MATHIAS VON ESSEN

CONTROL SOFTWARE FOR MICROROBOTIC PLATFORM

Master of Science Thesis

Examiners: Prof. Seppo Kuikka and Prof. Pasi Kallio
Examiners and topic approved in the Faculty of Automation, Mechanical and Materials Engineering Council meeting on 9th of December 2009

ABSTRACT

Tampere University of Technology
Degree Program in Automation
Essen, Mathias von: Control Software for Microrobotic Platform
Master of Science thesis, 75 pages, 14 appendix pages
May 2010
Major: Automation and Information Networks
Examiners: professor Seppo Kuikka, professor Pasi Kallio
Keywords: Micromanipulation, Microrobotics, Control software

This thesis is part of SMARTFIBRE project. The objective of the project is development of new functionalization concepts for smart fibre products. SMARTFIBRE is a collaborative effort divided into several subprojects. In the subproject assigned to Tampere University of Technology, a microrobotic platform (MP) capable of characterizing interactions of individual paper fibres is developed to determine the mechanical key factors effecting quality of paper.

Hardware of MP consists of three separate subsystems. The core of MP is Micromanipulation system including several microrobotic actuators. Vision system which consists of a camera and related optics is used to obtain visual information of an ongoing characterization procedure. The third subsystem, Data acquisition system, contains the sensors required to measure desired parameters of the studied interaction. The operator has to be able to control each subsystem of MP.

This thesis introduces *CoSMic*, a control software designed for the needs of MP. The final goal of CoSMic is autonomous characterization of paper fibres with throughput of several tens of paper fibres per hour. CoSMic is based on distributed architecture hosting different parts of the software on separate network nodes. The approach was selected to enhance scalability of the software. In its current state, CoSMic provides the operator with functionality required to control each of the subsystems of MP.

TIIVISTELMÄ

Tampereen Teknillinen Yliopisto
Automaatiotekniikan koulutusohjelma
Essen, Mathias von: Mikrorobottijärjestelmän ohjausohjelmisto
Diplomityö, 75 sivua, 14 liitesivua
Toukokuu 2010
Pääaine: Automaatio- ja informaatioverkot
Tarkastajat: professori Seppo Kuikka, professori Pasi Kallio
Avainsanat: mikromanipulaatio, mikrorobotijärjestelmä, ohjausohjelmisto

Tämä diplomityö on tehty osana SMARTFIBRE-projektia, jonka tarkoituksena on kehittää uusia toiminnallisia paperikuituja älykkäiden paperituotteiden valmistamiseen. Projektin osapuolina ovat Tampereen teknillisen yliopiston systeemitekniikan laitos, Åbo Akademin kuitu- ja selluloosateknologian laboratorio, Latvian valtiollinen puukemian tutkimuslaitos, UPM-Kymmene Oyj, Stora Enso Oyj sekä Metsä-Botnia Oy. Tampereen teknillisen yliopiston vastuualue projektissa on paperikuitujen välisten vuorovaikutusten tutkiminen yksittäisten paperikuitujen tasolla paperin mekaanisiin ominaisuuksiin vaikuttavien tekijöiden määrittämiseksi. Paperikuitujen mekaanisten ominaisuuksien karakterisointi suoritetaan projektissa kehitetyn mikrorobottijärjestelmän avulla.

Tampereen teknillisellä yliopistolla kehitetty mikrorobottijärjestelmä mahdollistaa yksittäisten paperikuitujen manipuloinnin, havainnoinnin sekä paperikuidun mekaanisten ominaisuuksien karakterisoinnin. Järjestelmä jakautuu kolmeen alijärjestelmään, joista kullakin on oma vastuualueensa. Paperikuitujen manipulointi tapahtuu mikromanipulaatiojärjestelmässä, joka mahdollistaa paperikuitujen manipuloinnin järjestelmään kuuluvien toimilaitteiden avulla. Paperikuitujen havainnointi tapahtuu järjestelmään liitetyn kamerasta, moottoroidusta objektiivista sekä valaisujärjestelmästä koostuvan konenäköjärjestelmän avulla. Havainnoinnin lisäksi konenäköjärjestelmällä voidaan suorittaa kuvaan perustuvia mittauksia. Mikrorobottijärjestelmään kuuluu myös anturijärjestelmä, joka kerää mittausinformaatiota karakterisoitavista paperikuiduista.

Tässä diplomityössä suunnitellaan ja toteutetaan mikrorobottijärjestelmän ohjaamiseen soveltuva ohjelmisto. Ohjausohjelmiston pääasiallinen tarkoitus on järjestelmään kuuluvien toimilaitteiden ohjaaminen sekä tiedonkeruu järjestelmän mittalaitteilta. Lopullisena tavoitteena on paperikuitujen karakterisointiprosessin automatisointi. Täysin automatisoidun mikrorobottijärjestelmän kapasiteetin on tarkoitus yltää useiden kymmenien paperikuitujen karakterisointiin tunnissa.

Tämä diplomityö esittelee hajautettuun arkkitehtuuriin perustuvan alustariippumattomuuteen pyrkivän mikrorobottijärjestelmän ohjausohjelmiston. Ohjausohjelmiston nimeksi on annettu CoSMic, joka on lyhenne sanoista **Co**ntrol **S**oftware for **Mic**rorobotic Platform. Vaikka lopullisena tavoitteena onkin kehittää täysin automatisoitu ohjausjärjestelmä, kehitetään tässä vaiheessa teleoperoinnin

mahdollistava ohjelmisto, joka tarjoaa graafisen käyttöliittymän jokaiseen mikrorobottijärjestelmän alijärjestelmään.

Työn selvitysosuudessa käsitellään hajautettuihin järjestelmiin liittyviä yleisiä hyöty- ja haittanäkökulmia. Hajautuksella voidaan saavuttaa huomattavia etuja järjestelmän suorituskyvyssä, skaalattavuudessa ja virheensietokyvyssä. Toisaalta järjestelmän moninaisuus ja hajanaisuus kasvaa. Hajautuksen mukanaan tuomia ongelmia voidaan vähentää ohjelmiston rakenteen huolellisella suunnittelulla.. Tähän osuudessa paneudutaan esittelemällä suunnittelumallin käsite. Suunnittelumallilla tarkoitetaan ohjelmistotekniikassa olio-ohjelmointiin liittyvää tapaa, jolla usein esiintyvä ongelma voidaan ratkaista. Suunnittelumalleja esiintyy usealla tasolla ja ne kuvaavat olioiden tai luokkien välisiä vaikutussuhteita ja kommunikaatiota.

Mikrorobottijärjestelmän turvallisuuteen liittyen paneudutaan törmäysten tunnistamiseen kolmiulotteisessa avaruudessa. Törmäysten tunnistamisen tarkoituksena on estää mikrorobottijärjestelmän eri osien törmääminen toisiinsa. Tyypillisesti törmäysten tunnistaminen suoritetaan mallintamalla todellisen laitteiston geometria virtuaalitodellisuudessa, jossa törmäykset havaitaan mallinnettujen geometrioiden leikatessa toisensa. Leikkauspisteiden etsiminen vaatii huomattavan määrän laskentatehoa ja useita erilaisia algoritmeja tarvittavan laskentatehon vähentämiseksi on saatavilla. Työssä esitellään törmäyksen tunnisteen liittyviä algoritmeja sekä luodaan katsaus olemassa oleviin avoimen lähdekoodin toteutuksiin.

Edellä mainittujen perustavanlaatuisten konseptien tutkimisen jälkeen esitellään mikrorobottijärjestelmän laitteisto, lähinnä sen ohjauksen kannalta sekä määritellään vaatimukset. Laitteiston ohjauksen yhteydessä tutustutaan saatavilla oleviin ohjelmakirjastoihin ja tutkitaan niiden soveltuvuutta mikrorobottijärjestelmään. Jokaisen alijärjestelmän ohjaamiseen valitaan oma ohjelmakirjasto, jonka tarkoitus on tukea koko järjestelmän pitkäaikaista kehittämistä. Lisäksi ohjelmiston kehitykseen valitaan erillinen ohjelmistokehitysympäristö, jonka tarkoituksena on tukea kehitettävän ohjausohjelmiston alustariippumattomuutta.

Työn soveltava osa keskittyy suurelta osin ohjausohjelmiston arkkitehtuuriin ja suunnitteluun. Ohjelmiston arkkitehtuuria suunniteltaessa tutkitaan hajautettuihin järjestelmiin soveltuvia arkkitehtuuritason suunnittelumalleja. Valittuja suunnittelumalleja käytetään perustana työssä kehitetylle ohjelmistokehykselle, jonka tarkoituksena on yhdenmukaistaa mikrorobottijärjestelmään liitettävien ohjelmistojen kehitysprosessi. Ohjelmiston eri osien välinen kommunikaatio pyritään irrottamaan erilliseksi osaksi, jotta ohjelmisto ei olisi riippuvainen käytetystä verkkotekniikasta. Ohjelmiston suunnitteluun liittyvässä osassa keskitytään tarkastelemaan mikromanipulaatiota ja mittausinformaation tiedonkeruuta ohjaavia ohjelmiston osia. Molemmat osat toteutetaan monisäikeisinä arkkitehtuurin yhteydessä kuvattuja suunnittelumalleja noudattaen. Lisäksi esitellään ohjausohjelmiston tämänhetkinen toteutus ja graafinen käyttöliittymä. Osuuden lopuksi käsitellään ohjausohjelmiston toteutuksessa havaittuja ongelmia ja kerrotaan ohjelmiston kehittämiseen liittyvistä tulevaisuudensuunnitelmista.

Työn tuloksena on suunniteltu hajautettu ohjausohjelmisto paperikuitujen mekaaniseen karakterisointiin tarkoitetulle mikrorobottijärjestelmälle. Ohjelmistosta on toteutettu kaksi alijärjestelmää, joiden avulla voidaan ohjata mikromanipulaatioon ja mittausinformaation tiedonkeruuseen liittyvää laitteistoa.

FOREWORD

This thesis has been made in the Department of Automation Science and Engineering at Tampere University of Technology (TUT). The work has been funded by the Finnish Funding Agency for Technology and Innovation (TEKES).

I would like to express my gratitude to Prof. Pasi Kallio who has supported and encouraged me throughout the thesis. His experience and knowledge has indeed helped me to accomplish this work. I am grateful to Prof. Seppo Kuikka whose hints and comments I have found invaluable. I would also like to thank all my colleagues in Micro- and Nanosystems Research Group – one could not wish for a better working atmosphere.

I would like to thank my parents for supporting me throughout my studies. I would also like to thank Evelína for all the inspirational trips we have had. Finally, I would like to express my deepest gratitude to my fiancée Magdaléna Vaňková.

Tampere, June 2010

Mathias von Essen

CONTENTS

Abstract II
Tiivistelmä III
Foreword VI
Symbols and Abbreviations IX
1. Introduction 1
1.1. Scope 2
1.2. Outline 2
2. Overview of Distributed Control Software for Microrobotic Platform 3
2.1. Introduction to Distributed Systems 3
2.2. Software Design Patterns 4
2.3. Real-Time Systems 6
2.3.1. Scheduling 7
2.3.2. Linux in Real-time Systems 7
2.4. Collision Detection 8
2.4.1. Collision Detection Pipeline 8
2.4.2. Collision Detection Implementations 12
3. Microrobotic Platform 14
3.1. Overview of the Microrobotic Platform 14
3.2. Requirements for Control Software 16
3.2.1. Requirements of Micromanipulation System 16
3.2.2. Requirements of Vision System 17
3.2.3. Requirements of Data Acquisition System 17
3.2.4. Requirements of Real-Time Controller 17
3.3. Related Hardware 17
3.3.1. Micromanipulation System 18
3.3.2. Vision System 19
3.3.3. Data Acquisition System 21
3.4. Control Modes 22
3.4.1. Manual and Semi-automatic Control Modes 22
3.4.2. Enhanced Manual Control Mode 22
3.4.3. Automatic Control Mode 23
4. Selection of Implementation Technologies 24
4.1. Qt – an Application Development Framework for C++ 24
4.1.1. Signals and Slots 25
4.1.2. Threading in Qt 25
4.1.3. Qt Integration with Real-time Operating Systems 26
4.2. Application Programming Interface for SmarAct Micropositioners 26
4.2.1. Communication Modes 26
4.2.2. Control Methods 28
4.3. Application Programming Interfaces for Data Acquisition 29
4.3.1. DAQmx 29

4.3.2. Data Acquisition in Real-time Linux ... 29
4.3.3. Selection ... 30
4.4. Selection of Collision Detection Library ... 30
4.4.1. CollDet ... 31
4.4.2. Testing of Selected Collision Detection Library ... 31
4.5. Key Findings ... 33
4.6. Selected Technologies and Design Principles ... 35
5. Architecture ... 36
5.1. Selected Architectural Patterns for Distributed Computing ... 36
5.1.1. Broker Pattern ... 37
5.1.2. Client Proxy Pattern ... 37
5.1.3. Invoker Pattern ... 38
5.2. Distributed Architecture for Microrobotic Platform ... 39
5.2.1. Network Communication ... 41
5.2.2. Communication on Network Node Level ... 43
5.3. CoSMic-Frame ... 46
5.3.1. Structure ... 46
5.3.2. Network Communication ... 47
5.4. Architecture of MiCo ... 48
5.5. Architecture of DAQCo ... 50
5.6. Summary ... 51
6. Design and Implementation ... 52
6.1. MiCo ... 52
6.1.1. Overview ... 52
6.1.2. MiCo API ... 55
6.1.3. Communication ... 58
6.1.4. User Interfaces ... 62
6.2. DAQCo ... 62
6.2.1. Callback Functions and Data Exchange ... 63
6.2.2. Graphical User Interface ... 65
6.3. Integration of MiCo and DAQCo With An Input Device ... 66
6.4. Current Implementation ... 67
7. Conclusions and Future Work ... 70
7.1. Conclusions ... 70
7.2. Future Work ... 71
8. References ... 73

SYMBOLS AND ABBREVIATIONS

Symbols

e_i	Sorting Axis End Index
n	Number of Bounding Volumes in Collision Detection
N	Degrees of Freedom in Actuator Assemblies
s_i	Sorting axis start index

Abbreviations

AABB	Axis Aligned Bounding Box
ACM	Automatic Control Mode
A/D	Analog-to-digital conversion
ADF	Application Development Framework
API	Application Programming Interface
CD	Collision Detection
CGC	Computer Graphics Group of Clausthal University of Technology
CPU	Central Processing Unit
DAQ	Data Acquisition
DAQCo	Control of DAQ System
DAQS	DAQ System
DLL	Dynamic-link Library
ECM	Enhance Manual Control Mode
FIFO	First In First Out
GUI	Graphical User Interface
LED	Light Emitting Diode
IEEE	Institute of Electrical and Electronics Engineers
IEEE1394	Serial bus interface standard defined by the IEEE
IIDC	1394 Trade Association and Industrial Control Working Group
MCM	Manual Control Mode
MiCo	Control of Micromanipulation System
MiS	Micromanipulation System
MP	Microrobotic Platform
OBB	Oriented Bounding Box
RS-232	Recommended Standard 232

RTAI	Real-time Application Interface for Linux
RTOS	Real-time Operating System
SAP	Sweep and Prune
SCM	Semi-automatic Control Mode
TEKES	Finnish Funding Agency for Technology and Innovation
USB	Universal Serial Bus
ViCo	Control of Vision System
ViS	Vision System
VR	Virtual Reality

1. INTRODUCTION

The development of the actuators used in microrobotics has undergone rapid evolution during the past decade. In its current state, the technology has reached a certain level of maturity and more commercial solutions are penetrating to market. The evolution however, is still on the hardware level and more resources are required in the development of control software. Software development for microrobotic hardware differs in many ways from that for conventional robotics. The products available are often immature – at least in the sense of software development. Moreover, the mere size and the possibly unknown characteristics of the actuator may hinder development of the software. The movements produced by the microrobotic actuators are often measured in micro- or nanometres and the movement may not be visible for naked eye. A lack of standardization and well established practises incurs a situation where the provided application programming interfaces (API) do not have common features, making creation of general-purpose control software virtually impossible, thus forcing the application developers to content with the manufacturer's API.

This thesis is part of SMARTFIBRE project funded by The Finnish Funding Agency for Technology and Innovation (TEKES). It is accomplished in Micro and Nanosystems Research Group at Department of Automation Science and Engineering, part of the Faculty of Automation, Mechanical and Materials Engineering of Tampere University of Technology.

The objective of the project is development of new functionalisation concepts for smart fibre products. The project is a collaborative effort of two research partners, responsibilities between the partners is divided as follows. Laboratory of Fibre and Cellulose Technology at Åbo Akademi is responsible for development of the new functionalization concepts. Main activities include design and multifunctionalisation of fibres and papers in order to enhance existing and to innovate new functionalities for fibre based materials.

Micro and Nanosystems research group is responsible for investigating individual fibre-fibre and fibre-chemical interactions using a novel Microrobotic Platform (MP) developed as part of the project. The final goal of the MP is fully automated characterization of paper fibres. In order to collect sufficient quantities of data, several hundreds of fibres should be characterized on a daily basis. Thus, the MP should be able to autonomously characterize several tens of fibres per hour.

Development of the microrobotic platform includes two separate phases. The first part, presented in [16], concentrates on selection and implementation of the hardware of the MP. In addition, control software capable of performing the autonomous characterization needs to be developed. This thesis work concentrates on the development of the control software, scope of the thesis work is described in detail in Section 1.1.

1.1. Scope

The objective of the work is to develop scalable, robust and partly real-time capable distributed control software for the purposes of paper fibre characterization. The scope of this thesis is limited to cover architectural design of **Co**ntrol **S**oftware for **Mic**rorobotic platform (CoSMic) and implementation of its two core parts, namely Control of Micromanipulation System (MiCo) and Control of Data Acquisition System (DAQCo).

1.2. Outline

The structure of the thesis is organized as follows. Chapter 2provides background relating to the concepts later implemented in this work. Chapter 3presents Microrobotic Platform (MP), the user requirements for the developed control software and related hardware. Chapter 4concentrates on selection of implementation technologies. Chapter 5elucidates the proposed architecture. Chapter 6describes design and implementation of MiCo and DAQCo. The final part, Chapter 7 concludes the thesis and presents proposals for future work.

2. OVERVIEW OF DISTRIBUTED CONTROL SOFTWARE FOR MICROROBOTIC PLATFORM

This chapter includes theoretical aspects involved in the development of control software for the microrobotic platform (MP). Section 2.1 introduces the concept of distributed systems followed by introduction of software patterns in Section 2.2. Section 2.3 encompasses the general aspects of a real-time system. Finally, Section 2.4 presents the concept of collision detection.

2.1. Introduction to Distributed Systems

Traditionally computer software was thought as a stand-alone system residing on a single computer. A typical stand-alone system has been responsible for reacting to inputs through a user interface, performing the desired processes and managing the persistent data. The constantly increasing complexity of the developed software has led to the point where the required level of computation is often too much to be handled by a single computer. Therefore, more and more systems are developed in a distributed manner. Distributed systems split the structure of the software into logical entities which are allocated to number of independent computers. The computers are able to cooperate over a communication network in order to achieve the desired objective.

The benefits of distributed systems over centralized solutions are widely recognized, some of the most important aspects include:

- Performance
- Economics
- Failure tolerance
- Scalability

Distributed systems have better performance when compared with mainframe solutions due to increased concurrency; different nodes of the distributed system are able to execute different tasks simultaneously. The parallel execution of several applications increases the system's performance in comparison with centralized solutions. In addition, a well-designed distributed system is easily scalable by adding components into the system. Also economical factors support usage of distributed systems because they offer a better price/performance ratio than mainframe systems. For example, nodes with specialized properties, such as expensive high speed data acquisition can provide

services for the other parts of the system. Conversely in a centralized solution, each system requiring the high speed data acquisition system would require its own hardware. Failure tolerance of a distributed system can be reached by introducing sufficient redundancy to the system; the most essential parts of the system can be replicated to several nodes. Sufficient redundancy delimits failures to subsystems, thus the entire system can survive crashes of the network or a single computer node. [1][2]

Distributed systems have also some disadvantages, most of which are tightly coupled with the benefits of networked computing:

- Complexity
- Heterogeneity

Distribution increases system complexity due to the increased level of concurrency and asynchronous communication. Failure of a single component might affect the entire network if appropriate mechanisms for preventing such a situation do not exist. Introduction of each new component increases the risk of affecting the entire network in case of a failure. [1]

Distributed systems are often used over large geographical areas and the development time might be calculated in years. Large geographical coverage increases the likelihood for incorporation of different implementation technologies in different parts of the system. The long development time often increases the heterogeneity of the system, as some parts of the old system might be incompatible with planned new features. [1][2]

2.2. Software Design Patterns

Software design patterns have been a largely discussed topic for more than a decade after reaching wide acceptance in 1994 followed by the publication of [3].

In object-oriented programming, the functionality of the developed program is provided through collaborative effort of several objects contributing into the system. Software developers often face problems which are identical or similar to issues solved in previous applications. Identification of recurring problems is even desirable, as object-oriented programming provides the means of reusing existing program code. Reusability provides obvious benefits through time saving, as the same program code can be used in multiple places. In addition, systems employing reusable components are likely to be less prone to errors due to the wider usage of the component; a large group of developer using the same library over long period of time is more likely to find the possible programming errors than a single developer. However *reusability* might be limited to tackle one single problem and cannot be used outside the original domain. [4][5]

For example in development of a distributed system, a developer of network related applications might have an off-the-shelf implementation of a client and a server class for socket communication over TCP/IP, which he employs in all network related applications. He couples the server class with the application code by mapping the

application's functions to the server. If the developer is asked to develop an application based on other network protocol, he might have to rewrite most of the server-side program code.

Reusability can be increased by *generalizing* the found solution into a set of rules which describes a solution on more general level. In object-oriented programming such collections of rules and guidelines solving abstract problems are known as *design patterns*. Design patterns are documentations which include both, the problem and the solution within a given context. Description of design pattern is given in a consistent textual format. The ground for preferring textual format over graphical notation is reusability; graphical representation is often able to catch only the end product, hiding the original reasoning from the viewer. A typical format of design pattern consists of eight sections describing the pattern from different aspects. A typical layout of design pattern is presented in Table 2.1. [3][6]

Table 2.1 Structure of a design pattern

Section name	Description
Name	Name of the pattern
Context	Motivation Domain of usage Example: *application has been distributed between three nodes which are connected together with a bus.*
Problem	Describes the original problem. Example: *how to change the bus standard without changing the application code*
Forces	Characterizes the pattern in detail. Describes what effective solution must take into account. Example: *Scalability: system may consists of hundreds of nodes.* *Reusability: the bus may change during the life-cycle*
Solution	Provides solution which solves the previously presented problem.

Consequences	Describes the benefits and the pitfalls of the proposed solution Example: - *Abstraction may increase latency* + *Increases the scalability of the system*
Resulting Context	Describes the results Example: *result is a highly scalable system where the bus standard can be changed without affecting the application code*
Related Patterns	Describes this patterns relation to other. The pattern may turn out to be more useful when combined with another pattern.
Known usage	List of known users of the pattern Example: *Complex platform for research purposes uses TCP/IP communication protocol between several nodes. The system is expected to be enhanced with real-time capable bus. Therefore a mechanism abstracting the network layer from the application code is required.*

Design patterns can be categorized by the domain they target and by the used level of abstraction. Patterns providing principle of solution for entire software architecture are called *architectural patterns.* Similarly, patterns related to the mechanistic design of the software are known as *mechanistic design patterns.*

2.3. Real-Time Systems

Real-time systems may include very different characteristics depending on their domain. Real-time systems are found in a variety of applications ranging from simple embedded systems to airplane manoeuvring systems and internet banking. The term real-time is often falsely though as a measure of high speed. In several real-time applications high speed is essential, but it does not define the system as a real-time system. By definition a real-time system performs given operations in timely manner – the system guarantees to fulfil the given performance constraints. Real-time systems can be categorized to *soft real-time* and *hard real-time* systems. Hard real-time systems are the stricter category of real-time systems. In these systems, a missed deadline is equal to a system failure. Soft real-time systems give more flexibility to the time constraints. In soft real-time systems, deadlines can occasionally be completely missed and missing the deadline by small time deviation is also allowed. [7]

Generally, all real-time systems interact with hardware in monitoring or controlling purposes. The interface between the application code and the hardware must have real-time implementation to maintain the system level real-time capabilities. For example, a standard Universal Serial Bus (USB) driver for Microsoft Windows can not comply with real-time constraints. Usage of such a driver in a real-time system will be problematic, since the behaviour is time wise undefined. However, different parts of a real-time system can have different level of constraints. In fact, most real-time systems include parts with soft and hard real-time requirements [7].

2.3.1. Scheduling

The scheduler is an instance which determines how to commit resources between several different tasks. Execution order of outstanding processes is based on pre-defined criteria, such as priority. Conventional operating systems serialize the processes based on their priority, thus the processes with highest importance are processed first. The described scheduling method is for real-time system – a high priority does not automatically convert to meeting the deadline. [8][9]

A typical real-time operating system (RTOS) also uses priorities for scheduling, but with additional timeline constraints. The highest priority task pending for processing always gets a time slot from central processing unit (CPU) within a fixed amount of time. Thus, the latency of the system depends only on tasks running at higher priorities. [8][9]

2.3.2. Linux in Real-time Systems

Linux is a Unix-like high-performance open-source operating system used globally by millions of users. In a typical case, Linux is shipped as a Linux distribution which consists of an operating system kernel and supportive software. Linux is considered to be one of the most stable operating systems available for servers and standard desktop computers.

Suitability of Linux for real-time systems is a widely discussed topic for which multiple solutions are available. The pure Linux kernel, often referred as *vanilla*, is not suitable for real-time systems as such. However, the open-source source code allows the developers to modify the kernel to suit better for the purposes of real-time systems. Currently there are several open-source real-time Linux implementations available. The following presents two well established open-source real-time kernel extensions for Linux.

RTAI

Real-Time Application Interface for Linux or shortly RTAI is a real-time kernel extension initially developed at Dipartimento di Ingegneria Aerospaziale / Politecnico di Milano. In its current state, RTAI is developed as a community effort. RTAI supports data acquisition (DAQ) through a real-time capable DAQ library called Linux Control

and Measurement Device Interface (Comedi). RTAI consists of two main parts: a Linux kernel patch introducing a hardware abstraction layer and a package of convenience services reducing the workload in development process of real-time applications. [11]

XENOMAI

Xenomai is another effort to bring RTOS capabilities to Linux. The main difference between Xenomai and RTAI is that the projects have slightly different focus. Xenomai considers extendibility, maintainability and portability as important goals. The portability is implemented as number of RTOS APIs referred as *skins*. Each skin supports one real-time API, currently available skins include POSIX, VxWorks, RTAI and several others. [10]

2.4. Collision Detection

Identification of colliding objects in a three-dimensional (3D) space is a fundamental problem in various areas of software development. In a typical application, such as a simulator or a computer game, *collision detection* (CD) might be required to model physical interactions of objects in the real-world. An additional step known as *collision handling* is required to determine appropriate steps in a case of a collision. For example, CD between a falling object and a surface is required in recognizing the event of the object hitting the surface. The collision handling might then calculate possible deformation and new trajectory for the object.

CD between hardware of the real-world requires modeling which maps the real-world situation into a virtual reality (VR). The modeling converts each real-world object into a VR object built from several polygons.

