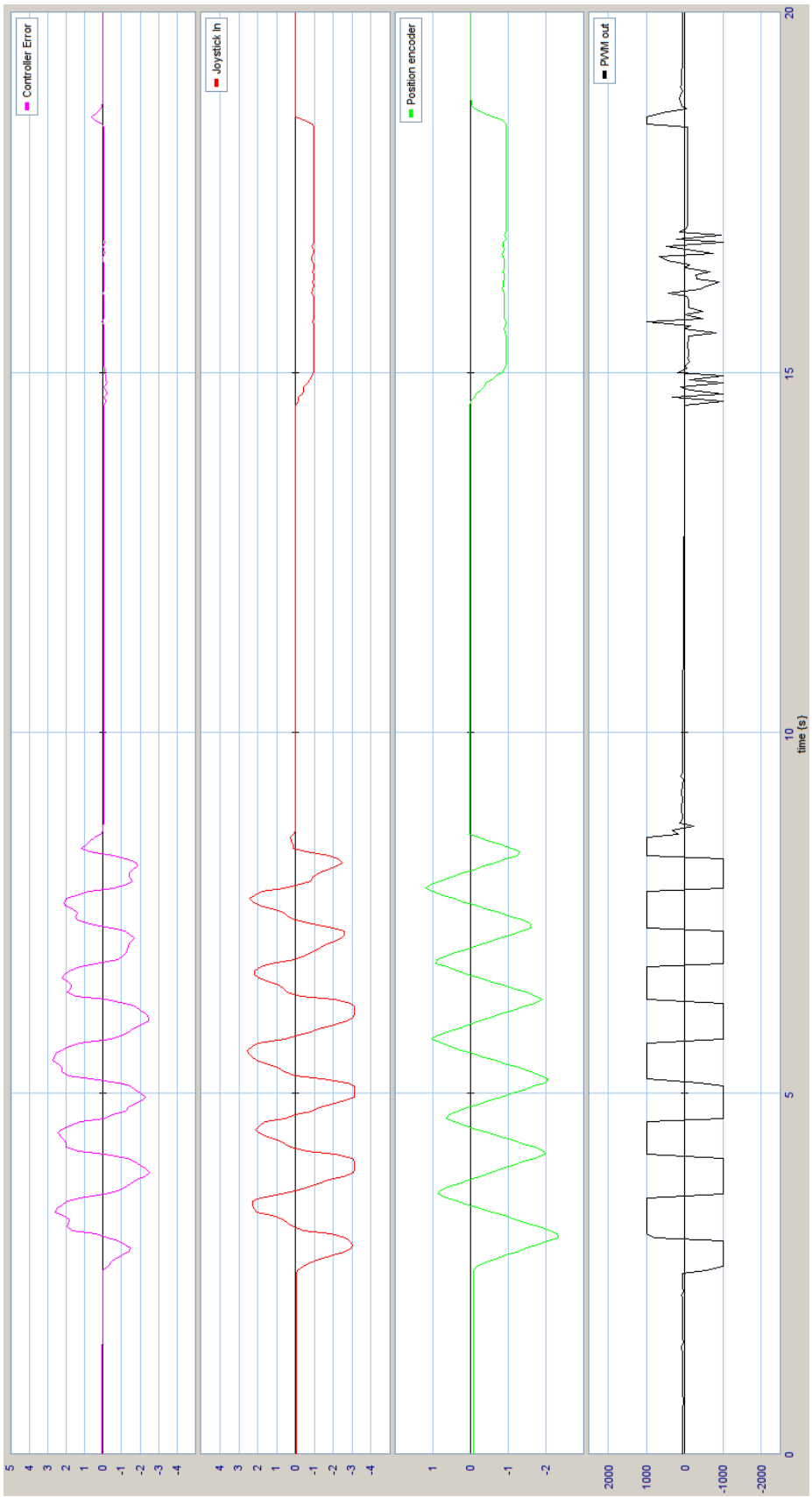

Figure H.6: 20-Sim Pan control signals

I Appendix - LUNA CPA 2011 conference paper

Communicating Process Architectures 2011
P.H. Welch et al. (Eds.)
IOS Press, 2011

LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework

M. M. BEZEMER, R. J. W. WILTERDINK and J. F. BROENINK

Control Engineering, Faculty EEMCS, University of Twente,
P.O. Box 217 7500 AE Enschede, The Netherlands.

{M.M.Bezemer , J.F.Broenink} @utwente.nl

Abstract. Modern embedded systems have multiple cores available. The CTC++ library is not able to make use of these cores, so a new framework is required to control the robotic setups in our lab. This paper first looks into the available frameworks and compares them to the requirements for controlling the setups. It is concluded that none of the available frameworks meet the requirements, so a new framework is developed, called LUNA.

The LUNA architecture is component based, resulting in a modular structure. The core components take care of the platform related issues. For each supported platform these components have a different implementation, effectively providing a platform abstraction layer. High-level components take care of platform-independent tasks, using the core components. And the execution engine components implement the algorithms taking care of the execution flow, like a CSP implementation. Next, the paper describes some interesting architectural challenges encountered during the LUNA development and their solutions.

The paper concludes with a comparison between LUNA, C++CSP2 and CTC++. It shows that LUNA is more efficient than CTC++ and C++CSP2 with respect to switching between threads. Also running a benchmark using CSP constructs shows that LUNA is more efficient compared to the other two. Furthermore, LUNA is also capable of controlling actual robotic setups with good timing properties.

Keywords. csp, framework architecture, hard real-time, performance comparison, rendez-vous communication, scheduling, threading

Introduction

Context

Nowadays, many embedded systems have multiple cores at their disposal. In order to be able to run more challenging (control) algorithms, embedded control software should be able to make use of these extra cores. Developing complex concurrent software tends to become tedious and error-prone. CSP [1] can ease such a task. Especially in combination with a graphical modeling tool [2], designing such complex system becomes easier and the tool could help in reusing earlier developed models. CTC++ [3] is a CSP based library, providing a hard real-time execution framework for CSP based applications.

When controlling robotic setups, real-time is an important property. There are two levels of real-time: hard real-time and soft real-time. According to Kopetz [4]: "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*".

Figure 1 shows the layered design, used in our Control Engineering group, for embedded software applications connected to actual hardware. Each layer supports a type of real-time,

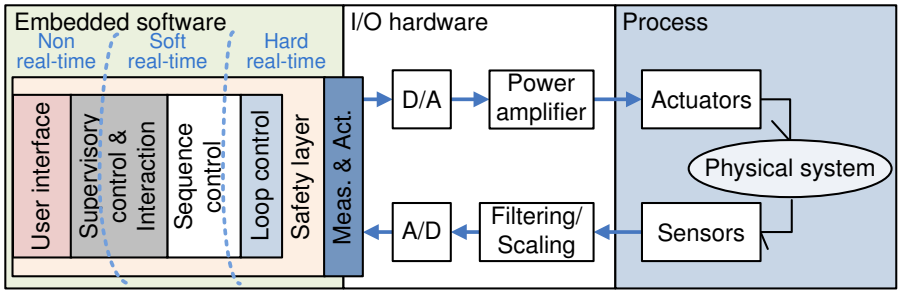

Figure 1. Software architecture for embedded systems [5].

varying from non real-time to hard real-time. The 'Loop control' is the part of the application responsible for controlling the physical system and it is realised in a hard real-time layer. The hard real-time layer has strict timing properties, guaranteeing that given deadlines are always met. If this for whatever reason fails, the system is considered unsafe and catastrophic accidents might happen with the physical system or its surroundings due to moving parts. The soft real-time layer tries to meet its deadlines, without giving any hard guarantees. If the design is correct nothing serious should happen in case such a deadline is not met. This layer can be used for those parts of the application which are more complex and require more time to run its tasks, like algorithms which map the environment, plan future tasks of the physical system or communicate with other systems. The non real-time layer does not try to meet any deadlines, but provides means for long running tasks or for an user interface. The left-over resources of the system are used for these tasks, without giving any guarantees of the availability of them.

