University of Twente

EEMCS / Electrical Engineering
Control Engineering

Design of the global software structure and controller framework for the 3TU soccer robot

Marten Lootsma

MSc report

Supervisors:
prof.dr.ir. S. Stramigioli
dr.ir. J.F. Broenink
dr. R. Carloni
ir. G. van Oort

June 2008

Report nr. 014CE2008
Control Engineering
EE-Math-CS
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Summary

Four identical humanoid soccer robots are currently developed by the three universities of technology in the Netherlands (3TU), with the intention to participate as the TeamDutchRobotics at the RoboCup 2008 competition. In order to move, each robot has 14 joints in its arms and legs, from which 12 are actuated by electric motors. The mechanical configuration is designed for dynamical walking; it is unstable and should continuously have stabilizing control. The 'brain' of the robot is a single processor computer, a PC-104 stack. The aim of this master thesis project is twofold:

- Design of the global software structure
- Design of the joint controller framework

The first part is about the software architecture, real-time behavior and the use of third-party software. Many software projects do exist for robots, the so-called robot software frameworks. The usability of these frameworks has been investigated, it remained in two candidates for using it in TUlip: Orocos and RoboFrame. However, no framework fully satisfied the requirements, because Orocos is not light weighted and RoboFrame does initially not support real-time. It has been chosen to use RoboFrame for the non-real-time part of the software and a self-written framework for the real-time part, with kept in mind that RoboFrame later can be ported to support real-time behavior. Furthermore it has been decided to use Linux/Xenomai as operating system to achieve the real-time execution of the software.

The second part is about the design of a software framework for stabilizing controllers: the joint controller framework. The movements of the robot exist of several basic behaviors, such as walking, kicking the ball, stand up etcetera. Each basic behavior might need different configurations of the joint controllers or even different types of joint controllers. The goal is to create a framework that gives control engineers the opportunity to switch among controllers and to be flexible in adjusting the parameters. The joint controller framework has been designed and a prototype has been realized. Instead of testing it on the real robot, models of the robot have been used to test the framework, because the robot was not assembled yet. Because of problems with the model of TUlip's body, the joint controller framework is tested on models of a humanoid head as well. The joint controller framework worked well and it has been used for realizing the head demo, as described in Visser (2008).

For future work, the prototype of the joint controller framework has to be extended to support switching between controllers while performing movements. It is expected that switching of controllers require an initialization mechanism for the new controllers in order to perform the transition of the controllers as smooth as possible.

Instead of having two separate frameworks for TUlip, it is an option to port RoboFrame to support real-time behavior, in order to create a light weight real-time framework that can be used for the whole system.

Although Orocos has chosen not to be used in TUlip because of uncertainties about its resource consumption, it might be interesting to do a full research on its capabilities and timing characteristics. Orocos is interesting, because does have real-time support and useful libraries for using it in robot applications.

Samenvatting

Vier identieke humanoid voetbal robots worden op dit moment ontwikkeld door de drie Nederlandse technische universiteiten (3TU), met de intentie om als de TeamDutchRobotics aan de RoboCup 2008 competitie mee te doen. The robots hebben elk 14 gewrichten in de armen en benen, waarvan 12 geactueerd door elektrische motoren. Het mechanische ontwerp is toegespitst op dynamisch lopen; het is onstabiel en heeft voordurend stabiliserende regelactie nodig. Het 'brein' van de robot is een computer met een enkele processor, een PC-104 stack. Het doel van deze master opdracht kan worden verdeeld in twee gedeelten:

- Ontwerpen van de globale software structuur
- Ontwerpen van de joint controller framework

Het eerste gedeelte gaat over de software architectuur, real-time gedrag en het gebruik van software van derden. Er zijn veel software projecten speciaal voor robots beschikbaar, de zogenaamde robot software frameworks. Het nut van het gebruik van deze frameworks is onderzocht. Het resulteerde in twee kandidaten voor het gebruik in TUlip: Orocos en RoboFrame. Alhoewel, geen van beide frameworks voldoet volledig aan de eisen, omdat Orocos waarschijnlijk een te zware belasting voor de processor is en RoboFrame ondersteund geen real-time gedrag. Het is gekozen om RoboFrame te gebruiken voor het niet real-time gedeelte van de software en om een zelf geschreven real-time framework te gebruiken voor de tijdkritische software gedeelten, met de gedachten dat RoboFrame later kan worden aangepast om real-time gedrag te ondersteunen.

Het tweede gedeelte gaat over het ontwerp van een software framework voor stabiliserende regelaars: de joint controller framework. De bewegingen van de robot bestaan uit verschillende basic behaviors, zoals lopen, tegen een bal schoppen, opstaan etcetera. Elke van die basic behaviors heeft wellicht een andere configuratie van regelaars of zelf verschillende type regelaars nodig. Het doel is om een framework te creëren dat regeltechnici de mogelijkheid geeft om van regelaars te wisselen en om flexibel te zijn met het aanpassen van parameters. De joint controller framework is ontworpen en een prototype is gerealiseerd. Omdat de mechanische constructie van de robot nog niet volledig af was, is het framework getest op modellen van de robot. Door problemen met het model van de lichaam van de robot, is het framework ook getest op een model van een humanoid hoofd. Het framework is later gebuikt voor het realiseren van de hoofd demo, als beschreven in Visser (2008).

Voor verder werk, de prototype van the joint controller framework moet worden uitgebreid om tijdens het bewegen van de robot het verwisselen van regelaars te ondersteunen. Het is te verwachten dat een initialisatie mechanisme nodig is om de transitie naar nieuwe controllers zo geleidelijk mogelijk te laten verlopen.

In plaats om twee verschillende frameworks voor TUlip te gebruiken, is het een optie om RoboFrame aan te passen om real-time gedrag te ondersteunen, zodat een framework kan worden gebruikt voor het gehele systeem.

Desondanks dat Orocos is niet gekozen om het te gebruiken in TUlip vanwege de verwachtte zware processor belasting, kan het interessant zijn om verder onderzoek te doen naar de mogelijkheden en timing karakteristieken van Orocos. Orocos is interessant, omdat het van zichzelf al real-time gedrag ondersteund en het verschillende bruikbare libraries heeft voor gebruik in robot applicaties.

Preface

With this report I conclude my Master's program in Electrical Engineering. Doing the Master's program has been a great addition to my HBO education. I am very thankful that I had the opportunity to do so.

First of all I want to thank my parents that they always believed in me and for their support. They made me who I am and they make me feel proud.

In general I want to thank the people of the Control Engineering group. It has been a great experience to participate with them in several interesting projects. I would like thank Rafaella Carloni, without her help this report could not be finished in time. I owe thanks to Gijs van der Hoorn from the University of Delft for his cooperation and interesting ideas.

I want to thank my fellow students and friends Harm de Boer, Hans van der Steen, Jeroen Warnas and Rein Hoekema for being with me this three years. I owe special thanks to Harm for being critical on how to document my work.

Last but not least, I want to thank my better half: Alfiya Lootsma. Thanks for standing by me and for your unlimited support and patience.

Marten Lootsma
Enschede, June 2008

Contents

1 Introduction

1.1 The TUlip project

TUlip is the Dutch Robotics soccer robot, which is a common project of the three technical universities of the Netherlands (3TU). It is a humanoid robot, whose body structure resembles that of a human. The goal of the Dutch Robotics Team is to participate in the TeenSize Humanoid League of RoboCupSoccer 2008 in China, an annual soccer competition for robots, where the TeenSize Humanoid League is a category for autonomous humanoid robots larger than one meter.

The Dutch Robotics Team consists of four identical robots. They should be able to walk dynamically and kick the ball while maintaining balance, processing human-like sensor information, making tactical decisions and communicating among each other.

At the beginning of this master thesis project, the mechanical parts were already designed and in production. The design of the electronics was almost finished as well. However, no decisions about the software had been made yet.

Together with students from Delft, the design and implement of TUlip software had to be realized.

1.2 Assignment

The assignment is to participate in the design and the implementation of the robot software architecture. The main point of interest is the implementation of the joint controllers. The aim of the assignment is split into two parts:

- To design a global software architecture and to exploit, if possible, suitable third-party software in order to realize the designed architecture.
- To design a framework for the joint controllers. The focus of this design is to be as flexible as possible in order to give the control designers the freedom to use and switch between different configurations of controllers.

1.3 Approach

1 The first phase of the assignment is the study on robotics, a survey on software in existing robots and especially how the control of the joints is performed in similar robots.

2 After that, the functions that the software of TUlip should cover had been investigated, and a list of requirement has been set up.

3 In the third phase, the usability of third-party software had been investigated.
   - The usability of several third-party software frameworks had been considered. Together with Deen (2008), a test-case has been set up in order to compare two different architectures: RoboFrame and Orocos.
   - The use of Linux/Xenomai as the real-time operating system had been evaluated.

   Together with the project members of the TUlip development team, the decisions have been made.

4 The design and implementation work have been divided over the project members. In the fourth phase of this assignment, the possible controller configurations have been investigated and a concept design for the organization of joint controller has been made.

5 In the last phase, a prototype of the joint controller framework had been realized and tested with models and controllers of the robot by Daemen (2008) and Visser (2008).

1.4 Report outline

- In Chapter 2, background information on the robot is provided.
- An analysis has been done on the requirement and functions for the software in Chapter 3. A global software architecture will be presented. The different functions of the software are categorized by the layered control structure in order to divide priorities.
- In Chapter 4 several robot software frameworks have been analysed for the usability in TUlip. A suitable robot software framework is suggested and explained. Furthermore the chose is made on which operating system the software will run.
- Chapter 5 describes the analysis of the possible controller configurations and a concept design is made.
- The design and implementation of the joint controller framework is discussed in Chapter 6.
- In Chapter 7 the validation of the joint controller framework is discussed.
- Appendix A is the recommendation report on which third-party robot software frameworks are useful to realize the designed software structure.
- Appendix B describes the real-time software abstraction, the timing tests of tasks and how model-in-the-loop is realized.
- In Appendix C the timing test of Xenomai discussed

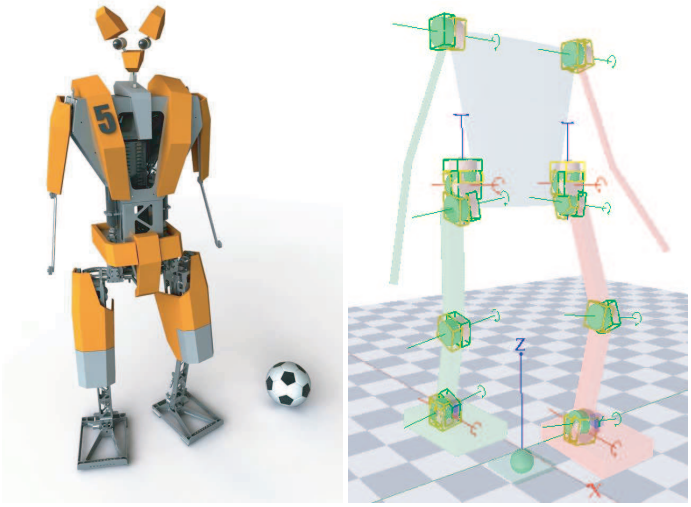
2 Background

In this chapter, background information to support understanding the context of this project is given.

2.1 Configuration of TUlip

The humanoid robot TUlip is about $1.20m$ high and weights, fully assembled, about $14kg$. The mechanical configuration is designed for dynamic walking (McGeer, 1988). It is unstable and should continuously have stabilizing control. A rendered picture of the TUlip design is shown in Figure 2.1(a).

In order to move, TUlip has a total of 14 rotational joints in legs and arms, as depicted in Figure 2.1(b). 12 joints are actuated by electrical motors, some of which are constructed as *series elastic actuators* (SEA), as described in Williamson (1995).



(a) CAD design

(b) Indication of the joints (green boxes with axis)

FIGURE 2.1 - The robot TUlip (a) and the simulated model (b)

TUlip should be able to walk dynamically and kick the ball while maintaining balance, processing human-like sensor information, making tactical decisions and communicating among each other. All these functions are performed by some sort of 'brain', which determines autonomously the tactical decisions that has to be made. These tactical decisions are based upon information about its environment and status. This information is provided by sensors, such as:

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

The tactical decisions are then translated to the necessary movements. To perform these movements, the joints need to be actuated. Joint controllers are needed to control this actuation in order to move the initially unstable robot in a proper way. Feedback for joint controllers are provided by rotational position sensors in the joints.

The brain of the robot is located in its chest and it is realized by a single processor PC-104 (2008) stack. The software on the stack needs to implement the functions The basic specifications of the stack are listed in Table 2.1.

TABLE 2.1 - Basic PC104 specifications

Type	Specification
Processor	low power, fan-less 1.0GHz VIA Eden ULV
Work memory	256MB 533MHz DDR2 RAM
Data storage	4GB flash card
Communication	Gigabit Ethernet
	4x USB 2.0
	4x Serial port
	33MHz PCI Bus

2.2 Software in similar robots

Similar dynamic walking robots have been analysed in order to evaluate the software in existing robots. The main point of interest is the computation of the joint controllers because TUlip has been designed to have a central computer on which the control algorithms of the joint have to be computed as well.

One of these projects is the two dimensional biped Dribbel. Dribbel has been designed at the University of Twente and it is controlled by a distributed network of micro-controllers (Dertien, 2005). A central computer is used, to control all the different distributed micro-controllers. Since the software of Dribbel does not compute the controller at a central computer, the software implementation of Dribbel is no used.

Flame, a robot of the University of Delft, computes the joint controllers in a central computer. The robot itself is in fact the standard used for the mechanical design of TUlip and therefore similar. The software is a self written set of libraries on a real-time Linux kernel. One thread, timed with the critical time period, processes all the tasks sequentially. So even the non-real-time tasks are performed on the highest frequency which leads to less computational space for the time critical tasks. Flame focuses on implementing a dynamic walking algorithm, while TUlip needs to perform, besides dynamic walking, several other functions. No software of Flame has been used for the implementation of TUlip, however, the software architecture has been studied in order to learn from it.