Mere detection of collisions does not suffice in cases where collisions may harm the system. Software interacting with hardware, such as robots, is a typical application where *collision avoidance* is essential. Moreover, autonomous systems should be able to reroute themselves in the case of a potential collision. Collision safe routing is often referred as *path-planning*. Collision avoidance and path-planning require collision detection in order to determine which actions would lead to a possible collision. The task is not easy, as the 3D space may contain hundreds or thousands of objects with complex geometries. Collision detection is computationally intensive task by definition. This section provides an overview of CD by introducing the CD process and its different phases.

2.4.1. Collision Detection Pipeline

The basic idea behind most of CD implementations is finding intersecting pairs of polygons between two objects. A collision occurs when a polygon of one object intersects a polygon of another object. However, comparison of all possible polygon pairs would lead to tremendous amount of computation. Therefore advanced algorithms

are required to avoid as many polygon/polygon tests as possible. CD process can be thought as a pipeline where the objects are an input which is handled by a collision pipeline containing the different phases of CD process. The outcome of the process is a physical response, such as a detected collision or distance of two objects. The structure of CD pipeline is illustrated in Figure 2.1.

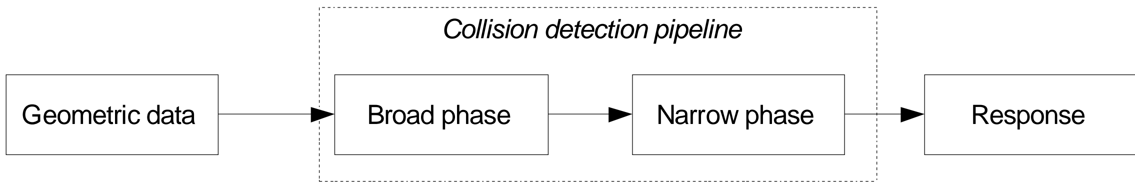

Figure 2.1 Collision detection pipeline

In order to relieve the computational load related to CD, the process is often divided into two stages, called *broad phase* and *narrow phase*. Broad phase can be thought as a filter which aims to avoid unnecessary intersection testing for objects that are far away from each other. Bounding volumes with simple geometry, such as box or sphere can be placed around each body to simplify the geometries analyzed in the broad phase; collisions may occur only if bounding volumes of two objects overlap. The objects which were found to have overlapping bounding volumes are passed to the narrow phase for further inspection. The narrow phase refines the previous collision detection to the level of individual polygons.

Broad Phase

The purpose of the broad phase algorithms is to quickly filter out as many objects as possible. Axis-aligned bounding boxes (AABB) and oriented bounding boxes (OBB) are typical approaches for implementation of the bounding volumes. Difference between the AABB and OBB is orientation of the bounding volume; AABB are aligned with the axis of the coordinate system, whereas OBB alignment is arbitrary. Figure 2.2 illustrates the difference between the orientation of AABB and OBB.

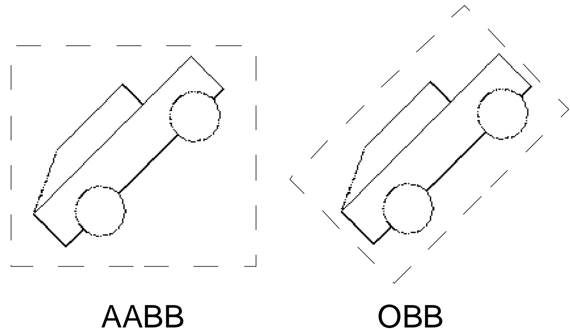

Figure 2.2 Axis-aligned bounding box (left) and Oriented bounding box (right)

The simplest method for testing collisions between two bounding boxes is known as the *brute-force algorithm*. The idea behind the algorithm is as follows. Compare each edge

of one bounding volume against all edges of the other bounding volume, and vice versa. The downside of this very simple algorithm is a lack of performance. The amount of comparisons required between two bounding volumes is n(n-1), where *n* represents the number of present bounding volumes. The performance of the broad phase algorithm can be optimized by several different strategies including spatial partitioning and the aforementioned bounding volumes.

Broad phase algorithm *Sweep and Prune* (SAP) presented in [39] uses AABB to determine whether two objects are sufficiently close to potentially collide. SAP determines overlapping bounding volumes by reducing the original three-dimensional problem into three one-dimensional problems; two AABB overlap only if all their projections overlap. For each sorting axis containing the projections, SAP stores intervals occupied by individual projections. The intervals are denoted as $[s_i, e_i]$, where s_i is the starting point for the interval of a single projection and e_i is the respective end point. Figure 2.3 presents a single sorting axis with three different objects.

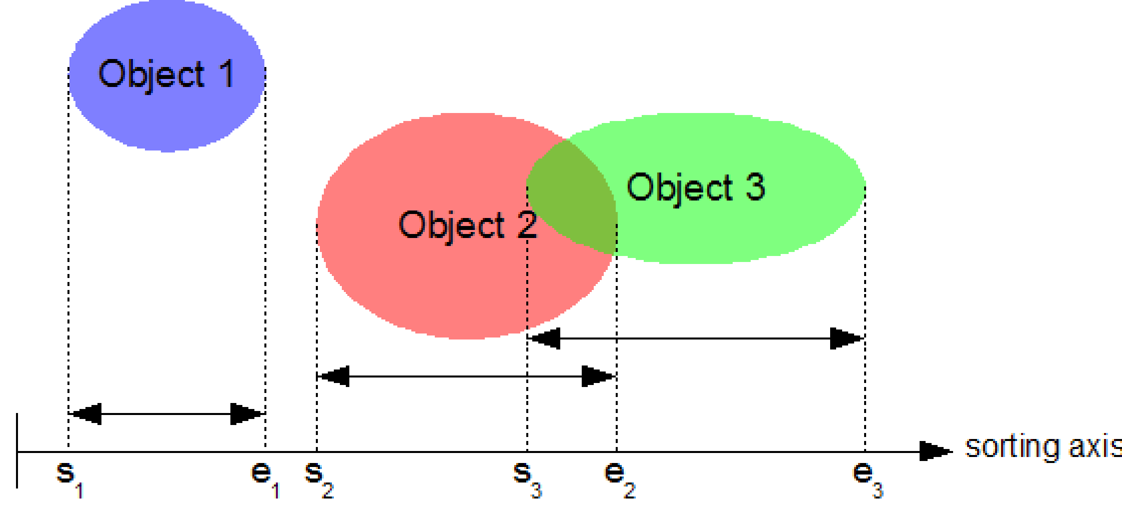

Figure 2.3 SAP sorting axis with three projections

The found intervals are stored in a list which is sorted in ascending order. Each node of the list includes a tag describing whether the node represents s_i or e_i of a particular object. The actual CD takes place by traversing the created list from the beginning to the end. Whenever the algorithm finds a s_i tag the object i is added to the active object list. In case of an e_i tag, the respective object is removed from the active object list. Thus each object is compared only against the objects currently stored in the active object list. Finally SAP finds the objects which collide in all projections and forms a list of candidates. This list can be forwarded to narrow phase algorithms for further inspection. SAP has proven to be an efficient broad phase algorithm and it is widely implemented in different CD libraries. However, the usage of AABB may lead to large amount of redundant space within the bounding volume. The problem becomes obvious if the bounded object has strong diagonal orientation as indicated in Figure 2.2. [40]

Narrow Phase

Narrow phase collision detection is responsible for detecting collisions between the pairs of objects which were found in the broad phase. The narrow phase should result in a list of individual polygons and exact coordinates of the points where collisions occurred. Several methods such as hierarchical methods and incremental distance computation have been proposed for the narrow phase collision detection.

Hierarchical methods decompose each object into a tree, where each node represents certain subset of the original object. The root node of the tree contains the whole object. An example describing possible decomposition of a simple object is provided in Figure 2.4. The decomposition should satisfy two opposing criteria guiding the selection of the bounding volume. The bounding volume should contain minimal amount of redundant space. However the intersection test should be as efficient as possible. That is the geometry of the bounding volume should be as simple as possible. [44][45]

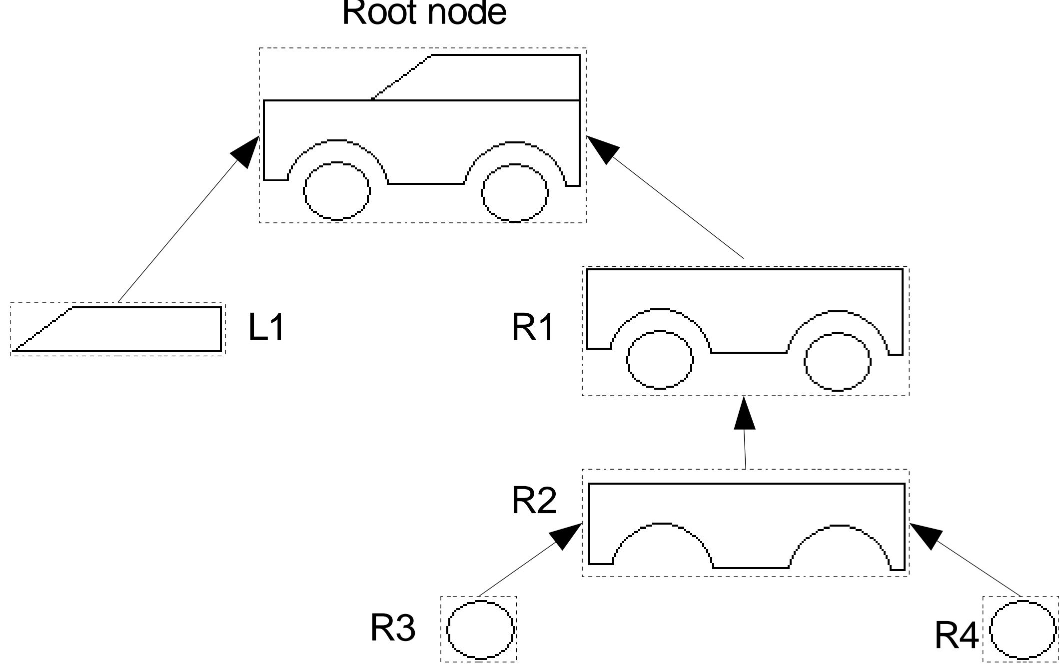

Figure 2.4 Hierarchical method – bounding volume tree

Hierarchical methods aim to further minimize the amount of polygons required to accurately determine the point of collision. In case where broad phase detects a collision between two objects, the hierarchical model is able to prune the irrelevant polygons by traversing the tree model of both of the colliding objects. Head-on collision of two cars based on the hierarchy presented in Figure 2.4 is taken as an example, the colliding cars are named as Car1 and Car2. Traversing of the trees starts by comparing the root nodes of Car1 and Car2. If the root nodes do not intersect, the objects cannot collide. If intersection between the root nodes is detected, the algorithm moves to next level by comparing the child nodes L1 and R1 of Car1 against the root node of Car2. If either of the child nodes of Car1 intersects with the root node of Car2, the bounding volume of

Car1 is replaced with the child node. In case of head-on collision, the node L1 of Car1 would replace the original bounding volume and traversing would stop. The traversing continues until the maximum depth of recursion is reached. [44][45]

Incremental distance computation is a probabilistic method assuming that objects move only small distance between successive calls of the collision detection algorithm. In such a case, methods of linear programming can be used, which yield linear performance time by definition. However, linear programming is only applicable for convex polygons. Thus in 3D space, the polyhedral models must satisfy the rules of convexity, that is all faces of each polyhedral must join together and form bounded 3D shapes. One of the most known algorithms within this category is Lin-Canny algorithm presented in [32].

2.4.2. Collision Detection Implementations

The following presents few examples of open-source collision detection libraries available. A more thorough list is available at [33].

OPCODE

Optimized Collision Detection or OPCODE is a small CD library developed for C++ developed by Pierre Terdiman. OPCODE uses AABB together with bounding volume tree hierarchy. The objective of OPCODE is to reduce the memory footprint in comparison with other similar collision detection libraries such as SOLID [34] and RAPID [35]. The broad phase collision detection of OPCODE provides implementation of SAP, in addition radix-based box pruning algorithm is available. [13]

SWIFT++

SWIFT++ is a C++ CD package developed by the Geometric Algorithms for Modeling, Motion, and Animation Group at University of North Carolina at Chapel Hill. The same group has produced numerous open-source collision libraries such as I-COLLIDE [36] , RAPID and SWIFT++ [15]. SWIFT++ is targeted for detection of intersection, computation of distances and determining contacts between pairs of objects. The objects are modelled by polyhedral geometries and allow several objects to share the same geometry. SWIFT++ employs SAP to detect overlapping of moving objects in the broad phase. The narrow phase collision detection is based on the Lin-Canny algorithm. [14][15]

CollDet

CollDet [46] is a C++ CD library developed by Computer Graphics group of Clausthal University of Technology (CGC). The primary application domain of CollDet is 3D real-time applications. The algorithms used in CollDet are developed at CGC. The

performance of these algorithms is in some cases significantly faster than the most typical approaches [40].

3. MICROROBOTIC PLATFORM

The microrobotic platform (MP) targeted for the characterization of different kinds of fibres has been developed as a part of the project SmartFibre. The final goal of the platform is to characterize several hundreds or thousands of fibres on a daily basis. MP consists of three separate subsystems: Micromanipulation system (MiS), Vision System (ViS) and Data Acquisition System (DAQS).

This chapter concentrates on presenting the microrobotic platform from several points of view. Section 3.1 presents an overview of the system, followed by the description of the user requirements in Section 3.2. The hardware related to the MP is described in Section 3.3. Different methods for performing paper fibre characterization on MP are proposed in Section 3.4.

3.1. Overview of the Microrobotic Platform

MP is built from three separate subsystems each responsible for a specific range of tasks, as illustrated in Figure 3.1. Micromanipulation System (MiS) containing a large number of actuators is used to manipulate the characterized object. Commonly performed MiS related tasks include grasping and moving of the characterized object. A single actuator is responsible for performing simple one-dimensional operations such as linear or rotational movement. The manipulation operations often require cooperation of several actuators in order to provide functionality in multiple dimensions. In such cases, the actuators may be physically coupled together to create a unit capable of providing movement in multiple dimensions. Such units are herein after referred as *assemblies*. Assemblies containing *N* actuators are denoted as *N*D assemblies, where N represents degrees of freedom of the particular assembly. The actual manipulation of the target object is performed with end-effectors attached to assemblies. Cooperation of multiple assemblies is required when multiple end-effectors are involved in the same manipulation operation.

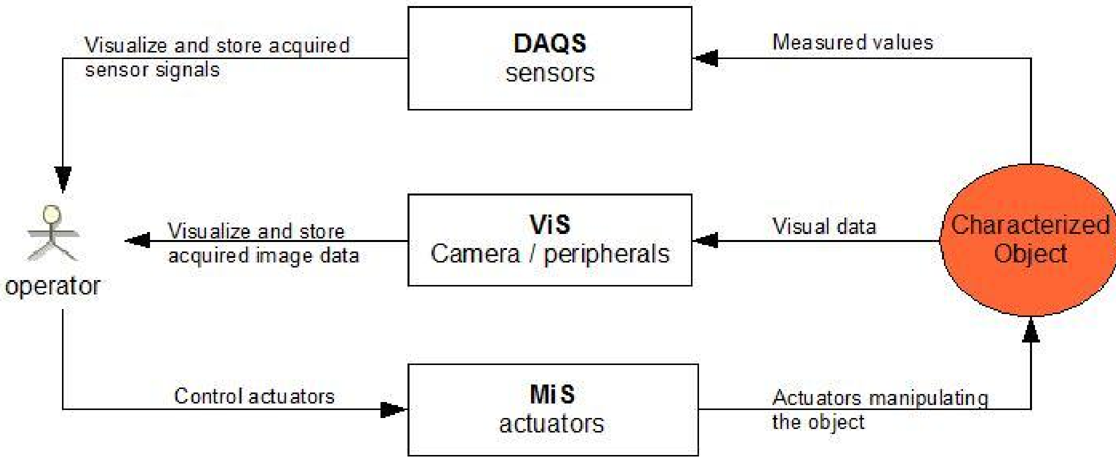

Figure 3.1 Overview of the system

Vision System (ViS) provides the user with image data regarding the characterized object and position of each end-effector. The hardware of ViS may include several cameras and related peripherals, such as objectives and illumination systems. The third part, Data Acquisition System (DAQS), is responsible for measuring different properties of the characterized object. Hardware of DAQS contains sensors for measuring different properties of the characterized object.

Characterization of an object with MP may include multiple phases depending on the characterized object and the measured properties. Prior to the actual characterization procedure, the characterized object must be located and identified using the ViS. After the object has been located, a sequence of micromanipulation operations using the MiS may be required. In a typical case, the MiS is used to grasp, move or align the object to desired position for further analysis. In the next phase, interesting properties of the object can be measured using the sensors of DAQS. Additionally, ViS can be used to measure properties, such as length of the object, from the acquired image data. Figure 3.2 presents a simplified characterization procedure. Responsible subsystem for each phase is indicated with respective abbreviation.

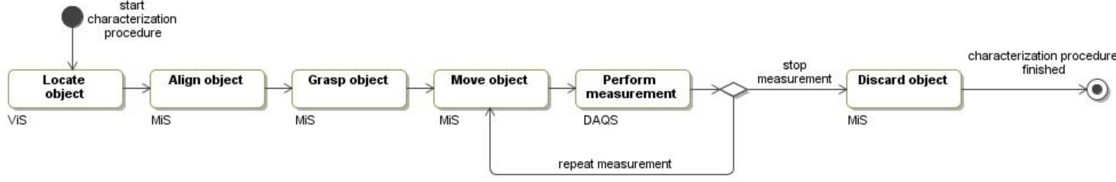

Figure 3.2 Simplified characterization procedure

The phases involving MiS may include cooperation of several actuators in order to accomplish the desired operation. For example, moving of a beam-like object may require separate actuators for both ends of the object.

3.2. Requirements for Control Software

The requirements for **Co**ntrol **S**oftware for **Mic**rorobotic Platform (CoSMic) were mainly derived from the description of usage presented in [16]. The requirements were further refined based on the user experience of series of test programs controlling different parts of the described MP. On general level, CoSMic is responsible for controlling and monitoring of all the hardware attached into MP. A high-level requirement common to all subsystems of CoSMic is modularity. Different parts of MP should be controllable through separate stand-alone applications and as a single application seamlessly integrating different parts of the system. In the single application case, the set of included subsystems should be customizable. The following presents more detailed requirements separately for each subsystem of MP. The requirements related to communication between different subsystems are presented in Section 4.5.

3.2.1. Requirements of Micromanipulation System

Most of the requirements of MP concentrate on Micromanipulation System (MiS) containing possibly a large number of actuators. The number of actuators connected to MiS is dependent on the performed characterization procedure. Thus the requirement of scalability is obvious.

The characterization procedures performed using MiS can be very complex and may include tens of different unit functions, such as moving and grasping of the characterized object. In addition, parts of the characterization procedure are often repeated multiple times. In order to reduce laborious and time consuming manual control, CoSMic should be able to record and repeat the performed procedures. Some of the procedures performed with MiS require simultaneous movement of several actuators. For example, when a fibre is stretched between two actuators, both of the ends should move in a synchronized manner to maintain the alignment and a correct stretch level of the fibre. CoSMic should provide a mechanism for synchronized movement of different actuators.

Another important aspect of the control of MiS is security. Depending of the hardware configuration of MiS, the actuators have a potential risk of colliding with each other or other parts of the system. Collisions might cause errors to the performed task or permanently damage the hardware. Therefore, CoSMic should be able to prevent such situations. The issue of security arises also in a case of a system failure due to malfunction of software or hardware. In both of the cases CoSMic should guarantee that the system remains in a safe state. The current requirements regarding the level of automation are minimal. However, additional automation related requirements may arise in the future, thus CoSMic should provide sufficient extendability in order to increase the level of automation. Real-time aspects of MiS are not discussed within this section due to the limitations of the device driver controlling the hardware of the MiS presented in Section 3.3.1. The device driver and the provided API are further discussed in Section 4.2.

3.2.2. Requirements of Vision System

Vision System (ViS) consists of cameras imaging the MP and related MiS from different angles. The main purpose of Vision System (ViS) is to provide the operator with visual feedback regarding the position and alignment of the manipulated object and the actuator. ViS may also contain peripherals, such as objectives and illumination systems, required to enhance the visual information acquired by the cameras.

CoSMic is responsible for controlling and monitoring all the ViS related hardware. The most essential features CoSMic should implement include visualization and recording of the acquired image data. In addition, CoSMic shall provide mechanisms for controlling all the peripherals attached into ViS. The implementation fulfilling the aforementioned requirements must be scalable; the number of cameras and peripherals attached to the system may vary depending on the requirements of a particular characterization process. The implementation should also support the most common communication busses for cameras.

Design of ViS should take into account the possibility of using the acquired image data to perform measurements, such as measuring the area or the length of the characterized object. In more general terms, ViS should provide an interface for future implementation of a machine vision system.

3.2.3. Requirements of Data Acquisition System

Data acquisition system (DAQS) contains several different sensors used for measuring different properties of the characterized object. The number of sensors attached into the system is entirely dependent on the characterization procedure. Moreover, the measured physical quantity might be different for each sensor.

The most important single feature CoSMic must comply with is visualization and recording of the acquired data. The data shall be provided in units corresponding to the measured physical quantity. The varying number of attached sensors implies that scalability should be included into the implementation.

3.2.4. Requirements of Real-Time Controller

MP is likely to be extended with additional features in the future. The features are reached through scaling up the system with additional hardware, such as different kinds of actuators. Some of the additional features may increase performance demands for the controlling software. For example, a PI or PID controller may require real-time implementation in order to accurately control a motor or an actuator. CoSMic should provide mechanism for easy integration of real-time controllers.

3.3. Related Hardware

The hardware of MP includes a set of actuators, cameras and multiple sensors. The hardware was carefully selected to meet the functionality required in fibre

characterization. The following gives a brief overview of the used hardware, more detailed description of the hardware and the selection process can be found in [16].

3.3.1. Micromanipulation System

MiS consists of several actuators performing the micromanipulation operations required to characterize the object of interest. MiS uses linear and rotational microactuators manufactured by SmarAct GmbH [17]. Teleoperation of the actuators requires a manufacturer specific control module, which couples the actuators and a computer via universal serial bus (USB). The control module is responsible for converting the commands transferred over the USB into analogue voltage signals used to actuate the connected actuators. Internally the control module consists of two different modules, namely an interface module and a driver module. Structure and communication between different parts of SmarAct modular control system is presented in Figure 3.3. [20]

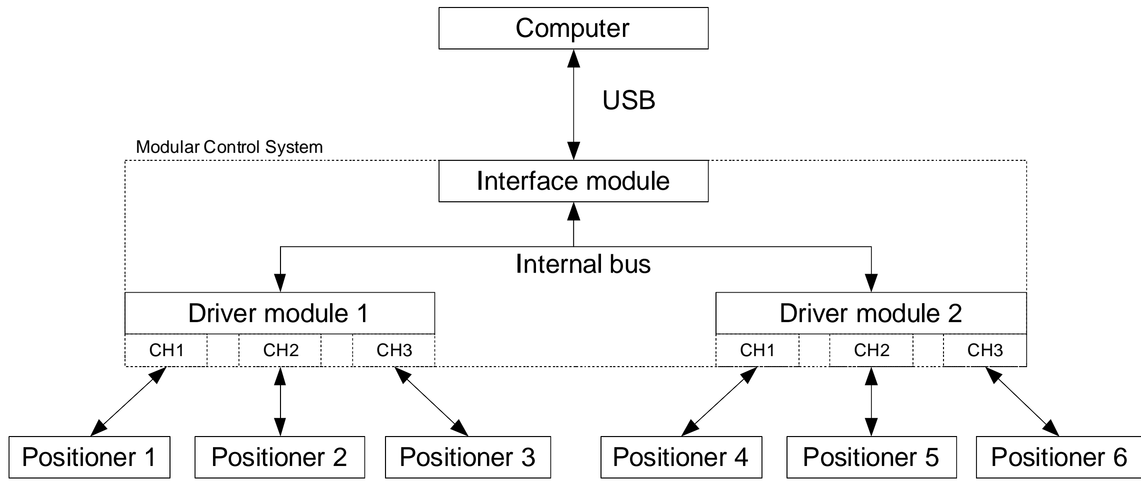

Figure 3.3 Structure of SmarAct modular control system

The interface module manages the actual communication between the computer and the control module. Each control system requires its own interface module, but several driver modules can use the same interface module. The driver module is responsible for creating the necessary signals for driving the attached actuators. A single driver module can control up to three actuators. The driver modules are capable of performing closed-loop control, providing that the controlled actuator is equipped with a position sensor and a sensor module for reading the position data is present. [17]

In its current state, MiS consists of eight linear micropositioners, two microgrippers and one rotational micropositioner. The linear micropositioners are used to create larger functional assemblies with several degrees of freedom. Two 3D assemblies with three degrees of freedom are used to move the characterized object. The 3D-assemblies include a microgripper which is used for grasping of the objects. The two remaining linear micropositioners form a 2D assembly which is used as a base for additional hardware. An overview of the system together with more detailed illustration of 3D assembly is provided in Figure 3.4.

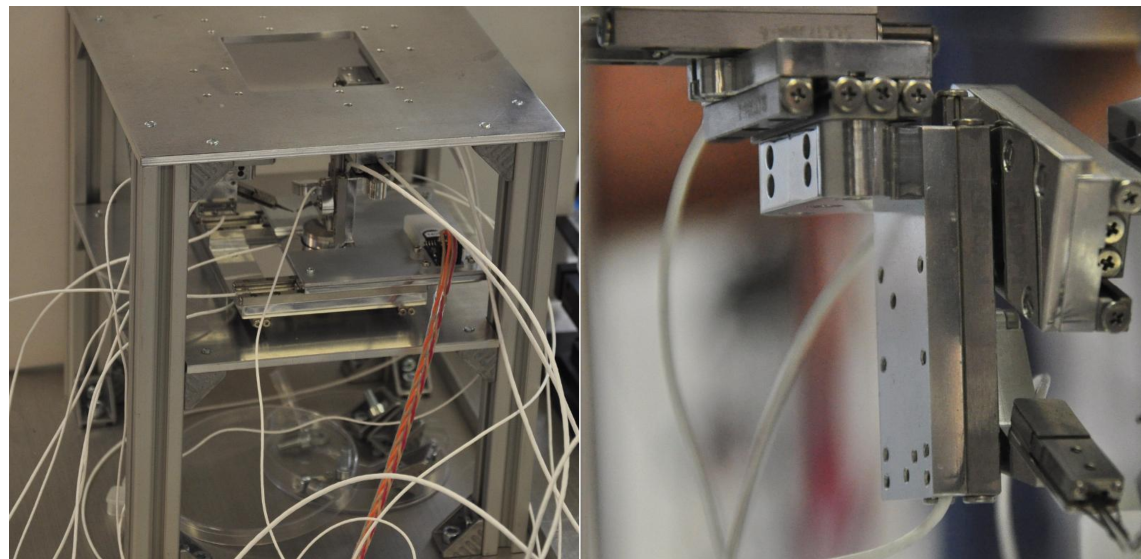

Figure 3.4 Overview of the Microrobotic Platform (left) and 3D assembly with a microgrippers as an end-effector (right)

3.3.2. Vision System

The current setup of ViS consists of a camera, an objective, and an illumination system. The communication between the controlling software and the hardware of the ViS requires several different communication lines. A schematic overview of the required communication is shown in Figure 3.5.

The used camera, SONY XCD-U100, is equipped with IEEE1394b serial bus interface compliant with the 1394 Trade Association and Industrial Control Working Group (IIDC) standard [18]. The camera provides an image size of 1600*1200 pixels with a maximum frame rate of 15 frames per second. In addition an IEEE1394b compliant card is attached into the controlling computer to enable communication between the computer and the camera.