Robotic and mechatronic setups like the ones in our lab require a hard real-time layer, since it is undesirable for the actual setups to go haywire. The use of Model Driven Development (MDD) tools makes developing for complex setups a less complex and more maintainable task [6]. For the multi-core or multi-CPU embedded platforms, we would like to make use of these extra resources. Unfortunately, the CTC++ library, as it is, is not suitable for these platforms, as it can only use one core or CPU. This paper evaluates possibilities to overcome this problem.

The requirements for a suitable framework that can be used for robotic and mechatronic setups are:

- *Hard real-time*. This incorporates that the resulting application needs to be deterministic, so it is possible to guarantee that deadlines are always met. The framework should provide a layered approach for such hard real-time systems (see Figure 1).
- *Multi-platform*. The setups have different kind of hardware platforms to run on, like PowerPC, ARM or x86 processors. Also different operating systems should be supported by the framework.
- *Thread support*. In order to take advantage of multi-core or multi-CPU capable target systems.
- *Scalability*. All kind of setups should be controlled: From the big robotic humanoids in our lab to small embedded platforms with limited computer resources.
- *CSP execution engine*. Although, it should *not* force the use of CSP constructs when the developer does not want it, as this might result in not using the framework at all.
- *Development time*. The framework should decrease the development time for complex concurrent software.
- *Debugging and tracing*. Provide good debugging and tracing functionality, so developed applications using the framework can be debugged easily and during development unexpected behaviour of the framework can be detected and corrected. Real-time logging functionalities could preserve the debug output for later inspection.

The CTC++ library meets most requirements, however as mentioned before, it does not have thread support for multi-core target systems. 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. This is an obstacle to use the library from a generic robotics point of view and results in ignoring the CTC++ library altogether, as is experienced in our lab. A future framework should prevent this tight integration. By adding a good MDD tool to the toolchain, the robotic oriented people can gradually get used to CSP.

It might seem logical to perform a major update to CTC++. But unfortunately the architecture and structure of the library became outdated over the years, making it virtually impossible to make such major changes to it. So other solutions need to be found to solve our needs.

Existing Solutions

This section describes other frameworks, which could replace the CTC++ library. For each framework the list with requirements is discussed to get an idea of the usability of the framework.

A good candidate is the C++CSP2 library [7] 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 a 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 unsafe for usage in hard real-time setups. Also as 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. And last, it makes use of features which are not commonly available on embedded systems. On such systems it is common practice to use the microcontroller C library (uClibc) [8], in which only commonly used functionality of the regular C library is included. Most notably, one of the functionalities which is not commonly included in uClibc is *Thread Local Storage*, but is used by C++CSP2.

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 [9]. Although, there is a new Java virtual machine, called JamaicaVM [10], which claims to be hard real-time and supporting 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.

Besides these specific CSP frameworks, there are also non-CSP-based frameworks, which might be used to add a CSP layer to. OROCOS [11] and ROS [12] are two of these frameworks and both claim to be real-time. But both will not be able to run hard real-time 1KHz control loops on embedded targets which are low on resources. Their claim about being real-time is probably true when using dedicated hardware for the control loops, which are fed by the framework with ‘setpoints’. Basically, the framework is operating at a soft real-time level, since it does not matter if a setpoint arrives slightly late at the hardware control loop. In our group we like to design the control loops ourselves and are not using such hardware control loop solutions. Furthermore, it is impossible to use formal methods to confirm that a complex application, using one of these frameworks, is deadlock or livelock free, because of the size and complexity of these frameworks [13].

Based on the research performed on these frameworks, we have decided to start over and implement a completely new framework. Available libraries, especially the CTC++ and C++CSP2 libraries, are helpful for certain constructs, ideas and solutions. The new framework can reuse these useful and sophisticated parts, to prevent redundant work and knowl-

edge being thrown away. After implementing the mentioned requirements, it should be able to keep up with our future expansion ideas.

Outline

The next section describes the general idea behind the new framework, threading, the CSP approach, channels and alternative functionality. Section 2 compares the framework with the other related CSP frameworks mentioned earlier, for some timing tests and when actually controlling real setups. In the next section, the conclusions about the new framework are presented. And the last section discusses future work and possibilities.

1. LUNA Architecture

The new framework is called LUNA, which stands for 'LUNA is a Universal Networking Architecture'. A (new) graphical design and code generation tool, like gCSP [14], is also planned, tailored to be compatible with the LUNA. This MDD tool will be called Twente Embedded Real-time Robotic Application (TERRA). It is going to take care of model optimisations and by result generating more efficient code, in order to reduce the complexity and needs of optimisations in LUNA itself.

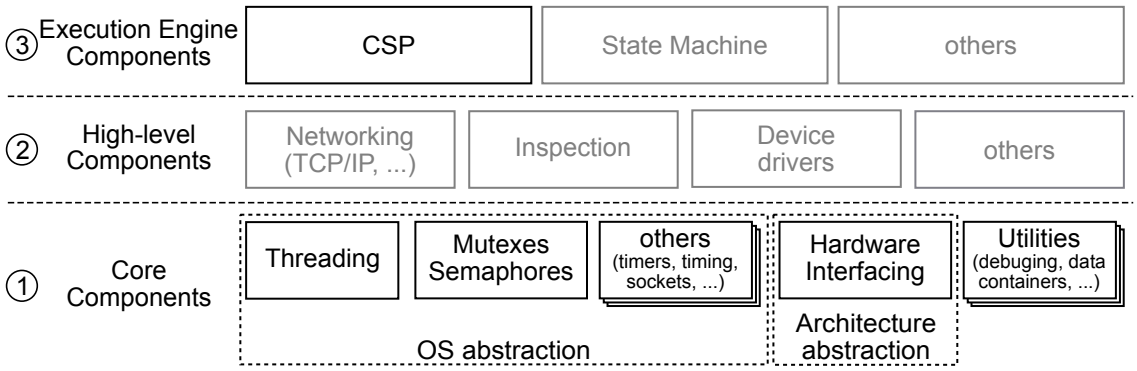

Figure 2. Overview of the LUNA architecture.

LUNA is a component based framework that supports multiple target platforms, currently planned are QNX, RTAI and Xenomai. To make development more straightforward, Linux and Windows will also be supported as additional platforms. Figure 2 shows the overview of the LUNA components and the levels they are on. The gray 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 are available to support the target operating system (OS), like threading, mutexes, timers and timing. The architecture abstraction components provide support for features specific to an architecture (or hardware platform), like the support for (digital) input and output (I/O) possibilities. Other components can make use of these core components to make use of platform specific features without knowledge of the actual chosen platform. Another group of core components are the utility components, implementing features like debugging, generic interfaces and data containers.

The next level contains the *High-level Components* (2). These are platform independent by implementing functionality using the core components. An example is the Networking component, providing networking functionality and protocols. This typically uses a socket component as platform-dependent glue and build (high-level) protocols upon these sockets.

The *Execution Engine Components* (3) implement (complex) execution engines, which are used to determine the flow of the application. For example a CSP component provides constructs to have a CSP-based execution flow. The CSP component typically uses the core components for threading, mutexes and so on and it uses high-level components like networking to implement networked rendez-vous channels.

Components can be enabled or disabled in the framework depending on the type of application one would like to develop, so unused features can be turned off in order to save resources. Since building LUNA is complex due to the component based approach and the variety of supported platforms, a dedicated build system is provided. It is heavily based on the OpenWrt buildroot [15,16].