2.3 Robot software frameworks and middleware

As defined by Douglass (2003): "A framework is a 'partially completed application that you customize or specialize for you specific application.' A framework differs from a library in that a library is a passive thing that you call to invoke a service. A framework is your application that invokes the problem-specific services that you want to perform. Fortunately frameworks for robot software exists."

As described in Section 2.2, software for similar robots have been developed in previous projects. Since the software in these projects are not reused in TUlip, there are two options:

- write the software from scratch and create a self-written framework,
- use a third-party software framework to start with.

Several third-party software frameworks for using it in robots are available. A significant part of the frameworks is the *middleware*. It operating system services like processes and inter-process communication. Developers of these frameworks all have a different philosophy and background, which results in various projects, all with their strong and weak points. Alliance, CLARAty, Miro, Orca, Orocos, Player, RoboFrame, URBI and YARP are robot software frameworks. In this project, the usability of these frameworks for the TUlip software have been investigated. This is described in Appendix A.

3 Software architecture analysis

This chapter presents the analysis of software architecture of TUlip. In order to set up a architecture, a list of requirements has been made.

3.1 Requirements

The RoboCup rules, hardware limitations and requests of the control engineers are analysed, and translated to the requirements for the robot software. These requirements are:

- Multiple functions: TUlip should make autonomous decisions, have a vision system, actuate the hardware for performing complex movements (e.g.: dynamic walking), have stabilizing control and communicate with team members and the referee. These functions have to be implemented in the software.
- Real-time: Some parts of the software (e.g.: stabilizing control) are time critical and need real-time execution.
- Multiple controller algorithms and topologies: It has to be possible to select different controllers for each joint and situation.
- Light weighted: As the hardware is already designed, the computational power of 1GHz and the available work memory of 256MB are restrictions that should be taken into account. Because of the high complexity of the system, it is a challenge to fit the software in the given hardware. Therefore, the software should be light weighted.

Furthermore requirements according to the organization of the software are:

- Reusable: The intention of Dutch Robotics is to participate each year at the RoboCup tournament. The goal is to evolve the hardware and software over the years. Therefore the software needs to be reusable.
- Easily configurable: It should be possible to configure and adjust the software easily. For minor changes of the system, for instance parameter adjustments, the system should not have to be recompiled.
- Modularity: Since the software needs to be reusable, it should be modular. With a modular structure, it is also easier to divide the software implementation over a large (changing) group of developers.

3.2 Functional specification

As mentioned in the requirements, the robot software should fulfil several functions and be modular. A graph is set up in order to get a general overview of the functions and their relation, it is depicted in Figure 3.1. The structure consists of function blocks, which are explained in the following paragraphs:

World modeling

The robot is supposed to play in a field with borders, goals, team mates and opponents. Static parameters are available on forehand, such as: field size, border indication, color of team members and opponents etcetera. Information about the active environment is collected in this function block to create an up-to-date model of the environment. The information is gathered by several inputs, e.g: vision, distance sensors, status of the robot, information from other robots.

Communication

Data exchange with team mates and with the referee needs to be possible. Team mates can exchange information about the world model (e.g.: position of the ball). The referee gives information about important events like starting and stopping the game, a score or maybe a foul.

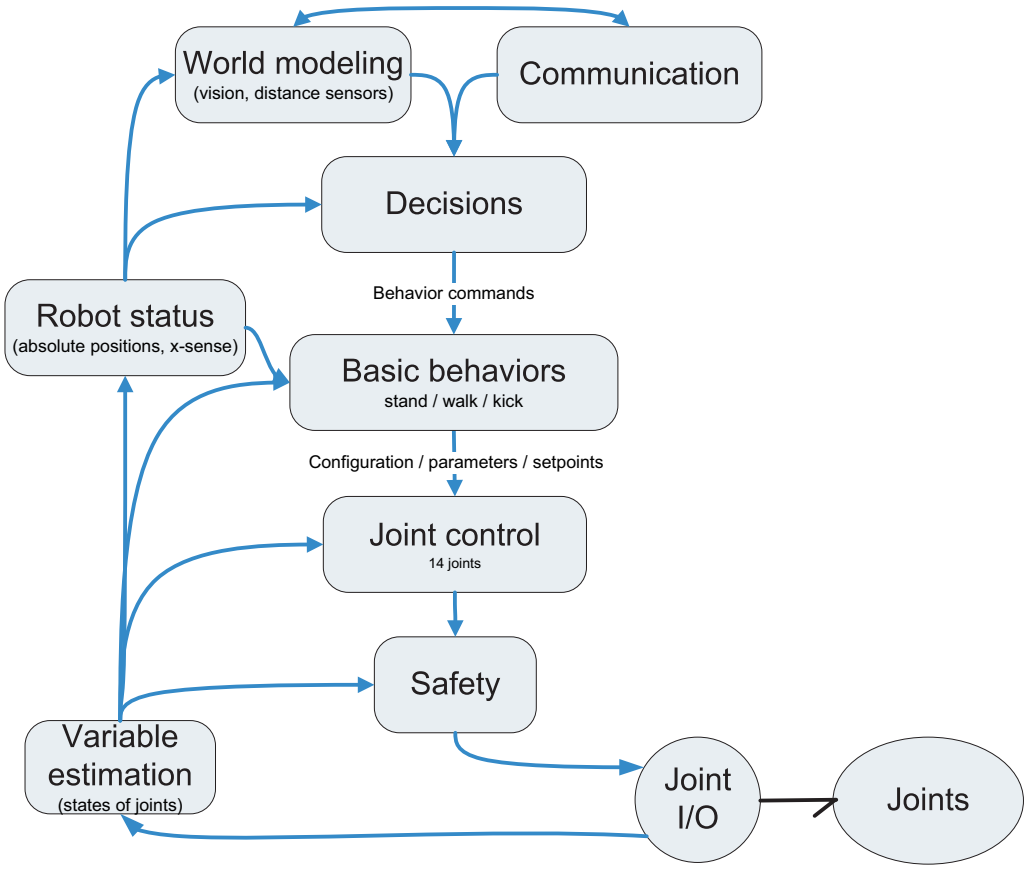


FIGURE 3.1 - Functions and their relations within the software.
A function block
The distribution of information is show by the arrows.
Ovals are the robot and the joint actuation/monitoring.
The bond-graph arrows indicate the physical interaction of the robot.

Decisions
Since it is an autonomous robot, TUlip needs to make decisions by itself. It has to decide which movements need to be performed, and give commands to the basic behaviors in order to perform the movements. The decisions can be based on information from the robot states, the world model and through communication with other robots and the referee.

Basic behaviors
The robot needs to move in various ways and perform different tasks (e.g.: walk to the ball and kick in direction of the opponent's goal). These movements are split up in simple movements (e.g.: walk, turn, kick), which are called basic behaviors. The idea and the term of basic behaviors came from a German RoboCup team as mentioned in Brunn et al. (2001). Each basic behavior is a set of motions which can be 'played'. A list of several basic behaviors can be played in order to perform a task. To perform motions, the basic behavior should be able to manipulate the joint controllers by:
- configure the controller algorithms and topologies
- set the parameters of the configured algorithms
- give setpoints to the controllers

Joint control
The joints of the robot are actuated by electric motors and need to be controlled continuously. The joint control should support several control algorithms and combination of control algorithms to perform the motions requested by the basic behaviors.

Safety
This function block has to prevent the robot from damaging itself, or its environment. Safety can be achieved by setting limitations to system states, such as maximum velocity, maximum torque and maximum position of the joints (also known as software end-stops).

Variable estimation
Information of position sensors on the joints can be used to estimate other useful information such as joint velocities, torques etcetera.

Robot status
This function block collects the information provided by the sensors on the status of the robot (e.g.: its current posture). This information can be requested by other function blocks.

3.3 Functions mapped on layered structure for priorities

Each function described in the previous section has a certain extent of real-time requirements. Since the software is running on a single central processing unit (CPU), computational time needs to be shared.

In digital control theory, real-time is an important issue. Real-time is spread over a range of non- and hard-real-time. Between these boundaries there is the area of soft-real-time. Hard-real-time stands for: not allowed to be to late, and soft real-time for: try not to be to late. Sharing of the CPU is done by means of processes with priorities. Software with hard-real-time requirements should have the highest priority.

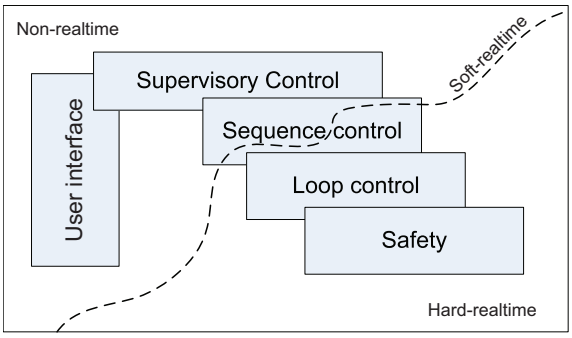


FIGURE 3.2 - Layered control structure

The Control Engineering group has experience with real-time control applications. The control processes should be divided over the range of hard and soft real-time. To make this visible a layered priority structure is used, as proposed by Broenink et al. (2007) and depicted in Figure 3.2. It consists of several control layers in order to indicate the priority of the control processes.

The function structure of Figure 3.1 have been mapped into the layered controller structure, which resulted in the new structure as depicted in Figure 3.3. The safety and loop control are hard-real-time, because missed deadlines may result in unstable control actions for the joints or in a system failure. It includes the function blocks: *Safety*, *Variable estimation* and *Joint control*. The sequence control configures the loop controllers and includes the *Basic behaviors* and *Robot status*. It takes care for the movements and thus the stability of the robot (that it does not fall), therefore it is soft-real-time. The supervisory control and user interface consist

of the function blocks *Decisions*, *World modeling* and *Communication*. These functions do not influence the stability of the robot and therefore they are non-real-time.

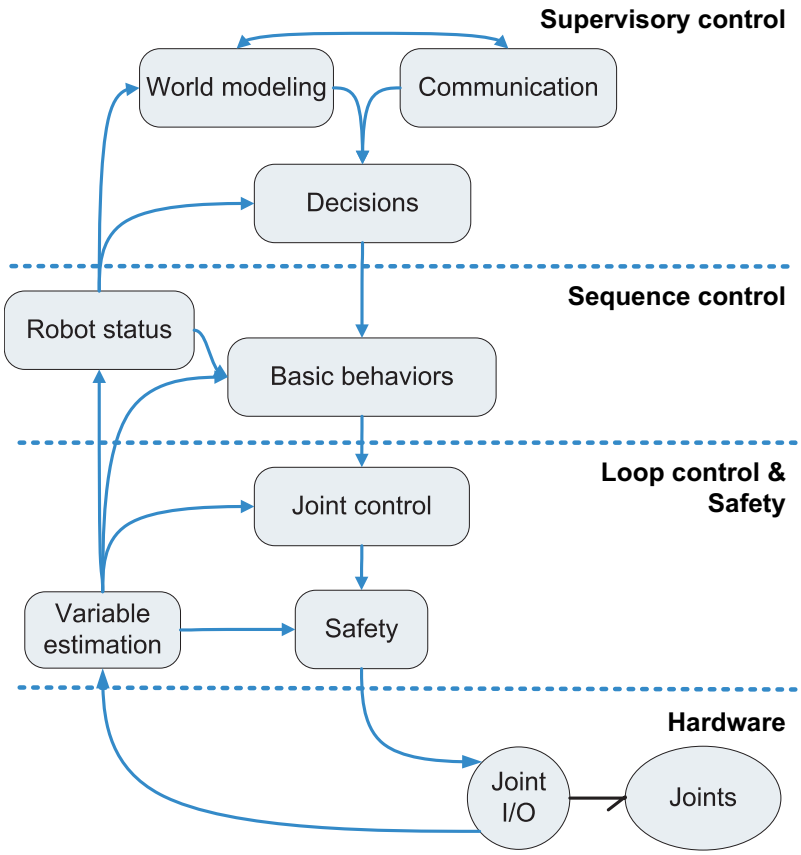


FIGURE 3.3 - Function structure mapped onto the layered control structure

3.4 Conclusions

In this chapter a software architecture that complies with the analysed requirements is proposed. This modular software architecture consists of function blocks in which all required functions are implemented. The function blocks have been mapped onto a layered structure to divide them in real-time priority layers.

The other requirements: reusability, light weighted and easy configurability depend on the software implementation. The architecture and the software requirements, will be used as a guideline for choosing an appropriate software framework solution. This will be discussed in Chapter 4

4 Software framework analysis

As mentioned in Section 3.1 one requirement is that the software should be reusable. Reusability of software can be achieved by using a framework, as described in Section 2.3.

As explained in Section 2.2, software from previous projects are not useful as a framework for TUlip. However, various third-party software frameworks especially for robots are available. For this assignment, several frameworks have been investigated for the usability in TUlip. In this chapter, the results of this investigation is discussed.

4.1 Comparison of third-party robot software frameworks

A literature study have been performed on the capabilities of the commonly used robot software frameworks, which are: Alliance, CLARAty, Miro, Orca, Orocos, Player, RoboFrame, URBI and YARP. In order to create a software architecture as given in Section 3.3, the main goal was to find an open framework with middleware that gives the opportunity to create executable modules that can communicate with each other. By investigating the software structure and the used techniques of each framework, as described in Appendix A, two possible candidates remained for use in TUlip:

- Orocos (Bruyninckx, 2001)
- RoboFrame (Petters and Thomas, 2005)

As described in Appendix A.3, the capabilities of these frameworks have been compared in detail. The result of this comparison is listed in Table 4.1.