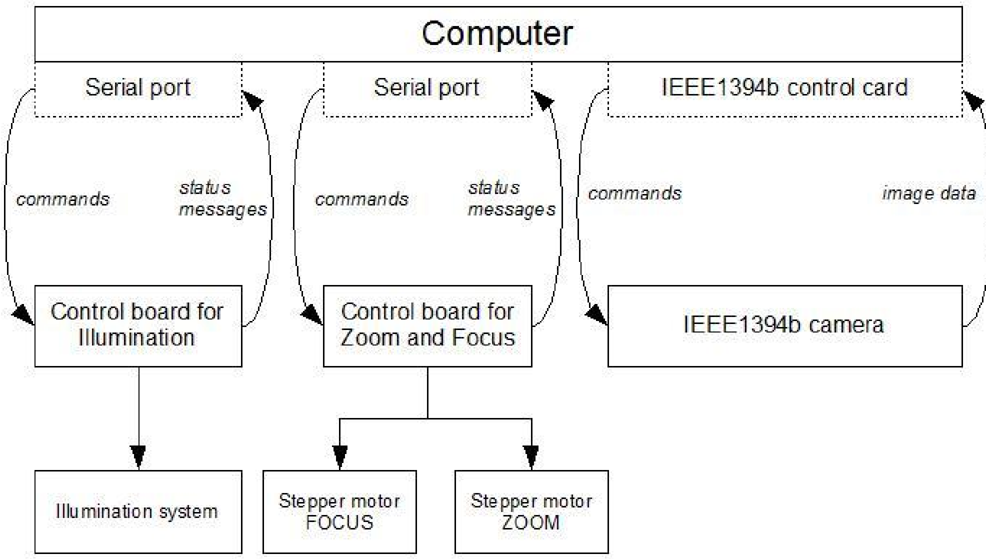

Figure 3.5 Structure of Vision System

The objective, Navitar 12x Zoom, selected to satisfy the needs of ViS includes two peripheral stepper motors allowing the adjustments of focus and zoom levels. The stepper motors are controlled via a control board including a serial communication interface using Recommended Standard 232 (RS-232) communication. Similar communication can be used to communicate with the illumination system, Navitar BrightLight coaxial illuminator based on ligh emitting diode (LED) technology . The different parts of ViS related peripherals are shown in Figure 3.5. [19]

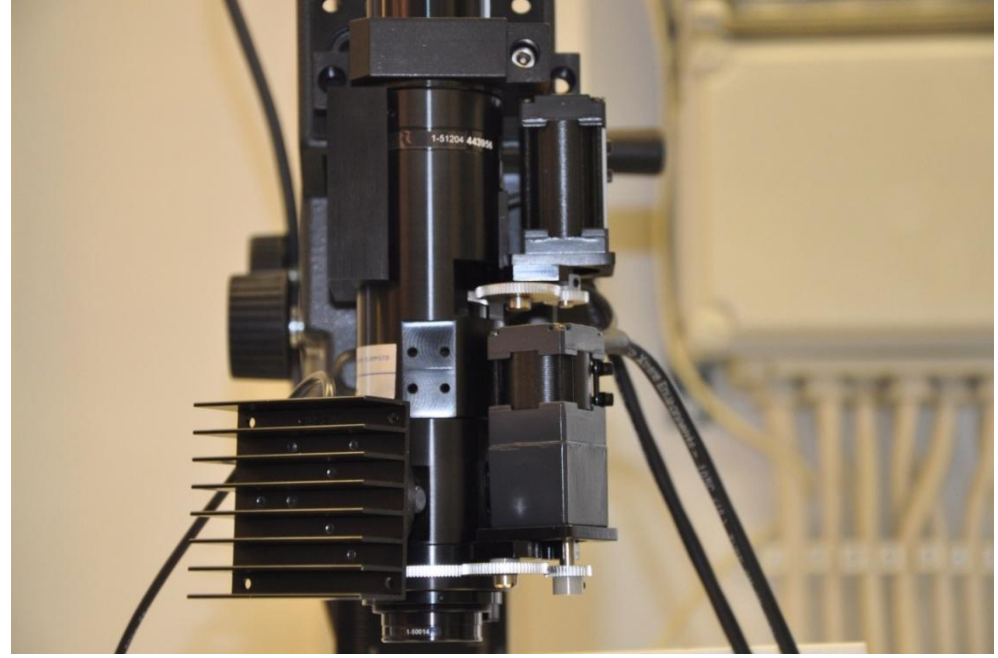

Figure 3.6 Navitar Motorized 12x Zoom objective

3.3.3. Data Acquisition System

DAQS consists of sensors, DAQ units and other DAQ related peripherals, such as amplifiers and filters. DAQS is used to measure the required properties of the analyzed object. In a typical case, the output received from a sensor is an analogue voltage signal varying in the range of ±10 Volts. In order to forward such signals to the computer system, an analog-to-digital (A/D) conversion is required.

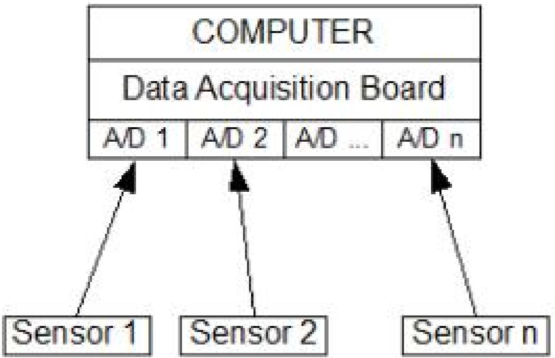

Figure 3.7 Scematic overview of the DAQS

MP resolves the issue of A/D conversions with a data acquisition (DAQ) board, National Instruments PCI-6229, providing an interface for up to 32 differential analogue voltage input channels. An overview of the data acquisition related hardware is shown in Figure 3.7.

Currently, there are two sensors attached into the system. Sensors, namely FT-S270-OEM and FT-S540-OEM, are capacitive force sensors manufactured by Femtotools. The sensors measure forces in the range of hundreds μN producing output voltage of 0-5V [16]. Figure 3.8 presents configuration of MP including a force sensor.

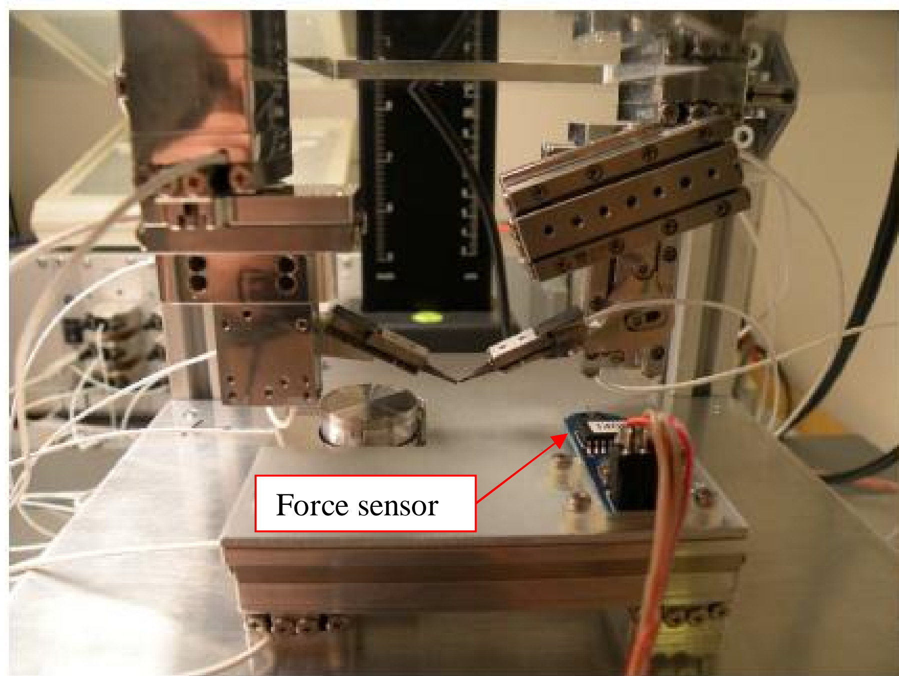

Figure 3.8 Force sensor attached to Microrobotic Platform

3.4. Control Modes

Beginning of Chapter 3 stated that the desired throughput of MP is from several tens to hundreds of fibres per day. In order to reach such a high throughput, a certain level of automation in the control of MiS is required. The following describes three different control modes for MiS proposed to be implemented in CoSMic.

3.4.1. Manual and Semi-automatic Control Modes

Manual control mode (MCM) is the simplest of the three proposed control modes. The operator controls MiS through a graphical user interface (GUI) or uses an additional input device, such as a joystick or a haptic device. In MCM, CoSMic is responsible for performing collision detection and transferring legal commands to the hardware. MCM allows the operator to be in charge of all operations carried out in the system and is ideal for testing new operations and solving possible error states of the system. In addition, MCM can be used by the developers during implementation of new parts of the system. The downsides of MCM are repeatability and performance. Repetition of an existing characterization procedure in this mode is difficult, if not impossible. In order to repeat even a single sequence, the operator should remember the exact location of each actuator throughout the entire sequence. However, MCM provides the operator with possibility of recording movements of each actuator. Working principle of MCM is presented in Figure 3.9.

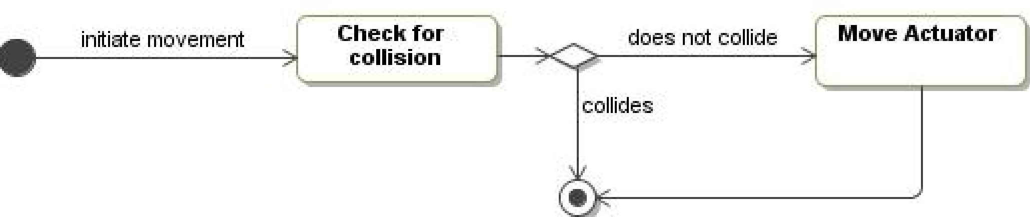

Figure 3.9 Manual Control Mode

Semi-automatic Control Mode (SCM) allows the operator to re-execute the movements recorded in MCM. SCM improves the repeatability of the performed characterization procedures. In addition, the movement sequences are performed faster. However, the system still lacks capability of decision making, which limits its performance. In a case of an abnormal situation, the system is not able to perform without user interference.

3.4.2. Enhanced Manual Control Mode

Enhanced Manual Control Mode (ECM), presented in Figure 3.10, introduces decision making in control of MiS. The goal of ECM is to provide significantly faster manual control, through optimization of the movement paths. In cases where the movement of an actuator does not cause collision, ECM is identical with SCM. The difference between the two control modes can be seen, when the movement of an actuator would cause collision. In such cases, ECM calculates optimum route to the destination and automatically redirects the actuator to newly calculated path.

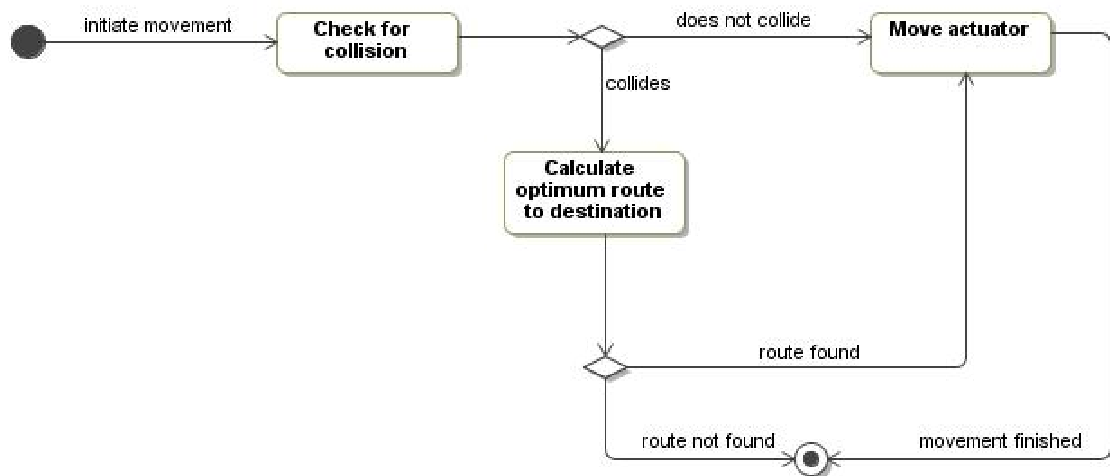

Figure 3.10 Enhance Manual Control Mode

3.4.3. Automatic Control Mode

Automatic Control Mode (ACM) aims to perform entire characterization procedures without any human intervention. ACM combines the previously presented modes to achieve a mixture of predefined trajectories and decision making capabilities. Several parts of the characterization procedures can be converted to simple trajectories using MCM. However, more complicated actions, such as picking up an object without a priori knowledge of the exact position cannot be performed with the methods of MCM. To overcome this issue, the ACM uses the feedback of ViS. The exact position of the characterized object is analyzed from the image data and can be converted into a trajectory for the actuators. Similarly, the output data of all the sensors of DAQS can be used as feedback when necessary.

4. SELECTION OF IMPLEMENTATION TECHNOLOGIES

Selection of implementation technologies for CoSMic involves several fundamental decisions such as selection of supported operating systems, possible usage of different software frameworks and selection of implementation technology for proposed real-time extension. Moreover, the possible limitations of the used hardware must be studied.

This chapter targets the aforementioned problems related to the selection of implementation technologies. Structure of the chapter is divided into three. Section 4.1 presents selection of the application development framework. Section 4.2 introduces an application programming interface used for the control of SmarAct piezoelectric actuators. Section 4.3 describes selection of the data acquisition library, followed by selection of the collision detection library presented in Section 4.4. The last part, Section 4.5, reports the key findings affecting the architecture and design.

4.1. Qt – an Application Development Framework for C++

Application development framework (ADF) can be defined as a collection of common software routines that provide a foundation for application development. Functionality provided by an ADF may cover several aspects such as cross-platform portability, network communication, concurrency and user interface technology. The developers benefit from usage of ADF through time saving and reduction of potential errors as the most used routines are implemented on the ADF level. Another clear benefit is unification of the produced source code; application development frameworks tend to guide the design process by promoting the usage of certain patterns and mechanisms. However, the aforementioned arguments also involve potential pitfalls. The selection of the ADF should be based on the requirements of the development team and the developed system. Rather than confining to the limitations of ADF, the development team should select an ADF which promotes their own thinking and the planned architecture. The following presents the application development framework selected to support the development of CoSMic.

Qt [kju:t] is a well established C++ application development framework suitable for development of high-performance cross-platform applications. Qt includes an extensive class library with over 400 classes and tools for application development. The framework has been used in commercial applications since 1995 and is currently estimated to be used by some 350 000 developers around the world. The wide range of

operating systems supported by Qt includes Microsoft Windows, Linux, Unix and OS X. Even though originally developed exclusively for C++ developers, the usage of Qt can be extended to other programming languages. Java is officially supported through binding known as Qt Jambi and a variety of third party solutions covers programming languages such as Python, C# and Ruby. [21][22][23][24]

Qt application development framework was selected for the purposes of CoSMic due to its strong cross-platform support. Qt brings several other benefits, which are briefly described in the subsequent sections.

4.1.1. Signals and Slots

The greatest strength that Qt brings to C++ is the used meta-object system, which enhances Qt objects with additional data at compile-time. The meta-object system enables Qt to provide extended run-time type information and other dynamic features, such as run-time object introspection. The most important single feature of the meta-object system is a flexible mechanism to interconnect objects known as "signals and slots". Objects can define *signals* that they emit when certain conditions are met. Signals appear as member functions prototypes, as they have only declaration. Objects can also have *slots* which are able to react upon a received signal. Slots look like normal member functions, but have gone through specific pre-processing. The pre-processing is further investigated in within this chapter. [24][26]

The usage of the signal-slot mechanism has several benefits. A single signal can be connected to any number of slots, allowing several objects to react on the same trigger. Respectively, a single slot can be connected to several signals, a useful feature if several different inputs should be processed in a similar manner. Signals are allowed to cross thread boundaries allowing a convenient way for asynchronous communication in concurrent environments. The signal-slot mechanism has a few restrictions, which must be fulfilled. The implemented class must [26]:

- directly or indirectly inherit *QObject*, which is the base class of all Qt objects
- use Q_OBJECT macro definition, which enables the signal-slot mechanism
- register emitted data types using Qt meta object system, unless primitive data types are used.

The mentioned requirements are illustrated in a form of an example in Appendix A. The example is not usable as is, but aims to highlight the usage of the signal/slot mechanism. The presented signal/slot mechanism has notable similarities with the Mediator design pattern presented in Section 2.2.

4.1.2. Threading in Qt

Qt supports multi-threaded applications through various classes which represent threads and the common mechanisms for protecting critical sections of the program, namely, mutex and semaphore. In addition concurrent programming with Qt benefits from reentrancy and thread-safety of most of the Qt classes [26].

The class for creating individual threads *QThread* is, like most of the Qt classes, inherited from *QObject*. *QThread* provides platform-independent threads by employing native threading mechanism of each platform. In Linux and Unix environments *QThread* is built to use POSIX threads, whereas in Microsoft Windows threads provided by the Win32 API (Win32 thread) are used [26]. Each instance of *QThread* has its own event loop, which is responsible for waiting and dispatching incoming events and messages. The previously presented signal-slot mechanism uses the event loop for communicating across thread boundaries.

4.1.3. Qt Integration with Real-time Operating Systems

The fact that Qt uses native threading for each platform allows execution of QThreads on real-time operating systems. The number of studied real-time operating systems compatible with QThreads was reduced to two potential options candidates on brief testing and the fact that both of the tested operating systems supported data acquisition. Two different Linux real-time kernel extensions, Rtai and Xenomai were tested with a small program executing a QThread in a real-time task. In both of the cases, QThread was proven to run as a real-time thread. Thus usage of Qt framework does not limit selection of real-time kernel extensions. Short test program used to run QThread under Xenomai is presented in Appendix A.

4.2. Application Programming Interface for SmarAct Micropositioners

Micromanipulation System (MiS) is based on piezoelectric micropositioners manufactured by SmarAct GmbH. The positioners can be teleoperated by using an additional control module coupling the positioners with the controlling computer. The control module and the computer communicate via USB. The control module is shipped with necessary drivers and an application programming interface (API) which allows the developers to write programs for controlling the positioners. The API, known as SCU3DControl, consists of a dynamic-link library (DLL) and a header file written in C.

4.2.1. Communication Modes

The SCU3DControl introduces asynchronous and synchronous communication modes for communication between the control module and the computer. Identical functionality is provided in both of the communication modes. The difference between the two modes is the mechanism how the calls block the calling program.

In synchronous mode, the calling thread is blocked until the called function has been finished. Result of the requested operation is passed as the return value of the performed function. In contrast, the function calls made in asynchronous mode return immediately and blocking does not occur. Responsibility for retrieving the resulting values is the left for the developer. SCU3DControl API differentiates functions of

asynchronous and synchronous communication modes by appending the function name with '*_A*' or '*_S*' in respective order.

Within this work, all communication between CoSMic and the SmarAct control module is performed in asynchronous mode. The goal is to prevent unnecessary blocking of the threads controlling the actuators. Furthermore, the asynchronous mode allows an event based mechanism for reacting upon finished movements or status changes.

Overview of Asynchronous Communication Mode

The asynchronous communication mode, presented in Figure 4.1, separates the sending of the commands and the answer retrieval to functionality provided by the API. When a function of the API is called, the DLL transmits corresponding command to the control module, which invokes the required functionality on the hardware level. The return value of the initially called function contains only information describing whether the hardware received the command or not. Further error handling is made in the answer retrieval process.

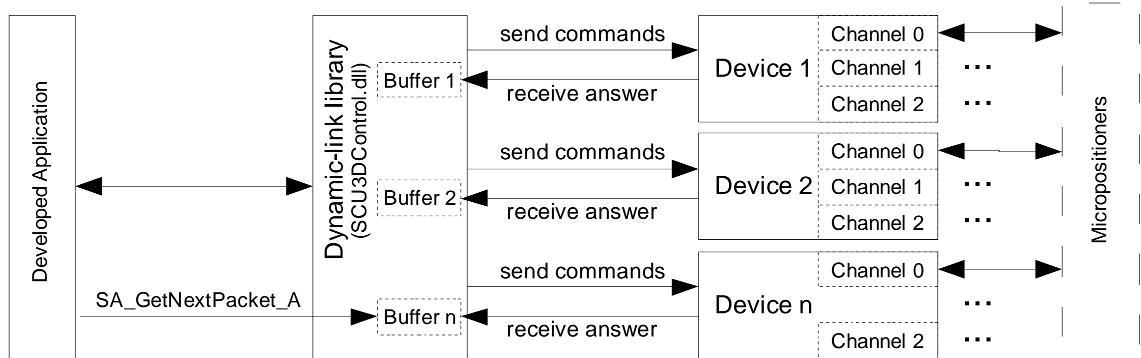

Figure 4.1 Communication between SCU3DControl API and SmarAct micropositioners

The data packet, containing the response of the hardware, is forwarded through the control unit to the DLL, which stores the answer data in a FIFO buffer, dedicated to the particular device. The API provides functions for inspecting and retrieving data packets from the device specific data buffer. In a typical case `SA_GetNextPacket_A` function can be used to fetch the next data packet of the buffer. Both directions of the communication are multi-threaded, thus providing parallel communication between the developed application and each of the attached devices.

Event Driven Communication

An additional benefit from the asynchronous communication mode can be gained through the usage of event driven answer retrieval implemented in the SCU3DControl API. The application developer may create event objects which are registered to one or several of the data buffers. Whenever a new packet is stored into the buffer, the event is activated. Thus the application may wait for an incoming event without need for constantly polling the incoming traffic.

Benefits of the event driven communication come more obvious in development of a multithreaded application. One of the threads can listen to the incoming message, while other threads are performing other operations, such as sending commands to move the positioners. When answer messages are received from the hardware, the listening thread awakes and required operations can be performed. Optionally the listening thread may forward the answer message to other threads. The event driven communication mode also provides possibility of receiving an answer message whenever movement of a positioner is accomplished.

The event based answer retrieval is based on usage of Windows API event objects. In order to use the event driven answer retrieval, the developer must create and register an event object, wait for activation of the created event object and inspect the content of the received answer message. The usage of the required functions is further presented in the following section.

4.2.2. Control Methods

Section 3.3 described the possibility of closed-loop control for positioners equipped with a sensor. The SCU3DControl API provides the two different functions for closed-loop control. The first method for closed-loop control is movement relative to current position. For example, in the situation illustrated in Figure 4.2, a relative movement of 200µm from initial position p_1 would move the positioner to position p_1+200. The second method for closed-loop control performs absolute movement against the zero position of the positioner. For example, absolute movement to position p_1 from any given starting point has always the same outcome. The method is useful for reaching exactly same position multiple times. However, system shutdown causes position reset for all positioners. When the hardware is reinitialized, the zero position is moved to the last position prior to shutdown. SCU3DControl API provides a function capable of moving the zero position to a more suitable location, such as one of the ends of the trajectory.

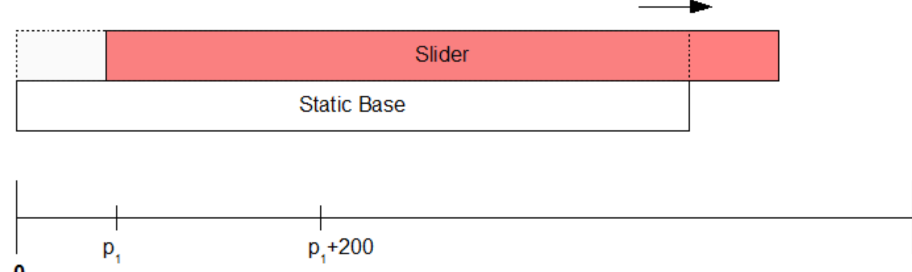

Figure 4.2 Schematic view of SmarAct linear micropositioner

Before calling functions performing closed-loop control, the type of the attached positioner should be verified. The SCU3DControl API provides functions for acquiring the type of the positioner. A simple example program describing the positioner type acquisition is presented in Appendix C..

4.3. Application Programming Interfaces for Data Acquisition

CoSMic may require data acquisition in several subsystems. Section 3.3.3 suggested that the real-time extension of CoSMic should be based on Linux with a real-time kernel extension. On the other hand, the previous section presented the SmarAct API, which runs only on Microsoft Windows operating systems. Finding a common solution fulfilling the requirements of both of the subsystems may be impossible. Hence usage of two separate software may be the most feasible solution.

4.3.1. DAQmx

National Instruments (NI) provides high-level cross-platform driver software known as NI-DAQmx, for development of data acquisition applications based on NI DAQ hardware. The driver software is mainly designed for to be incorporated with LabVIEW or LabWindows. However, the NI-DAQmx also includes an API written in C which enables development of DAQ applications with standard C and C++. Currently, NI-DAQmx is available free of charge. [27]

Data acquisition with NI-DAQmx Base C API is based on callback functions. Callback functions are implemented by the developer, but used by the NI-DAQmx; Callback function is assigned to the NI-DAQmx by providing a function pointer to the callback function. NI-DAQmx calls the functions through the pointer when acquired data is ready. This approach allows the developer to implement data exchange separately for each application.

4.3.2. Data Acquisition in Real-time Linux

Linux control and measurement device interface or shortly Comedi, is an open-source project developing device drivers and tools for data acquisition. Currently Comedi supports more than two hundred different DAQ boards including several different manufacturers. Comedi consists of three separate parts. The first part including the core functionality is a package of device drivers which are loaded into the kernel space. The second part, **comedilib**, is a separately distributed package enabling user space access to the loaded drivers. In addition, a variety of different utilities is included in comedilib. Kernel mode **kcomedilib** is the third part of Comedi providing same interface as comedilib, but in kernel space. Kcomedilib is suitable for situations, where Comedi is used from real-time tasks of the supported real-time kernel extensions RTAI and RTLinux. [comedi.org]

However, Comedi cannot be used together with Xenomai – a Linux real-time kernel extension derived from RTAI. The main reason why Xenomai and Comedi cannot be incorporated is the device driver model used in Xenomai. The functionality of Comedi is provided in Xenomai through Analogy which is a fork of the Comedi project.[10]

4.3.3. Selection

All three abovementioned data acquisition APIs may be used in development of CoSMic. However, few restrictions should be followed. DAQmx should be used only, if support of Microsoft Windows operating systems provides clear benefits. In all other cases, especially when RT capability is required, Comedi or Analogy should be selected.

4.4. Selection of Collision Detection Library

Selection of suitable CD library was based on brief testing of three different open-source collision detection libraries. The tested libraries included RAPID, SWIFT++ and CollDet. All of the mentioned CD libraries are designed for CD of rigid bodies, satisfying the current requirements of CoSMic. The quantitative requirements for CD in CoSMic remain unknown as the entire concept of CD is new to the development team. However, the performance of the aforementioned libraries is expected to satisfy the requirements of CoSMic. In the current state, the most important factors affecting the selection of suitable CD library include:

- Usability
- Programming language of the library should be C or C++
- Cross-platform support
- Possibility to visualize the moving objects and collisions

The first tested library, RAPID, is developed in C and does not have any dependencies to other libraries. Pure C implementation indicates cross-platform support, which was further verified on Microsoft Window XP and Linux operating systems. API of RAPID is very simple, the developer is provided with only four functions. Three of these functions are used in building of objects and the remaining function is responsible for performing collision queries. Despite the simplicity of the API, RAPID appears to be relatively complicated to use; the modeled objects must be constructed from triangles, which can be very difficult task in case of complex objects.