The initially supported platform is QNX [17], which is a real-time micro-kernel OS. QNX natively supports hard real-time and rendez-vous communication. This seemed ideal to start with, relieving the development load for an initial version of LUNA. As QNX is POSIX compliant, a QNX implementation of LUNA would result in supporting other POSIX compliant operating systems as well. Or, at least it would support parts of the OS which are compatible, as not many operating systems are fully POSIX compliant.

1.1. 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 User threads are light on resources, but must run in a OS thread and are thus running on the same core as the OS thread. A big advantage of using OS threads is the preemptive capabilities of these threads: Their execution can be forcefully paused anywhere during its execution, for example due to a higher priority thread becoming ready. 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. Combining resource-heavy OS threads and non preemptive capable User threads results in a hybrid solution. This allows for constructing groups of threads which can be preempted but are not too resource-heavy.

As the term already implies, the OS threads are provided and maintained by the OS. For example, the QNX implementation uses the POSIX thread implementation provided by QNX and for Windows LUNA would use the Windows Threads. Therefore, 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 [7,18], except the LUNA User threads are not run-time portable to other OS threads. There is no need for it and this will break hard real-time constraints.

Figure 3 shows the LUNA threading architecture. Two of the components levels of Figure 2 are visible, showing the separation of the threading implementation and the CSP implementation.

UThreadContainer (UTC) and OSThread are two of the available thread types, both implementing 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.

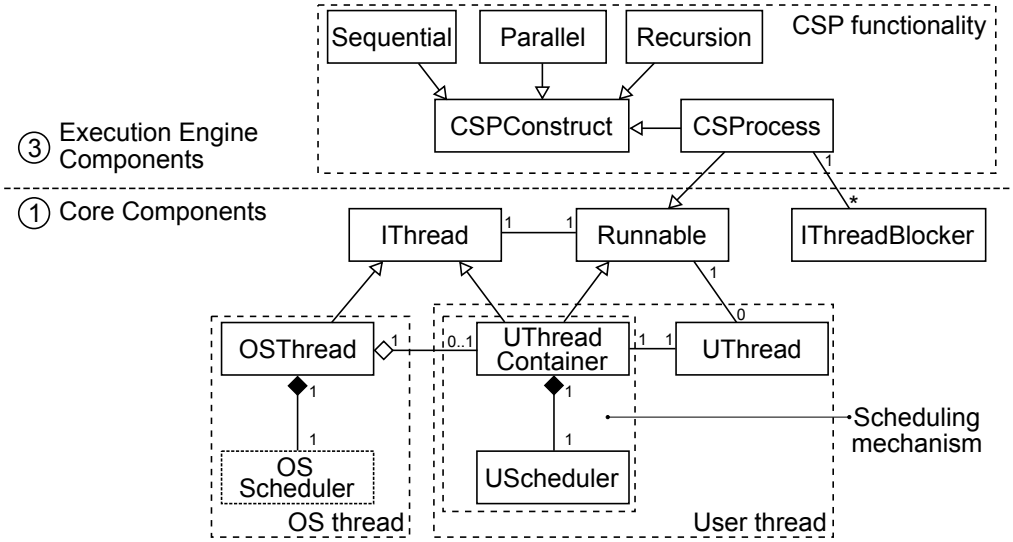

Figure 3. UML diagram of threads and their related parts.

2. the UScheduler which contains the ready and blocked queue and decides which User thread is the next to become active.

The UTC also contains a list with UThreads, which are the objects containing the 'context' of a User thread: the stack, its size and other related information. Besides this context relation data, it also contains a relation with the Runnable which should be executed on the User thread.

For the CSP functionality a 'separation of concerns' approach is taken for the CSP processes and the threads they run 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 models.

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 and CSProcess 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.

The UTC implements the Runnable interface so that 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 architectural decision, the scheduling mechanism is not running on a separate thread, but makes use of the original thread, in between the execution of two User threads.

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.

1.2. 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 a CSP-based execution engine. It is completely separated from the threading model, so it will run on any platform that has threading support within LUNA.

Each CSP process is mapped on a thread. Because of the separation of CSP and the threading model, 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 will also facilitate code generation, since code generation needs to be able to decide how to map the CSP processes on the available cores in an efficient way without being limited by thread types.

Figure 4 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, of which process **Q** is one. Process **Q** is a sequential process and also has some child processes. Process **T** is one of these child processes and it does not have any child processes of its own.

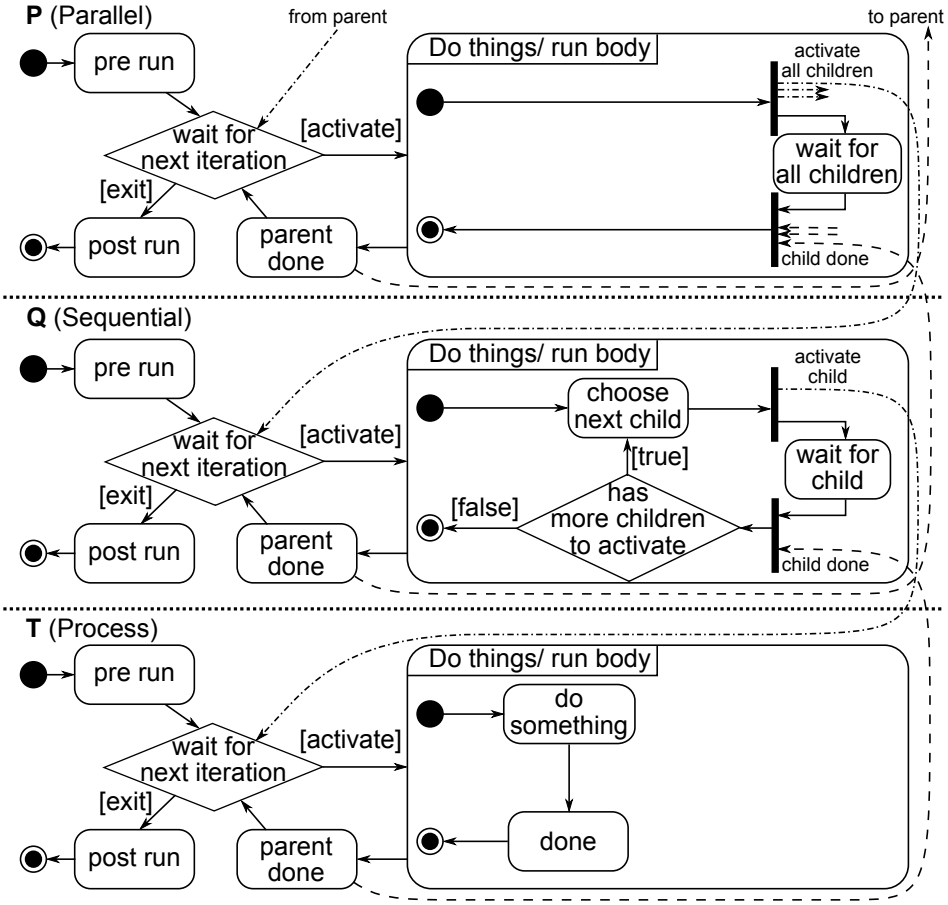

Figure 4. Flow diagram showing the conceptual execution flow of a CSProcess.

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 and next the process will wait until all children are done before signalling the parent that the process is finished. Process **Q** is only 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, because **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. So at some point 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.