TABLE 4.1 - Comparison list of RoboFrame and Orocos

Requirements	RoboFrame	Orocos
Multi layered structure	+	+
Light weighted	+	-
Inter module communication		
- local	+	+
- through network	+	+
Real-time	-	+
Open	+	+

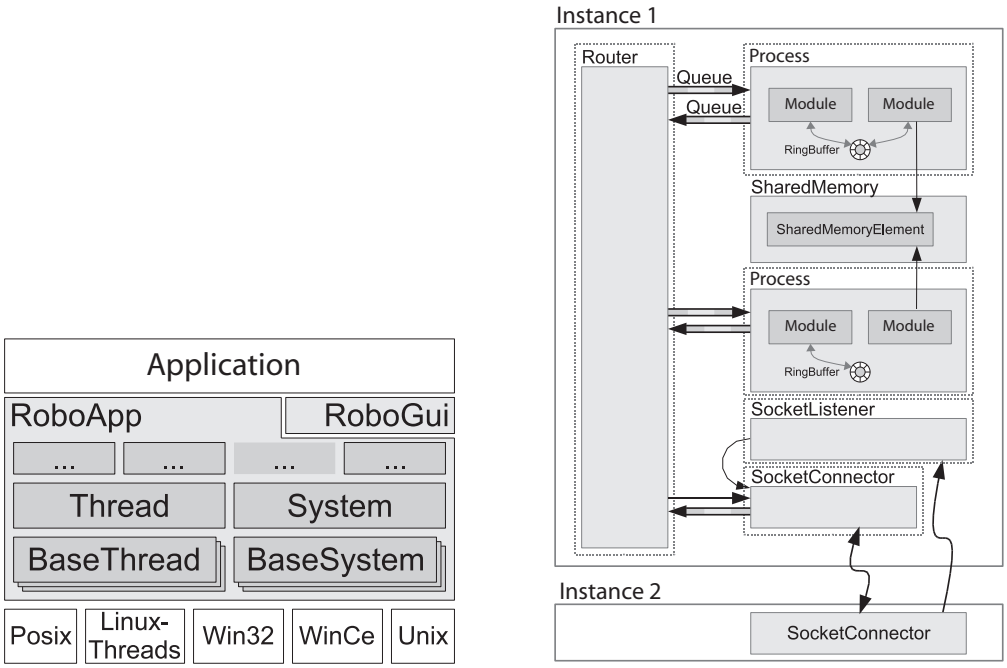
From the nine investigated robot software frameworks, Orocos is the only framework which supports real-time, however it is known to be resource consuming. This is based on the projects where Orocos is used and from information from their mailinglist (2008). As stated in Section 3.1, one of the requirements is that the software needs to be light weight, in order to fit on the hardware. Therefore, Orocos has not been chosen to be used as a framework for the TUlip software.

Although RoboFrame does not initially support real-time, it seems to be the only appropriate third-party framework, that can be used for the TUlip software. RoboFrame is discussed in more detail in the next section.

4.1.1 RoboFrame

RoboFrame is a modular software framework for lightweight autonomous robots, created by the Technical University of Darmstadt (2008). RoboFrame started as a master thesis project of Petters and Thomas (2005). It is developed especially for the soccer robot competitions, basically for statical actuated small robots like Aibo from Sony. RoboFrame focuses on serving an infrastructure between several modules, such as: Vision, Behavior, Communication and Actuation.

As depicted in Figure 4.1a, RoboFrame is a framework that allows the programmer to create a modular application, which is platform independent. The RoboApp library can be used for the actual robot application, which is build from modules (depicted in Figure 4.1a as blocks with 3 dots). RoboGUI is a QT (2008) graphical interface library, to create a user interface that can look into the process status. RoboFrame offers logging and configuration functionality.



(a) Abstraction layers. The blocks with 3 dots are custom modules.

(b) Example of implementation

FIGURE 4.1 - Structure of RoboFrame

Figure 4.1b shows an example of implementation. The object which is called a *process*, is a thread. A process can consists of multiple modules, which can use global data objects to communicate among each other within one instance or through a router even among modules in different instances. Modules are C++ classes where its functions can be triggered by the scheduler or a change in its connected data objects.

Sharing data among several modules can be done in two ways. One way is to store data in shared memory, the so called *Blackboard*. In that case the data object needs to be locked for writing and reading. Data on the blackboard can be accessed by multiple readers and writers. The second way is a *Data Buffer*, which is a one-to-one data exchange and is based on the ring-buffer principle. The ring-buffer has a predefined number of elements and can handle besides standard data types also custom data types (which within RoboFrame are called *Representations*).

Furthermore, data can be exchanged between different instances which might be on the same computer or connected via a network. Each instance has a router which can be reached via

a socket connection. In this way it is possible to create user interface application (e.g.: with RoboGUI) which can read and even adjust data within the robot application.

Furthermore it provides services for:
- data logging
- configuration file interpreter
- finite state machine engine (with use of XABSL (2008))

4.2 Software framework choice for TUlip

To use RoboFrame for all the software function blocks, it also needs to provide real-time support for the time critical parts. RoboFrame initially does not support real-time. However it has a well defined OS abstraction layer, which makes it possible to port it to a real-time OS. It is expected that this porting cost a lot of time to implement and to evaluate. Another option, which is less time consuming, is to implement a self-written real-time module for the time critical parts. These are the sequence control, joint control and safety layers as described in Section 3.3. RoboFrame can be used only for the supervisory priority layer. This separation is depicted in Figure 4.2. Because of the limited time resources, this option have been chosen to be implemented.

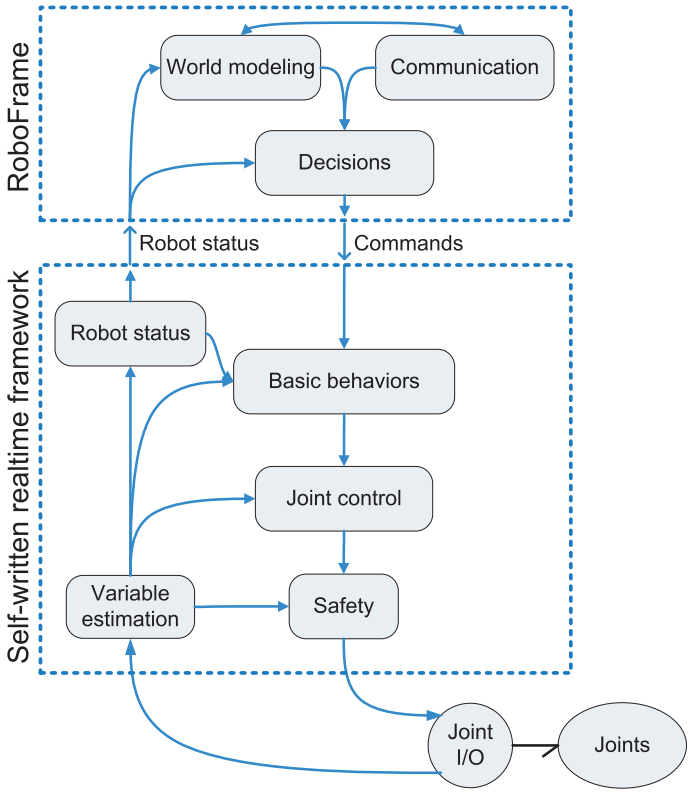


FIGURE 4.2 - Use of software frameworks in TUlip

4.3 Real-time operating system

In order to execute real-time software on the computer stack of TUlip, a real-time OS is needed. Several real-time OS's are available. Since RoboFrame has a OS abstraction layer which supports windows and POSIX OS's, it leaves the choice open for any real-time OS. With many projects at the Control Engineering department the real-time operating system Linux/RTAI (2008) have been used. It seems to be sufficient for projects with similar hardware and timing requirements. RTAI is known to have a conservative attitude to be fully compatible with older versions. It lacks support and documentation as well. A spin-off of RTAI, called Xenomai

(2008), known before as RTAI Fusion, has a progressive development and a large developers community. Xenomai has several skins to be compatible with other popular real-time OS's.

Is has been chosen to use Linux/Xenomai, because a lot of support is available and is easy to implement. However, Xenomai can only be used when the timing performance are satisfying the requirements. In order to evaluate the timing performance of Xenomai, a jitter test has been carried out, as explained in Appendix C. Jitter is the unwanted variation in periodical timing. It is the difference between the scheduled time and the actual execution time. For hard-real-time the worst-case jitter is important, however, no timing constraints are yet available for the function blocks which need hard-real-time. The timing constraints are determined by the digital controllers , which are expected to run at a frequency of $1kHz$. Since the digital controllers are not designed yet and not timing constraints are available, it is chosen to use sufficient criterion: as reference, worst-case jitter should not exceed 1% of the periodic time .

On the computer stack of TUlip, a periodical task is scheduled at $1kHz$ and the jitter is measured while the system was fully loaded, as described in Appendix C. From the results of the test, Xenomai have been found to be suitable because the worst-case jitter stayed below $7\mu s$, which is below 1%. With this result, it proven that Xenomai can achive the required real-time behavior and thus can be used.

4.4 Conclusions

In this chapter, RoboFrame and Orocos are analysed for usability in TUlip. From this analyses, it has been decided to use RoboFrame because of its modularity and lightweight characteristics. Since RoboFrame does not have real-time support, it has been chosen to implement a self-written real-time module for the time critical function blocks. In order to provide real-time support, a real-time OS is needed. After a short analysis, Xenomai is chosen to be used because of its good support and easy implementation. With a short test, it was proved that the jitter of a periodic task, running on Xenomai, is within acceptable bounds.

This concludes the design of the global software architecture, which is the aim of the first part of the assignment. The second part of the assignment is to design the joint controller framework, which consist of the analysis and design of the functions in the loop control & safety layer. This will be discussed in the next chapters.

5 Analysis of joint controller organization

This chapter focuses on the analysis of configuring the joint controller by the basic behaviours. These two function blocks are part of the software architecture as discussed in Section 3.2. In Figure 5.1 the software architecture is represented with a more detailed view of the joint controller and basic behaviours blocks.

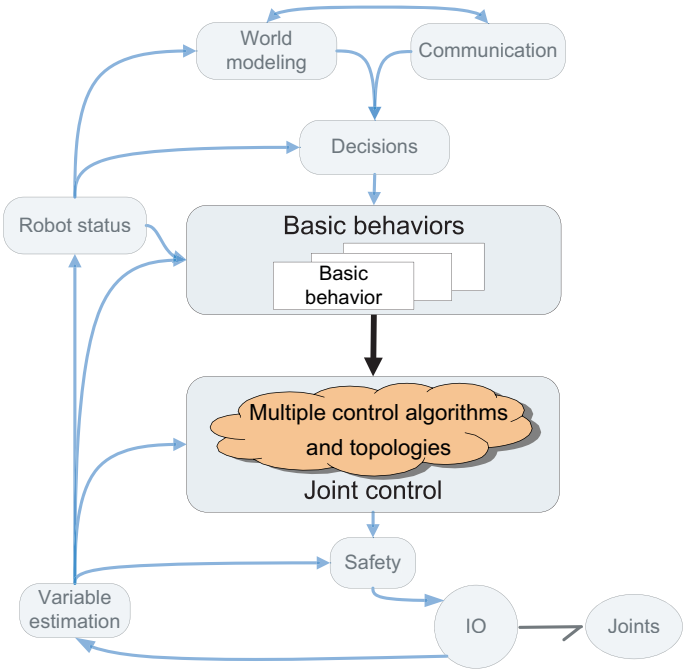


FIGURE 5.1 - How the controllers have to be organized

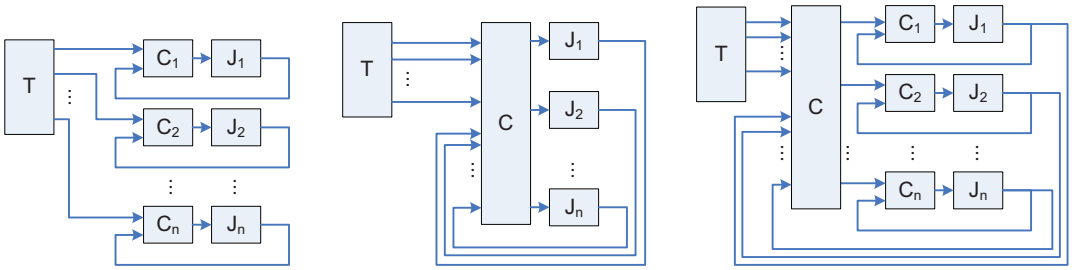
The basic behaviors block, consists of multiple basic behaviors. Each basic behaviour represents a certain movement like walking or kicking. In order to perform these movements, the different joints have to be actuated. This actuation will be controlled by the joint controller function block. Each basic behavior configures and activates the necessary controllers to perform the requested movement. How the controllers have to be organized to be fully adjustable by the basic behaviors, is the question which this chapter tries to answer.

5.1 Multiple controller configurations

In general most control problems are solved by designing a single-loop controller for an often linearized plant. Such a single-loop controller has one mode: it contains one (optimized) algorithm to make a joint reach a certain setpoint (e.g. a desired torque, angular position or angular velocity). However, in a multi-joint, multi-mode system such as TUlip a lot of interaction among the joints exist and the required controller response can differ per situation.

From the software requirements, as discussed in Section 3.1, it needs to be possible to support multiple control algorithms and topologies, so that the basic behaviors are not restricted by a single controller algorithm. Each basic behavior might have its preferred control algorithms, and some basic behaviors need two types of controllers (e.g. walking motion of a robot, called Flame, implemented by Hobbelen (2008), which switches between position and torque control). Also different controller algorithms can have different topologies. Three commonly used topologies are shown in Figure 5.2. Figure 5.2(a) shows a topology for a multi joint system, with independent control algorithms per joint, it is an array of the single-input-single-

output (SISO) controllers. In order to compensate for interaction among the joints a multi-input-multi-output (MIMO) controller can be used. In Figure 5.2(*b*) such a MIMO topology is depicted. It is possible to combine these two topologies as shown in Figure 5.2(*c*).



(a) SISO (b) MIMO (c) MIMO and SISO cascade

FIGURE 5.2 - Examples of several common used controller topologies. T – motion trajectories; C – control algorithm; J – joint actuations.