To overcome the problem of building models from single triangles another library, SWIFT++ was tested. SWIFT++ is completely written in C++ and includes cross-platform support similar to RAPID. SWIFT++ implements an object importer allowing usage of objects modeled with different tools. The greatest issue found in the testing of SWIFT++ was lack of support for current compilers. In both, Windows XP and Linux, the provided source code did not compile without several modifications. Even after successful compilation, the library did not function as reported.

The third library – CollDet appeared to be the most promising solution for the requirements of CoSMiC. The following section provides an overview of CollDet.

4.4.1. CollDet

CollDet is a CD library developed by Computer Graphics group of Clausthal University of Technology. The primary application domain of CollDet is 3D real-time applications. The cross-platform support of CollDet is similar to the two previously presented CD libraries. One of the key differences of CollDet when compared against SWIFT++ or RAPID, is that CollDet does not manage the models itself. An additional library called OpenSG is used to move and store the objects.

OpenSG is an open-source cross-platform scene graph[1] library designed to provide an API for development of real-time 3D graphics programs, such as VR applications [41]. CollDet uses OpenSG to store the objects, their alignments and positions.

Functionality of CollDet is based on a simplistic API providing only functions necessary for the developer. In a general case the developer is required to use only three different functions of the API. Firstly, object of class describing collision pipeline is constructed to establish a new CD pipeline. In the second phase, each participating object is registered as an input of the pipeline by calling. The final step is to provide the pipeline with information regarding collision response, which CollDet handles through virtual callback class. Thus different collision responses can be defined for each object pair.

4.4.2. Testing of Selected Collision Detection Library

The usability and basic functionality of CollDet was tested by developing a simple program simulating collision detection between a moving 3D assembly and a static target object. The used object models are presented in Figure 4.3 Graphical representation of the objects used to test functionality of CollDet library, where the 3D assembly is constructed from three heptahedrons, each representing single linear actuator. The static target object is modeled as a sphere.

[1] Scene graph is a hierarchical data structure storing representation of a graphical scene.

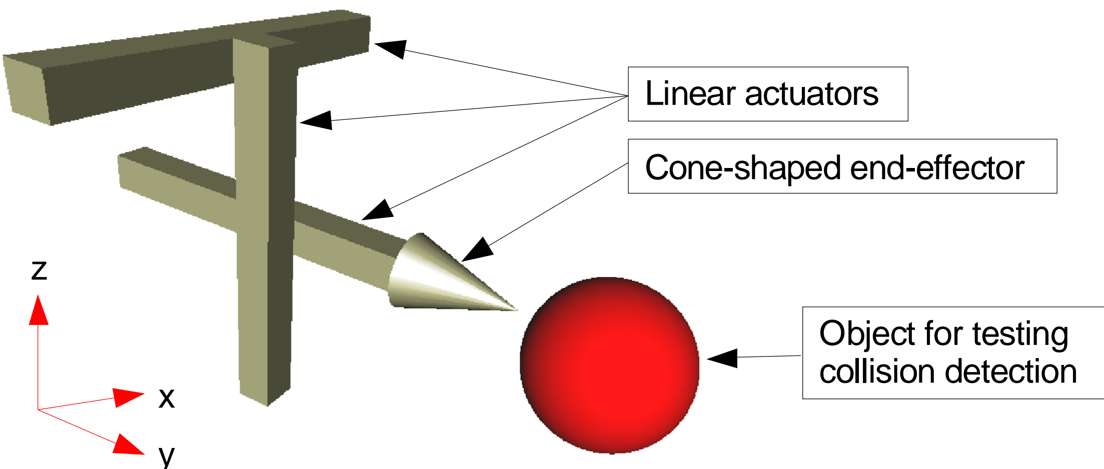

Figure 4.3 Graphical representation of the objects used to test functionality of CollDet library

The scene graph storing the information required to produce the presented graphical scene is described in Figure 4.4. The graph scene is built mainly from two different kinds of nodes, namely transformation node and geometry node. The latter nodes are responsible for describing the geometry of a single object, whereas the first contains information regarding position and rotation of the object. In addition, a root is required to maintain the tree structure. CollDet must be provided with the geometry nodes of each object of interest, transformation information is retrieved automatically by traversing the scene graph tree.

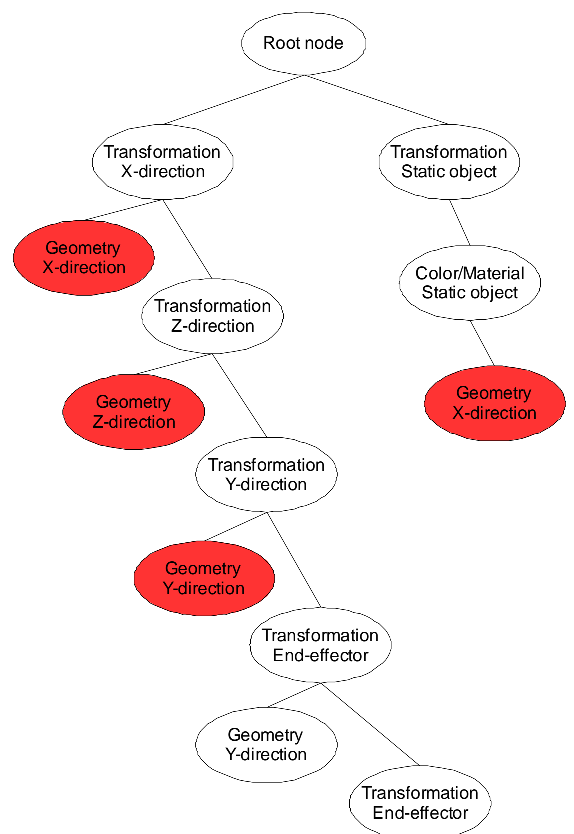

Figure 4.4 Structure of a scene graph representing a 3D assembly and spherical target object

Creation of simple objects and scenes using OpenSG was found to be relatively easy, especially when in-built primitive geometries, such as heptahedrons, cylinders and spheres provide sufficient accuracy. If more complex objects are required, third-party modeling tools, such as Autodesk 3ds Max [42] or Blender [42] can be used to create the objects.

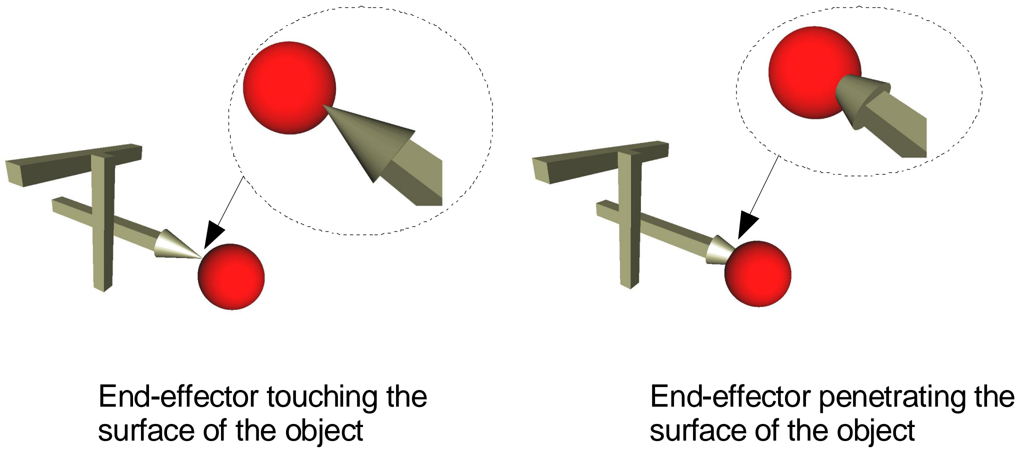

Figure 4.5 Collision detection with CollDet

Implementation of CollDet on top of the OpenSG scene graph was found to be effortless. Collision pipeline was created and the geometry nodes were registered with the respective function. A simple callback class calculating the amount of occurred collisions was implemented to test how accurately CollDet performs in this particular case. Different parts of the 3D assembly object were collided to the spherical target object in order to find out whether CollDet detects all collisions or not. Figure 4.5 illustrates two different cases where collision was found. In the first case, the end-effector touches the surface of the sphere, but does not penetrate it. In the second case, the end-effector completely penetrates the surface of the sphere.

4.5. Key Findings

The requirements relating to scalability and performance of each subsystem are very different. In addition, one of the required APIs is available only in Microsoft Windows operating systems. Many of the requirements indicate that selection of a distributed architecture would be justified. Moreover, the requirements for real-time capable subsystems as well as the proposed machine vision functionalities call for high computational capabilities. The functionality required from the entire control software is recapitulated in Figure 4.6. The key components in the scope of this work are indicated with red colour.

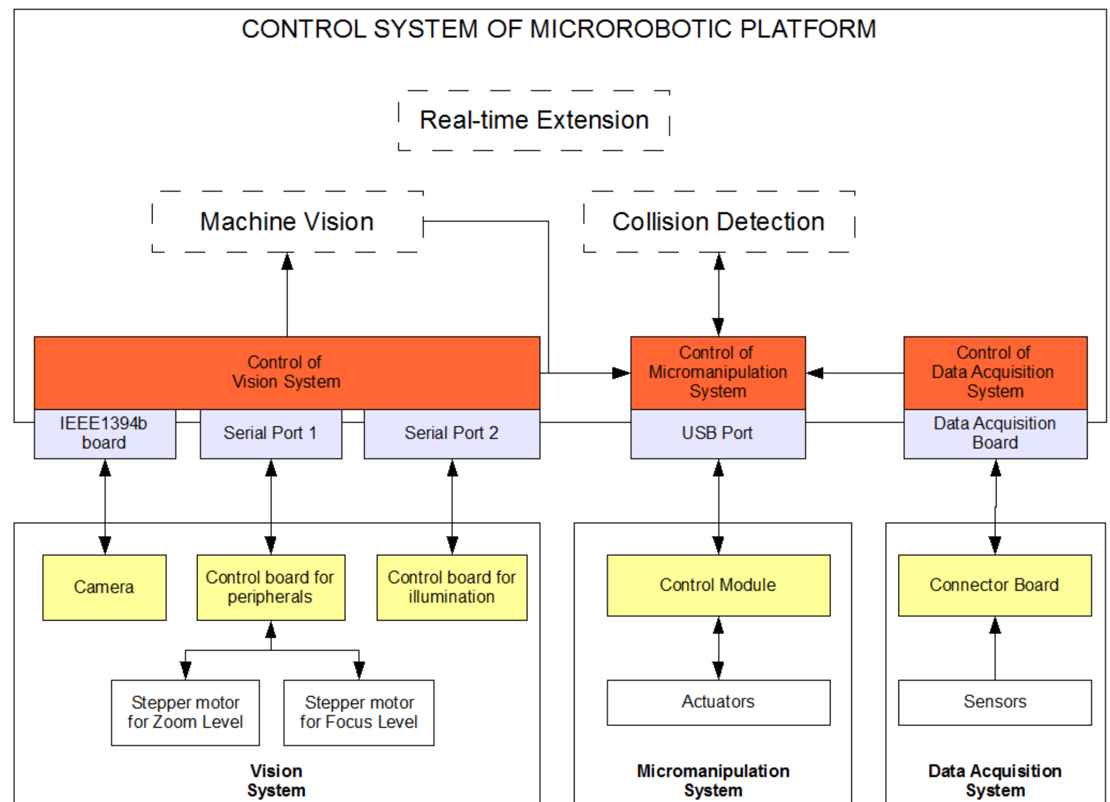

Figure 4.6 Proposed functionality for the Control System of Microrobotic Platform

The limitations of the hardware were studied mainly from the perspective of support for the selected operating systems. Other possible hardware related constraints were left to be further studied in the architectural design. Existing hardware were found not to severely limit the development as most of the presented components are capable of communicating with standard bus technologies implemented in most of the modern operating systems. In most cases, the required API is available directly in the OS and further libraries are not required. Both of the control boards related to the ViS are capable of standard serial communication using the RS-232. Moreover, the used camera employs communication based on the IEEE1394 standard, which is also well established in most of the operating systems. [18][19]

The proposed structure of DAQS does not pose any operating system related issues; a large variety of different DAQ devices are supported by all of the operating systems as presented in the previous section.

The MiS is the only part of the system that limits the selection of operating system. The communication over USB between computer and the SmarAct interface module requires proprietary libraries available only for Microsoft Windows. However, the requirements of MiS can be limited to the computer node responsible for controlling MiS, other parts of the system may run on different OS, providing that the communication between the nodes is platform independent.

4.6. Selected Technologies and Design Principles

The presented user requirements and available hardware interfaces led to selection of a distributed architecture. CoSMic will propose usage of Linux on all possible computer nodes. The subsystems with real-time constraints shall run on a dedicated computer powered by Linux and a real-time kernel extension. Selection of used real-time kernel extension is left for the developer of the particular subsystem.

CoSMic will be developed in C++ incorporating the Qt application development framework. The proposed primary selection for data acquisition is Comedi or Analogy, depending on the selected kernel extension. However, if a computer node running Microsoft Windows requires DAQ capability, the NI-DAQmx from National Instruments should be selected.

CollDet library is selected as the current approach for CD. Selection of the library is supported by the performance evaluation presented in [46]. Further, CollDet is easy to use and allows visualization through OpenSG. Other aspects promoting CollDet is multi-thread support and compatibility with Qt [47]. Implementation of CD for CoSMic is not in the scope of this thesis. However, the architecture should be designed in a manner which allows an easy integration of CD implementation in further stages of the development.

5. ARCHITECTURE

Defining the concept of software architecture is a relatively difficult task; there are literally tens of different definitions by different authors. Within this work, the following definition is used:

Software architecture describes systems organization and functionality on high level of abstraction. The most important aspect of software architecture is to collect the development team's key decisions regarding the structure, behaviour and relationships of the contributing components. [29][30]

This chapter describes the software architecture of **Co**ntrol **S**oftware for **Mic**rorobotic Platform (CoSMic). Section 5.1 presents three architectural patterns which were considered as the most prominent options during the architectural design process. Section 5.2 describes the architecture of Control Software for Microrobotic Platform on general level. Section 5.3 introduces CoSMic-Frame – a simple framework designed to be used in application development for MP. The two remaining sections concentrate on describing two subsystems of CoSMic. Section 5.4 presents the architecture of a subsystem responsible for controlling the MiS related hardware. Section 5.5 discusses the architecture of a subsystem controlling DAQS.

5.1. Selected Architectural Patterns for Distributed Computing

One of the most important architectural issues related to distributed systems is coupling of network technology and the actual program code. The issue becomes significant in scenarios where the program code should be ported to other environment or if changes are made to the used network technology. For example, in some cases the physical implementation of the used network might be changed due various reasons, such as insufficient performance of the network or companywide upgrading of the network technology. In such a case, tight coupling of the program code and the network related functionality may lead to complete rewriting of the program. A similar scenario applies for cases where application is ported to a domain where the used network technology differs from the development environment. Therefore decoupling of the program code and the network related functionality is an important topic. The following subsections introduce three different architectural patterns proposed to overcome the aforementioned issue.

5.1.1. Broker Pattern

The first pattern, known as broker pattern, is proposed for systems where independent cooperation between two distributed components is required. Main motivation of the pattern comes from separating the actual application code and the network related details; an application residing on server or client should not require knowledge regarding the implementation or physical location of the counterpart. The mechanism presented in this pattern allows the client programs to invoke methods of remote services as if they were local. [2]

In broker pattern, each node of the network should include an instance known as the broker. The main purpose of the broker is to register interfaces and locations of the local components. Registration is required to gain visibility throughout the distributed system. If client wishes to invoke functionality provided by a remote component, it invokes the local broker in order to obtain a client-side proxy, which acts as a local substitute of a registered remote component. The client-side proxy collaborates with the client and server-side brokers to forward the client's request to the remote component. Same route is used to pass the possible results back to the client. [2]

The broker pattern is often presented together with an additional component called bridge, which is responsible for encapsulating the network-specific functionality. Figure 5.1 illustrates structure of the broker pattern including the bridge instance. [2][28]

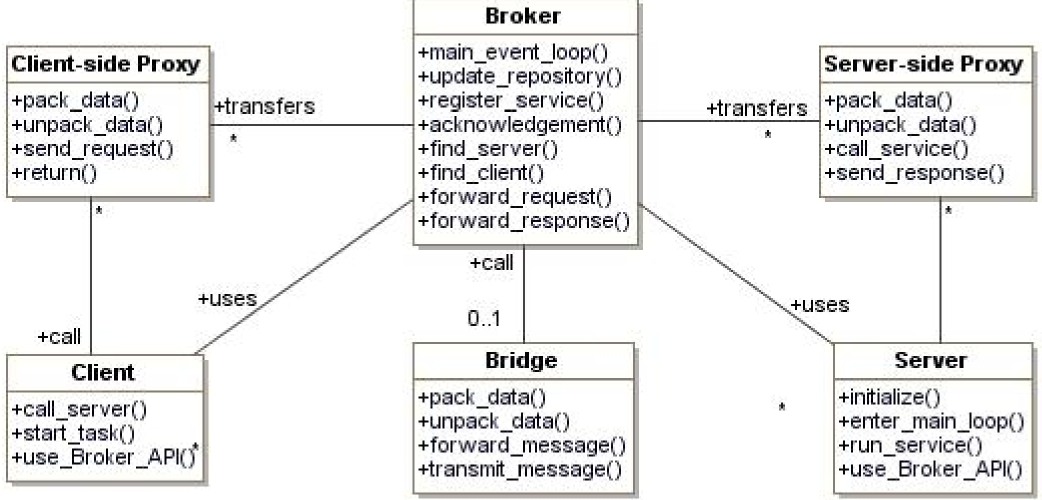

Figure 5.1 Relationships of the classes participating in broker pattern

5.1.2. Client Proxy Pattern

The second architectural software pattern presented in Figure 5.2 is known as client proxy. Client side application willing to access the services provided by a remote component must comply with data format and network protocol used at the server side. To enhance the reusability of the client side application the Client Proxy pattern adds an additional component, a client proxy, on the client sides address space. Purpose of the client proxy is to provide the client application with an interface identical with the one provided by the remote component. The client proxy is responsible for mapping all the

client side invocations to the remote component. Further, the client proxy must reinterpret the possible return messages into a format understood by its client. [2]

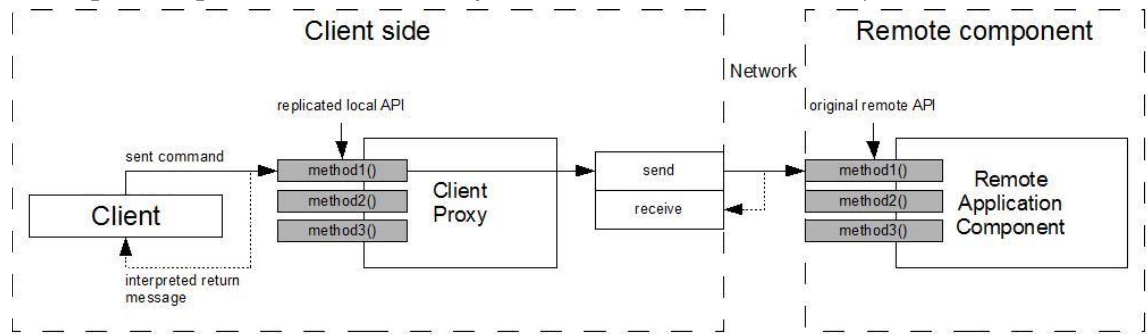

Figure 5.2 Overview of client – remote component communication with client proxy pattern

Unlike the broker pattern, client proxy is unable to achieve location-independent communication. In client proxy pattern, the client must obtain the client proxy prior to initiating communication with the remote component. Therefore the client side must be aware of the remote component's location. Client-side proxy pattern can be used when constructing a client-side broker of the broker pattern. However the pattern is also usable as such. [2]

5.1.3. Invoker Pattern

The last presented architectural pattern is called invoker pattern. Invoker pattern resembles client proxy, but the network related functionality is encapsulated on the server side. Invoker pattern encapsulates the server-side application component from the network related tasks. If the application component would manage the network related tasks itself, portability and reusability of the component would be difficult – especially if the used network technology changes. Figure 5.3 presents an overview of the invoker pattern.

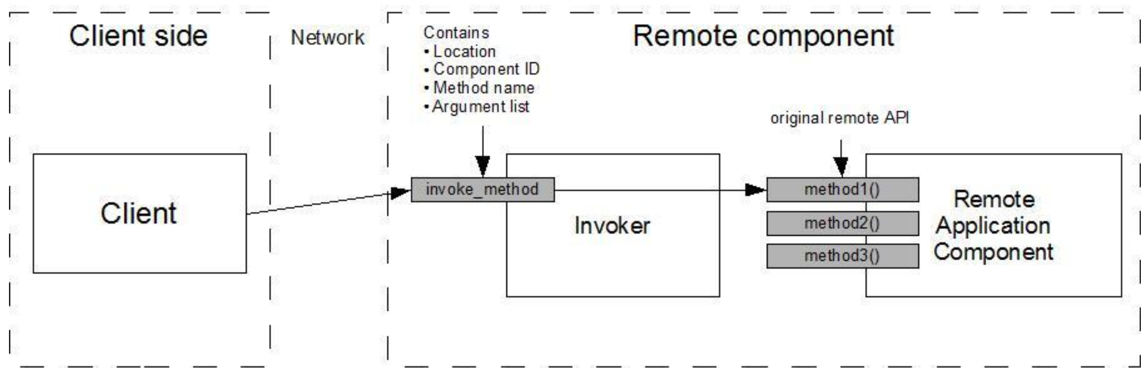

Figure 5.3 Overview of client – remote component communication using invoker pattern

Invoker pattern can be deployed in several different ways, depending on the desired level of complexity. In the simplest solution, a single invoker is deployed to serve all the components residing in the server. This solution might be feasible when the number of served components remains low. However, if the number of components is increased

dramatically, a design where each component gets served by dedicated invoker might be preferred.

5.2. Distributed Architecture for Microrobotic Platform

The distribution of the CoSMic presented herein, is based on the observation made in the previous chapters. CoSMic is divided into four packages each of which is responsible for controlling a specific part of MP. Responsibilities of the packages are as follows. Control of MiS (MiCo) is responsible for controlling all MiS related hardware. MiCo provides the operator with the functionality required in control of the MiS actuators, recording and re-executing actuator trajectories and preventing the actuators from colliding with other hardware of CoSMic. Control of ViS (ViCo) implements functionality required in image acquisition, visualization and image analysis with MV. Control of DAQS (DAQCo) is responsible for acquiring, storing and visualizing data from the sensors attached to DAQS. Real-time extension implements possible future real-time constrained subsystems. For example, the real-time extension might be needed when implementing closed-loop control with strict performance requirements.

The use cases of each part of CoSMic are presented in Figure 5.4, which also presents the intentioned distribution.

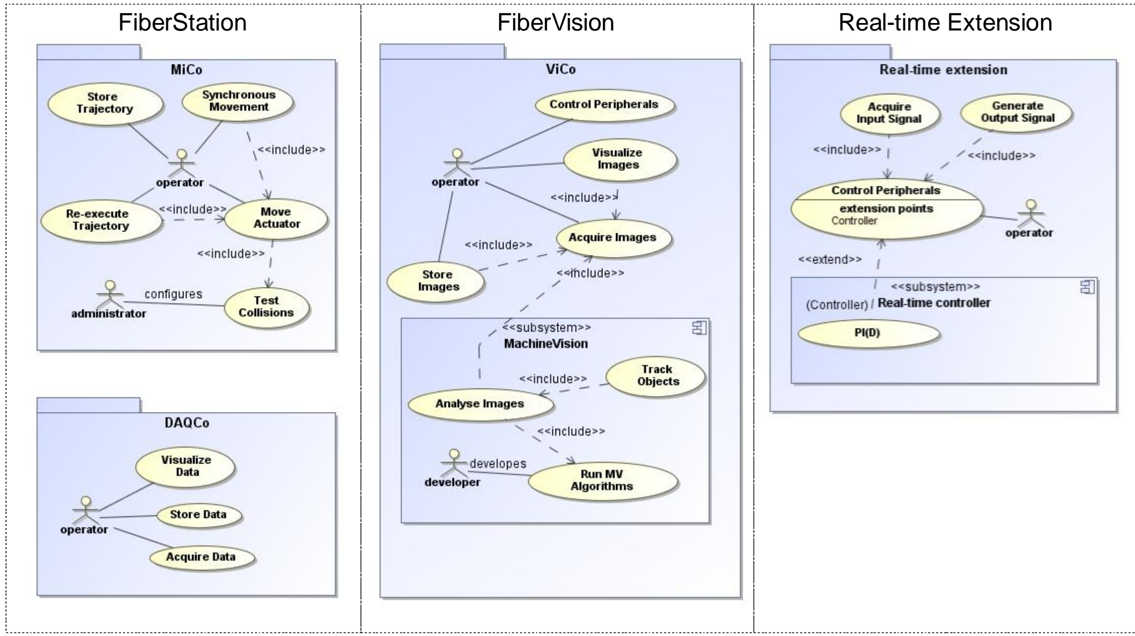

Figure 5.4 CoSMic - required packages and related use case diagrams

The presented uses cases aim to highlight the domain of each package. In MiCo, the most essential functionality is illustrated by use case *Move actuator. Move actuator* represents the functionality required to move single SmarAct actuator. It includes detection of potential collisions, described in *Test Collisions*. The functionality illustrated in *Move actuator* is also required in use cases *Re-execute Trajectory* and *Synchronous Movement*. The first provides means for re-executing trajectories stored in

Store trajectory and the latter allows the operator to move multiple actuators in synchronous manner. DAQCo contains three use cases, namely *Acquire data*, *Visualize data* and *Store data*. The first represents the functionality required in data acquisition from a sensor attached to DAQS. The second describes how the acquired data should be visualized. The third use case illustrates storing of the acquired data into a file. ViCo is divided into four separate use cases. *Acquire images* is a use case comparable to DAQCo *Acquire data*, it describes how image acquisition can be started, what parameters must be taken into account and what high-level operations ViCo performs during image acquisition. *Store images* illustrates storing of the acquired images as video file or as a sequence of image files. *Control peripherals* represents the functionality required to control the motorized objective and the illumination system. *Visualize acquired images* describes how the acquired images should be visualized. ViCo subsystem called Machine Vision represents the proposed MV system. The most significant MV related use case is *Analyse images* which analyses each acquired image prior to visualization. *Analyse images* includes another use case, *Run MV algorithms*, which describes how selected MV algorithms are executed. Possible images analysis related object tracking is described by *Track objects*. Real-time extension and related use cases are herein provided for merely illustrative purposes. More detailed analysis of RT requirements related to CoSMic must be conducted prior to further designing the RT capable subsystems.

Control of ViS (ViCo) is assigned with a dedicated computer due to the possibly high computational requirements of the proposed MV system. The actual requirements remain unknown until MV software has been implemented. However, it is safe to assume that at least one desktop computer is required to suffice for the needs of the ViCo. The computer where the implementation of ViCo resides is referred in this thesis as FiberVision. The MV is presented in Figure 5.4 as a subsystem of ViCo to emphasize the requirements it may have towards the core functionality of ViCo. The MV may require several computation intensive operations, such as copying and converting images to different formats. Architecture of ViCo should take into account the possible effects by implementing thread-safe core functionality and data buffers. This thesis does not address the architecture or the design of ViCo into more details, as ViCo is developed in a thesis work parallel to this work. The functional requirements presented in Section and the architectural requirements based on CoSMic-Frame are followed in the development of ViCo. In its current state, most of the functionality of the ViCo has been implemented. More detailed description of the ViCo is available after the related thesis work has been finalized [31].