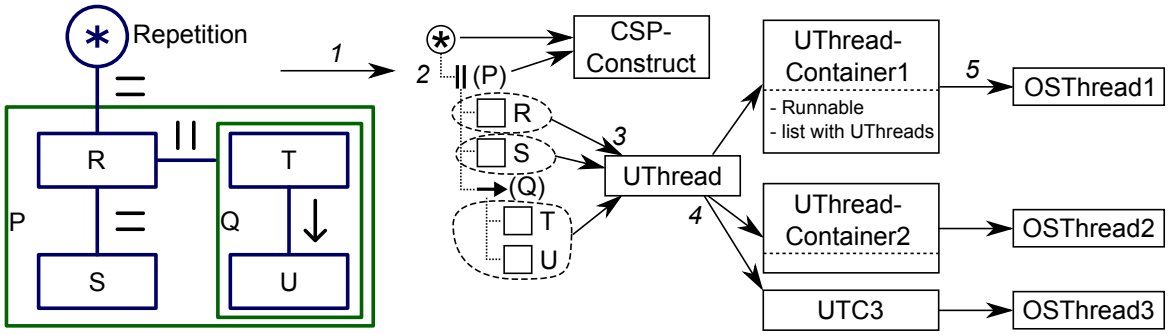

Figure 5. Steps from a model to a LUNA based mapping on OS threads.

Since the CSP processes are indifferent to the type of thread they run on and how they are grouped on OS threads, LUNA needs to provide a mechanism to actually attach these processes to threads. When looking at a gCSP model (left-most part of the figure), a compositional hierarchy can be identified in the form of a tree (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 and generate code. Because of the 'separation of concerns' code generation is straightforward as the interoperation of OS and User threads is handled by LUNA. Figure 5 shows the required steps to map the model to OS threads.

First, the model needs to be converted to a model tree (number *1* in the figure). This model-tree contains the compositional relations between all processes. Second, the user (or the modeling tool) needs to group processes (*2*) which are put on the same OS thread, for example criteria for grouping could be processes which heavily rely on communication or try to balance the execution load. Each process is mapped to a UThread object (*3*). Except for the compositional processes mentioned in the previous section, they are mapped onto CSPConstructs. Next, each group of of UThreads is put in an UThreadContainer (UTC) (*4*).

Finally, each UTC is mapped to an OS thread (*5*), 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 [19].

1.3. Channels

One of the initial reasons for supporting QNX was the availability of native rendez-vous communication support between QNX threads. This indeed made it easy to implement channels for the OS threads, but unfortunately it was *not* for the User threads. Main problem is that two User threads which want to communicate may be placed on the same OS thread. If one User thread wants to communicate over a rendez-vous channel and the other side is not ready, the QNX channel blocks the thread. But QNX does not know about the LUNA implemented scheduler and its User threads, so it blocks the OS thread instead. The other User thread which is required for the communication now never becomes ready and a deadlock occurs. So unfortunately, for communication between User threads on the *same* OS thread the QNX rendez-vous channels are not usable and the choice to initially support QNX became less strong.

Figure 6 shows the 2 possible channel types. Channel 1 is a channel between two OS threads. The QNX rendez-vous mechanism can be used for this channel. Channel 2a and 2b are communication channels between two User threads; it does not matter whether the User threads are on the same OS thread or not. For this type of channel the QNX rendez-vous mechanisms cannot be used as explained earlier, as it could block the OS thread and therefore prevent execution of other User threads on that OS thread. An exception could be made for

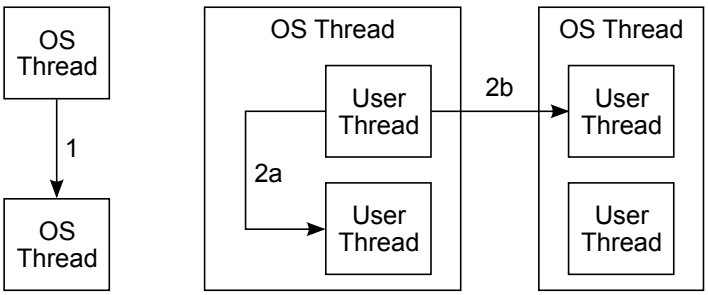

Figure 6. Overview of the different channel situations.

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 in between. Guarded channels are also not supported by QNX, so for this type of channels a custom implementation is also required.

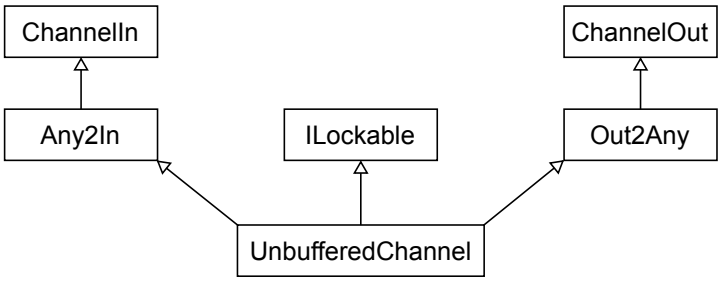

Figure 7. Diagram showing the channel architecture.