To be able to support multiple control algorithms and topologies, the controller organization should be as generic and flexible as possible. Besides these requirements, the requirement of light-weighted software has to be kept in mind for designing a suitable controller architecture. Furthermore, parameters and setpoints have to be set by the basic behaviors.

Three different options for organizing the controller algorithms are considered and evaluated on flexibility and generality:

- Configurable pool of controllers: The basic behavior can configure the joint control per joints to use certain algorithms and topology through an interface. As long as the system needs only SISO controllers, it is straight forward. The configuration would be: per joint a selection among several algorithms. However, as soon as combinations of MIMO and SISO controllers in different topologies are needed, the configuration becomes complex. It is highly generic, although it will be difficult to implement MIMO control topologies, this reduces the flexibility.
- Selectable sets of controllers: The basic behavior can select among different predefined sets of algorithms and topologies per system. It reduces the complexity compared with the previous alternative. However, in this case it is not possible to change the algorithm for a single joint with changing the controllers for all the joints.
- Controllers per basic behavior: Each basic behavior has its own customized control objects which implements a certain algorithms, in this way it can be totally on the demands of the basic behavior. The active basic behavior gives the pointer of its controller objects to the joint controller. The joint control can call a special function of the control objects in order to compute the controller algorithms per joint. This option is general and flexible, because the control objects can be totally customized per basic behavior.

It is chosen to design and implement a controller framework with use of control objects, because it is a flexible and generic solution in order to organize controller algorithms, fully adjustable by the basic behavior.

5.2 Principle of control objects

This section explains the basic principle and the use of the control objects. In Figure 5.3 the function blocks basic behavior and joint control are depicted in detail. The control objects are represented by the Joint 1, 2 ,3 blocks.

The joint control has for each joint a controller interface. Such an interface is a reference to a custom joint controller in the active basic behavior. Although the joint controller interfaces are per joint, it is possible to create custom MIMO controller objects, which simply serve mul-

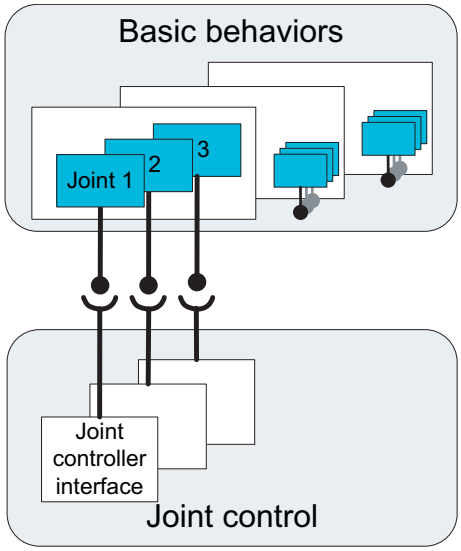


FIGURE 5.3 - Customized joint controllers per basic behavior, called: control objects.

tiple interfaces. The basic behavior adjust the custom joint controllers by setting parameters and setpoints. On the other side, the joint controller calls a function of the joint controller interface, which is associated with a custom control object, in order to compute the appropriate algorithm.

In Figure 5.4 a control object is depicted. The principle of a controller object is based on low level and high level controllers. As shown in Figure 5.2, controller configuration have certain topologies. These configuration can be split up in two levels, high and low. The distinction between low and high level is in general:

- *Low level* contains the control algorithm that is responsible for the dynamics of the joint and often realized as SISO PID controllers.
- *High level* contains the less time critical and CPU time consuming algorithms, such as setting parameters, generation of motion profiles and computation of a cascaded MIMO controller.

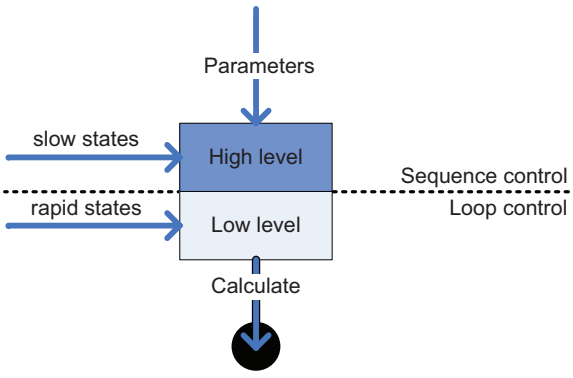
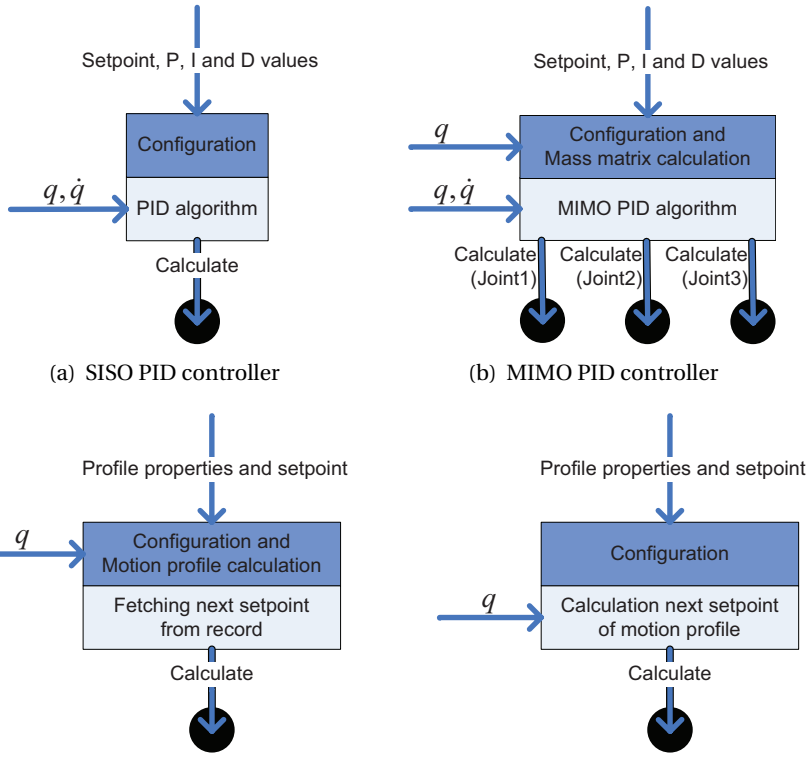


FIGURE 5.4 - A control object

In Figure 5.5 a few examples of loop control objects are given. The control objects in Figure 5.5(*a*) and (*b*) are PID controllers, where (*a*) is a single joint controller and (*b*) a multi joint controller. Figure (*a*) computes the output of the PID on a request from the loop control, depending on the parameters set by the configuration. Computing of a mass matrix, which is needed for a multi joint PID controller, is in general a resource consuming job. In (*b*) the mass



(a) SISO PID controller

(b) MIMO PID controller

(c) Motion generator, computed by the high level part

(d) Motion generator, computed by the low level part

FIGURE 5.5 - Example joint control objects. q and $\dot{q}$ are measured joint positions and velocities.

matrix will be computed by the high level part, requested by the basic behavior. As soon as the joint control request a computation of one of the joints, the low level part computes the output for all the joints depending on the mass matrix and returns the requested one. The remaining request of the other joints simply return the already computed values. Figure 5.5 (c) and (d) represents motion profile generators. Where in (c), the computation will be done by the high level and put in a list, so that the low level part can just fetch the next setpoint each time on a request of the joint control . In (d) the computation of the motion profile is done each time the joint control requests computation.

These object are just examples and can be implemented as needed.

5.3 Conclusions

In this chapter, the possible organizations of the joint controllers has been analysed and a concept for implementation the controllers is proposed, which complies to the stated requirements. It is chosen that each basic behavior has its own set of control algorithms for the joints, in the so-called control objects. It is chosen because it is a flexible and generic solution in order to organize control algorithms so that they are fully adjustable by the basic behavior. The joint control reach the control objects through a generalized joint controller interface, in order to compute the control algorithms. The principle and use of the control object concept are explained by examples, which show that it is possible to implement different controller algorithms in a generic way.

In order to execute the algorithms of the control objects, a execution engine is needed. The design of it will be discussed in the next chapter.

6 Design and implementation of the joint controller framework

The joint controller framework is the execution engine for the sequence control, loop control and safety layer, as discussed in Section 3.3. As described in Section 3.1, a requirement is that the loop controller and safety layer need to be executed hard-real-time. The sequence control layer has the requirement to be executed soft-real-time.

This chapter explains how the software is executed, and how the data is exchanged between the layers.

6.1 Time driven tasks with data storages

All the function blocks of the joint controller framework have to be implemented with use of a self-written real-time module, as discussed in Section 4.2.

Since controllers need to be executed periodically and the context of the basic behaviors (which is outside of the scope of this assignment) needs to be executed periodically, it is chosen to create periodical executed software components, which are called *tasks*.

Each tasks has a priority. To achieve hard-real-time for tasks that implement function blocks that require hard-real-time, these tasks are given a higher priority than the tasks with function block that require soft-real-time.

In order to reduce complexity and overhead due to communication mechanisms, the goal is to create a minimum number of tasks. Therefore, all functions with the same timing requirements are handled by the same task. Timing requirements are the period time and priority. For instance, the data loop of: reading sensors, calculate the joint controllers and actuating the joints are all handled sequentially in one task grouped by joints with the same sampling rate.

The function blocks within the different tasks, need to be able to configure each other and exchange data. Since the tasks have different priorities and different period time, it is undesirable to couple the tasks for data communication. Therefore it has been chosen to use asynchronous communication. Communication is realized by shared memory in the so-called *data storage*. The concept of data storages is that only one task is responsible for one data storage. It is based on the concept that each function block in Figure 3.1 is responsible for one set of data, while multiple function block might use the data, as shown in Figure 3.1.

In Figure 6.1, a *task* and a *data storage* are shown. The arrows indicate data transfer. Each data storage can only be written by one task. The task can update the storage by processing data from other data storages or external data. Each data storage can be read by a number of tasks.

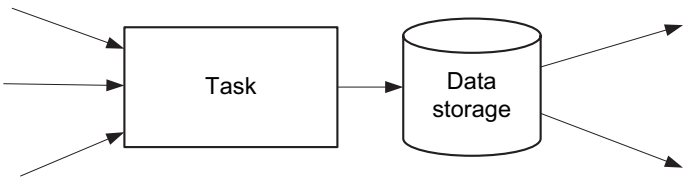


FIGURE 6.1 - A task responsible for a data storage

Figure 6.2 gives a representation of the tasks and data storages within the real-time software module. It consist of three tasks:

- Supervisory control interface
- Behavior engine

• Control engine

which correspond respectively to the control priority layers supervisory, sequence and loop control.

The task *supervisory control interface* communicates with the supervisory software modules written with RoboFrame. It provides the robot status data for the supervisory software and receives commands to perform certain the basic behaviors. The implementation of this task is outside the scope of the assignment and will not be discussed any further.

Each basic behavior consists of a few motions. These motions are processed by a hierarchical state machine, where each state performs a motion. The state machine transitions and the state contents are periodical executed by the task *basic behavior engine*. The state contents configure and execute the high level of the control objects, as described in Section 5.2.

The low level algorithms of the control objects are executed by the task *control engine*. Furthermore, the control engine performs the function block safety, the function block filtering and communication with IO of the robot hardware as well.

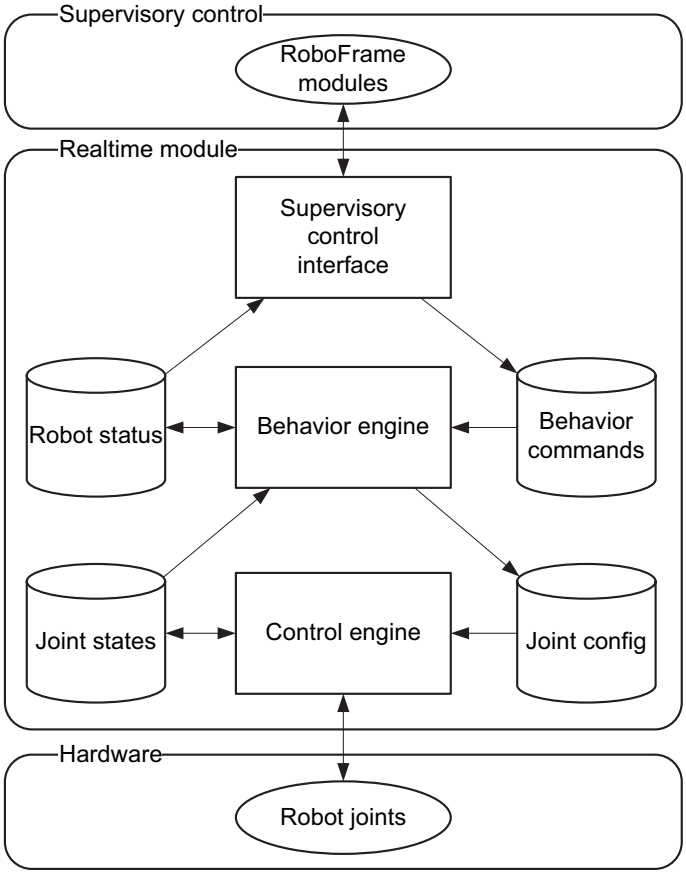


FIGURE 6.2 - Tasks and data storages within the real-time software module

6.2 Execution of the control objects

As mentioned by Visser et al. (2006) two different control approaches exists within digital control. The so called 'sample-compute-actuate' and the 'sample-actuate-compute' in which the names refer to the order of the loop control processes.

In the control engine, it is not known what the computation time of the control objects will, because the executing code can change. Because of the variating computation time, jitter might occur when using the common 'sample-compute-actuate' approach, as shown in Figure 6.3.