The second computer, known as FiberStation, is proposed to host Control of MiS (MiCo) and Control of DAQS (DAQCo). The grounds for having MiCo and DAQCo in the same domain lie in the required communication between the packages. Some of the sensors connected to DAQCo may damage, if their operation range is exceeded. Fast communication between these two parts ensures that the devices of MiCo may quickly react upon certain outputs of DAQCo. The second reason is the

possible implementation of force-feedback with haptic device using sensors attached into DAQCo. Physical relocation of DAQCo would greatly increase the network traffic, as some input devices may require update frequencies of several hundred Hertz.

5.2.1. Network Communication

As mentioned previously, the communication between the different computers of network is initially planned to be implemented using TCP/IP sockets. However the architecture should allow the development team to change the used network protocol without affecting the functionality of the application code. In order to fulfill this requirement, an invoker-like pattern with a multithreaded approach is proposed.

Motivation behind the multi-threaded approach is to enable execution of several server-side applications on one single computer. Furthermore, the server hosting the applications should provide an efficient mechanism for communication between multiple clients and the server-side applications. In single threaded server, instructions sent from multiple remote locations are executed in a serialized manner. Each connection must wait until the instructions of the previous connection have been processed and forwarded to respective application. The pattern presented in the following aims to enhance performance of the server by serving each incoming connection in a different thread.

The pattern, presented in Figure 5.5, encapsulates the network related functionality into three instances. `ClientSideConnection` implements the required client-side functionality, such as client-side TCP/IP socket. `ServerListener` is responsible for listening and accepting incoming client-side connections. The third instance, `ConnectionHandler` is responsible for handling the run-time communication between client-side and server-side.

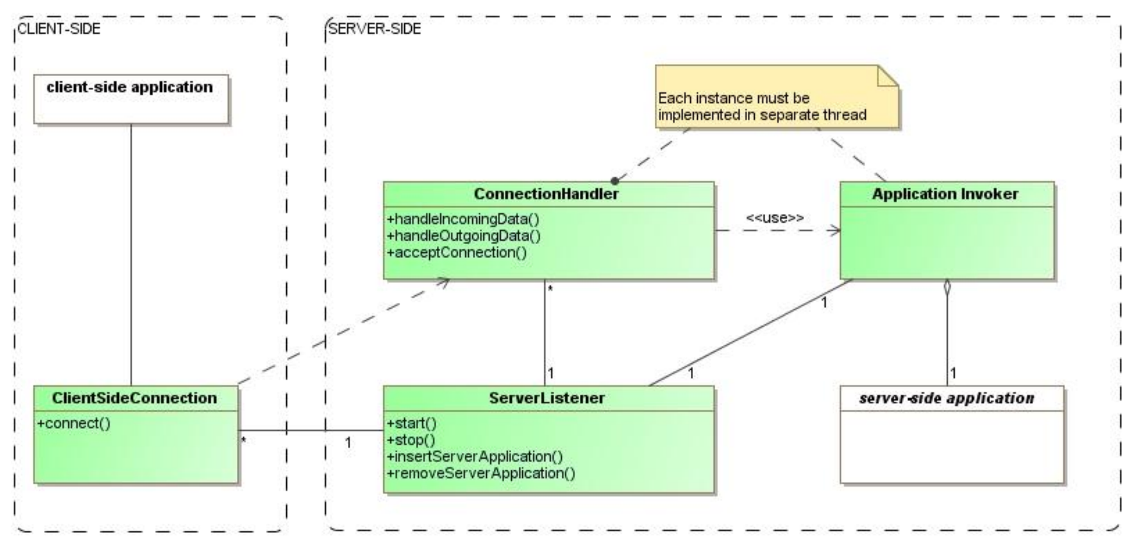

Figure 5.5 Structure of the communication related functionality

Division of network related functionality between `ServerListener` and `ConnectionHandler` was made in order to achieve concurrency between the client-

side connections. Each connection accepted by `ServerListener` is handled in separate thread dedicated to `ClientSideConnection`.

The fourth instance participating in the pattern is `Application Invoker` which provides the functionality similar to the invoker instance presented in Section 5.1.3. Serialization is avoided by creating `ApplicationInvoker` per server-side application. Thus incoming connections are queued only when multiple `Connection Handler` instances are required to call same instance of `ApplicationInvoker`.

The intended initial communication between participating instances is shown in the sequence diagram presented in Figure 5.6. The sequence assumes that an instance of Server Listener has been created and started to listen incoming connections. Before client-side application may communicate with server-side application, a communication line between server-side and client-side must to be established. The initialization of a new connection is triggered by client-side application which calls `ClientSideConnection` initiating the connection to given address of `ServerListener`. Upon incoming connection request, the Server Listener listening to incoming connections wakes up and creates new instance of `ConnectionHandler` representing the server-side of the communication of a single client. Possible tasks related to initialization of the connection, such as hand-shaking are performed by the `ServerListener`. `ConnectionHandler` replies to the `ClientSideConnection` and new connection is established. Finally `ClientSideConnection` notifies client-side upon successfully established connection.

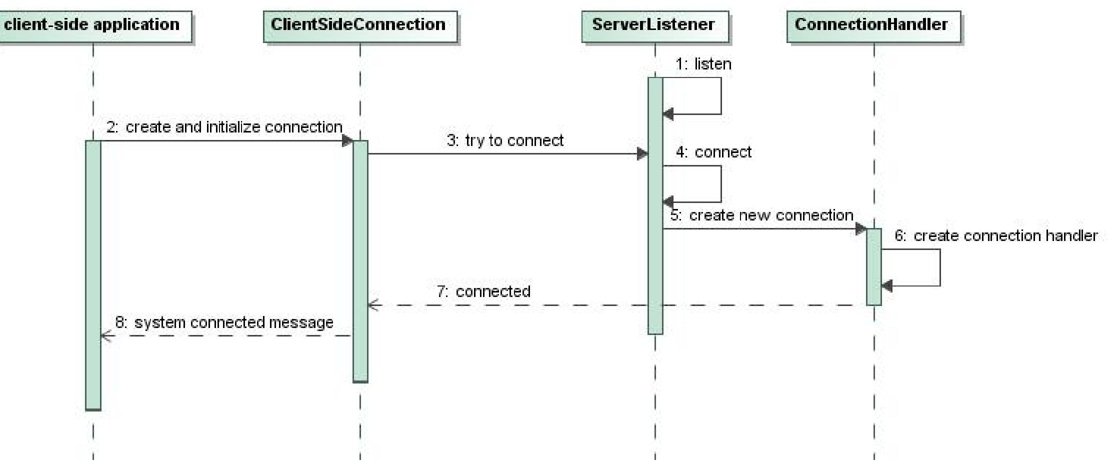

Figure 5.6 Sequence diagram presenting initialization of communication between client-side application and server.

The run-time communication between client-side and server-side applications is presented in Figure 5.7. The sequence is initiated by the client-side application, which calls the `ClientSideConnection`. The function call arguments must include identifier of the invoked server-side application, identifier of the invoked function and the required parameters.

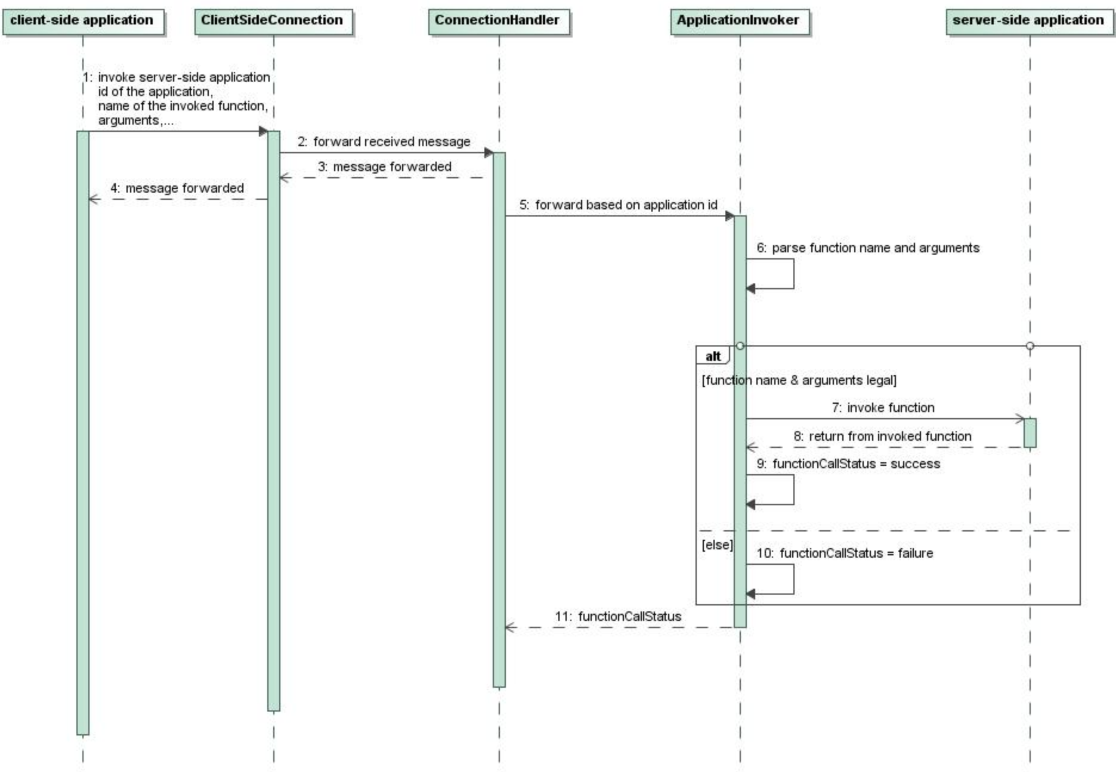

Figure 5.7 Run-time communication between client-side application and server-side application

`ClientSideConnection` parses the arguments into a message which is forwarded to the respective server-side `ConnectionHandler`. `ConnectionHandler` forwards the message to respective `ApplicationInvoker` which converts the received message into name of the invoked function and argument list. In the final phase, `ApplicationInvoker` invokes the desired function of the server-side application.

5.2.2. Communication on Network Node Level

The subsystems residing on the same physical location require communication to exchange data related to measurement results and possible error states. The data exchanged between different parts of CoSMic is typically either event based messaging or continuous flow of data from an external source, such as DAQ hardware or camera. The difference between the two methods of data exchange is the quantity of the transferred data. Therefore different approaches are proposed to satisfy the requirements of both of the presented cases. Within this section the event based messaging is not further discussed, but the Qt signal-slot mechanism provides excellent possibilities for forwarding messages between different parts of CoSMic. The possibilities of the signals and slots are presented more in details in Chapter 6.

Figure 5.8 presents a Qt framework based pattern for data exchange of continuous measurement data using a data buffer. The pattern is mainly designed for

cases, where large amounts of data are produced and only the latest data is in the interest of the consumer instance. The pattern involves three main participants `ProducingApplication`, `ConsumingApplication` and `DataBuffer`, which couples the two aforementioned. The pattern aims to hide the producer and consumer instances from each other by using another instance called `Connector` to connect the required methods. In addition, the pattern can be used for resolving the classical producer-consumer problem by implementing a container class to store several measurement values at the `DataBuffer`. Moreover, implementation of mutual exclusion (mutex) operations is required. The functionality of the pattern is defined by the structure of the `DataBuffer`.

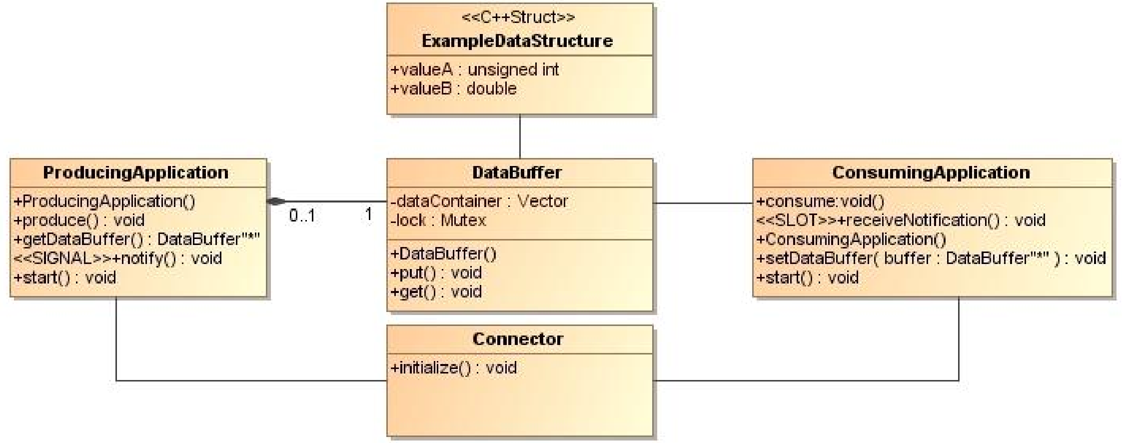

Figure 5.8 Data exchange between application threads using a data buffer

In order to maintain desired encapsulation, the initialization of the instances participating in the pattern must be performed by a third party. Figure 5.9 presents a simple initialization sequence, where an instance of `Connector` performs the initialization. The sequence starts by creating an object of `ProducingApplication` and `ConsumingApplication`. The `ProducingApplication` is the owner of `Data Buffer`, thus responsible for its creation. When all required instances have been created, the `Data Buffer` is connected to the `ConsumingApplication` through the `Connector`. `Connector` performs a query to `ProducingApplication` and receives a pointer to the `DataBuffer`. After successful acquisition of the pointer, the `Connector` forwards the pointer to `ConsumingApplication` and connects the producer-side `notify` signal to consumer-side `receiveNotification` slot.

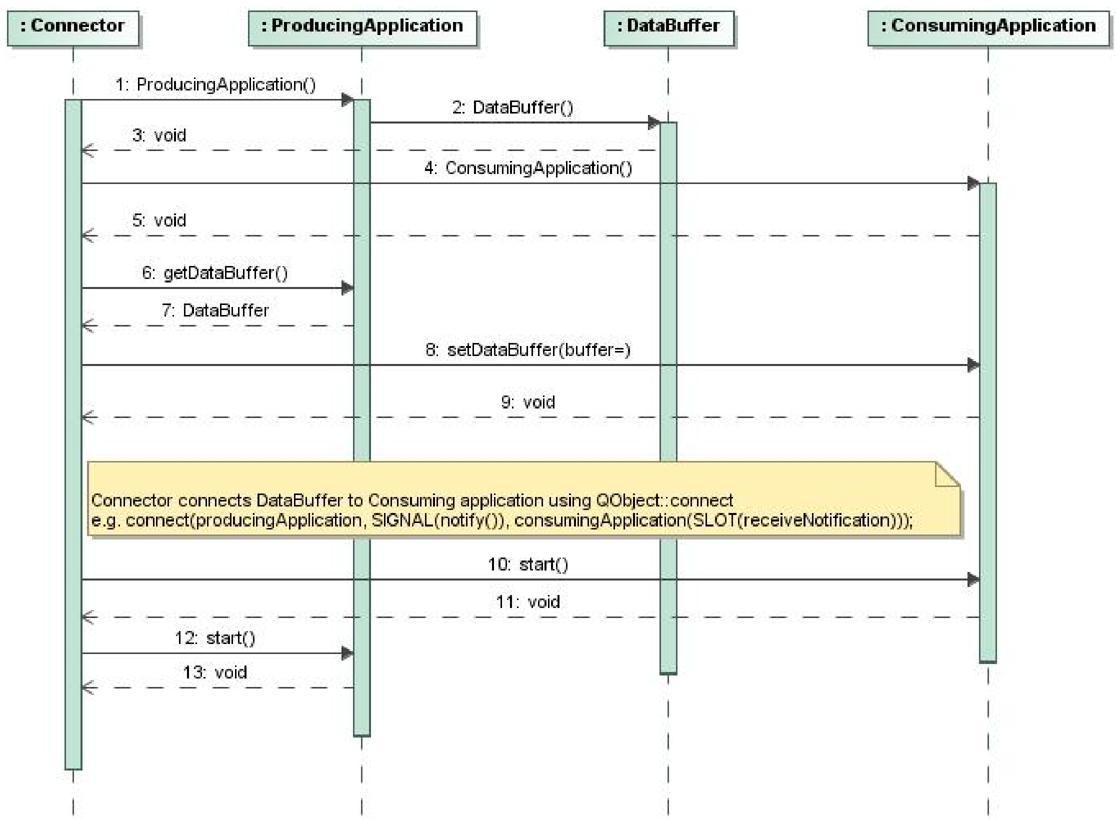

Figure 5.9 Initialization sequence of continuous data exchange using buffer

In some cases, the buffering of acquired data may require additional operations, which are not in the scope of the producer-side. For example, the consumer-side may require different data format than the producer is able to provide. Tight coupling of the required data conversions with the producer may lead to a situation where implementation of new functionality would require complete rewriting of the producer-side program. In order to prevent such situations, it might be useful to implement the data conversions into the used buffer. However, the pattern presented in Figure 5.8 should not be used in such a case; if the data conversion is sufficiently complex, the *notify* signal would be emitted prior to accomplishment of the data conversion. Figure 5.10 presents Active buffer, a modification of the previously presented *DataBuffer*, where the responsibility of notifying the consumer-side applications is assigned to *DataBuffer* instead of *ProducingApplication*. Functionality of both variations is almost identical, but Active buffer guarantees finalizing of the required data conversions prior to notification.

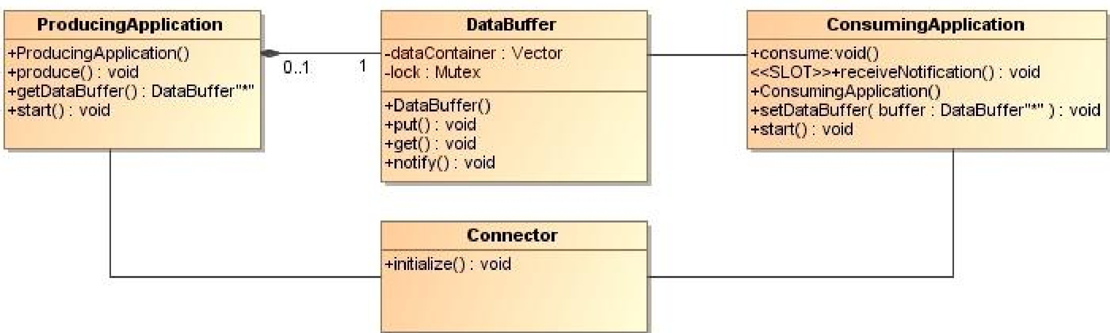

Figure 5.10 Data exchange with Active buffer

5.3. CoSMic-Frame

CoSMic-Frame is a simple framework designed to be used in application development for Microrobotic Platform (MP). The framework describes the general requirements and design principles including threading and data exchange related issues. Further, the framework aims to enable execution of any CoSMic-Frame based application on a server and as a stand-alone application.

5.3.1. Structure

All applications based on CoSMic-Frame inherit an abstract base class *CosmicApplicationBase*. Figure 5.11 presents the *CosmicApplicationBase*, which is in fact inherited from *QThread*. The reasoning behind the used inheritance is identical with the case presented in network communication. Structure of the *CosmicApplicationBase* is simple and aims to provide the developer with relatively free hands – it does not interfere with the internal structure of the application. Each application developed according the rules of the framework may include only one GUI component. However, the GUI component may be a composite of several GUI components. When running in a stand-alone mode, the possible GUI is automatically loaded. The GUI component can also be used when running the application on a server, by passing the application specific GUI components to *CosmicServerGui*, which acts as the main GUI.

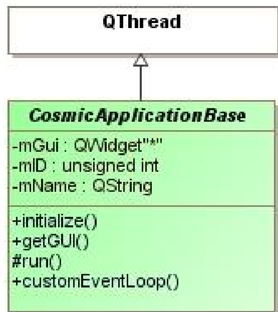

Figure 5.11 Abstract base class for applications complying with CoSMic-Frame

The implementation of the abstract `CosmicApplicationBase` class is presented in Figure 5.11.

5.3.2. Network Communication

Network communication in CoSMic-Frame is based on the pattern proposed in Section 5.2.1. Schematic overview applying the pattern to the communication scheme between the FiberVision and FiberStation is shown in Figure 5.12. The client-side, FiberStation includes always minimum of two separate threads, called Application logic thread and Main thread. Application logic thread is responsible for executing instances common for the entire client-side computer node. For example, the required client-side socket object resides in the Application logic thread. The amount of threads on the client-side is increased by one per each hosted CoSMic application. Thus deployment of MiCo and DAQCo would increase the thread count to four. The Main thread, also known as the GUI thread, is reserved exclusively for instances of graphical user interfaces.

The server-side contains minimum of three threads: Socket Server thread, Main thread and Application thread running a single application. When a client-side socket is connected, the number of Client Connection threads is increased by one per each new connection. The restrictions of the Main thread are similar on the server-side as described for the client-side.

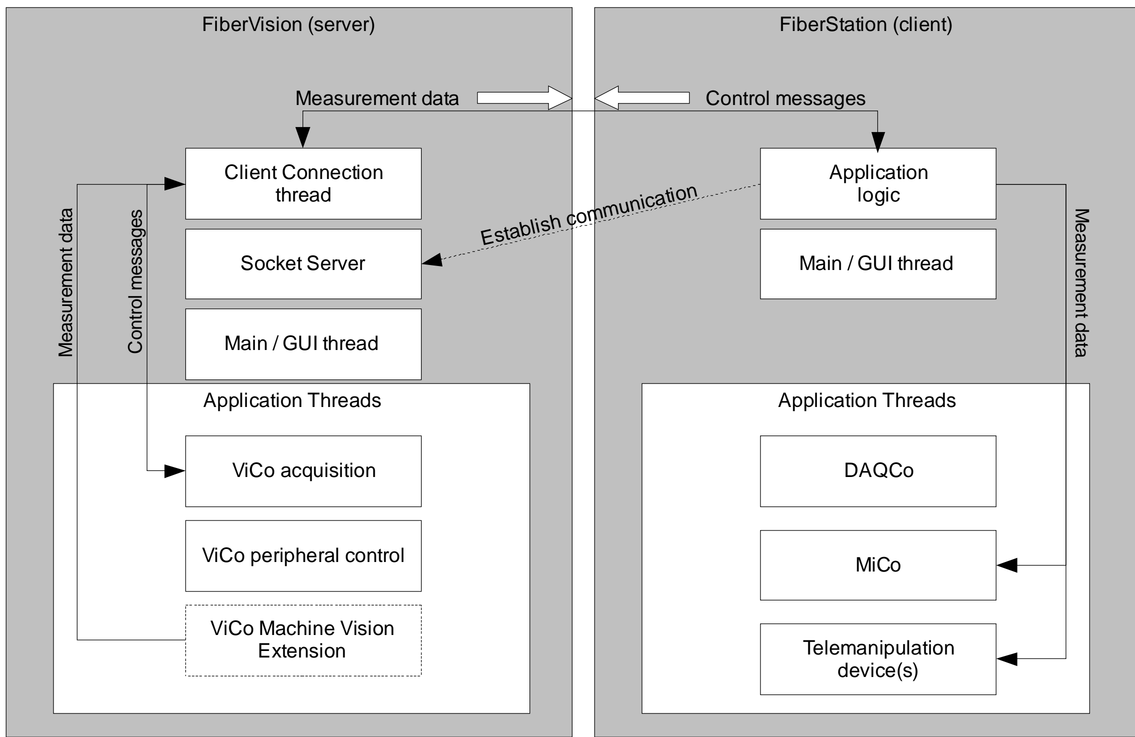

Figure 5.12 High-level architecture of FiberVision and FiberStation

CosMiC-Frame provides ready implementation of the required server-client pair. The implementation uses TCP/IP sockets, but the used communication protocol can be changed with relatively small effort.

5.4. Architecture of MiCo

Architecture of MiCo is multi-threaded employing threading and synchronization mechanisms provided by the Qt framework. The core functionality of the MiCo is distributed between several classes, some of which run in separate threads. The architecture of MiCo aims to maintain the threading provided by the SCU3DControl API for each individual SmarAct device. Serialization is avoided whenever possible and the API of MiCo provides communication with each device through different thread. The architecture of MiCo presented in Figure 5.13 is largely inspired by the Qt signal-slot mechanism which is used in event based communication between objects residing in separate threads. Figure 5.13 simplifies the architecture by ignoring the user interface classes of MiCo, which are discussed in Section 6.1. The following describes core functionality of the system, which is distributed between `DeviceManager`, `Device` and `DeviceListener` classes. Classes `DeviceCommander` and `CollisionManager` are further discussed in Section 6.1.2.

`DeviceManager` can be described as the business logic of the MiCo. It initializes the hardware, owns the objects used in communication between SCU3DControl API and provides some of the MiCo API functions. However the `DeviceManager` does not contribute to the run-time communication with SCU3DControl API, but reassigns the responsibility to `Device` objects.

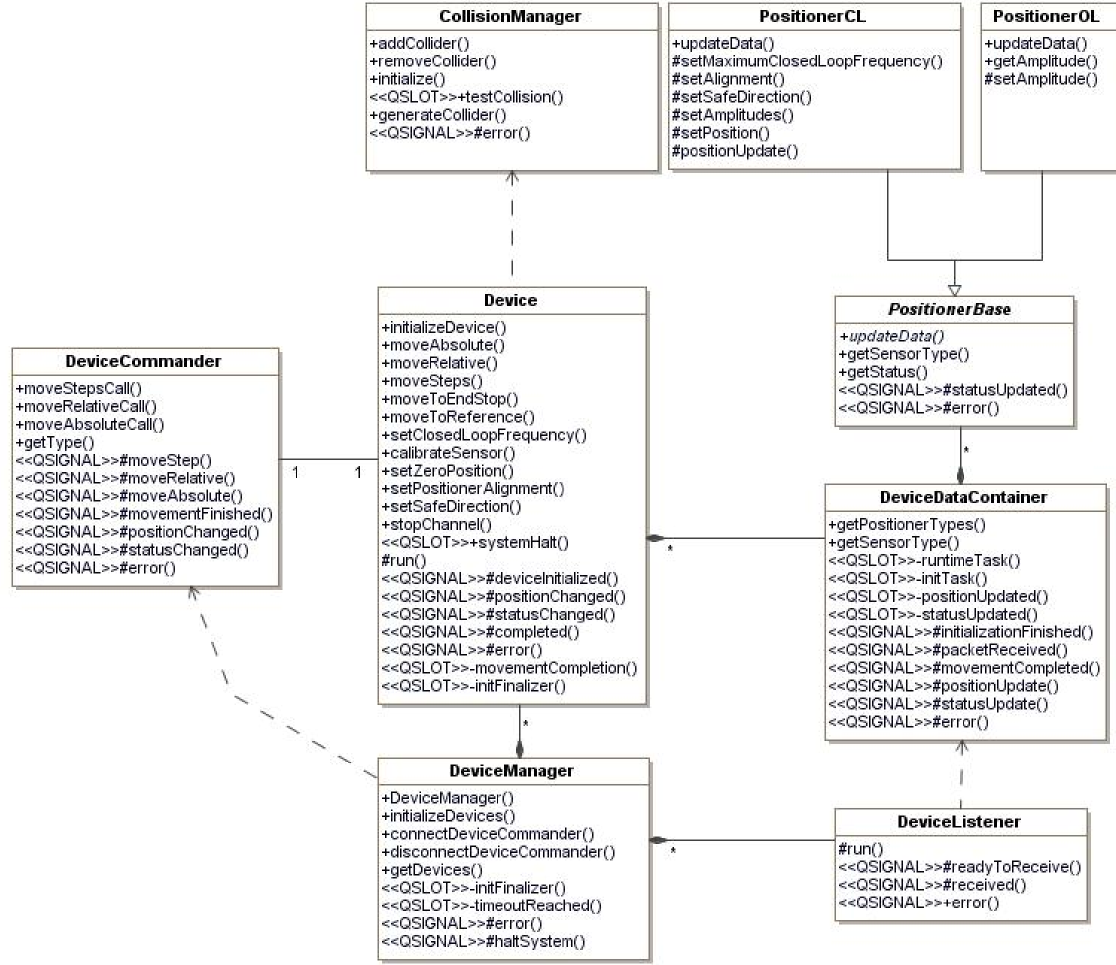

Figure 5.13 Architecture of MiCo

`Device` represents an individual SmarAct device and runs in its own thread. The counterpart required to receive incoming messages from the API is `DeviceListener` which also has a dedicated thread. Instances of these two classes are responsible for all run-time communication with the SmarAct hardware. Data exchange between `Device` and respective `DeviceListener` objects is provided through `DeviceDataContainer`. `DeviceDataContainer` contains information of each SmarAct actuator connected into the particular SmarAct device. Actuators with an in-built sensor are represented by `PositionerCL` class and actuators without sensor have own dedicated class called `PositionerOL`. Implemententation of new actuator types is supported through abstract base class `PositionerBase`.