Figure 7 shows the architecture of the channel implementation. A channel is constructed modularly: The buffer, Any2In and Out2Any types can be exchanged with other compatible types. The figure shows an unbuffered any-to-any channel, but a buffered any-to-one is also possible, along with all kinds of other combinations.

```
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 1. Pseudocode showing the channel behaviour for a write action.

Listing 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) 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 ready for communication. This list is ordered on process priority. And, the writer needs to be blocked until a reader is present. The same goes for reading a channel, but exactly the other way around.

The *findReadyReaderOrBuffer()* method checks if there is buffered data available, otherwise it calls a *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. So depending on the input type of the 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 rendez-vous 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 3). 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.

As mentioned in the start of this section, there are different implementations of channels: the QNX implementation used for communication between OS threads and the LUNA implementation for communication between User threads and/or OS threads. It would be cumbersome for a developer to have to remember to choose between these types, especially when the User threads are not yet mapped to their final OS threads. So a channel factory is implemented in LUNA. When all CSP processes are mapped on their threads, this factory can be used to determine what types of channels are required. Having the information of the type of threads to map the CSP processes on is sufficient to determine the required channel implementation. At run-time, before the threads are activated, the factory needs be invoked to select a correct implementation for each channel. If a developer (or code generation) moves

a CSP process to another OS thread, the factory will adapt accordingly, using the correct channel implementation for the new situation.

1.4. Alternative

The Alternative architecture is shown in Figure 8. It is a CSProcess itself, but it also has a list of other CSProcesses which implement the IGuard interface. Alternative uses the list when it is activated and will try to find a process which meets its IGuard conditions. Currently, the only guarded processes that are available are the GuardedWriter and GuardedReader processes. But others might be added, as long as they implement the IGuard interface.

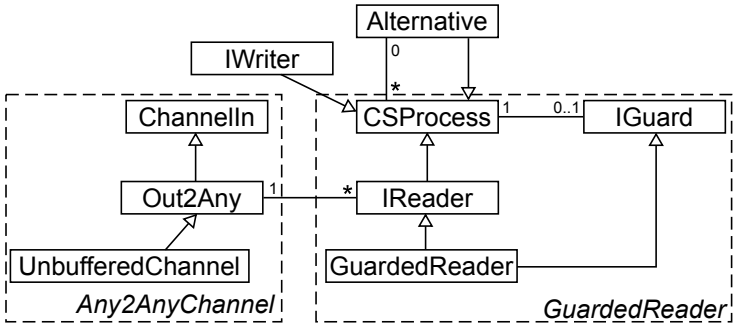

Figure 8. Diagram showing the relations for the Alternative architecture.

In the case of channel communication, it first checks if a reader or writer is guaranteed to perform channel communication without blocking and makes sure this guarantee stays intact. Next, it performs the communication itself. The Alternative implements a sophisticated protocol in order to make sure the communication is guaranteed, even though different threads are part of the communication or some of the processes on the channel might be not guarded.

First in Figure 9 a situation is shown, where a guarded reader gains access on a channel, but blocks when it should actually read the contents, as another reader came in-between. Some of the objects in Figure 8 are grouped by the dashed boxes, they are shown in Figure 9 as a single object to keep things simple.

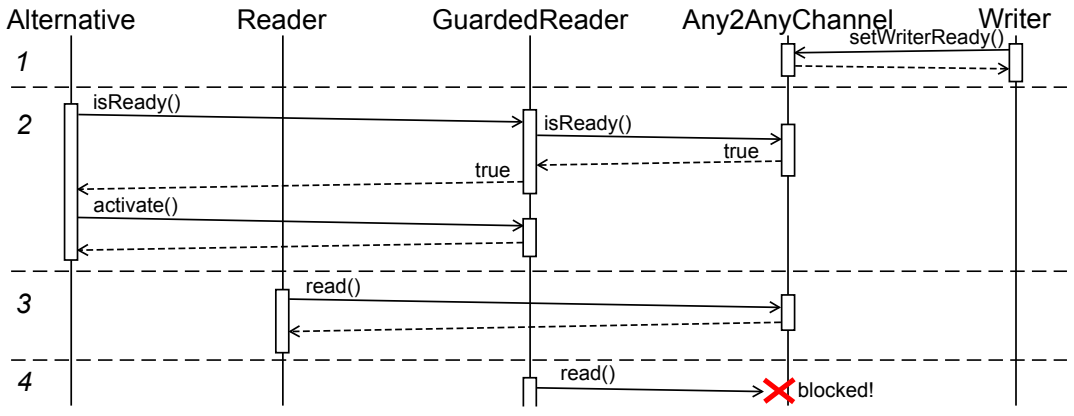

Figure 9. Sequence diagram showing a situation were a guarded reader blocks.

Assume we have an any-to-any channel, which has a writer waiting to communicate (*1* in the figure). The Alternative is activated (*2*) and checks if the GuardedReader is ready. 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 so it can be scheduled by the scheduler to actually perform the communication.

Unfortunately before the communication takes place, another Reader is activated and wants to communicate on the channel as well (*3*), since there is a writer present the communication takes place. Later, the GuardedReader is activated (*4*), but the writer is not available anymore and the GuardedReader is blocked, even though it gained access to the channel through the Alternative.

To prevent such behaviour a more sophisticated method is used, shown in Figure 10. This example describes a situation where both channel ends are guarded to be able to describe the protocol completely. Whether this situation is used in real applications or not is out of scope.

Again the writer registers at the channel, telling that it is ready to write data (*1*). There is no reader available yet, so the write is put in the *ready_list* and gets a *false* as result. Next, Alternate2 continues to look for a process which can be activated, but this is not interesting for the current situation.

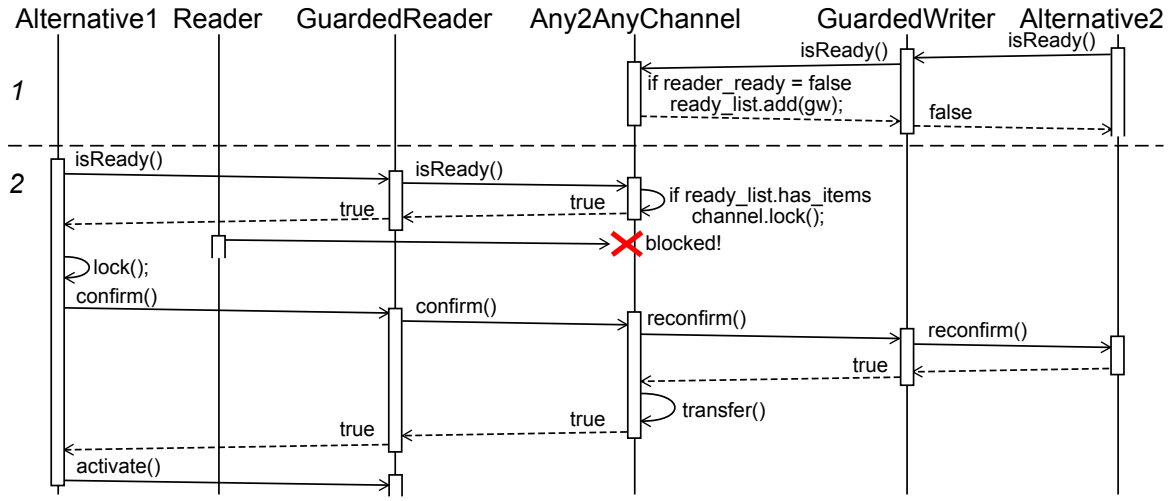

Figure 10. Sequence diagram showing the correct situation.

Alternate1 checks whether the GuardedReader is ready or not, when it becomes active (*2*). Since the *ready_list* has items on it, the channel is ready for communication. To prevent that other readers are interfering with our protocol, the channel gets locked. If Reader wants to read from the channel it gets blocked due to the lock. This is in contrast with the previous example, where the GuardedReader got wrongly blocked.

When the *isReady()* request returns positive, the Alternative1 checks whether another, previously *isReady()* requested, guard has not been reconfirmed. If this is not the case, it will *lock()* the Alternative for exclusive reconfirm request, preventing other guards taking over the current communication.

Before the actual transfer, Alternative1 needs to check whether the GuardedWriter is still ready to write. It might be possible that Alternate2 found another process to activate and the GuardedWriter is not ready anymore. Using the *confirm()* method, Alternative1 asks the channel for this and the channel forwards the question at Alternate2 via GuardedWriter with the *reconfirm()* method. Assuming that the GuardedWriter is still ready, the channel directly performs the *transfer* of data. This is not necessary, but is more efficient as the channel becomes available for other communications earlier.

In the end, Alternative1 revokes the *isReady()* requests of its other guarded processes, since a process was chosen, and it activates GuardedReader. For this example situation it is unnecessary, since the transfer is completed already, but for other (non reader/writer) processes it is required to run the guarded process code. Also, the GuardedReader might be used to activate a chain of other processes.

The described alternative sequence of Figure 10 has been tested for some basic use cases. Although it is not formally proven, it is believed that this implementation will satisfy the CSP requirements of the alternative construction.

2. Results

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

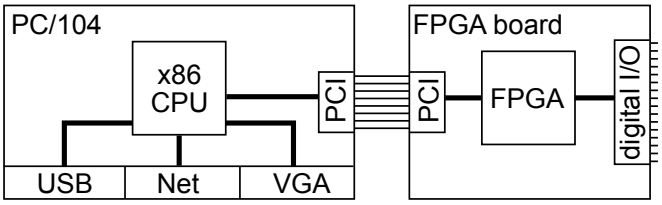

Figure 11. Overview of the used test setup.

All tests in this section are performed on a embedded PC/104 platform with 600 MHz x86 CPU as shown in Figure 11. It is equipped with an FPGA based digital I/O board to connect it with actual hardware when required for the test. While implementing and testing LUNA, QNX seemed to be slower than Linux. To keep the test results comparable, all presented tests are executed under QNX (version 6.4.1) and compiled with with the corresponding qcc (version 4.3.3) with the same flags (optimisation flag: -O2) enabled.

2.1. Context-switch Speed

After the threading model was implemented, a context-switch speed test was performed to get an idea of the efficiency of the LUNA architecture and implementation. To measure this speed, an application was developed consisting of two threads switching 10,000 times. The execution times were measured and the average switching time was calculated to get a more precise context-switching time. Table 1 shows these times.

Table 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, but only the required parts for the test are made available. In order to be able to compile the CTC++ library for QNX, some things needed to change:

- The `_setjmp`/`_longjmp` implementation used when switching to another User thread. The Stack Pointer (SP) was changed to use the correct field for QNX.
- Linux does not save the signal mask by default when executing `_setjmp` and `_longjmp`. QNX does, which slows down the context switches considerable. Therefore, the '`_`' versions of `setjmp` and `longjmp` are used for the QNX conversion.
- The compiler and its flags in order to use the QNX variants.