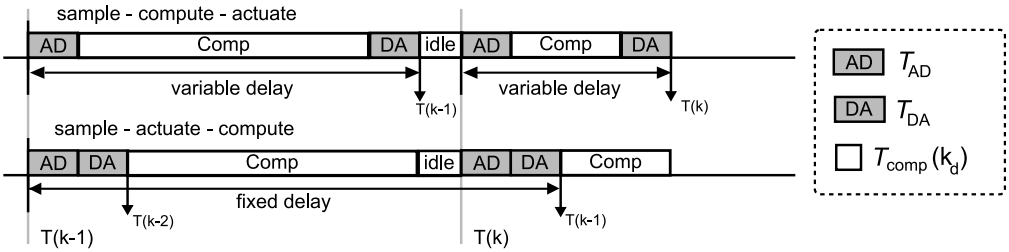


FIGURE 6.3 - Timed-control implementation approaches (from Visser et al. (2006))

As suggested by Visser et al. (2006), the 'sample-actuate-compute' approach has been implemented.

Figure 6.4 shows the sequence diagram of the control engine task. On each periodic trigger, the joint sensors are read first. The information is stored in the *Joint states* data storage. The data is filtered, in order to to estimate states (e.g.: joint velocities). This information is stored in the Joint states as well. The function safety checks if the current joint states are not out of their boundaries and that the actuation signal is appropriate. If it discovers that something is wrong (e.g.: joint exceeded a software end-stop, or a maximum velocity), it can perform a graceful degradation (Cooling, 2000) instead of failing the whole system. The actuation signal, checked by safety, is written to the actuators. These performed functions have practically a fixed execution time. Also, they can not be interrupted, since they are executed in a real-time tasks with the highest priority. In this way the jitter of actuation is tried to be minimized. The algorithms of the control objects are computed in the spare time of the execution period. It calls for each joint the joint controller interface, which refers to a customized controller object. The lower part of each control object can reach the joint states in the data storage which are updated in the beginning of this sequence.

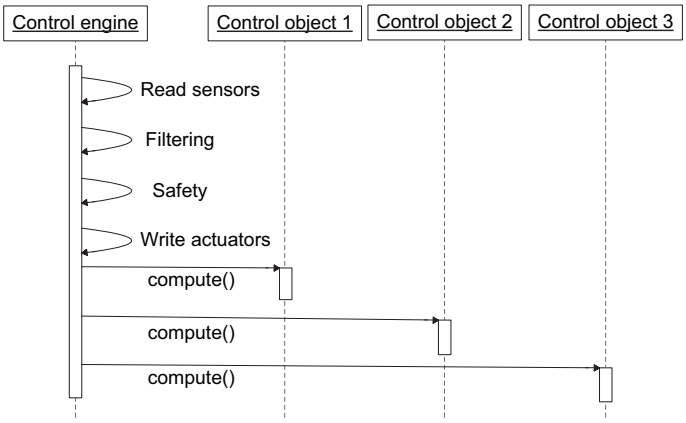


FIGURE 6.4 - The sequence of functions executed by the control engine

In Figure 6.5, the sequence diagram of the behavior engine task is shown. The behavior engine task updates the state machine according to the joint and robot states. The context of the current state is executed, which configures the control objects. The higher part of each control object can reach the joint states and the robot states in order to perform it configuration algorithm.

6.3 Implementation

A prototype of the joint control framework has been made. It consists of two real-time tasks,

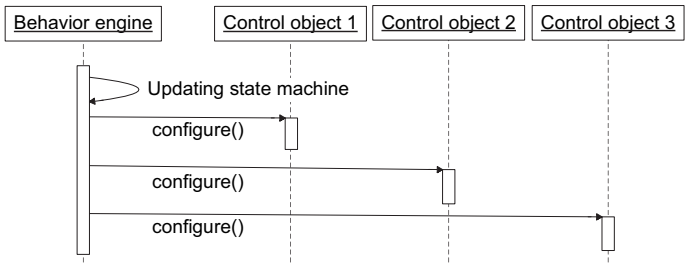


FIGURE 6.5 - The sequence of functions executed by the behavior engine

the basic behavior engine and the control engine.

In order to create real-time tasks easily, an object-oriented abstraction layer for the Xenomai application programming interface has been created, as discussed in Appendix B. The communication through data storages are realized by shared memory.

Implementation of two controller configurations have been realized, as discussed in Chapter 7. The classes used in the prototype are depicted in Figure 6.6. The prototype is focused on execution of the control objects and the functions safety and IO are not implemented yet. For both controller configurations, the basic behavior engine is composed of control objects. At a periodic trigger, it executes the high level algorithms (configuration) of the control objects. The control engine is composed of control object interfaces, which are associated with the actual control objects. On the periodical trigger of the control engine, it calls for every control object interface the `compute()` function, in order to compute the low level algorithm of the associated control objects. The control objects can make use of the joint states in their algorithms.

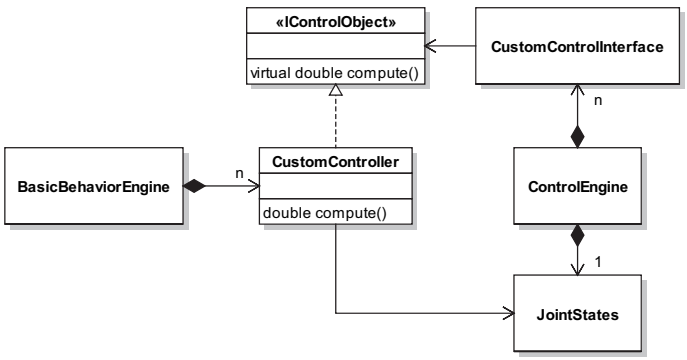


FIGURE 6.6 - Implemented classes of the joint controller framework prototype

6.4 Discussion on the extension of features

Since the implementation of the joint controller framework is in a experimental phase, two not yet fully implemented features will be discussed.

Non-blocking shared memory

Data exchange with use of the data storages is implemented as shared memory. Different tasks might perform a read or write request to the same data at the same time. Without a protection mechanism it might result in corrupted data.

Protection mechanisms can be 'blocking' or 'non-blocking'. Blocking shared memory is used commonly, a well-known example is mutual exclusion (Dijkstra, 1965). It is currently used for the shared memory.

In a task with hard-real-time requirements, it is not desirable to wait for data that is locked by another task. When a higher priority tasks is blocked because a lower priority thread has locked the data, it is called priority inversion. Although blocking priority inversion can be solve by priority inheritance (the lower priority task inherits the priority of the highest priority task), it is highly undesirable because it decreases the predictability of the timing of the higher priority task. Especially, when

- the data is large
- locks are performed inefficiently
- data is locked frequently

In these cases it is desirable to use non-blocking shared memory.

In Soetens (2006b) an principle for non-blocking shared memory is described and evaluated. A simple and clear picture is given in the presentation of Soetens (2006a). Basically it is having multiple copies of the data, in such a way that it is always possible for a writer to find 'free' memory. As soon as a writer has updated the data, it marks the data as 'most recent'. A reader always reads the most recent data.

The principle, as explained by Soetens (2006a), is for many-writer-many-reader setups. However, the maximum number of readers and writers should be known in order to allocate enough memory, which grows linear with the number of readers and writers.

In case of hard-real-time threads sharing
- a large data object
- frequent read data

it is worth to consider using non-blocking shared memory.

It can be useful, for instance, in a MIMO control object, where the high level computes the mass matrix and the low level need to access the mass matrix in order to calculate its algorithms. Since the computation of the mass matrix can be time consuming, with blocking shared memory it should – calculate the mass matrix locale – lock data – copy – unlock data – in order to minimize the time of locking the data. While with non-blocking shared memory it can – calculate mass matrix to free buffer – mark buffer as most recent –. The second case does not have any extra overhead, while the benefit is that there is not chance on priority inversion.

Switching between loop controllers
Switching among control objects creates a potential undesired effect and in worst case instability. In case of switching to a control object with an algorithm that uses memory, it could be initialized. For example switching to, for instance, a PID controller, which has one integrating action, it could be for initialize by the reverse calculation of the previous state, setpoint and actuation.

This is a hypothetical solution to a potential problem, it had not been proved nor tested. It only have been an trigger for implementing the possibility for initialization of controllers by the previous output before switching.

6.5 Conclusions

In this chapter, the design of the joint controller framework is discussed, which consists of real-time tasks in order to execute the function block with real-time requirement. In order to implement the data flow as in Figure 3.1, data storages have been implemented. Only one task is responsible for a data storage, while the others can read from it. The data storages have been implemented as shared memory in order to reduce overhead, compared to message based data exchange. Furthermore, the actuation of the hardware is done with the sample-actuate-compute approach in order to reduce jitter.

The joint controller framework and the control objects will be validated by discussing the implementation of two controller configuration, in the next chapter.

7 Realization of test setups and control configurations

In this chapter, the realization of two test setups is discussed. For the test setups two different control configurations have been used.

The test setups have been create in order to:
- test development iteration of the joint controller framework,
- test the use of 'controller objects'.

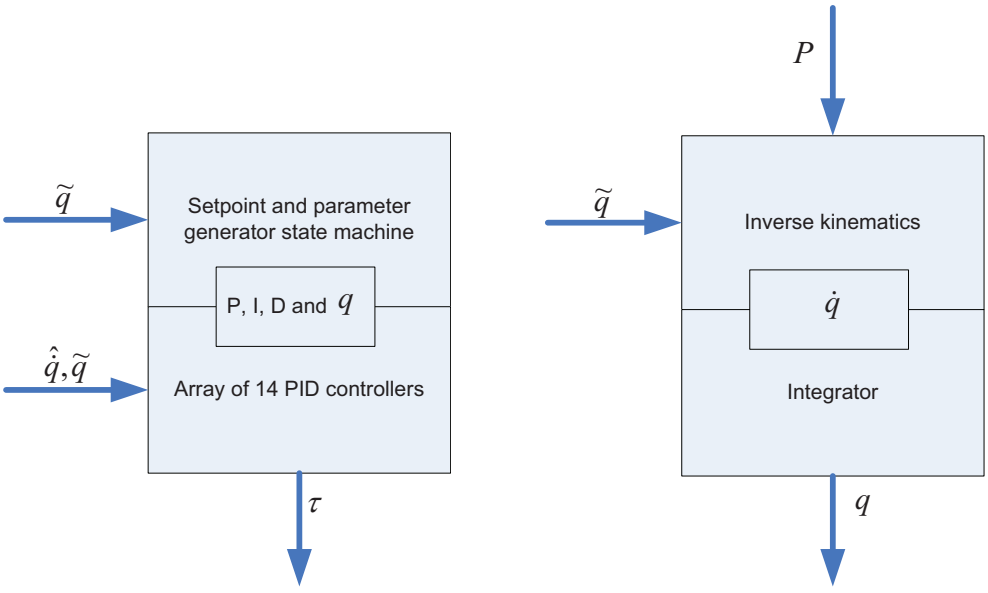
7.1 Implementation of the control objects

The use of control objects have been tested by implementing two controller configurations.

1 A controller configuration for the 12 joints of TUlip's body, in order to stand up from prone position. It was created by Daemen (2008) as a Master Thesis project. It consists of 12 SISO PID controllers with a sequence controller, which changes the setpoints and parameters of the PID controller according to the current state of the robot.

2 A controller configuration for a humanoid head, which is created by Visser (2008) as a Master Thesis project. The neck, head and eyes are connected to each other with 7 joints. The setpoint to the controller is a point in a three-dimensional space (e.g. location of an interesting object). By inverse kinematics it steers the joints in order to focus the eyes to that point.

These controller configuration were designed with use of 20-sim. Both controller configurations are split up in a high and low control loop, to fit in the concept of the control object (as designed in Section 5.2).



(a) Body: Get up from prone position.

(b) Head: Follow target with the vision system.

FIGURE 7.1 - Two controller configurations implemented in control objects. Explanation of the variables in the figures: P, I and D – controller parameters; q – desired joint position; $\tilde{q}$ – measured joint positions; $\dot{q}$ – desired joint velocity; $\hat{\dot{q}}$ – estimated joint velocity; τ – desired joint torque; P – 3D setpoint (target position). The variables in the erea between are stored in shared memory of the control object, it is the configuration of the low level algorithm.

The controller object created for the first controller configuration is shown in Figure 7.1. The higher level of the controller object consists of a state machine, which depends on joint posi-

tions ($\tilde{q}$). It generates parameters (P, I and D) and setpoint (q) for PID controllers in the lower level. The PID controllers compute the torques (τ) for actuation, according to the current joint positions ($\tilde{q}$) and velocities ($\hat{\dot{q}}$).

The second controller configuration is implemented as shown in Figure 7.1b. A target position (e.g.: gathered by the vision system) is the setpoint for the higher level controller. The desired joint velocities for the joints ($\dot{q}$) are computed by inverse kinematics and proportional control. The lower level is a simple integrator to obtain the desired joint positions (q).

Both controller objects have been implemented.

7.2 Testing the joint control framework with a model of the robot

Unfortunately, at moment of implementation of the controller objects, the robot was not operational yet. To bypass the missing robot, simulation models of the robot have been used in order to test the controller implementations.

Both the controller configurations mentioned in the previous section are developed and tested with use of models in 20-sim. These models are used in the test setup.

7.2.1 Computation of the robot model

In order to test the real-time framework, the model should be computed in real-time as well. This can be realized by Hardware-in-the-Loop (HIL), as described by Visser et al. (2004).

A problem is that the computation of the given models on standard lab computers take longer than the simulated time. In case of the model of the body, a modern desktop computer computed $12ms$ for a model time of $1ms$. In order to use these models for testing the real-time setup, a solution need to be found. Possibilities are:

1 Simplify the model and code
2 Use more computational power
3 Simulation of the framework

Both the models are designed with the 3D Mechanics Toolbox of 20-sim (2008). It has been chosen to not simplify the model, because it would require a revalidation of the model. However, the C-code generated by 20-sim could be optimized, which resulted in a computation time of $9ms$, but it is still –of course– not real-time. For the second option, in order to compute the robot model real-time, a system should be used that is capable of compute ten time faster then a modern desktop computer. It is possible, for instance, to increase the computational power by using a cluster of computers. The third possibility is to do a step back, instead of testing the framework by HIL, to test the framework by simulation. Since the second option would require a lot of effort of setting up a HIL system, it has been chosen to test the framework by simulation. Simulation of the framework can be realized by modifying it in such a way, that it is possible to suspend the whole system. The feature of suspending the real-time tasks have been implemented and it is described in Appendix B.2. In Figure 7.2, the timing diagram of the basic behavior and control engine tasks is depicted. On the vertical dotted lines, the tasks get suspended in order to compute a time interval of the model, after it the system resumes again. In this way the real-time relation between the tasks is maintained. Compared with HIL, simulation of the framework has the advantage that no second computer is needed for computation of the model, because the computation of the model can be done at the stack as well.