Purpose of the data exchange class `DeviceDataContainer` is complete decoupling of `Device` and `DeviceListener`. Decoupling is desirable due to the OS requirements of SCU3DControl API. The answer retrieval procedure discussed in Section 4.2 included usage of Windows API event handles. In fact, the answer retrieval is the only procedure related to SCU3DControl API which requires usage of Windows specific functions. Decoupling ensures that platform specific functionality is

encapsulated to one single class. This may be beneficial if the API is later on released to other platforms.

5.5. Architecture of DAQCo

The main responsibilities of DAQCo include data acquisition, buffering, visualization and storing the acquired data into a file. Each of these tasks is assigned with a dedicated thread as illustrated in Figure 5.14. The core of DAQCo is data acquisition, which is performed in Daq thread. The acquired data is stored in a data buffer residing in Buffer thread. After buffering, the data can be written into a file in Recorder thread. Visualization of the acquired data is performed in GUI thread.

The high-level architecture of DAQCo is similar with MiCo. The most significant difference between these two subsystems is the amount of data that needs to be exchanged. Communication in MiCo is event based and requires only small amounts of data to be transferred between different instances. On the other hand, DAQCo may acquire several thousands of samples per second. Thus efficient and reliable data buffering is important. Structure of the DAQCo data buffer is extended from the buffer presented in Figure 5.10.

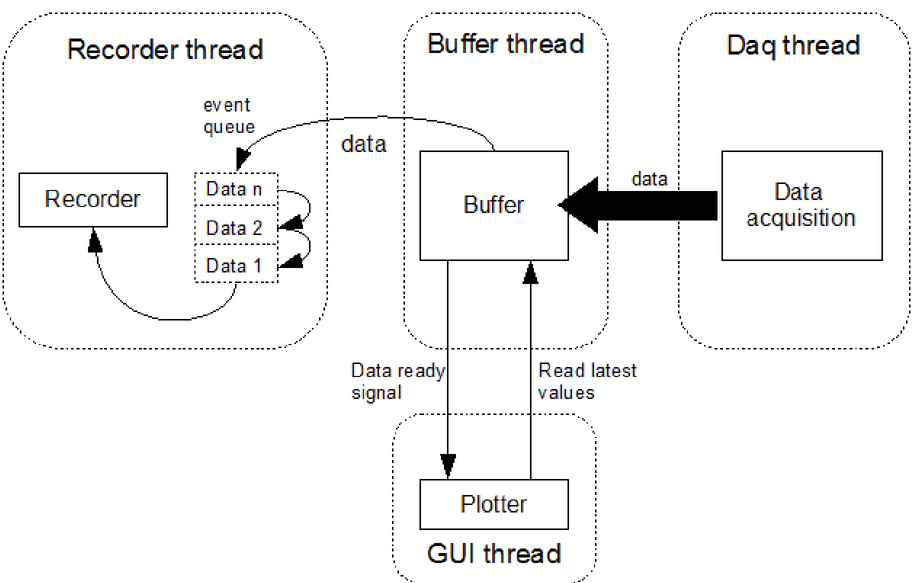

Figure 5.14 High-level architecture of DAQCo

The DAQCo data buffer includes two different methods of data exchange designed for different purposes. Small volumes of data can be exchanged using a Qt signal and a function retrieving the latest data from the buffer as presented in Figure 5.10. However, the mechanism is not suitable for large quantities of data. In DAQCo, a mechanism employing `QSharedPointer` and the event queue of `QThread` is used for exchanging large quantities of data between the data buffer and the instance responsible for storing the data into a file. The mechanism is further discussed in Section 6.2.

The DAQCo uses NI-DAQmx API presented in Section 4.3. The data exchange of the API and DAQCo employs callback functions, which the developer must provide

to the API in form of function pointers. Figure 5.15 presents the architecture of DAQCo on class level. The communication between the NI-DAQmx API and DAQCo is directed through `UDaqCore`, a class which is responsible for all data acquisition related activities. `UDaqCore` also owns the data buffer *`UDaqBuffer`* which is responsible for providing the acquired data to other parts of CoSMic. The functionality required in storing the acquired data into a file is provided through *`UDaqFileWriter`*.

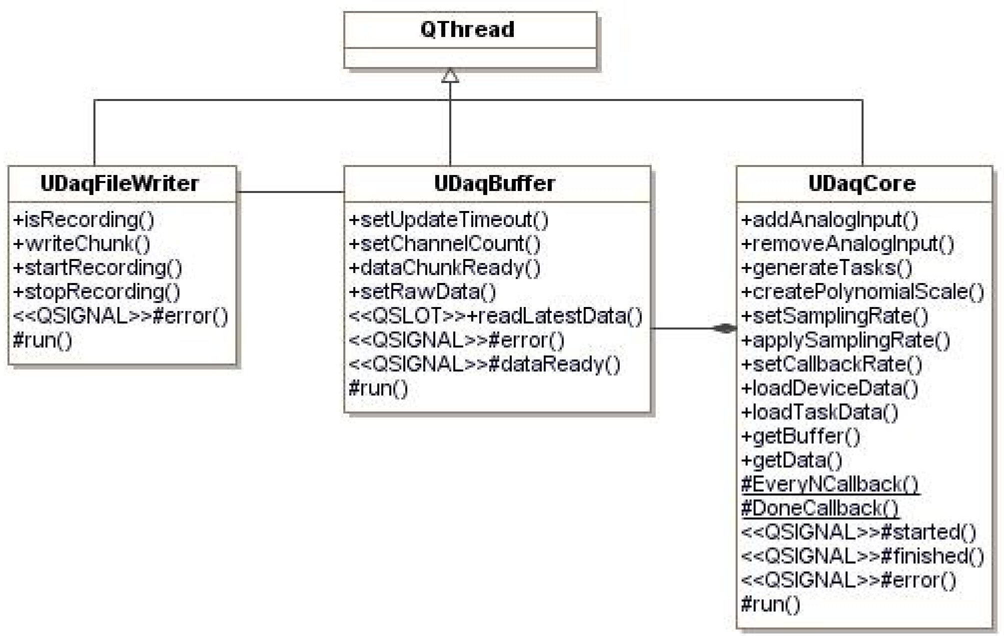

Figure 5.15 Architecture of DAQCo

The presented architecture of DAQCo disregards all GUI related classes. An overview of the GUI is provided in Section 6.2.

5.6. Summary

This chapter described the distributed architecture of CoSMic including three subsystems MiCo, ViCo and DAQCo. Section 5.1 revised a selection of design patterns, which were considered as suitable starting point for development of the architecture. Section 5.2 presented the overall architecture together with use cases illustrating the required functionality of each subsystem. Two of the subsystems, MiCo and DAQCo, are assigned to a single computer named as FiberStation. MiCo implements control of the micropositioners attached to MiS, whereas DAQCo is responsible for storing and visualizing the measurement data. Collaboration between the two subsystems is required to protect the hardware from damages. The third subsystem ViCo runs on dedicated computer called FiberVision. ViCo provides detailed visualization of the MP by interfacing cameras attached to the system. General guidelines and rules regarding development of CoSMic were laid out in form of a simple framework in Section 5.3. Finally two subsystems, MiCo and DAQCo, were discussed more in details in Section 5.4 and Section 5.5 .

6. DESIGN AND IMPLEMENTATION

This chapter presents an overview of the design and implementation for two subsystems of CoSMic. Section 6.1 concentrates on the design of MiCo, followed by description of DAQCo in Section 6.2. Section 6.3 discusses integration of the two subsystems. Finally Section 6.4 describes the state of current implementation.

6.1. MiCo

The Control of Micromanipulation System (MiCo) is designed to control the SmarAct micropositioners attached to MiS. The main purpose of MiCo is to implement an API and a user interface, which provides high-level functionality for the needs of CoSMic. Another important aspect in the design of MiCo is tracking the position of each actuator attached into MiS. Tracking of the positions is essential, since the actuators do not have a global coordinate system making development of some of features very tedious. For example, collision prevention is impossible without the knowledge of each actuator's current position.

6.1.1. Overview

A typical characterization process from the view of MiCo starts by grasping the object with microgrippers attached to two 3D assemblies. Before the object can be grasped, the 3D assemblies must be moved to correct positions. After grasping the object, the 3D assemblies move the object to the measurement area. Synchronous movement of the 3D assemblies is to maintain correct alignment of the object. In the next stage of the characterization process, the desired properties are measured. A single measurement procedure may include several repetitions. Thus the 3D assemblies are required to move the object several times from one place to another. In the final phase, the 3D assemblies move the object to a predefined location to wait for disposing.

The characterization procedure described above is converted into a high-level use cases in Figure 6.1. The most important single functionality required from MiCo is clearly communication with the SmarAct devices through SCU3DControl API.

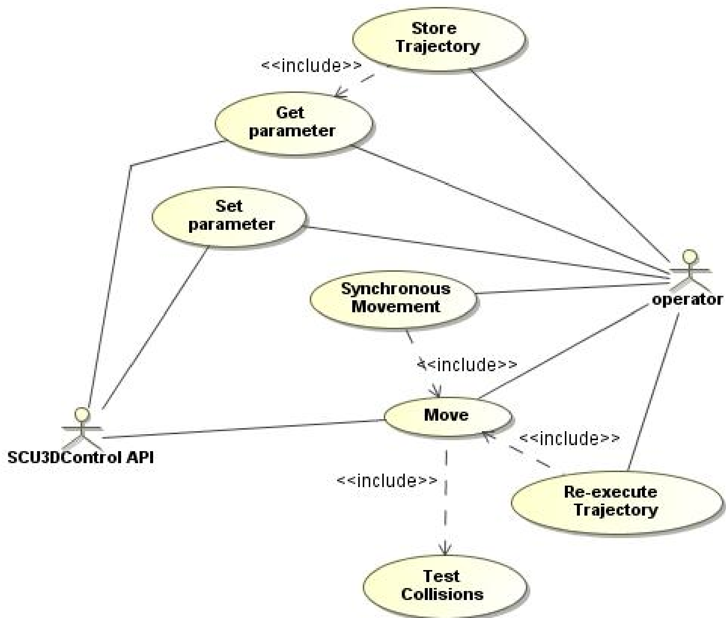

Figure 6.1 Core functionality of MiCo

All of the control modes presented in Section 3.4 require the functionality described by use case Control Actuator, which represents all run-time communication from MiCo to the SCU3DControl API. Implementation of SCM is represented by two additional use cases *Store Trajectory* and *Re-execute Trajectory*. The two remaining control modes, ACM and ECM require implementation of CD indicated by the use case *Test Collisions*.

Figure 6.2 recapitulates the core classes of MiCo providing asynchronous communication between MiCo and SCU3DControl API. Class *Device* represents a single SmarAct device controlling up to three individual actuators. All function calls to the API are sent from *Device*. Thus the class provides the functionality described in the use cases *Set parameter* and *Move*. The *DeviceListener* class is the counterpart of *Device*; it waits for incoming data from the API and forwards received data to *DeviceDataContainer*. Each *Device* has its own *DeviceDataContainer* which is responsible for storing all device specific data, including positions and statuses of each individual positioner. *DeviceDataContainer* can be thought as the class that provides the functionality required to accomplish the use case *Get parameter*. *DeviceDataContainer* has also an important role in the implementation of position-awareness required by the CD.

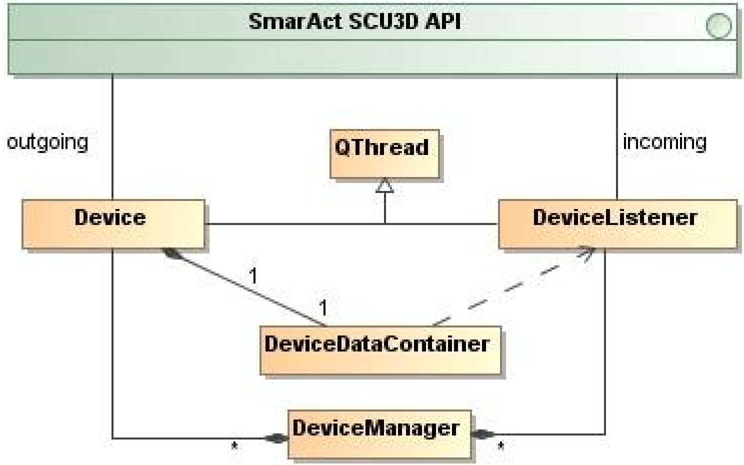

Figure 6.2 MiCo core classes

Use cases *Synchronous Movement*, *Re-execute Trajectory* require the functionality described in *Move*. *Synchronous Movement* is similar to use case *Move*, but moves several actuators in synchronous manner. Thus the functionality described in *Move* is required for several actuators simultaneously. *Synchronous Movement* is implemented in `DeviceManager`. Use case *Store Trajectory* records movements of the SmarAct actuators into a file. Whenever an actuator is moved, its new position is stored to *`DeviceDataContainer`* . *Store Trajectory* uses *Get Parameter*, which retrieves actuator's parameters from *`DeviceDataContainer`*. *Re-execute Trajectory* re-executes the trajectories stored by *Store Trajectory*. Re-execution is performed as sequence of *Move* use cases.

The number of active threads in MiCo depends on the number of devices attached to the SmarAct control module. Figure 6.3 illustrates threading of MiCo through an example where one SmarAct device is present.

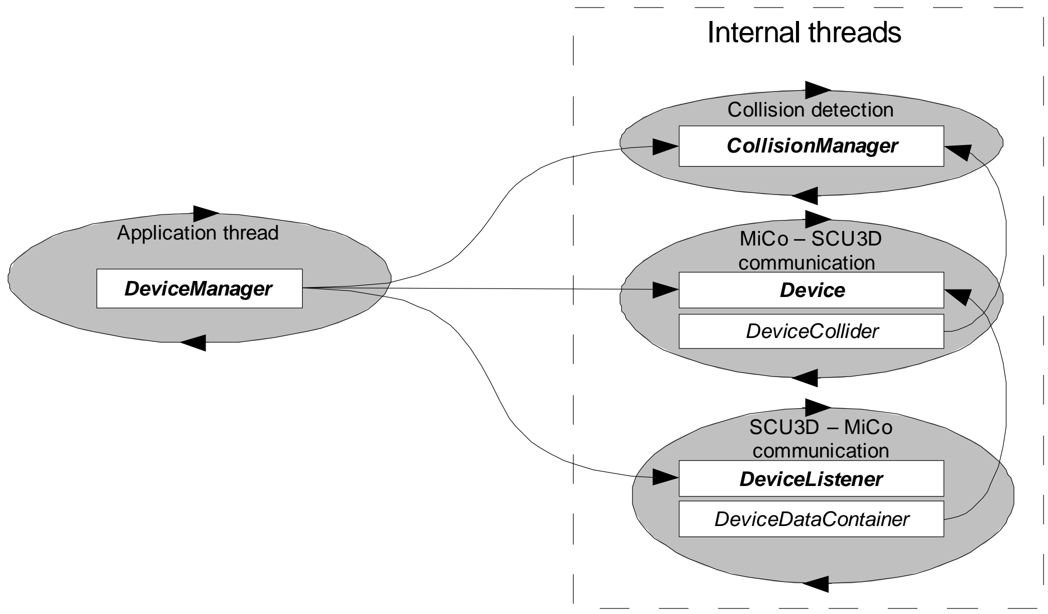

Figure 6.3 Threading and cross-thread communication in MiCo with one SmarAct device attached

The number of threads is always increased by two when an additional SmarAct device is attached into the SmarAct control module as each new SmartAct device requires a `Device` and a `DeviceListener`.

6.1.2. MiCo API

Application programming interface of MiCo consists of *DeviceManager*, *DeviceCommander* and *CollisionManager*, presented in Figure 5.13. Functions provided by the *DeviceManager* are mainly intended for initializing and shutting down MiCo. In addition a function providing access to *Device* objects is provided for configuration purposes. *DeviceCommander* provides the run-time access to *Device* and `DeviceDataContainer`. The functions provided by *DeviceCommander* have identical functionality with the functions of *Device*, but *DeviceCommander* only emits a Qt signal to respective *Device*, thus decoupling the API from the internal implementation of `Device`. Implementation of *moveAbsoluteCall*, which is used to invoke *moveAbsolute* function at *Device*, is shown as an example in Program 6.1.

```
void DeviceCommander::moveAbsoluteCall(unsigned int channelIndex, int
          position, unsigned int holdTime) {
    emit moveAbsolute(channelIndex, position, holdTime);
}
```

Program 6.1 Example function of DeviceCommander class emitting the received values as Qt signal

DeviceCommander objects do not include any information regarding the *Device* they interact with. The *DeviceManager* is used as a mediator to connect *DeviceCommander* and *Device* objects. The sequence required to establish communication between an application using MiCo API and *Device* is illustrated in Figure 6.4.

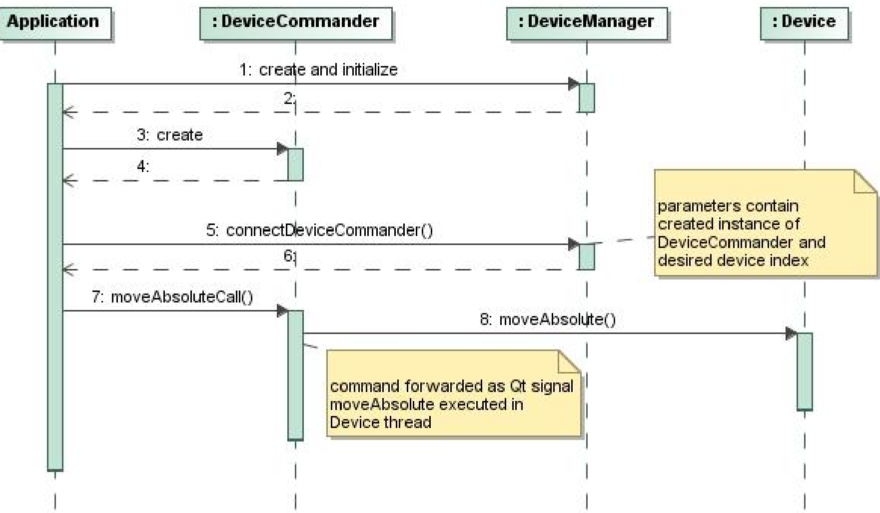

Figure 6.4 Sequence of connecting DeviceCommander *and given* Device

DeviceManager provides two types of communication between the *DeviceCommander* and *Device* objects. If *DeviceCommander* is used only to receive status and position messages from *Device*, it can be declared as an input device. Respectively *DeviceCommander* used only to move the actuators can be declared as an output device. The combination of the two communication types is also possible. Implementation of the *connectDeviceCommander* function is presented in Program 6.2 to enlighten the situation.

```
void DeviceManager::connectDeviceCommander(unsigned int deviceIndex,
          DeviceCommander *commander,
          QVector<UManipPositionerType> &positionerData) {
    Device* device = mDevices.at(deviceIndex).device;
    if (DeviceCommander::InputCommander ||
                    DeviceCommander::InputOutputCommander) {
        connect(commander, SIGNAL(moveStep(uint,int,uint,uint)), device,
                    SLOT(moveSteps(uint,int,uint,uint)));
        connect(commander, SIGNAL(moveRelative(uint,int, uint)), device,
                    SLOT(moveRelative(uint,int,uint)));
        connect(commander, SIGNAL(moveAbsolute(uint,int, uint)), device,
                    SLOT(moveAbsolute(uint,int,uint)));
    }

    if (DeviceCommander::OutputCommander ||
                    DeviceCommander::InputOutputCommander) {
        connect(device, SIGNAL(completed(uint,uint)), commander,
                    SIGNAL(movementFinished(uint, uint)));
        connect(device, SIGNAL(positionChanged(uint,uint,int)),
                    commander, SIGNAL(positionChanged(uint,uint,int)));
    }
}
```

Program 6.2 Function connectDeviceCommander *connecting provided* DeviceCommander *object to given* Device *object*

Third class involved in MiCo API, *CollisionManager* is responsible for collision detection (CD) and collision prevention. The implementation of CD of MiS hardware is

based on the CollDet and OpenSG scene graph libraries presented in Section 2.4. Figure 6.5 presents an overall structure of MiCo CD.

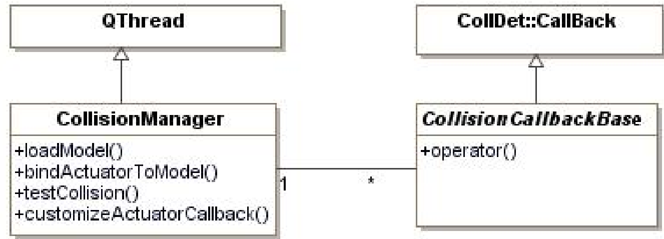

Figure 6.5 MiCo collision detection

`CollisionManager` contains all the graphical objects, which describe all geometrical features of CoSMic. In other words, `CollisionManager` creates a VR representing CoSMic. All objects, including each actuator are included in a scene graph constructed according to the principles described in Section2.4. The other participant of the CD is `CollisionCallbackBase`, which handles occurred collisions. `CollisionCallbackBase` is a virtual class, thus allowing customizing of the callback events. In addition, the developer may write different collision response implementation for each object. Purpose of MiCo CD is to prevent collisions by simulating movements of the actuators prior to moving them in the real-world. Sequence describing the CD procedure is presented in Figure 6.6.

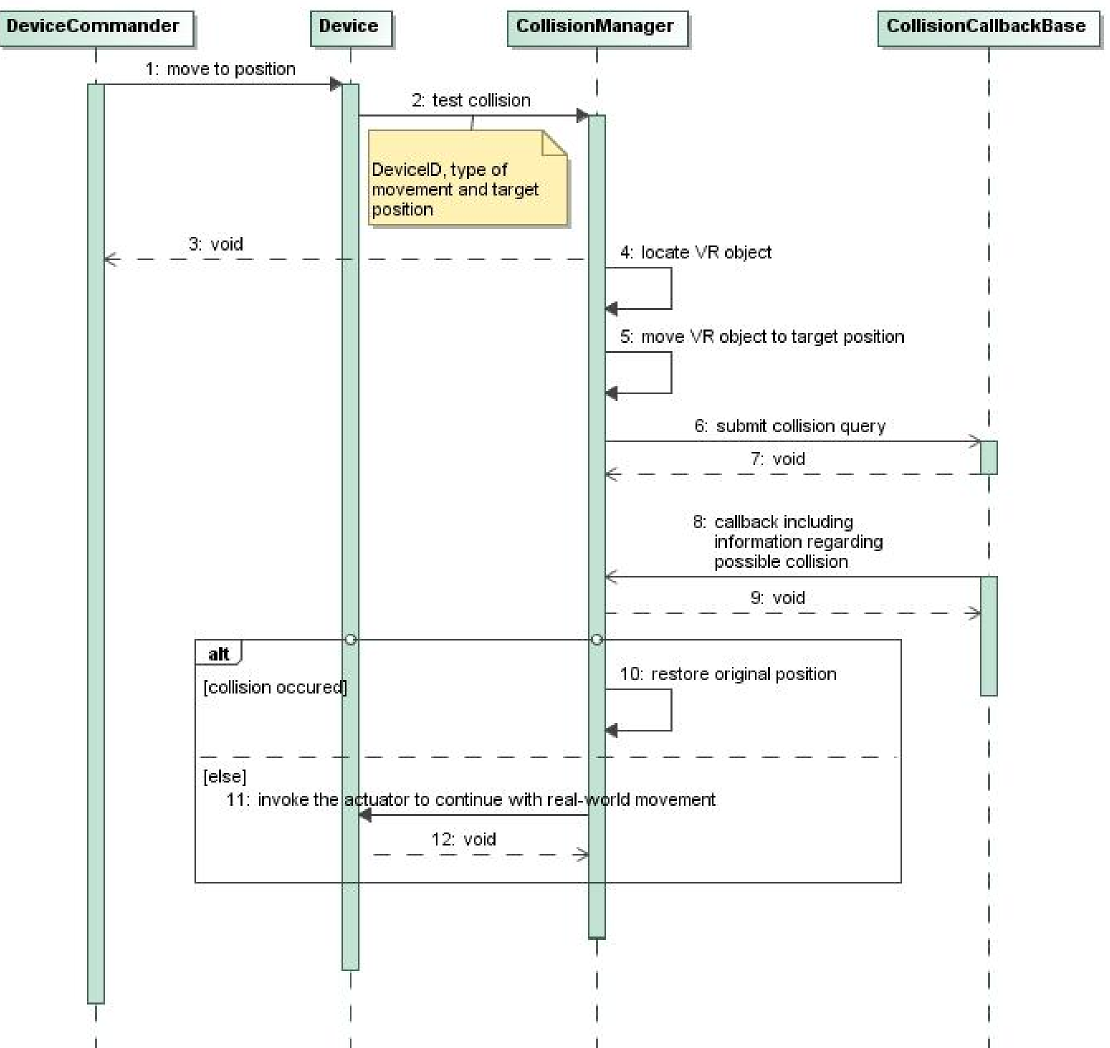

Figure 6.6 MiCo collision detection procedure

When `Device` receives instructions to move an actuator to a new location, the legality of the movement is evaluated by forwarding the instructions to *`CollisionManager`*. *`CollisionManager`* tests possible collisions by moving VR object respective to the actuator into the given location. After moving the actuator to correct position, *`CollisionManager`* calls the CD pipeline which calculates the possible collisions and calls the *`CollisionManager`* through a callback function. If a collision was detected, the *`CollisionManager`* moves actuator's VR object back to its original position. Otherwise the `Device` is instructed to continue to movement in the real-world.

6.1.3. Communication

Communication between MiCo and SCU3DControl API is routed through different classes during initialization and run-time. The subsequent sections describe the difference between the two communication phases.

Initialization

Initialization process of MiCo involves initialization of the hardware, creation of *Device* and *DeviceListener* objects for each found device and data exchange regarding each individual actuator attached to the MiS. Communication between MiCo and the SCU3DControl API differs from the run-time communication; the hardware initialization commands and queries regarding the amount of attached devices are performed by *DeviceManager*.

The initialization of MiCo described in Figure 6.7 starts by calling *initializeDevices* from *DeviceManager* which performs a query acquiring all available devices attached to the SmarAct control module. The command provided by the API explicitly states that the query must be made prior to actual initialization of the devices. In the next step the hardware is initialized using respective function of the API. After successful initialization, the *DeviceManager* creates *Device* and *DeviceListener* objects for each SmarAct device found in the first step of the initialization process. The created objects automatically move to new threads and the communication between MiCo and the API is reassigned from the *DeviceManager* to *Device* objects. However the *DeviceManager* remains in initialization mode and sleeps until each *Device* has been successful initialized. The created *Device* objects start their own initialization process by sending a query regarding type of attached positioner to each channel. The *DeviceListener* objects react upon incoming data packets and process the content. If the data packet carries information regarding the type of the positioner it is passed to the *DeviceDataContainer*. Otherwise, the packet is discarded.