- 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.

To use the C++CSP2 library with QNX, the same changes were made as for CTC++ library except the SP modification, as it was not required for C++CSP2. As mentioned the `_setjmp`/`_longjmp` are used for the quick conversion to QNX, although the library already used `_longjmp`, but not `_setjmp`. This might indicate that the author knew of this difference and intended different behaviour. The QNX implementation for the C++CSP2 library is also not complete, only the required parts are tested, all other parts are not tested for compatibility. For the test a custom application was created as the provided C++CSP2 test suite did not contain a pure context switching test.

CTC++ QNX [20] 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.

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 switches, which is expected and the reason for the availability of User threads. All 3 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 well.

The context-switch time for the LUNA User threads is much lower 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.

2.2. Commstime Benchmark

To get an better idea of the scheduling overhead, the commstime benchmark is implemented, as shown in Figure 12. This test passes a token along a circular chain of processes. The Prefix process starts the sequence by passing the token to Delta, which again passes it on to the Prefix via the Successor process. The Delta process also signals the TimeAnalysis process, so it is able 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.

Table 2 shows the cycle times for each library for the commstime benchmark. The commstime tests are taken from the respective examples and assumed to be optimal for their CSP implementation. LUNA QNX has two values: the first is for the LUNA channel implementation and the second value for the QNX channel implementation. It is remarkable that the QNX channels are slower than the LUNA channels. This is probably due to the fact the QNX channels are always any-to-any and the used LUNA channels one-to-one. The amount of context-switches of OS threads is unknown, since the actual thread switching is handled

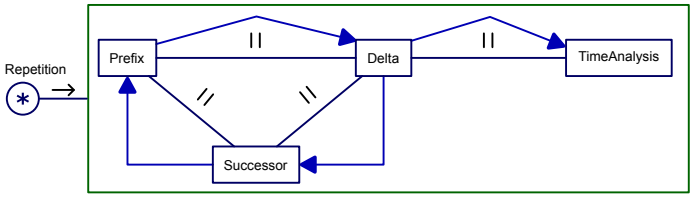

Figure 12. Model of the commstime benchmark.

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

Table 2. Overhead of the schedulers implemented by the 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

Normally, the library is used with modeling tools in combination with code generation. This would result in a different implementation of the commstime benchmark. In general, the readers and the writers become separate processes, instead of integrated within the Prefix, Delta, Successor and TimeAnalysis processes. For example, in this situation the Successor is implemented using a sequential process containing a reader, an increment and a writer process.

Table 3 shows the results when gCSP in combination with code generation is used to design the commstime benchmark application. gCSP code generation is only available for CTC++, so for LUNA the CTC++ code is rewritten manually as if it would have been generated.

Table 3. Commstime results when using MDD tools to create the test.

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. Since C++CSP2 threads are destroyed when one cycle is done, they need to be recreated for each cycle. The processes added to sequential process, for example in the Successor, cannot contain a loop, since this would prevent the execution of the second and the third process because those processes need to wait on preceding processes. Due to this limitation, the C+CSP2 implementation needs to recreate 15 threads each cycle, hence the +15 in the table. The construction and destruction of these threads generates a lot of overhead, resulting in the high cycle times around $12.5ms$. It was not possible to prevent this behaviour when using the 'code generated' code, due to differences in the design ideas behind the libraries.

The table also shows that the close result of the CTC++ 'original' and the CTC++ QNX libraries 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. For the first results, the optimized channels of the QNX variant probably resulted in the small difference between the two.

The benchmark results of LUNA are much better compared to the CTC++ library. Furthermore the LUNA results are better than the C++CSP2 results when looking at Table2. This is due to the efficient context-switches, as described in the previous section. When compensating for the required context-switch times, the results for C++CSP2 and LUNA are similar.

When comparing both tables, it is clear that using MDD tools with code generation results in slower code. For simple applications 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. For this 'code generated' results, the LUNA framework has good cycle times, which is encouraging since the planning of TERRA, the new MDD tool, which will feature code generation for LUNA. It is advisable for such an MDD tool to invest effort into optimizing code generation to get good performance on the target system.

2.3. Real Robotic Setup

Next, an implementation for a real robotic setup was developed with LUNA, to see whether it is usable in a practical way. To keep things easy for a first experiment, a simple pan-tilt setup is used, with 2 motors and 2 encoders. These 2 degrees of freedom can be controlled using a joystick. The control algorithm of this setup requires about 50 context switches to completely run one cycle.

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

Table 4 shows the timing results of LUNA and the CTC++ implementation. The experiments have been performed with $100Hz$ and $1kHz$ sample frequencies, so each control loop cycle should be respectively $10ms$ and $1ms$ long. 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 is 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.

Table 4. Timing results of the robotic implementation.

Platform	Frequency (Hz)	Cycle time (ms) Mean	Min	Max	Standard deviation (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

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.

Due to periodically missed clock ticks, the maximum cycle time of the $1kHz$ measurements is twice the sample time. This phenomenon can be explained by the mismatch between the requested timer interval and the PC/104's hardware timer [21]. The timer can not fire exactly every $1ms$, but instead it fires every $0.999847ms$ and for every 6535 instances the timer will not fire. In the $100Hz$ case this will not be noticed, because the cycle time is large enough and these kind of errors are relatively small.

When looking at the CTC++ 'original' implementation, it is seen that the $100Hz$ results are good as well, although the mean cycle time, shows that the obtained frequency is $90.9Hz$ instead of $100Hz$. Same goes for the $1kHz$ measurement where a $847.5Hz$ frequency was obtained instead. From this it can be concluded that CTC++ has problems to closely provide the requested frequencies. For a frequency of $1kHz$ the standard deviation becomes very large as well.

A third frequency was also measured, $1000.15Hz$, which is an exact match with the available frequency of the setup. This solves the very large standard deviation and the incorrect mean cycle times for the CTC++ library. It should be noted that this frequency is setup dependent and therefore needs to be measured for each setup separately, in order to gain these good results.

The frequency of $1000.15Hz$ indeed solves the maximum cycle times of LUNA being two periods long. For setups which needs to be extremely accurate this is important, as it can make the difference between an industrial robot moving smoothly or scratching your car. The other values are not much different, showing that LUNA is more robust for all frequencies than the CTC++ library and frequency tuning is not required to get reasonable hard real-time properties.

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 resources are available for the controlling code. Even the LUNA OS threads processing times are comparable with the CTC++ User thread processing times.

3. Conclusions

Good results are obtained using LUNA, it 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, the real-time behaviour of the library, is much more important when controlling robotic setups. The simple robotic setup indeed performed as expected; it reacts smoothly on the joystick commands. The maximum and minimum cycle time values are close to the (requested) mean cycle time and the standard deviation values are low, showing that the hard real-time properties of LUNA are good as well.