7.3 Test setup

In Figure 7.3 the test setup is depicted. The left box represent the PC-104 stack, which is the computer of the robot. On the PC-104 stack, the joint controller framework is running. Instead of actuating the hardware, the model of the robot is computed on the stack as well. While computing the model, the real-time tasks are suspended. A separate non-real-time thread contains a socket server, which reads on each request the model states and sends it over a network con-

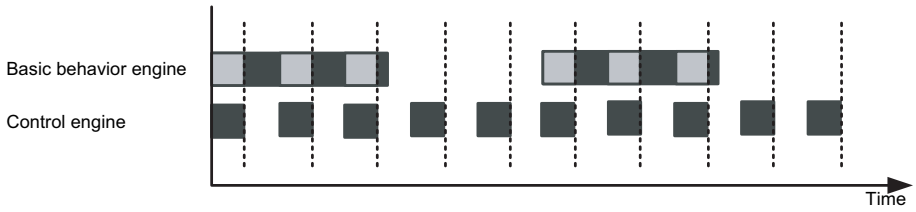


FIGURE 7.2 - Timing diagram of the basic behavior engine and control engine tasks. The dark blocks represents the execution of the tasks, while the light gray blocks represents preemptions of a task.

nection to the second computer, depicted as the right box. On the second computer a instance of 20-sim is running. Through a customized DLL, the data can be requested and brought into the 20-sim simulation environment. Where the 3D visualization is the representation of the model states. In this way 20-sim can monitor the states of the model and is able to start and stop the joint controller framework.

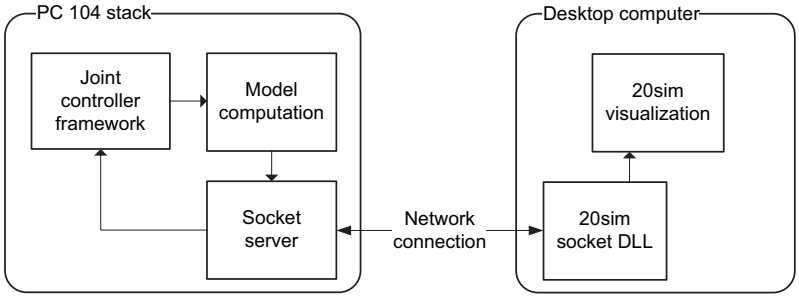


FIGURE 7.3 - Overview of the test setup

Both controller configurations have been tested with this setup, they will be further described in the next sections.

7.3.1 Joint controller framework with model-in-the-loop: Body

As shown in Figure 7.4, the joint controller framework exists of two asynchronous real-time tasks: Behavior engine and control engine. Which are periodical executed at $100Hz$ and $1kHz$ respectively. The behavior engine executes the higher level of the control object, which contains the setpoint and parameter generator, implemented as a state machine. The lower level of the control object is executed in the control engine. From the generated parameters (P, I and D) and setpoints, it computes the PID control algorithm. Instead of actuating the hardware, the controller 'actuates' the model, by computing $1ms$ of simulation. While the model is computed, both real-time tasks are suspended. The joint positions q from the model are the feedback signals for the PID controllers and motion controller. On request of the 20-sim application, the transformation matrices H of the limbs from the model and sends it over a network connection to the second computer, where the 3D visualization is the representation of the transformation matrices form the model.

7.3.2 Joint controller framework with model-in-the-loop: Head

The second setup, as shown in Figure 7.5, is similar to the first, but instead of controlling the humanoid body it controls a humanoid head. In this case the joint controller framework consists of three real-time tasks. The extra tasks reads the joystick device in order to manipulate a 3D target position. This target is the reference for the inverse kinematics controller, which tries

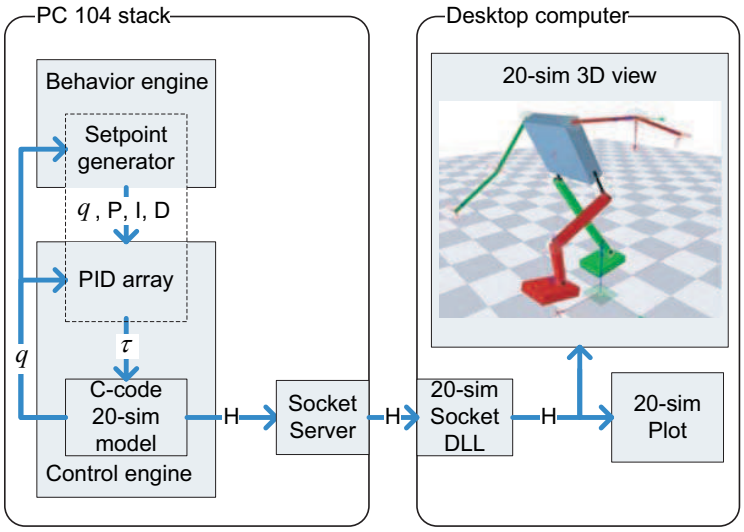


FIGURE 7.4 - Joint controller framework with model-in-the-loop: Body

to focus the 'eyes' to the target. The control engine integrates the output of the inverse kinematics by executing the lower level of the control object. The tasks are periodical executed at $10Hz$, $20Hz$ and $1kHz$ respectively. Instead of sending the transformation matrices, only the joint positions are send to the visualization computer. The visualization computer computes locale the transformation matrices H in order to visualize the model.

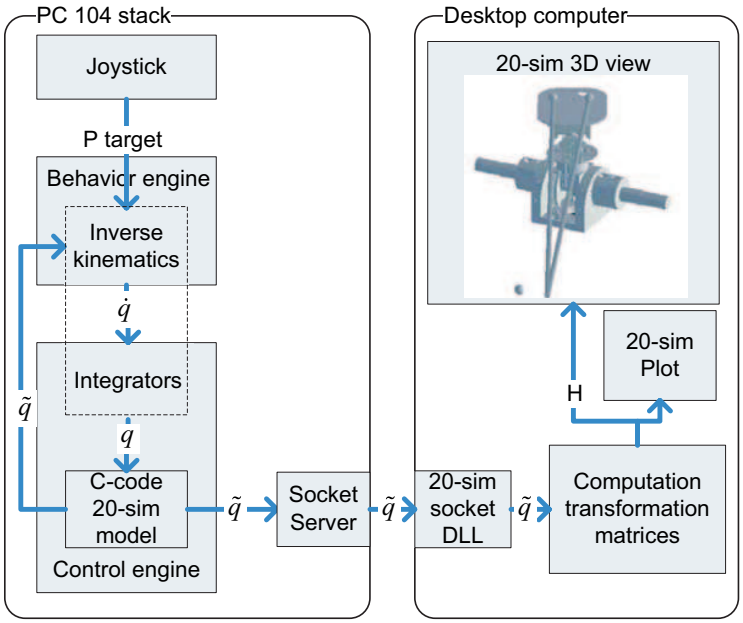


FIGURE 7.5 - Joint controller framework with model-in-the-loop: Head

7.4 Results

Both control configurations are implemented. Unfortunately, the C-code generated from the model of the body generates errors as soon the actuation of the joints become to large. These errors occur because parts of the model are based on the timing of zero-crossings, which can be easily determined by variable step algorithms, but not by the fixed step algorithms that are available for the C-code generation in 20-sim. In order to ommit the errors, several changes

have been made:
- Calculate the model at a higher frequency
- Tried different integrator algorithms for the model
- Weakening the controller parameters
- Interpolating the signals from the controllers to the model

However, even with these changes it did not succeeded to control the model in a proper way.

The head model is handled well by a fixed step integrator algorithm. The control configuration was set up with success. The 'eyes' of the head follow the with the joystick manipulated target. After the successful simulation of the framework with the head model, the framework is used to control the real head, as described in Visser (2008).

7.5 Conclusions

In order to test the joint control framework and the concept of controller objects, two controller configuration have been implemented. The controllers from 20-sim have been divided in a lower and higher controller part to fit in the concept of the control objects.

The concept of the higher and lower controller of the control object, allows to split up the controller in two parts. So that the hard-real-time control loop does not have to compute time consuming algorithms, which are able to run a lower sampling time. The concept of the control objects serves a high flexibility of combining control algorithms as well. Two examples are the two implemented controller configurations.

A simulation setup with models of the robot have been realized to perform the test of the software, because the robot was not present yet. The setup might also be used later when the robot is operational to prevent the robot from damaging caused by buggy experimental code.

The model of the body has not been successfully tested, because the C-code of the model did not work properly. However, the model of the head has been tested successfully in the framework simulation and the framework is used for the implementation of the head demo, as described in Visser (2008).

On the other hand, the test setup has been used to test iterations of the joint controller framework, in order to improve and add functionalities.

8 Conclusions and recommendations

8.1 Conclusions

A software architecture for TUlip is proposed. This modular software architecture consists of function blocks in which all required software functions are implemented. The function blocks have been mapped to a layered structure to divide them in real-time priority layers.

Several third-party robot software frameworks are analysed for usability in TUlip. From this analysis, it has been decided to use RoboFrame because of its modularity and lightweight characteristics. Since RoboFrame does not have real-time support, it has been chosen to implement a self-written real-time module for the time critical function blocks.

Xenomai is chosen to be used in order to provide real-time support, because of its good support and easy implementation. With a short test, it is proved that the jitter of a periodic task, running on Xenomai, is within acceptable bounds.

A controller architecture is proposed which complies to the stated requirements. It is chosen that each basic behavior has its own set of control algorithms for the joints, in the so called: Control objects. It is chosen because it is a flexible and generic solution in order to organize controller algorithms so that they are fully adjustable by the basic behavior. The concept of the higher and lower controller of the control object allows to split up the controller in two parts. So that the hard-real-time control loop does not have to compute time consuming algorithms, which are able to run a lower sampling time. In this way the load of the hard-real-time software part is minimized. The concept of the control objects serves a high flexibility of combining control algorithms as well.

A design has been made of the joint controller framework, which consists of real-time tasks in order to execute the function blocks with real-time requirements. A prototype of the joint controller framework has been made.

In order to test the prototype of the joint control framework and the concept of controller objects, two controller configuration have been implemented in the simulated framework setup. The model of the body has not been successfully tested, because the generated C-code of the model did not work properly. However, the model of the head has been tested successfully in the framework simulation and the framework is used for the implementation of the head demo, as described in Visser (2008).

8.2 Recommendations

8.2.1 Porting RoboFrame to support real-time behavior

Instead of continuing with two separate frameworks, it might be useful to port RoboFrame to a real-time operating system, in order to create a light weighted framework that simplifies creating real-time applications. The well-defined OS abstraction layer of RoboFrame allows relatively easily porting.

8.2.2 Investigation performance of Orocos

It might be interesting to do a full research about the timing characteristics of Orocos. From documentation of Orocos (2008), it can be concluded that it is possible to optimize the software for timing, by using the so-called ‘hard-coded deployment’. As can be seen in Appendix A.3, Orocos is more focused on general robotics than RoboFrame, since it has very useful libraries and support real-time behavior. However, it has not chosen to be used for TUlip, because the uncertainties about its resource consumption (which has not been proven nor unproven), because of limited time resources). Since the only drawback of Orocos is its expected resource consumption, it would be a huge improvement if it is possible to reduce its resource

consumption.

8.2.3 Switching between controllers while operational

For this assignment, a generic and flexible solution for organizing joint controllers has been proposed. However, it has not been tested what happen on switching between different controller configurations while the robot is operational. As mentioned in Section 6.4, a potential problem is undesired effect or even instability when switching between controllers, which might be solved by initializing the controllers by inverse computation of the algorithm. Furthermore, switching between loop controllers can cause instability. The timing of switching from loop controller should be chosen carefully in order to omit instability.

8.2.4 Testing new controller implementations by simulating the framework

A simulation setup has been realized to perform the test of the software, because the robot was not present yet. The setup might also be used later when the robot is operational to test the implementation of new controllers, to prevent the robot from damaging caused by buggy experimental code.

A Robot software framework to use for TUlip

In this chapter, several third-party robot software frameworks will be discussed.

A.1 Requirements for a robot software framework for TUlip

In the software architecture design, several layers are defined which implement different functionalities, as shown in Figure A.1 and explained in Section 3.3. The main requirement for a framework is the possibility to create such a modular and layered structure. The software that need to be created should be an executable that runs at the given computer stack and work with the given hardware. It is a challenge to implement all the functionalities discussed in Section 3.2 on the PC-104 stack, because of its limited resources. Therefore, the software framework (if used) should be light weighted. Among the modules, direct communication should be possible. The loop control layer which is responsible for the joint controllers is time triggered and requires hard real-time behavior in order to guarantee low latency for actuation of the joints. Another issue is the licencing of the software. It should be free to use and adjustable.