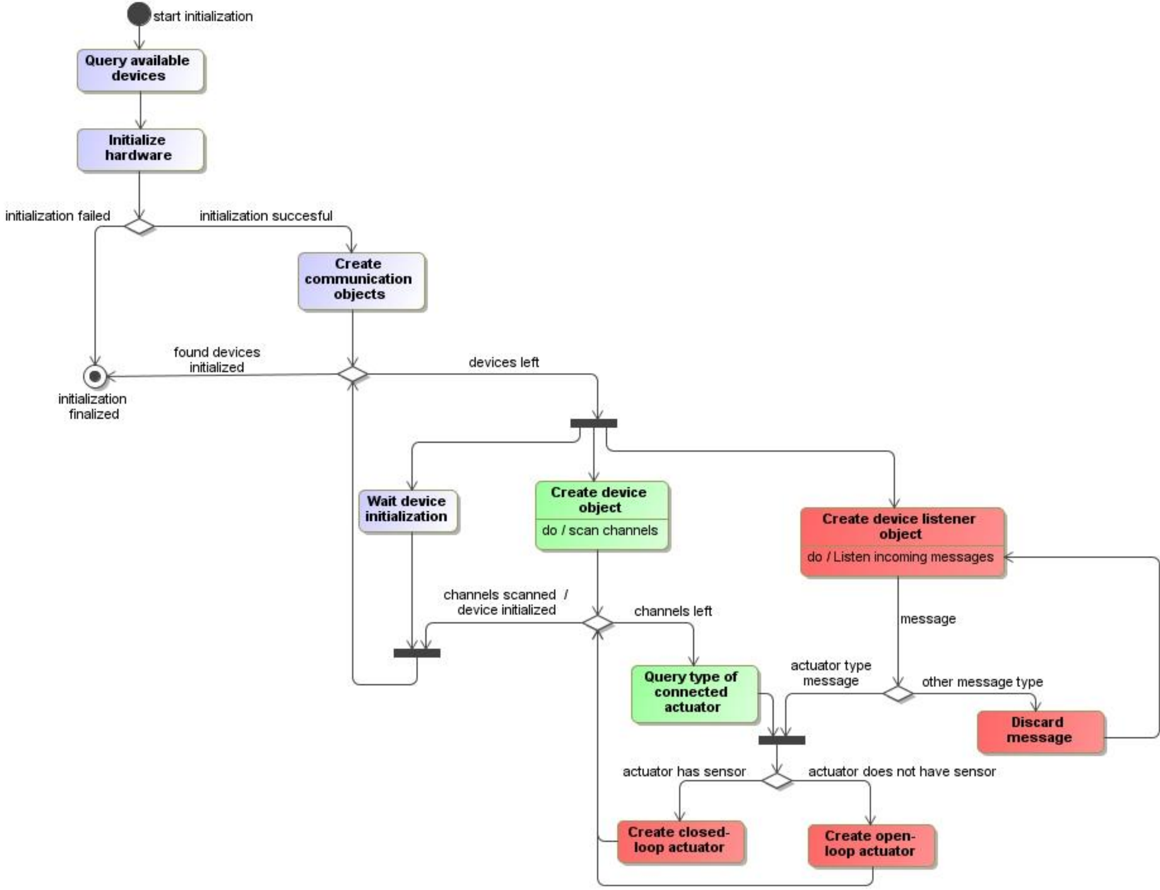

Figure 6.7 MiCo initialization procedure

Depending on the content of the packet received by the `DeviceDataContainer` object, an object representing positioner with or without sensor is created. After each positioner attached to each channel has been studied, the `DeviceDataContainer` objects inform the respective `Device` objects, which then report to the `DeviceManager`. After each `Device` has reportedly finished the initialization `DeviceManager` forces the system to run-time mode.

Moving Single Positioner

Movement of a single positioner involves several steps. Type of the commanded positioner must be checked and collision detection must analyze possible obstacles on the path of the positioner prior to sending the command to hardware. Figure 6.8 presents a sequence of function calls required to move a positioner from one position to another.

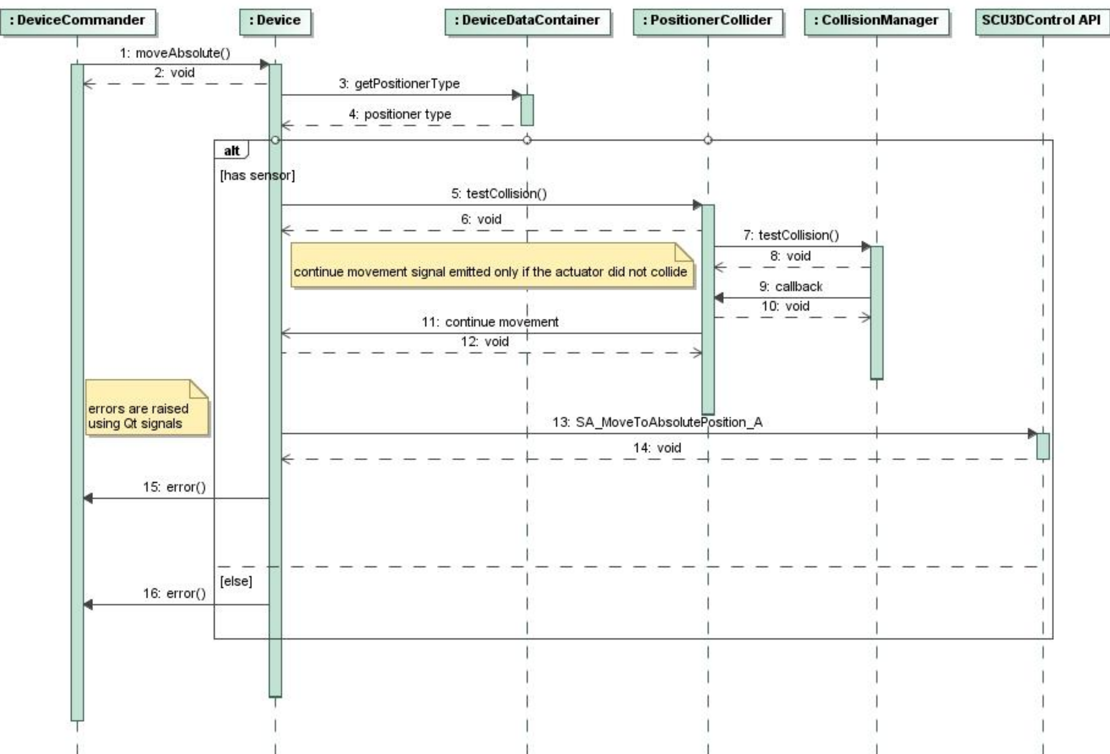

Figure 6.8 Call sequence for moving a positioner from MiCo API to SCU3DControl API

The presented sequence does not discuss the answer retrieval after the movement has been finished. Whenever a movement is successfully finished, MiCo queries the position of the positioner and updates the data to `DeviceDataContainer`, which passes the information to all objects connected to `positionUpdate` signal. Figure 6.9 presents the sequence performed after SCU3DControl API reports completed movement.

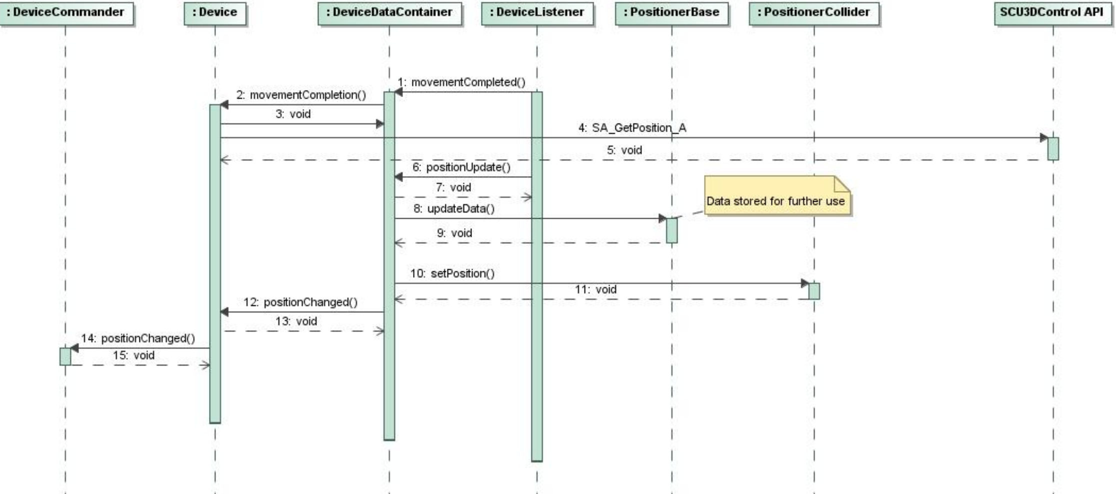

Figure 6.9 Activities after DeviceListener *object has received message indicating completion of movement*

The sequence presented above describes only those targets of the `positionChanged` signal which are within the implementation of MiCo. However the Qt signals can be

forwarded to other parts of the system, by simply connecting the signal at DeviceCommander to a slot or signal of another instance.

6.1.4. User Interfaces

MiCo includes an in-built GUI which provides the most common functionality required in the characterization process. Design of the GUI is component-based and scalable. In addition, the GUI can be easily integrated to any other Qt based GUI. The architecture of the GUI presented in Figure 6.10 includes three types of GUI components. On the lowest level resides PositionerBaseGUI, a base class for a single positioner. The class itself is able to describe a positioner which does not include a sensor. The positioners with a sensor are described by ClosedLoopGUI *class* which is specialized from PositionerBaseGUI. In order to enable easy creation of new types of positioner GUI classes, virtual functions are used in implementation of the PositionerBaseGUI. DeviceGUI describes the GUI on device-level, each *Device* is described by individual DeviceGUI which is connected to signals and slots of the respective *Device*.

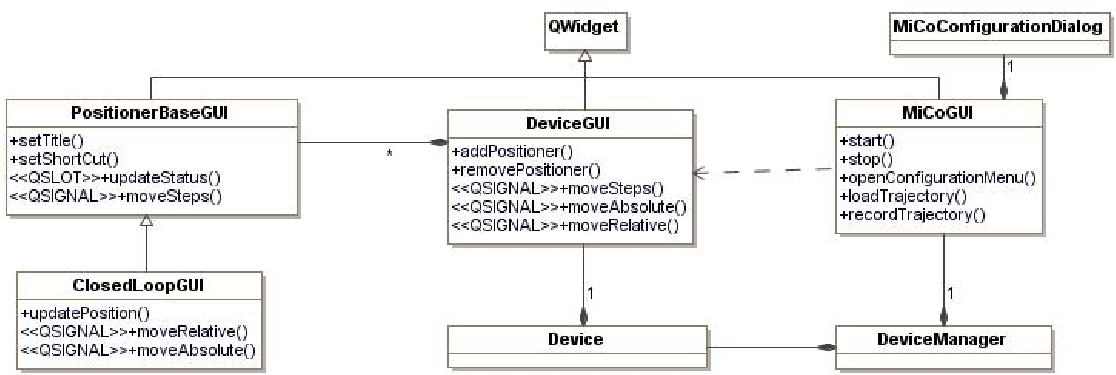

Figure 6.10 Architecture of MiCo GUI

The ownership of the GUI components is assigned to Device. However due to the requirements of Qt framework, the GUI components must be created in the main thread. This requirement is ensured by passing the main thread through all participating classes in their constructors as a QObject. For example, declaration DeviceManager constructor is DeviceManager::DeviceManager(QObject* parent =0).

MiCo allows easy implementation of custom user interfaces through DeviceCommander class. Similarly, the mechanism can be employed in integration of ViCo and MiCo.

6.2. DAQCo

The Control of Data Acquisition System (DAQCo) is designed to gather data from several sensors attached into DAQS. The used sensors and respective physical quantities may vary depending on the used configuration. Also several different sensor types may be used to measure same phenomenon. Therefore DAQCo must provide the operator

with the possibility of configuring each sensor individually. Some sensors require relatively complex conversions between the given output and the physical quantity of interest. The operator should be able to create custom polynomial scales which can be used to convert acquired electrical signals into the measured physical quantity. The main purpose of the DAQCo is to acquire, store and visualize data. The use cases indicating the required functionality are presented in Figure 6.11.

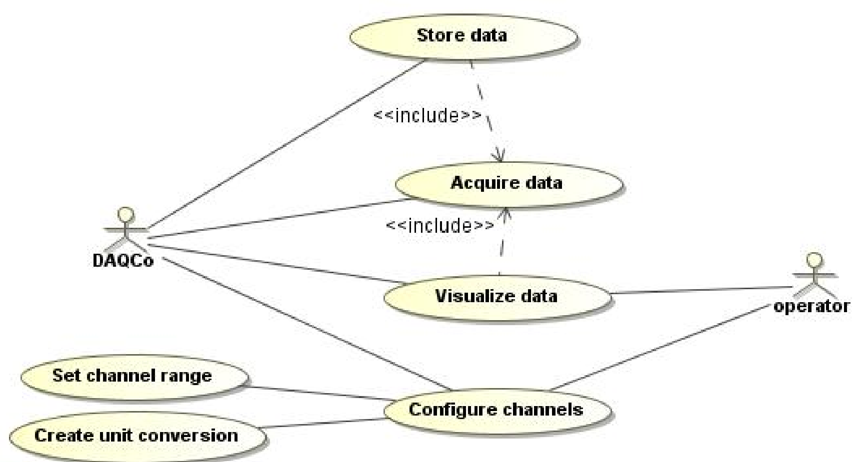

Figure 6.11 Core functionality of DAQCo

6.2.1. Callback Functions and Data Exchange

The DAQ functionality of DAQCo is based on two classes, namely `UDaqCore` and `UDaqBuffer`, presented in Figure 5.15. `UDaqCore` implements the functionality related to data acquitision, whereas `UDaqBuffer` is responsible for buffering the acquired data. The `UDaqCore` data acquisition is based on callback functions provided by NI-DAQmx API. The run-time execution of the data buffer `UDaqBuffer` and the core class `UDaqCore` is presented in Figure 6.12.

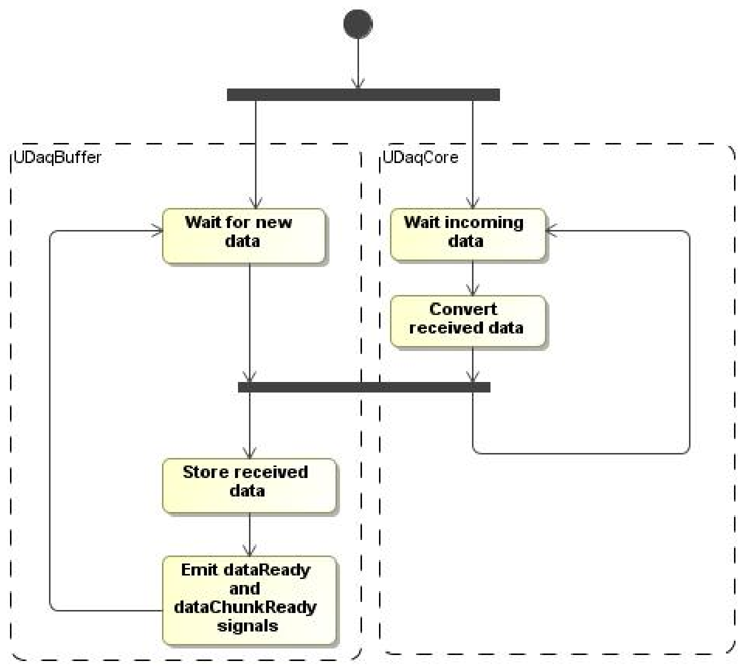

Figure 6.12 Run-time execution of `UDaqBuffer` *and* `UDaqCore`

After the initialization, *UDaqCore* and *UDaqBuffer* sleep waiting for external signaling. *UDaqCore* awakes upon callback function called from NI-DAQmx API. The callback function converts the received data and forwards it to the data buffer, which awakes when incoming data is detected. *UDaqBuffer* stores the latest values of the received data and informs other instances by emitting *dataReady* and *dataChunkReady* signals. The conversion performed at *UDaqCore* aims to provide the data in more convenient form to the *UDaqBuffer*; NI-DAQmx API callbacks provide the data in static C array, which is inconvenient to use with Qt signals. *UDaqCore* creates a *QVector* container and copies the received data chunk into the container object. The container is wrapped into a *QSharedPointer* which deletes the data after all references to it have been deleted. [26]

The *dataChunkReady* signal transfers a reference to the created *QSharedPointer*, thus allowing receivers of the signal to access the entire data chunk. When all instances have stopped using the data chunk, it is automatically deleted. Figure 6.13 illustrates the usage of QSharedPointer together with the *dataChunkReady* signal.

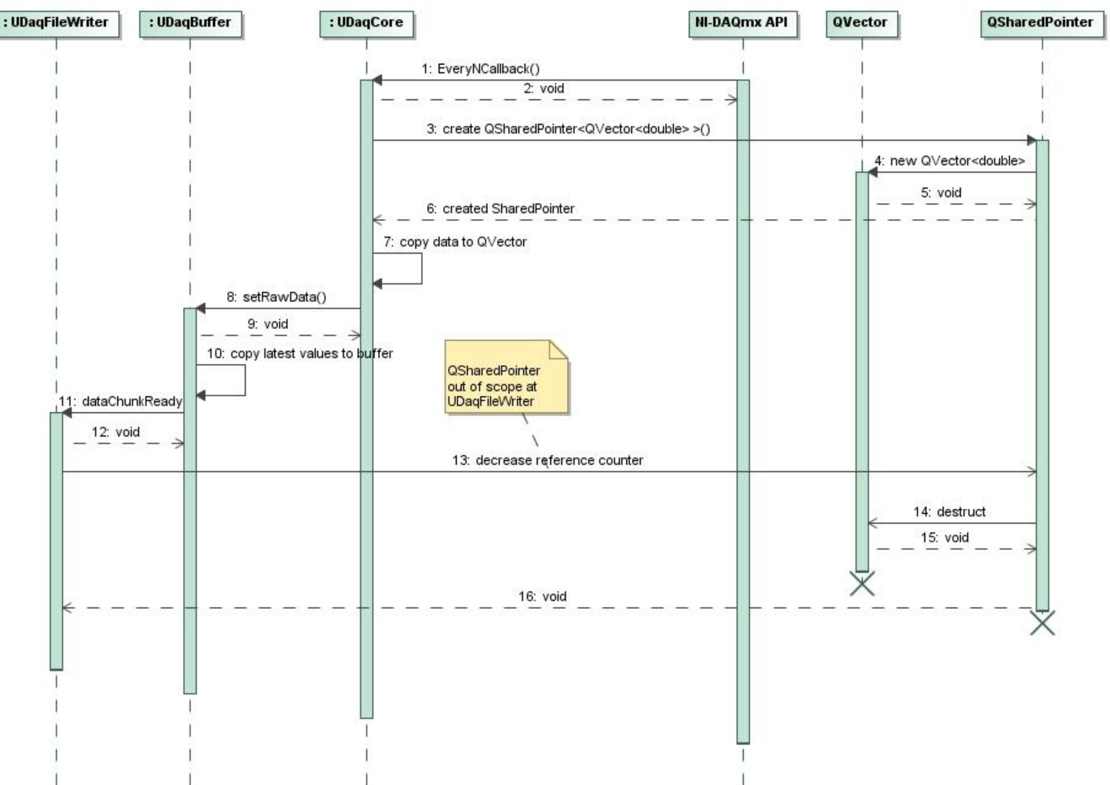

Figure 6.13 Usage of QSharedPointer in communication between `UDaqCore`, `UDaqBuffer` *and* `UDaqFileWriter`

If the thread receiving the *dataChunkReady* signal is busy, the signal stores the *QSharedPointer* in the event loop of the thread. Event loop acts as FIFO buffer, thus guaranteeing that the receiving thread gets the data chunks in same order as they were sent [26]. The presented design for forwarding large data chunks from the data buffer was implemented due to mere curiosity of the developer. The design could as well be replaced with traditional ring buffer and set of synchronization objects to guarantee proper access rights for all participating instances.

The other method accessing the acquired data through the data buffer is `dataReady` signal. The signal does not pass any parameters to receiver and is purely informative. The instances connected to the `dataReady` signal must call the `getLatestData` function the access the stored data. The reading and writing functions of the data buffer are protected with mutual exclusion to prevent simultaneous reading and writing.

6.2.2. Graphical User Interface

GUI of DAQCo consists of four separate GUI components. Two of the components, *UDaqMainWidget* and *UDaqFileWidget* are standard Qt GUI components, thus they inherit Qt's user interface base class *QWidget*. *UDaqMainWidget* is the base of the DAQCo GUI, it contains only menus and does not include any functionality visible to operator. *UDaqFileWidget* implements a GUI for storing the acquired data into a file. The third component *UDaqConfigureDialog* is a configuration dialog and

inherits `QDialog`. `UDaqConfigureDialog` allows the user to configure each channel of the DAQ unit, to assign different conversions between units and to alter active channel configuration. The fourth component `UDaqPlotter` is a data plotter which visualizes the data received from the data buffer. `UDaqPlotter` is based on *Qwt* – a library specifically designed for scientific plotters [48]. The architecture of the GUI is presented in Figure 6.14.

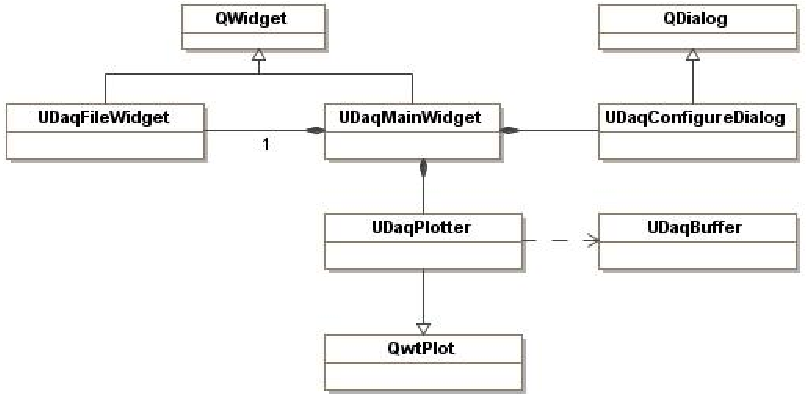

Figure 6.14 Architecture of DAQCo GUI

The GUI is a composite of several objects where `UDaqMainWidget` has the ownership of all participating instances. The structure enables easy integration of the GUI with other Qt based GUI.

6.3. Integration of MiCo and DAQCo With An Input Device

This section presents an example application integrating MiCo and DAQCo with a haptic device. The libraries for the haptic device have been developed in-house and the design is in line with CoSMic-Frame [29]. The high-level architecture of the application demonstrating the integration of DAQCo and MiCo is presented in Figure 6.15.

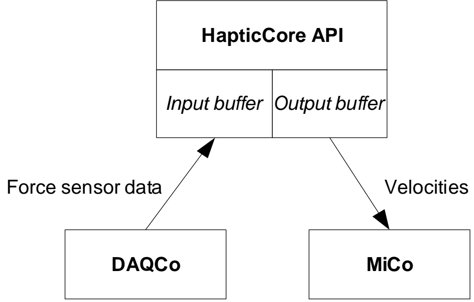

Figure 6.15 High-level architecture of the developed application

The application developed for this demonstration includes three classes. `TestApplication` is the core of the application and responsible for connecting all the required instances to each other. Two additional classes `TestAppDAQCoConverter` and `TestAppMiCoConverter` are used to enable communication between the `HapticCore`, `UDaqBuffer` and `DeviceCommander`. `TestAppMiCoConverter` is responsible for assigning the output values of the

HapticOutputBuffer to each positioner of the 3D assembly. Respectively, *TestAppDAQCoConverter* assigns the sensor readout values to different axis of the haptic device. The architecture is presented more in details in Figure 6.16.

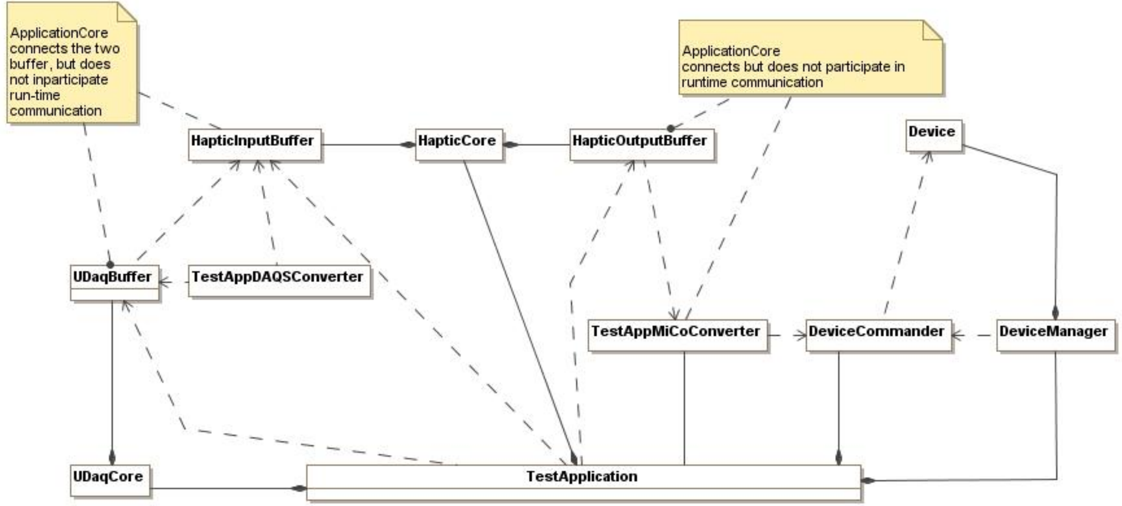

Figure 6.16 Architecture of the developed application

As presented in Figure 6.16 the *TestApplication* is not required in the run-time communication. The two additional instances *TestAppDAQCoConverter* and *TestAppMiCoConverter* are proposed to run in same thread with the sender or receiver of the data to prevent creation of unnecessary threads.

6.4. Current Implementation

The current implementation of CoSMic consists of stand-alone applications of MiCo and DAQCo. Both of the applications have successfully been used in characterization of paper fibres [16]. Figure 6.17 presents the GUI of MiCo, which customized to control two 3D assemblies, 2D assembly, rotary actuator and two end effectors. The GUI is constructed from several GUI components as proposed in Section 6.1.4.