The choice for QNX is not that obvious anymore when the provided rendez-vous 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. The first three of them are obvious: LUNA is a hard real-time, multi-platform, multi-threaded framework.

Scalability is also met, even though LUNA was not yet tested with a big (robotic) setup, early scalability tests showed that having 10,000 processes poses no problem.

The CSP execution engine is the only implemented execution engine at the moment. But

the requirement to not be dependent on it is met, as it is possible to turn it off and use the User and OS threads in a non CSP related way. Using the provided interface it is also possible to add other execution engines like a state machine execution engine.

Developing applications using LUNA is straightforward, for example one does not need to keep the type of threads and channels in mind while designing the control application. It is possible to just create the CSP processes, connect them with channels and let the LUNA factories decide on the actual implementation types.

And finally, Debugging and Tracing is also not a problem, it is possible to enable the debugging component if required. This component contains means for debugging and tracing the other components as well as the application is being developed. It is also possible to send the debug and trace information over a (local) network to a development PC, in order to have run-time analysis or to store it for off-line analysis.

The required logger does not influence the executing application noticeable as it is a real-time logger. 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.

Especially logging the activation of processes is interesting, as this could provide valuable timing information, like the cycle time of a control loop or the jitter during execution. So it is possible to influence the application with external events and directly see the results of such actions. It is also possible to following the execution of the application by monitoring the states (running, ready, blocked, finished) of the processes. This information could also be fed back to the MDD tool, in order to show these states in the designed model of the application.

For *future work* an implementation for Linux (and Windows) would be convenient. It is much faster to try out new implementation on the development PC than on a target. Of course this requires more work, hence the choice to support QNX first, but it certainly pays off by reducing development time. The flexibility to easily move processes between the groups of OS and User threads reduces development time even more, as the developer does not required to change his code when moving processes.

Building the simple robotic setup took some time. There are only about 51 processes to control this setup. Of course this could be less, but it takes too much time to develop controller applications by hand, so code generation for LUNA is required. In order to attract users to start using LUNA, also for educational purposes, code generation is also required. So, soon after LUNA evolves into an initial/stable version, TERRA needs to be build as well, to gain these advantages to properly use LUNA.

When TERRA and code generation are available, algorithms to optimize the model for a specified target with known resources can be implemented. Before code generation, these algorithms [19] can schedule the processes automatically in an optimal manner for the available resources. These scheduling algorithms are also interesting for performing timing analysis of the model, in order to estimate whether the model will be able to run real-time with the available resources.

To see whether LUNA is capable of controlling setups larger than the example setup, it is planned to control the Production Cell [22] with it. It is already partially implemented, but the work is not completely finished yet.

Performing similar tests, as done in Section 2.3, really shows the advantages of using LUNA. Another planned test with the Production Cell is to control it with Arduinos [23]. The Production Cell has 6 separate production cell units (PCUs), each PCU is almost a separate part of the setup. Using one Arduino for each PCU seems like a nice experiment for distributed usage of LUNA. This would require support for a new platform within LUNA, which does not use operating system related functionalities.

References

[1] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[2] D.S. Jovanović, B. Orlic, G.K. Liet, and J.F. Broenink. gCSP: a graphical tool for designing CSP systems. In I. East, Jeremy Martin, P.H. Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures 2004*, volume 62, pages 233–252, Amsterdam, September 2004. IOS press.

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

[4] H. Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[5] J.F. Broenink, Y. Ni, and M.A. Groothuis. On model-driven design of robot software using co-simulation. In E. Menegatti, editor, *SIMPAR, Workshop on Simulation Technologies in the Robot Development Process*, November 2010.

[6] M.A. Groothuis, R.M.W. Frijns, J.P.M. Voeten, and J.F. Broenink. Concurrent design of embedded control software. In T. Margaria, J. Padberg, G. Taentzer, T. Levendovszky, L. Lengyel, G. Karsai, and C. Hardebolle, editors, *Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling (MPM2009)*, volume 21 of *Electronic Communications of the EASST journal*. EASST, ECEASST, October 2009.

[7] N.C.C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In A.A. McEwan, W. Ifill, and P.H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, July 2007.

[8] uClibc website, 2011. `http://www.uclibc.org/`.

[9] P.H. Welch, N.C.C. Brown, J. Moores, K. Chalmers, and B. Sputh. Integrating and Extending JCSP. In A.A. McEwan, W. Ifill, and P.H. Welch, editors, *Communicating Process Architectures 2007*, pages 349–369, July 2007.

[10] JamaicaVM website, 2011. `http://www.aicas.com/jamaica.html`.

[11] OROCOS website, 2011. `http://www.orocos.org/`.

[12] ROS website, 2011. `http://www.ros.org/`.

[13] E.W. Dijkstra. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured programming*, chapter 1, pages 1–82. Academic Press Ltd., London, UK, 1972.

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

[15] OpenWRT website, 2011. `http://www.openwrt.org/`.

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

[17] QNX website, 2011. `http://www.qnx.com`.

[18] Mordor website, 2011. `http://code.mozy.com/projects/mordor/`.

[19] M.M. Bezemer, M.A. Groothuis, and J.F. Broenink. Analysing gcsp models using runtime and model analysis algorithms. In P.H. Welch, H.W. Roebbers, J.F. Broenink, F.R.M. Barnes, C.G. Ritson, A.T. Sampson, D. Stiles, and B. Vinter, editors, *Communicating Process Architectures 2009*, volume 67, pages 67–88, November 2009.

[20] B. Veldhuijzen. Redesign of the CSP execution engine. MSc thesis 036CE2008, Control Engineering, University of Twente, February 2009.

[21] M. Charest and B. Stecher. Tick-tock - Understanding the Neutrino micro kernel's concept of time, Part II, April 2011. `http://www.qnx.com/developers/articles/article_826_2.html`.

[22] M.A. Groothuis and J.F. Broenink. HW/SW Design Space Exploration on the Production Cell Setup. In P.H. Welch, H.W. Roebbers, J.F. Broenink, and F.R.M. Barnes, editors, *Communicating Process Architectures 2009, Eindhoven, The Netherlands*, volume 67 of *Concurrent Systems Engineering Series*, pages 387–402, Amsterdam, November 2009. IOS Press.

[23] Arduino website, 2011. `http://www.arduino.cc/`.

Bibliography

Alexandrescu, A. (2001a), *Modern C++ design: generic programming and design patterns applied*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, chapter 8, ISBN 0-201-70431-5.

Alexandrescu, A. (2001b), *Modern C++ design: generic programming and design patterns applied*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, chapter 6, ISBN 0-201-70431-5.

Anderson, J. H., S. Ramamurthy and K. Jeffay (1997), Real-time computing with lock-free shared objects, *ACM Transactions on Computer Systems*, **vol. 15**, pp. 28–37.

Baase, S. and A. Gelder, van (2000), *Computer Algorithms - Introduction to Design & Analysis*, Addison-Wesley, 3 edition.

Bennik, J. (2008), Mechatronic design of a humanoid head and neck, Msc Thesis 019CE2008, Control Laboratory, University of Twente.

Berg, van den, L. (2006), Design of a Production Cell Setup, Msc Thesis 016CE2006, Control Laboratory, University of Twente.

Bezemer, M., M. Groothuis and J. F. Broenink (2011a), Way of Working for Embedded Control Software using Model-Driven Development Techniques, in *ICRA Workshop on Software Development and Integration in Robotics (SDIR VI)*, Eds. D. Brugali, C. Schlegel and J. F. Broenink, IEEE, IEEE, pp. 1 – 6.