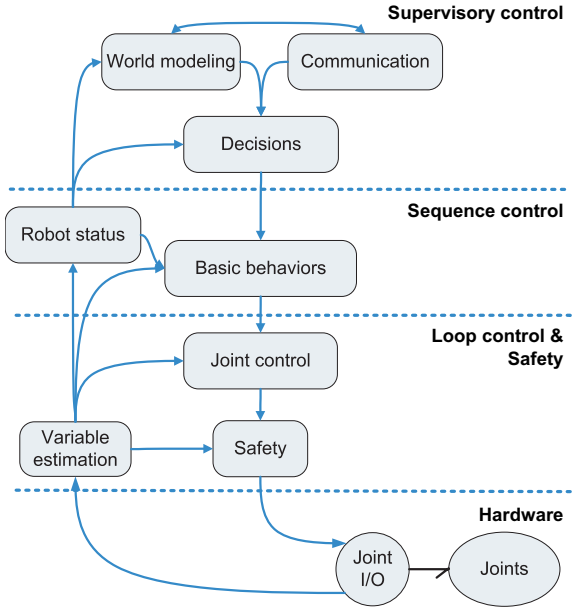


FIGURE A.1 - Software layers with responsibilities

The requirements that are used in order to make a selection from the available frameworks are summarized in the following list:

- Multi layered structure
- Light weighted
- Inter-module communication
- Real-time support
- Open

Furthermore additional requests were used to indicate the usability of the framework in global, these are:

- Scripting engine: To adjust software at runtime
- State machine engine: For executing state dependent software
- Library components: Already developed components that might be used
- Library of basic functions: For example mathematical function
- Examples available: Examples are useful to learn the capabilities of the software

- Well documented: To learn more about the software it should be well documented

A.2 Evaluation available frameworks

Popular robot software frameworks at the moment were Alliance, CLARAty, Miro, Orca, Orocos, Player, RoboFrame, URBI and YARP. In the following paragraphs the available robot software frameworks will be shortly discussed and evaluated according to the requirements.

Three very common used frameworks are *Yet Another Robot Platform* YARP, *Universal Robotic Body Interface* (URBI) and *Player*. These framework are similar to each other and respectively describes by Fitzpatrick et al. (2008); Baillie (2005); Vaughan et al. (2003). They all are sensor-actuator interface networks and are focused on reusability. That means that programs are able to connect certain hardware by connecting to a server. A lot of already made interfaces for common hardware does exist.

Middleware for Mobile Robots (Miro) is developed by the computer science department of the university of Ulm in Germany (Utz et al., 2002). It focused on distributed systems. All internal communication is through CORBA.

The Coupled Layer Architecture for Robotic Autonomy project (CLARAty) is the two layered software framework from NASA. It is a framework for reusable robotic components and it is focused on abstraction of devices.

Open Robot Control Software (Orocos) is a framework for robot and machine control (Bruyninckx, 2001). It was started as an EU-sponsored project of three universities to create an open and free framework. Eventually the work of three universities has never been merged. The project referred here is of K.U.Leuven in Belgium. Applications bases on Orocos are constructed from real-time components. Several already made components for common purpose are available. Furthermore two mathematical libraries are available which can be used within the real-time components.

Orca was initially part of the Orocos project, but it split-off and was renamed to Orca. It is part of the Australian Centre for Field Robotics now. The framework of Orca is highly modular and there exists a lot of reusable modules (Kaupp et al., 2007). In 2006 the internal communication system was replace by ICE, the Internet Communication Engine. Orca is focused on reusability of modules and in contrast with its sister project Orocos it does not support real-time behavior.

RoboFrame is a a modular software framework for lightweight autonomous robots Petters and Thomas (2005). It has a very general setup, although it was specially designed for soccer robot. The focus of it is to be able to create time or data triggered threads and the communication among the threads. The data object are usual shared within one executable, but can be reached through a network connection. RoboFrame is a promising framework because it is used very successful by the GermanTeam at the RoboCup competitions for several years. The drawback of RoboFrame is that it does not support real-time behavior.

Alliance is an framework that is focused on multi robot cooperation and higher level intelligence (Parker, 1998), it is used mainly for wheeled robots. By its design it is not usable for multi layered structure.

A.2.1 Selection of candidates

In Table A.1, the comparison of mentioned frameworks are listed.

With Orocos, it is possible to create the modular and multi layered architecture with direct inter-module communication. It has to be noticed that Orocos is the only framework that has a middleware with real-time support. However, it is known to be resource consuming, this is based on the projects where Orocos is used and from information from their mailing list [1]. As stated in Section A.1, one of the requirements is that the software need to be light weighted, in

[1] Orocos mailing list: orocos-users@lists.mech.kuleuven.be

TABLE A.1 - Comparison list of several robot frameworks

	RoboFrame	Orocos	Orca	Alliance	CLARAty	Miro	Player/Stage	URBI	YARP
Multi layered structure	+	+	+	-	+	+	-	-	-
Light weighted	+	-							
Inter-module communication									
- direct	+	+	-	-	+	-	-	-	-
- distributed	+	+	+	+	+	+	-	-	-
Real-time	-	+	-	-	-	-	-	-	-
Open/Availability	+	+	+	-	-	+	+	+	+

order to fit on the hardware. Because of this uncertainty, it might be possible that no framework can be used for the real-time part of the software. Therefore the usability of the other (non-real-time) frameworks is still of interest.

The hardware interface frameworks URBI, YARP and Player are not useful to create the modular and multi layered structure, because inter-module communication is not possible. CLARAty seems a promising candidate, because in Volpe et al. (2001) issues as reusability, object-orientation, decision layer and functional layer are mentioned. However, it is not an open framework. The internal communication of Miro, Orca and Alliance are based on network communications layer for distributes systems, which leads to a loss of efficiency as most of the advantages of it cannot be used. RoboFrame does not support real-time behavior and therefore it fails the requirement. However, it does have a well-defined OS abstraction layer, which makes it possible to port it to a real-time OS. Furthermore is capable of creating modular software in a layered priority structure as Orocos.

That leaves two possibilities with both their disadvantages:
- Orocos: uncertainty about its resource consuming
- RoboFrame: it does not initially have real-time support

In order to make a deliberative choice, these two frameworks have been compared in detail. This will be discussed in the next section.

A.3 Orocos versus RoboFrame

First of all RoboFrame and Orocos are with repect to modularity and infrastructure similar. Executables bases on these frameworks are standalone programs which consists of components that implements the robots activities. RoboFrame is used mainly in wheeled robots, although it is also used in two small humanoid soccer robots (Stelzer and von Stryk, 2008). Orocos is focused on industrial robotics and have not been used in public humanoid projects yet.

In order to compare the two frameworks in detail, the support of each requested feature will be discussed. After that, the test-case will be discussed, which is done to experience the usability of the frameworks to create an robot application.

A.3.1 Comparing by requested features

In this section Orocos and RoboFrame will be compared to each other. This is done by the list of additional request given in Section A.1.

Orocos does have a scripting engine that allows adjusting source code online (while the executable is running). They both have a finite state machine engine, although RoboFrame makes use of an external but close related state machine engine, which is called *The Extensible Agent Behavior Specification Language* (XABLS) and created by Lötzsch et al. (2006).

Orocos is the only framework which support real-time, however it is known to be resource consuming. This is based on the projects where Orocos is used and from information from their mailing list [2]. Although RoboFrame does not support real-time behavior, RoboFrame is used by the competitor GermanTeam (who created RoboFrame) in a humanoid robot as well. Their solution was to use a separate real-time module to handle the time critical software.

Orocos comes with two mathematical libraries, the *Kinematics and Dynamics Library* (KDL) and *Bayesian Filtering library* (BFL). The website of Orocos hosts also a set of already made components which can be directly used in a self made Orocos program, this set of components is called the *Orocos Components Library* (OCL). In contrast with RoboFrame, that really is only a modular infrastructure framework.

Examples and documentation are available for both frameworks. Also the source code was in both cases documented with Doxygen[3]. Besides that Orocos does have an active community on the user mailing list.

A.3.2 Test-case

To get to know more about RoboFrame and Orocos, it was decided to perform a simple test-case: implementing existing 20-sim (2008) controllers for the JIWY setup to control the movements with a joystick (similar as in Jovanovic et al. (2002)). The idea of this test-case was to get familiar with the frameworks in order to judge about how useful the frameworks could be. The test with RoboFrame was performed by the writer of this report, the test with Orocos was performed by Deen (2008). Time needed for installation of the framework, studying of the API and implementation will be taken as measures.

Test setup for RoboFrame: Controlling JIWY

The JIWY setup, as shown in Figure A.2, has 2 degrees of freedom. The setup consists of a PC-104 computer with RoboFrame modules and a Anything IO FPGA board. All the software is running in RoboFrame modules, and each module is within a process (as described in Section 4.1.1). The joystick process is periodical executed on a $10Hz$, the module reads the non-bocking joystick device and stored the setpoints in a blackboard data object. The control process consists of a 20sim controller, its reads out the setpoint from the black board and by data from the encoders it computes the next actuation for the joints. The IO process consists of two modules. First, a module to write the PWM to the AnythingIO board. Second, a module which reads the encoder values from from the Anything IO board. This process it scheduled at $100Hz$. By writing data to the buffer of the 20sim controller, the 20sim controller module gets triggered. After computation it writes the result to the actuation buffer. In this way the IO is executed at $100Hz$, and in the spare time the controller is computed.

Results

The results of RoboFrame test was that the joystick interface, controllers and IO was implemented in one week after receiving the source code and documentation. While the test with Orocos by Deen (2008) took longer, which had two reasons: first, the longer installation process, due to the needed real-time libraries; and second, the higher complexity of the strong structured Orocos.

It have to be noticed that the loop controllers in RoboFrame were not implemented to support real-time. Since RoboFrame is does not have real-time support (yet), and option is to create a separate real-time module for the controllers, but would take significant more time.

From this test-case can be concluded that Orocos has a steeper learning curve. Our opinion is that both frameworks were extremely usable for creating modular software, while Orocos does

[2] Orocos mailing list: orocos-users@lists.mech.kuleuven.be

[3] Doxygen website: `www.doxygen.org`

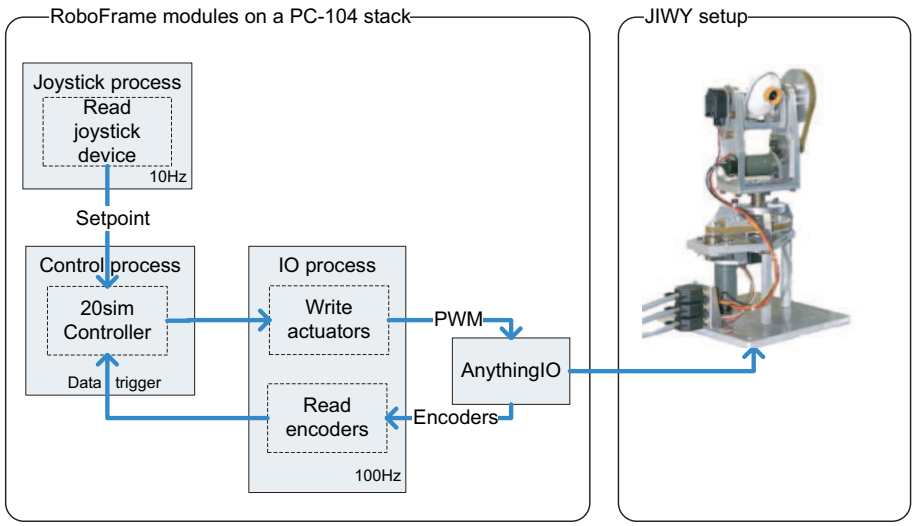


FIGURE A.2 - Test of RoboFrame on JIWY setup with 20sim controllers

a step closer toward robotics due to its real-time support. In theory, Orocos does support real-time, however, from timing tests performed by Deen (2008) can be concluded that the real-time characteristics of Orocos (about 20% variation in time on $1\,kHz$ periodical execution, measured on a similar computer as the TUlip stack) are dubious.

A.4 Conclusion

In this chapter, several popular robot software frameworks have been investigated. No framework could meet all the requirements. However, Orocos and RoboFrame are capable of creating modular software that has inter-modular communication without the intervention of a network protocol. Although, both with their disadvantages:

- Orocos: uncertainty about its resource consumption
- RoboFrame: it does not initially have real-time support

RoboFrame and Orocos have been compared in detail. From where can be concluded that Orocos does support much more features than RoboFrame. However, Orocos has a steeper learning curve because of its higher complexity.

The uncertainties about the resource consumption of Orocos have been nor proven, nor unproven. However, from results of the timing tests from Deen (2008), it is even unsure if Orocos will meet the timing requirement of TUlip. Therefore Orocos has not chosen to be used as the framework for TUlip.

Since RoboFrame has a well-defined OS abstraction layer it might be ported to an API of a real-time OS. However, this is probably a time consuming job and requires extensive validation. Therefore it has been chosen to use RoboFrame only for the non-real-time software parts.

An overview of the frameworks against the requirement and requests is given in Table A.2.

TABLE A.2 - Comparison list of several robot frameworks

		RoboFrame	Orocos	Orca	Alliance	CLARAty	Miro	Player/Stage	URBI	YARP
Requirements	Multi layered structure	+	+	+	-	+[a]	+	-	-	-
	Light weighted	+	-							
	Inter module communication									
	- local	+	+	-	-	+	-	-	-	-
	- through network	+	+	+	+	+	+	-	-	-
	Real-time	-	+	-	-	-	-	-	-	-
	Open/Availability	+	+	+	-	-	+	+	+	+
Requests	Scripting (runtime edit)	+[b]	+[c]							
	State machine	+	+[d]							
	Available modules	-	+							
	Available libraries	-	+[e]							
	Examples	+	+							
	Well documented	+	+							

[a] A two layered structure: Decision and Functional
[b] The Extensible Agent Behavior Specification Language
[c] Orocos Program Scripts
[d] Orocos State Descriptors
[e] Kinematics and Dynamics Library

B Real-time module

An object-oriented real-time tasks library was made, in order to create real-time tasks in an easy way. It was chosen to create an abstraction to the timer API from Xenomai, to be able to port it later to any other timing instructions such as from RTAI, or just from the not realtime posix.

Figure B.1 shows that the timing classes inherit from the interface class `ILoopingTask`. The classes `XenomaiLoopingTask`, `RTAILoopingTask` and `PosixLoopingTask` implement the `start()` `stop()` and the timing construction. This allows users of the timing classes to create a task and only implement the `run()` function, which will be triggered at the requested time interval.