Bezemer, M., R. 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*, Ed. P. Welch, IOS Press BV, Amsterdam, volume 68 of *Concurrent System Engineering Series*.

Boost (2011), Free peer-reviewed portable C++ source libraries.
http://www.boost.org/

Broenink, J., Y. Ni and M. Groothuis (2010a), 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.

Broenink, J. F., M. Groothuis, P. Visser and M. Bezemer (2010b), 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. Johansson, IEEE, IEEE, pp. 73 – 77.

Broenink, J. F., M. A. Groothuis, P. M. Visser and B. Orlic (2007), A Model-Driven Approach to Embedded Control System Implementation, in *2007 Western Multiconference on Computer Simulation*, SCS, San Diego, CA.

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

Buit, E. (2005), *PC104 stack mechatronic control platform*, MSc thesis 009CE2005, Control Engineering, University of Twente.

C++CSP2 (2009), Easy Concurrency for C++.
http://www.cs.kent.ac.uk/projects/ofa/c++csp/

Charest, M. and B. Stecher (2011), Tick-tock - Understanding the Neutrino micro kernel's concept of time, Part II,
http://www.qnx.com/developers/articles/article_826_2.html.

Controllab Products (2011), 20-sim - Graphical modeling and simulation tool.
http://www.20-sim.com/

Cooling, J. (2000), *Software Engineering for Real-Time Systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 0201596202.

CORBA (2011), Common Object Request Broker Architecture.
http://www.omg.org/spec/CORBA/3.1/

Cygwin (2011), Cygwin - ports of the popular GNU development tools for Microsoft Windows.
http://www.cygwin.com/

DIAPM (2011), RealTime Application Interface for Linux.
http://www.rtai.org

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

Douglass, B. P. (2002), *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 0201699567.

Fainelli, F. (2008), *The OpenWrt embedded development framework*, Free and Open source Software Developers' European Meeting (FOSDEM).
http://downloads.openwrt.org/people/florian/fosdem/openwrt_cfp_fosdem2008.pdf

Formal Systems (Europe) Limited (2008), FDR2.
http://www.fsel.com/software.html

Groothuis, M. (2004), *Distributed HIL simulation for BodeRC*, MSc thesis 020CE2004, Control Engineering, University of Twente.

Groothuis, M. A., J. J. Zuijlen, van and J. F. Broenink (2008), FPGA based Control of a Production Cell System, in *Communicating process architectures 2008 : WoTUG-31*, volume 66 of *Concurrent Systems Engineering Series*, IOS Press, Amsterdam, volume 66 of *Concurrent Systems Engineering Series*, pp. 135–148, dOI (not active): 10.3233/978-1-58603-907-3-135.

GSL (2011), GNU Scientific Library.
http://www.gnu.org/software/gsl/

Hilderink, G., J. Broenink and A. Bakkers (1997), Communicating Java Threads, in *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50, IOS Press, Netherlands, University of Twente, Netherlands, volume 50, pp. 48–76.

Hilderink, G., J. Broenink and A. Bakkers (1999), Communicating Threads for Java, in *Architectures, Languages and Techniques*, Ed. B. Cook.

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice Hall International.

IEEE Std 1003.1 (2004), IEEE Standard for Information Technology - Portable Operating System Interface (POSIX). Base Definitions, *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Base*, doi:10.1109/IEEESTD.2004.94570.

Jones, E. (2003), Implementing a Thread Library on Linux,
http://www.evanjones.ca/software/threading.html.

Jovanović, D., B. Orlic, G. Liet and J. 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, volume 62, pp. 233–252, ISBN 1-58603-458-8, ISSN 1383-7575.

Jovanovic, D. S., B. Orlic, G. K. Liet and J. F. Broenink (2004), Graphical Tool for Designing CSP Systems, in *Communicating Process Architectures 2004*, pp. 233–252, ISBN 1-58603-458-8.

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

Lethbridge, T. and R. Laganiere (2001), *Object-Oriented Software Engineering: Practical Software Development using UML and Java*, McGraw-Hill, Inc., New York, NY, USA, 1 edition, ISBN 0077097610.

Lootsma, M. (2008), Design of the global software structure and controller framework for the 3TU soccer robot, Msc Thesis 014CE2008, Control Laboratory, University of Twente.

Masmano, M., I. Ripoll, A. Crespo and J. Real (2004), TLSF: A New Dynamic Memory Allocator for Real-Time Systems, in *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, IEEE Computer Society, Washington, DC, USA, pp. 79–86, ISBN 0-7695-2176-2, doi:10.1109/ECRTS.2004.35.

Mesa electronics (2011), Mesa Anything I/O FPGA cards, http://www.mesanet.com/fpgacardinfo.html.

Meyers, S. (2005a), *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*, Addison-Wesley Professional, ISBN 0321334876.

Meyers, S. (2005b), *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*, Addison-Wesley Professional, chapter 47, ISBN 0321334876.

Meyers, S. (2010), *Effective C++ in an Embedded Environment*, volume 1, Artima Press. http://www.aristeia.com/c++-in-embedded.html

Michael, M. M. and M. L. Scott (1996), Simple, fast, and practical non-blocking and blocking concurrent queue algorithms, in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, ACM, New York, NY, USA, PODC '96, pp. 267–275, ISBN 0-89791-800-2.

Molanus, J. (2007), Redesign of the Linux driver for the Mesa Anything I/O cards, Pre-doctoral 023CE2007, Control Engineering, University of Twente.

Molanus, J. (2008), *Feasibility analysis of QNX Neutrino for CSP based Embedded Control Systems*, MSc thesis 032CE2008, Control Engineering, University of Twente.

Mozy (2011), Mordor website, http://code.mozy.com/projects/mordor/.

Nissanke, N. (1997), *Realtime systems*, Prentice Hall series in computer science, Prentice Hall, ISBN 978-0-13-651274-5.

Open Group, The (1998), Unix98 mark - the mark for systems conforming to version 2 of the Single UNIX Specification integrates the industry's Open Systems standards. http://www.unix.org/unix98.html

OpenWrt (2011), OpenWrt buildroot - a GNU/Linux distribution for embedded devices, http://www.openwrt.org/.

Orlic, B. and J. 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.

Orocos (2011), Open Robot Control Software. http://www.orocos.org/

Peeters, K. (2009), tree.hh: an STL-like C++ tree class, website.

Petters, S. and D. Thomas (2005), *RoboFrame - Softwareframework for mobile autonomous robotic systems*, Master's thesis, Technical University Darmstadt.

POCO (2010), The POCO C++ Libraries overview v1.3. http://pocoproject.org/documentation/PoCoOverview.pdf/

POCO (2011), POCO C++ libraries.
http://www.pocoproject.org/

QNX (2010), Writing a Resource Manager, http://www.qssl.com/developers/docs/6.3.2/neutrino/prog/resmgr.html.

QNX Software Systems (2011), The QNX hard real-time operating system.
http://www.qnx.com

Reilink, R. (2008), Realtime Stereo Vision Processing for a Humanoid, Msc Thesis 030CE2008, Control Laboratory, University of Twente.

Risler, M. and O. Stryk, von (2008), Formal Behavior Specification of Multi-Robot Systems Using Hierarchical State Machines in XABSL, in *AAMAS08-Workshop on Formal Models and Methods for Multi-Robot Systems*, Estoril, Portugal.

RoboCup (2011), robot soccer competition.
http://www.robocup.org

RoboFrame (2010), A software framework tailored for heterogeneous teams of autonomous mobile robots.
http://www.roboframe.info

ROS (2011), Robotic Operating System.
http://www.ros.org

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.

Shann, C.-H., T.-L. Huang and C. Chen (2000), A Practical Nonblocking Queue Algorithm Using Compare-and-Swap, in *Proceedings of the Seventh International Conference on Parallel and Distributed Systems*, IEEE Computer Society, Washington, DC, USA, ICPADS '00, pp. 470–, ISBN 0-7695-0568-6.

Silberschatz, A., P. B. Galvin and G. Gagne (2004), *Operating System Concepts*, John Wiley & Sons, ISBN 0471694665.

Steen, van der, H. (2008), *Design of animation and debug facilities for gCSP*, MSc thesis 036CE2008, Control Engineering, University of Twente.

Steen, van der, H., M. Groothuis and J. Broenink (2008), Designing Animation Facilities for gCSP, in *Communicating Process Architectures 2008 : WoTUG-31*, Concurrent Systems Engineering Series, p. 447, dOI (not active): 10.3233/978-1-58603-907-3-447.

Sutter, H. (2009), Volatile vs volatile.
http://www.drdobbs.com/high-performance-computing/212701484

Tsigas, P. and Y. Zhang (2001), A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems, in *in Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures*, ACM, pp. 134–143.

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

Visser, L. (2008), Motion control of a humanoid head, Msc Thesis 016CE2008, Control Laboratory, University of Twente.

Zuijlen, van, J. (2008), FPGA-based control of the production cell using Handel-C, Technical Report 008CE2008, Control Laboratory, University of Twente.