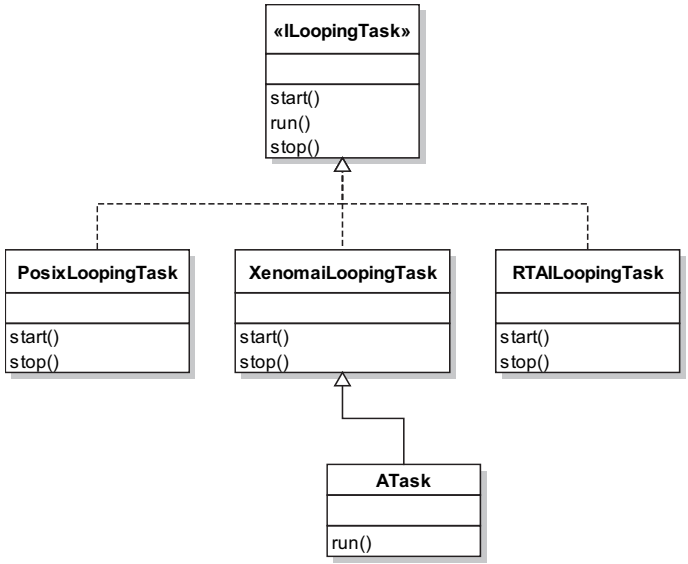


FIGURE B.1 - OS independent loopingtask

B.1 Timing measurements of the tasks

In order to test the timing characteristics of tasks, a timing measurement feature is build in the `XenomaiLoopingTask`. There are two timing measurements possible:

1 scheduling time if the task and
2 computation time of the task.

By measurements of the scheduling time, it can be monitored if the task is scheduled as expected. It measures the time at each execution of the task and stores the maximum and average absolute jitter.

It is also possible to monitor the computational time of a task. In this way it is, for instance, possible to see which task causes heavy load. It measures the time before and after the execution of the `run` function.

B.2 Model-in-the-loop

A simple loop controller framework has been set up. As shown in Figure B.2, it consists of two tasks classes: `LoopTask` and `MotionTask`, which are inheritances of `XenomaiTask` and therefore are real-time. `LoopTask` is running on a higher frequency and higher priority then `MotionTask`.

To test the loop controller the robot was needed. Unfortunately at that time the robot was not fully assembled yet. Therefore was decided to use existing physical models of the robot created by 20-sim. As shown in Figure B.3 the `LoopTask` have been inherited by `ModelLoopTask`

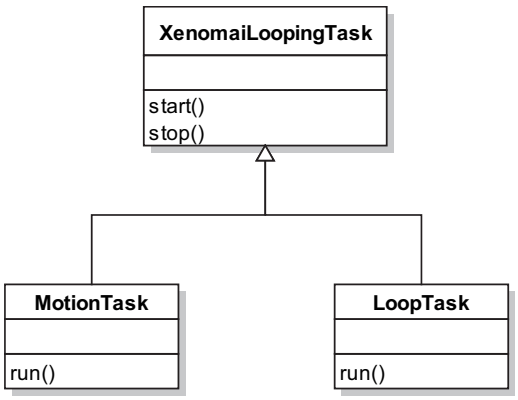


FIGURE B.2 - Used tasks

in order to calculate the model each time after the loop controllers. A non-real-time tasks send periodically the states of the robot to another computer which displayed the model.

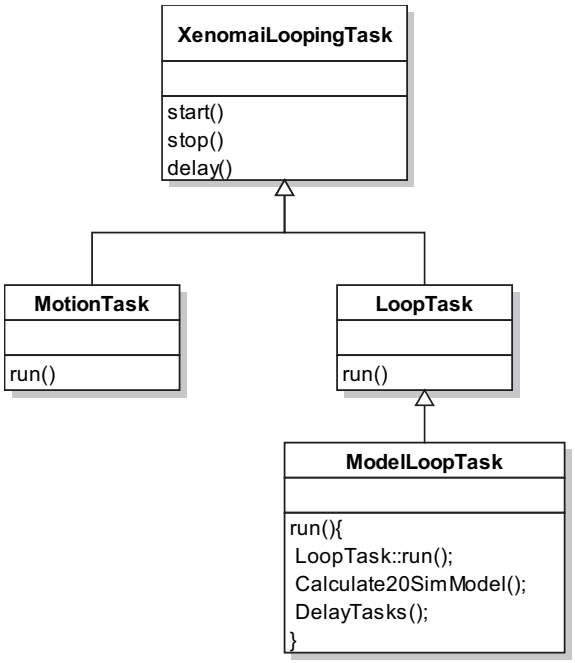


FIGURE B.3 - Delay of the tasks

However, the computation of the models took a lot CPU time. Even more then the actual time, for example the model was calculating $9ms$ for a physical time of $1ms$. To bypass that problem a delay function in the class tasks was introduced (as shown in the class `XenomaiLoopingTask` in Figure B.3). In the run function of `ModelLoopTask` the time of model calculation was measured and all the tasks were delayed by that time. In that way the joint controller framework could be tested.

C Timing test of a task running on Xenomai

In order to validate the timing performance of the realtime tasks based on Xenomai, a realtime tasks have been created, as explained in Appendix B. The software is executed on the PC-104 stack of TUlip. No hard criteria are available yet, since the timing constraints are determined by the digital controllers, which are not designed yet. Since it is expected to run at 1kHz, the test task is scheduled at 1kHz as well. As a reference, the jitter should not exceed ±1% of the periodic time.

It implements a run function which measures the timing compared with the previous execution and stores the data.

The test results, as depicted in Figure C.1, are produced by a measurement of 1000.000 samples. The test is done during 15 minutes (1000.000 samples at $1kHz$) with heavy system load. The heavy system load is produced by reading bulk data from the storage device which causes many interrupts, and continuously reading the random number generator for the computational load.

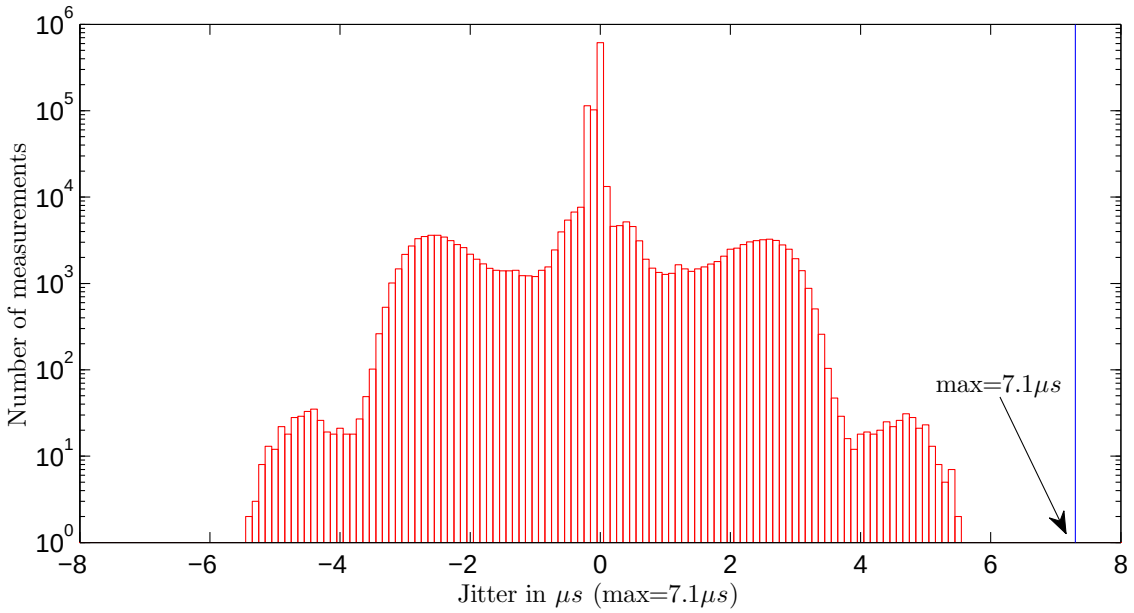


FIGURE C.1 - Jitter of a $1kHz$ task, with full system load

The worst-case jitter is $7.1\mu s$ according to the previous execution, it is below 1% of the scheduled period.

Bibliography

20-sim (2008), A modeling and simulation program, URL `www.20sim.com`.

Baillie, J.-C. (2005), URBI: towards a universal robotic low-level programming language, *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pp. 820–825, doi:10.1109/IROS.2005.1545467.

Broenink, J., M. Groothuis, P. Visser and B. Orlic (2007), A Model-Driven Approach to Embedded Control System Implementation, in *Proceedings of the 2007 Western Multiconference on Computer Simulation WMC 2007, San Diego*, Ed. H. R. Anderson, J., pp. 137–144.

Brunn, R., U. Düffert, M. Jüngel, T. Laue, M. Lötzsch, S. Petters, M. Risler, T. Röfer, K. Spiess and A. Sztybryc (2001), Lecture Notes in Artificial Intelligence 2377, in *RoboCup 2001*, Springer, pp. 705–708.

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

Cooling, J. (2000), *Software Engineering for Real-Time Systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 0201596202.

Daemen, P. (2008), *ZMP based control in 3D passive dynamic walking*, Master's thesis, Univsersity of Twente.

Darmstadt (2008), URL `http://www.sim.tu-darmstadt.de/`.

Deen, B. (2008), *Developing a WLAN Orientation Platform (WOP)*, Master's thesis, University of Twente.

Dertien, E. (2005), *Realisation of an energy-efficient walking robot*, Master's thesis, University of Twente.

Dijkstra, E. W. (1965), Solution of a problem in concurrent programming control, *Commun. ACM*, **8**, 9, p. 569.

Douglass, B. P. (2003), *Software available for reusability, frameworks Real-Time design patterns*, Addison Wesley Longman.

Fitzpatrick, P., G. Metta and L. Natale (2008), Towards long-lived robot genes, *Robot. Auton. Syst.*, **56**, 1, pp. 29–45, ISSN 0921-8890, doi:http://dx.doi.org/10.1016/j.robot.2007.09.014.

Hobbelen, D. (2008), *Limit Cycle Walking*, Ph.D. thesis, TU Delft.

Jovanovic, D., G. Hilderink and J. Broenink (2002), *Communicating Process Architectures*, Reading UK, chapter A communicating Threads -CT- case study: JIWY, pp. 321–330.

Kaupp, T., A. Brooks, B. Upcroft and A. Makarenko (2007), Building a Software Architecture for a Human-Robot Team Using the Orca Framework, *Robotics and Automation, 2007 IEEE International Conference on*, pp. 3736–3741, ISSN 1050-4729, doi:10.1109/ROBOT.2007.364051.

Lötzsch, M., M. Risler and M. Jüngel (2006), XABSL - A Pragmatic Approach to Behavior Engineering, in *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, Beijing, China, pp. 5124–5129.

mailinglist (2008), URL mailto:orocos-users@lists.mech.kuleuven.be.

McGeer, T. (1988), Passive dynamic walking, Technical report, Simon Fraser University, Burnaby, British Columbia, Canada.

Orocos (2008), online, URL http://www.orocos.org/stable/documentation/rtt/v1.4.x/doc-xml/orocos-components-manual.html#id2624787.

Parker, L. E. (1998), ALLIANCE: An architecture for fault tolerant multirobot cooperation., *IEEE Transactions on Robotics and Automaton*.

PC-104 (2008), URL http://www.pc104.org/.

Petters, S. and D. Thomas (2005), *RoboFrame - Softwareframework für mobile autonome Robotersysteme*, Master's thesis, TU Darmstadt, FB Informatik.

QT (2008), URL http://trolltech.com/products/qt/.

RTAI (2008), URL https://www.rtai.org/.

Soetens, P. (2006a), Lock-Free Data Exchange for Real-Time Applications, Presentation: http://people.mech.kuleuven.be/~psoetens/doc/Lock-Free-FOSDEM.pdf.

Soetens, P. (2006b), *A Software Framework for Real-Time and Distributed Robot and Machine Control*, Ph.D. thesis, Katholieke Universiteit Leuven.

Stelzer, M. and O. von Stryk (2008), Walking, running and kicking of humanoid robots and humans, in *From Nano to Space - Applied Mathematics Inspired by Roland Bulirsch*, Eds. M. Breitner, G. Denk and P. Rentrop, Springer Verlag, pp. 175–192 and 337.

Utz, H., S. Sablatnog, S. Enderle and G. Kraetzschmar (2002), Miro - middleware for mobile robot applications, *Robotics and Automation, IEEE Transactions on*, **18**, 4, pp. 493–497, ISSN 1042-296X, doi:10.1109/TRA.2002.802930.

Vaughan, R., B. Gerkey and A. Howard (2003), On device abstractions for portable, reusable robot code, *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, **3**, pp. 2421–2427 vol.3, doi:10.1109/IROS.2003.1249233.

Visser, L. (2008), *Motion Control of a Humanoid Head*, Master's thesis, University of Twente.

Visser, P., J. Broenink and J. van Amerongen (2006), *Boderc: model-based design of high-tech systems*, Embedded Stystems Institute, chapter 17, Design trajectory and controller-plant interaction, pp. 205–214.

Visser, P. M., M. A. Groothuis and J. F. Broenink (2004), *Proceedings of the 5th Progress Symposium on Embedded Systems*, STW, Nieuwegein, chapter Multi-Disciplinary Desing Support using Hardware-in-the-Loop Simulation, pp. pp 206 – 213.

Volpe, R., I. Nesnas, T. Estlin, D. Mutz, R. Petras and H. Das (2001), The CLARAty architecture for robotic autonomy, *Aerospace Conference, 2001, IEEE Proceedings.*, **1**, pp. 1/121–1/132 vol.1, doi:10.1109/AERO.2001.931701.

Williamson, M. M. (1995), Series Elastic Actuators, Technical Report AITR-1524, URL citeseer.ist.psu.edu/williamson95series.html.

XABSL (2008), URL http://www2.informatik.hu-berlin.de/ki/XABSL/.

Xenomai (2008), URL http://www.xenomai.org/.