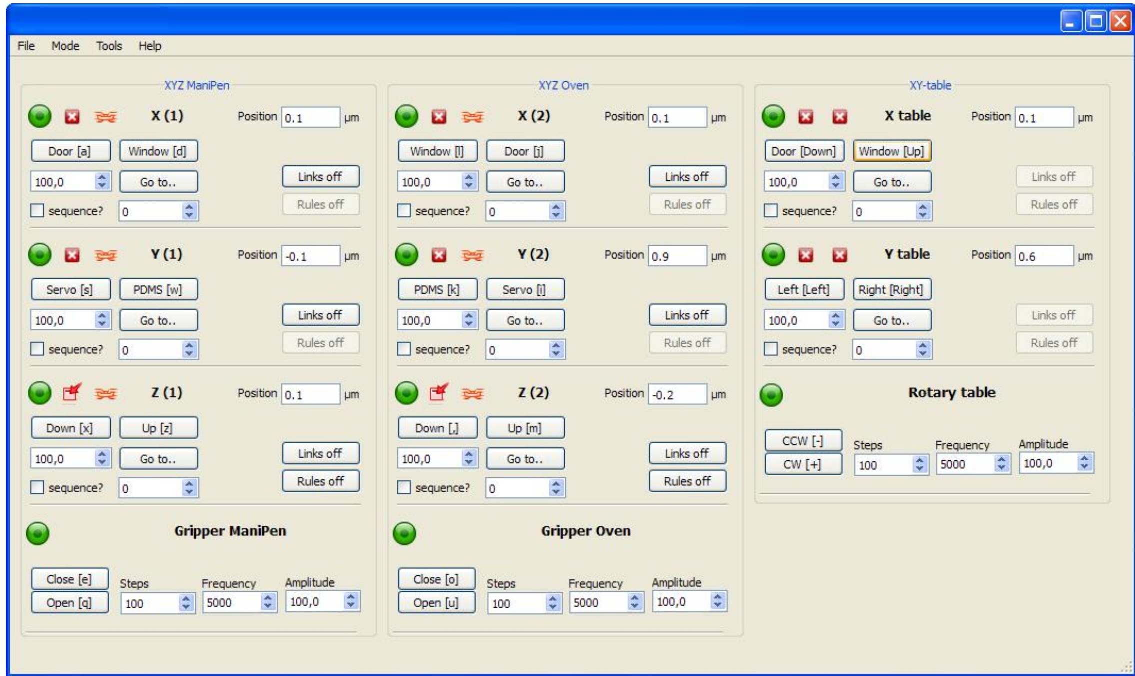

Figure 6.17 User interface of MiCo

An example of a MiCo GUI component, presenting control of single actuator with an in-built sensor is presented in Figure 6.18. The GUI component provides functionality required to move the actuator. In addition, the operator may exclude the actuator from collision detection and decide whether the actuator moves synchronously with another actuator. The functionality of the MiCo GUI component is described more in details in Appendix E.

Figure 6.18 MiCo GUI component for closed-loop controlled SmarAct micropositioners

In its current state, MiCo is capable of producing the functionality required in MCM and SCM. MiCo provides automatic detection of different actuator types. The functionality provided to the operator is selected according to the detected actuator type. In addition the GUI can be configured through a simple configuration file, which defines structure of the GUI.

In addition to in-built GUI, MiCo provides an API allowing the developer's to integrate the functionality of MiCo into other application. The developed API is thread-safe and complies with the CoSMic-Frame described in 5.3. MiCo API has been demonstrated by integrating MiCo API with an in-house developed API for haptic device.

The current implementation of DAQCo provides the operator with a GUI capable of handling multiple DAQ units and multiple sensors simultaneously. The GUI implements the functionality, which is required to visualize and store data received from different sensors through DAQ unit. An overview of DAQCo GUI is presented in Figure 6.19, more detailed description of the GUI is available in Appendix F.

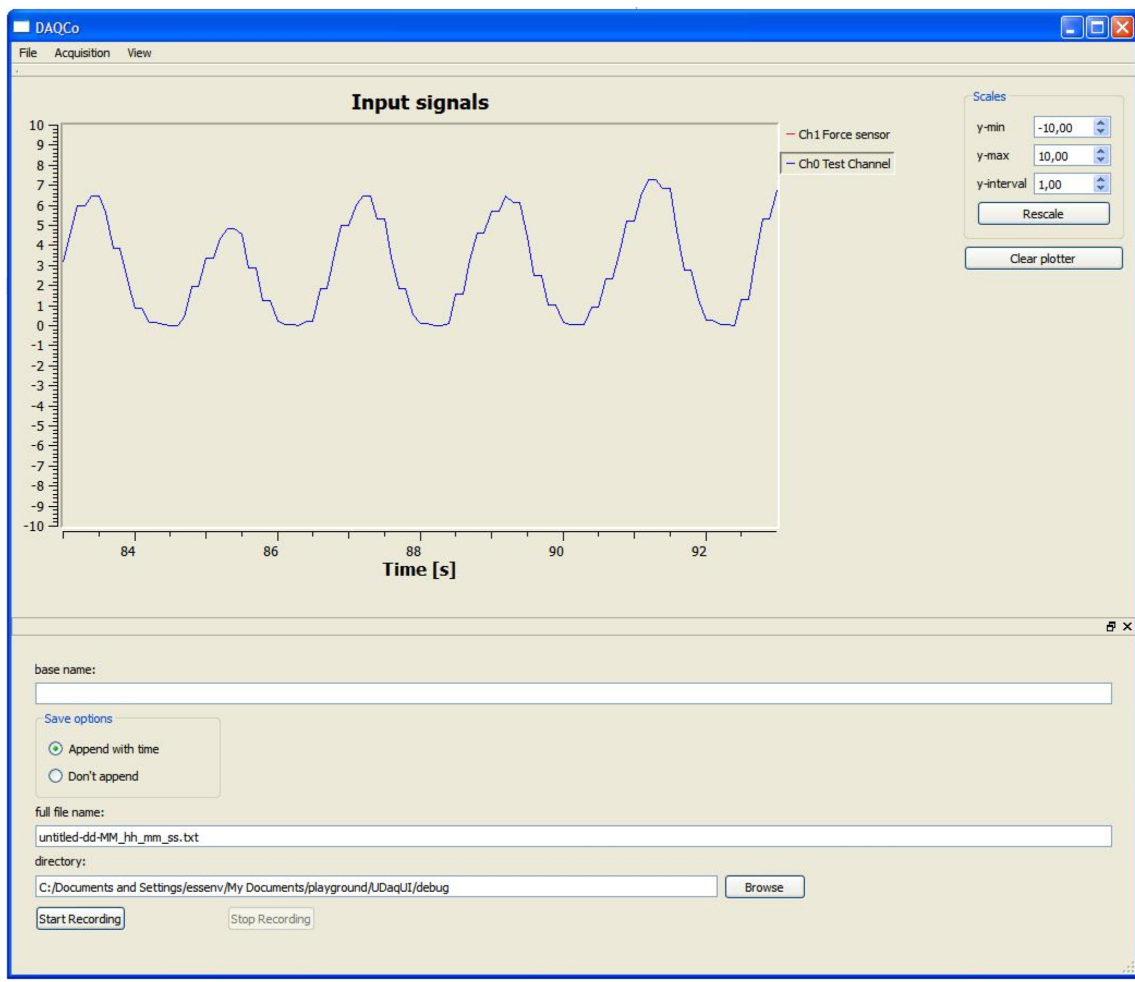

Figure 6.19 DAQCo user interface

Like MiCo, DAQCo also provides an API including all the functions that are provided through the GUI.

7. CONCLUSIONS AND FUTURE WORK

This chapter includes the conclusions of this thesis and future work for development of Control Software for Microrobotic Platform.

7.1. Conclusions

This thesis presented CoSMic, a control software designed for the needs of Microrobotic Platform (MP) used in the characterization of paper fibres. Current implementation of CoSMic, including two separate applications, provides GUI for two subsystems of the MP. The first application, Control of Micromanipulation System (MiCo), allows the operator to manipulate the characterized paper fibres. The latter application known as Control of Data Acquisition System (DAQCo), is responsible for acquiring data from different sensors attached to MP. In addition, DAQCo is capable of storing and visualizing the acquired data.

The presented work concentrates on several issues of application development for MP. The most important aspects cover cross-platform support, scalability and reusability of the produced program code. Cross-platform support is reached in several parts of the control software through careful selection of the used third party program libraries. Furthermore, Qt – a cross-platform application development framework is used to enhance portability of the developed program code. Scalability is supported through the distributed architecture of CoSMic. A design pattern that decouples the network technology from the actual program code is implemented to guarantee reusability of the developed core classes. The design and architecture of CoSMic provides a solid basis for application development on MP using well-established design principles.

However, several proposals for improvements were found during the course of this thesis. Based on the feedback of the operators, need of an additional feature for MiCo was identified. A configurable hardware initialization sequence for MiCo is proposed. Purpose of the sequence is to move all MiS related hardware to predefined starting position, thus enhancing the repeatability of the characterization procedure. In addition, the initialization sequence should include the possibility to calibrate of each actuator in given order.

SCU3DControl API used in MiCo to control MiS hardware was found to lack the possibility of moving several actuators in a synchronized manner. Therefore, the current version of CoSMic omits synchronization and executes the movement related API function calls in series. This approach does not guarantee synchronous behavior. If the API does not return sufficiently fast from previous function call, the behavior of the

actuators is unpredictable. In order to reach genuinely synchronized behavior, changes in the implementation of SCU3DControl API are required. Another issue related to SCU3DControl API is its extendibility. Extensive usage of C++ `#define` directive within the API may result in tedious changes of the developed program code, when a new version of the API is released. The problem could be avoided through implementation of an additional layer of abstraction between the SCU3DControl API and the program code.

Selection of Qt framework as the base of CoSMic has proven to be effective solution from the developer's point of view. The Qt signal-slot mechanism and the cross-platform multithread support have greatly reduced the time required in the implementation of CoSMic. In addition, cross-thread communication is less prone to errors as Qt signal-slot mechanism provides a thread-safe method for cross-thread communication. Furthermore, Qt's object-oriented approach for GUI development enhances reusability of developed GUI components. Especially in the implementation of MiCo, the component based GUI has been an effective and easily configurable solution. Operators using the MiCo are required to make only minor modifications to MiCo configuration file when the hardware configuration is altered.

Section 5.3 presented CoSMic-Frame simple framework designed to be used in application development for Microrobotic Platform. One of the most significant deficiencies of the framework is lack of guidelines for exception handling. In Qt-based applications exception handling is a topic of particularly high importance, as Qt does not fully support C++ standard exceptions. Hence creation of guidelines for exception handling within CoSMic-Frame should be concerned as a high priority task.

7.2. Future Work

Continuation of this thesis aims to fulfill the requirements of a fully automated paper fibre characterization process. However, in order to reach the functionality required in ACM, each subsystems of CoSMic requires additional features.

In MiCo, the development of collision prevention for MiS hardware has been started and the fundamental concept together with the required CD library has been selected. Implementation of the collision prevention will continue the multithreaded approach of CoSMic. The development of collision prevention should be directed towards path-planning, which is an essential feature for fully automated fibre characterization. A thorough investigation of suitable path-planning libraries should be conducted in order to reach optimal solution. However, implementation of path-planning does not have high priority; the collision prevention alone has a large impact on the usability of CoSMic. After successful implementation of collision prevention, integration of DAQCo and MiCo should take place. Implementation of collision prevention prior to integration is proposed to avoid unnecessary modification of the GUI; collision prevention is likely to result in large changes of the MiCo GUI, as the visualization of the VR may be desired. An additional MiCo related task is the

implementation of the aforementioned hardware initialization sequence. This task is has high priority as the impact to usability is obvious.

Visualization and image based measurements are important issues for usability as well as for automatic fibre characterization. The ongoing implementation of ViCo will provide a solid base for development of the required machine vision (MV) algorithms. The developed MV algorithms should be able to recognize single fibre, to measure its length and to provide MiCo with position data and to enable automatic grasping of the characterized fibre. Integration of the FiberVision and FiberStation network nodes is required in order to enable communication between ViCo and MiCo. The communication will be based on the CoSMic-Frame communication presented in 5.3.2. Communication between the ViCo and MiCo is thought to have high priority, as it may reveal the possible deficiencies of the proposed network communication. Development of MV algorithms may run parallel with all other activities, as it is independent from CoSMic. Thus algorithm development may be assigned to another party in order to reduce workload of the development team.

The future vision of MP includes extension of the existing hardware with different kinds of actuators. Some of these actuators may require real-time control in order to reach reliable behaviour. This aspect promotes implementation of the real-time capable extension presented in 5.2.

This section presented the most significant features required to automate the paper fibre characterization with MP. The development team is in the belief that the goal of fully automated paper characterization can be reached within a year.

8. REFERENCES

[1] A. Puder, K. Römer, F. Pilhofer, *Distributed systems architecture: a middleware approach*, ISBN-10: 1558606483, Morgan Kaufmann, 2005

[2] F. Buschmann, K. Henney, D. C. Schmidt, *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*, ISBN-10: 0470059028, Wiley, 2007

[3] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ISBN-10: 0201633612, Addison-Wesley, 1994

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, ISBN-10: 0471958697, Wiley, 1996

[5] M. Rintala, J. Jokinen, *Olioiden ohjelmointi C++:ll*a, ISBN: 952-14-0936-3, Talentum Media, 2005

[6] V-P. Eloranta, J. Koskinen, M. Leppänen, V. Reijonen, *A Patternlanguage for Distributed Machine Control Systems*, ISBN: 978-952-15-2318-2, Tampereen Yliopistopaino, 2010

[7] B. P. Douglass, *Real-time UML Advances in The UML For Real-Time Systems*, ISBN: 0321160762, Addison-Wesley, 2006

[8] N. Audsley, A. Burns, *Real-time System Scheduling*, Technical Report No. YCS 134, Department of Computer Science, The University of York, UK, 1990

[9] *Linux kernel documentation* [www][cited 1.4.2010]. Available at http://www.kernel.org/doc/

[10] *Xenomai: Real-Time Framework for Linux* [www][cited 1.4.2010]. Available at http://xenomai.org

[11] *RTAI - the RealTime Application Interface for Linux from DIAPM* [www][cited 1.4.2010]. Available at http://rtai.org

[12] D. J. Tracy, S. R. Buss, B. M. Woods, *Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal.* [cited 1.4.2010]. Available at http://sweepandprune.com/Daniel_Joseph_Tracy/Sweep_and_Prune_files/SAP_paper.pdf

[13] OPCODE documentation [www][cited 1.4.2010]. Available at http://www.codercorner.com/Opcode.htm

[14] S. A. Ehmann, M. C. Ling, *Accurate and Fast Proximity Queries Between Polyhedra Using Convex Surface Decomposition*, Eurographics 2001, Volume 20, Issue 3, 2001

[15] *Speedy Walking via Improved Feature Testing for Non-Convex Objects*. [www][cited 1.4.2010]. Available at http://gamma.cs.unc.edu/SWIFT++

[16] P. Saketi, "Microrobotic Platform for Manipulation and Flexibility Measurement of Individual Paper Fibers", Master of Science Thesis, 2010

[17] *SmarAct* [www][cited 5.4.2010]. Available at http://www.smaract.de

[18] *Sony XCD-U100 data sheet* [www][cited 5.4.2010]. Available at http://pro.sony.com/bbsc/ssr/product-XCDU100/

[19] *12x Zoom Vision System* [www][5.4.2010]. Available at http://www.machinevision.navitar.com/catalog/?c=372

[20] *SCU3D Simple Control Unit Reference Guide*, SmarAct, 2009

[21] *Korundum / QtRuby* [www][cited 10.4.2010]. Available at http://rubyforge.org/projects/korundum/

[22] *About PyQt* [www][cited 10.4.2010]. Available at http://wiki.python.org/moin/PyQt

[23] *Qt Jambi Reference Documentation* [www][cited 10.4.2010]. Available at http://qt.nokia.com/doc/qtjambi-4.4/html/com/trolltech/qt/qtjambi-index.html

[24] J. Thelin, *Foundations of Qt Developement*, ISBN-10: 1590598318, APress, 2007

[25] *Qt – a Cross-platform application and UI framework* [www][cited 1.4.2010]. Available at http://qt.nokia.com

[26] *Qt 4.6 Reference Documentation* [www][cited 1.4.2010]. Available at http://qt.nokia.com/doc/4.6

[27] *NI-DAQmx Software* [www][cited 10.4.2010]. Available at http://www.ni.com/dataacquisition/nidaqmx.htm

[28] *Pattern: Broker* [www][cited 1.4.2010]. Available at: http://www.vico.org/pages/PatronsDisseny/Pattern%20Broker/

[29] P. Kruchten, *The Rational Unified Process – An Introduction*, Addison-Wesley-Longman, 1999

[30] (ISO/IEC 42010:2007). Systems and software engineering – Recommended practice for architectural description of software-intensive systems.

[31] L. Podivin, *Unknown title*, Master of Science Thesis, 2010

[32] M. Ling, J. Canny, *A Fast Algorithm for Incremental Distance Calculation", IEEE International Conference on Robotics and Automation*, 1991

[33] *Robotic Motion Planning* [www][cited 15.4.2010]. Available at http://www.cs.cmu.edu/~motionplanning/

[34] *Software Library for Interference Detection* [www][cited 1.4.2010]. Available at http://www.win.tue.nl/~gino/solid/

[35] *RAPID - Robust and Accurate Polygon Interference Detection* [www][cited 1.4.2010]. Available at http://gamma.cs.unc.edu/OBB/

[36] *I-COLLIDE* [www][cited 1.4.2010]. Available at http://gamma.cs.unc.edu/I-COLLIDE/

[37] M. Ramaekers, *Introduction to Advanced Computer Architecture: Parallel Collision Detection* [electronical book][cited 1.4.2010]. Available at http://parallel.vub.ac.be/documentation/pvm/Example/Marc_Ramaekers/parallel.html

[38] P. Terdiman, *Sweep-and-prune* [electronical document][cited 10.4.2010]. Available at http://www.codercorner.com/SAP.pdf

[39] D. Baraff, *Dynamic Simulation of Non-Penetrating Rigid Bodies, Ph. D dissertation*, 1992

[40] G. Zachmann, *Optimizing the Collision Detection Pipeline*, Proceedings of the First International Game Technology Conference, 2001

[41] *OpenSG* [www][cited 10.4.2010]. Available at http://www.opensg.org

[42] *Autodesk 3ds Max Products* [www][cited 15.4.2010]. Avaible at http://usa.autodesk.com/adsk/servlet/pc/index?siteID=123112&id=13567426

[43] Blender [www][cited 1.4.2010]. Available at http://www.blender.org

[44] S. LaValle, *Planning Algorithms* [electronical book][cited 15.4.2010]. Available at http://planning.cs.uiuc.edu/

[45] G. Zachmann, *Virtual Reality in Assembly Simulation – Collision Detection, Simulation Algorithms, and Interaction Techniques*, Ph.D dissertation, 2000

[46] *CollDet – A Library for Collision Detection* [www][cited 14.4.2010]. Available at http://cg.in.tu-clausthal.de/research/colldet/index.shtml

[47] *CollDet Reference Manual 1.0* [www][cited 22.5.2010]. Available at http://cg.in.tu-clausthal.de/research/colldet/data/CollDetDoc.pdf

[48] *Qwt - Qt Widgets for Technical Applications* [www][cited 23.5.2010]. Available at http://qwt.sourceforge.net/

APPENDIX A QT SIGNAL/SLOT MECHANISM

Program A.1 presents usage of Qt signal/slot mechanism in form of an example. Class `MySender` inheriting `QObject` represents an object capable of emitting signal *mySignal*. The signal is emitted upon calling member function `shoot`, which is declared as a slot. It is important to understand that slots are member functions; behavior of a slot differs from ordinary member function only when connected to a signal using `QObject::connect`. `MyReceiver` class represents a class capable of connecting to a signal by the slot `mySlot`.

Fundamentally, the most important part of the example presented in , resides in the *main* function. After creating an object from both of the presented classes, `mySignal` and `mySlot` are connected. The `QObject:connect` is provided with pointers to both of the objects and with names of the respective signal and slot. The mechanism indicates that the *participating objects do not require information about each other*. After connecting the two objects together with the signal-slot mechanism, the program does a normal function call to `MySender::shoot`, which commands the `MySender` object to emit the signal `mySignal`. Based on the meta-object data, the `MyReceiver::myReceiver` slot is called in response to the emitted signal.

```
class MySender: public QObject {
        // required macro
        Q_OBJECT
        public:
                MySender();
        public slots:
                void shoot();
        signals:
                // no implementation!
                void mySignal();
};


class MyReceiver: public QObject {
        // required macro
        Q_OBJECT
        public:
                MyReceiver();
        public slots:
                // must have implementation!
                void mySlot();
};


void MySender::shoot() {
        emit mySignal();
}

int main() {
        MySender sender;
        MyReceiver receiver;
        connect(&sender, SIGNAL(mySignal()),
                &receiver, SLOT(myReceiver)));
        sender.shoot();
        return EXIT_SUCCESS;
}
```

Program A.1 Usage of Qt signal/slot mechanism

Compatibility of QThread and Xenomai was tested with a small program presented in Program A.2.

```
class XenomaiTestThread: public QThread {
        RT_TASK rt_this_task;
protected:
        void run();
};

void XenomaiTestThread::run() {
        mlockall(MCL_CURRENT | MCL_FUTURE);
        rt_task_shadow(&rt_this_task, "Task 1", 10, 0);
        rt_print_init(4096, "Task 1");
}

int main() {
        XenomaiTestThread testTread;
        testThread.start();
        testThread.wait();
        testThread.quit();
        return 0;
}
```

Program A.2 Test program running `QThread` *under Xenomai*

APPENDIX B SCU3DCONTROL API

Program B.1 presents a simple example program describing acquisition of the positioner type with the `SA_GetSensorType_A` function. Prior to sensor type query, the initialization function `SA_InitDevices` must be called. In addition, the asynchronous communication mode requires creation of an event object which is activated in case of data reception.

The event object is registered to the specific device using `SA_SetReceiveNotification`, a function which is able to inform an application when a data packet has been received from the hardware. After the described initial steps are taken, the `SA_GetSensorType_A` can be called. The hardware performs the actual query, stores the acquired data into the data buffer of the device and uses the event handle to inform the application program. The application retrieves the packet from the queue using the `SA_ReceiveNextPacket_A` function.

```
#include "SCU3DControl.h"
#include <windows.h>

int main() {

        unsigned int channelIndex = 0;
        unsigned int deviceIndex = 0;
        SA_PACKET packet;
        HANDLE handle;
        handle = CreateEvent(NULL, false, false, NULL);

        SA_InitDevices(SA_ASYNCHRONOUS_COMMUNICATION);
        SA_MovePositionAbsolute_A(0, 0, 100, 0);
        WaitForSingleObject(handle, INFINITE);
        SA_ReceiveNextPacket_A(deviceIndex, channelIndex, &packet);
        return EXIT_SUCCESS;
}
```

Program B.1 Asynchronous retrieval of positioner sensor type

APPENDIX C HIGH-LEVEL USE CASES

The following describes the high-level use cases of MiCo and DAQCo. Table C.1 presents MiCo related use cases *Move Actuator*, *Record Trajectory* and *Run Stored Trajectory*. The functionality related to these use cases is presented in Section 6.1. The GUI component related to herein presented use cases of MiCo is described in Appendix E.

Table C.1 Description of MiCo high-level use cases

Use case name	**MiCo1: Move Closed-Loop Actuator**
Performer	Operator
Preconditions	MiS hardware is turned on MiCo is initialized
Description	Operator enters desired coordinates into input field MiCoCL.8.. Operator presses push button MiCoCL.6, MiCoCL.7 or MiCoCL.8 depending on which type of movement is desired. MiCo moves the actuator to selected position
Exceptions	1. MiCo cannot communicate with SmarAct interface module. Operator is informed with an error message. MiCo is closed and reserved memory is released. 2. An error occurs at the actuator during the movement. Operator informed with an error message, which describing the error.
Result	Actuator moved to given position
Use case name	**MiCo2: Record Trajectory**
Preconditions	See use case MiCo1
Performer	Operator
Description	Operator activated trajectory recording mode from MiCo GUI and moves the actuators as described in use case MiCo2. MiCo records each movement into a file and wait for next movement. Operator finalizes the recording process by deactivating trajectory recording mode from the GUI.
Exceptions	See use case MiCo1 Error in generating or opening file for the created trajectory. Operator informed with error message. Trajectory recording mode deactivated.
Result	A new trajectory recorded into a file
Name	**MiCo3: Run Stored Trajectory**
Performer	Operator
Preconditions	See use case MiCo1 Trajectory with legal format is available
Description	Operator loads predefined trajectory into MiCo through MiCo GUI. MiCo tests legality of the trajectory and re-executes it.
Exceptions	See use case MiCo1 Trajectory format is illegal or cannot be opened. Operator informed with error message.
Result	Actuators are moved to target positions
Use case name	**MiCo4: Teleoperation**
Performer	Operator
Preconditions	See use case MiCo1 Teleoperator / telemanipulator is initialized

Description	Operator assigns a target actuator for the telemanipulator. In addition, parameters describing the relation between the movements of the telemanipulator and the actuator are given. Operator moves the telemanipulator (e.g. joystick) and the assigned actuator moves respectively.
Exceptions	See use case MiCo1 Telemanipulator fails to communicate with MiCo. Operator informed with error message. Actuators assigned to the telemanipulator are stopped.
Result	The actuator is moved to position, which is relative to position of the telemanipulator.
Use case name	**MiCo5: Test Collisions**
Performer	MiCo / Administrator
Preconditions	Collision detection data loaded into MiCo by administrator Operator has performed the steps described in use case MiCo1
Description	MiCo forwards the parameters received through the GUI to collision detection system, which performs the given movements in virtual reality.
Exceptions	None
Result	If no collisions where found, MiCo moves the actuator to given position as described in use case MiCo1. Otherwise operator receives an error message due to illegal movement.

Table C.2 presents the high-level use cases related of DAQCo, which have been used as a basis for the design of DAQCo. The design of DAQCo is presented in Section 6.2 and the GUI components related to DAQCo use cases are described in Appendix F.

Table C.2 Description of DAQCo high-level use cases

Use case name	**DAQCo1: Visualize Data and DAQCo3: Acquire Data**
Performer	DAQCo
Preconditions	DAQS hardware initialized with required sensors DAQCo initialized
Description	DAQCo starts acquiring data from DAQ unit immediately after initialization. A/D converted signals are stored in a data buffer. The signals stored in the data buffer are automatically visualized in DAQPlot.1.
Exceptions	DAQCo fails to retrieve data from DAQ unit. Operator is informed with error message. DAQCo is closed and reserved memory is released.
Result	A/D converted signals are continuously stored into data buffer
Use case name	**DAQCo2: Store data**
Preconditions	See use case DAQCo1
Performer	Operator
Description	Operator activates data storing by entering desired file name (DAQFile.1.) and directory (DAQFile.4.). The file name can be appended with a timestamp by selecting “Append with time” option from DAQFile.2. Operator starts data storing by pressing DAQFile.5. Data storing is stopped by pressing DAQFile.6.
Exceptions	DAQCo fails to write to given file. Operator is informed with error message. See use case DAQCo1
Result	Incoming signals are stored in the given file

APPENDIX D ABSTRACT BASE CLASS FOR COSMIC APPLICATIONS

Implementation of abstract base class for CoSMic application is presented in Program D.1. The declaration shows that only few functions are implemented on the base class level. Thus only minimal restrictions for the developer are made. The developer may change the implementation of the event loop by writing replacement for the virtual function *customEventLoop*.

```
class CosmicApplicationBase: public QThread {
Q_OBJECT
public:
        CosmicApplicationBase(QObject *guiParent=0);
        virtual void initialize()=0;
        virtual QWidget *getGui() { return mGui; }

protected:
        virtual void run() {
                if (!customEvenLoop() ) {
                        exec();
                }
        }

private:
        virtual bool customEvenLoop() { return false; }
        QWidget *mGui;
        CosmicApplicationInvoker *mInvoker;
};
```

Program D.1 Implementation of abstract CosmicApplicationBase *class*

APPENDIX E MICO USER INTERFACE

This appendix describes functionality of MiCo GUI components known as `PositionerBaseGUI` and `ClosedLoopGUI`. The first describes a base class capable of providing the functionality required to control an actuator, which does not contain an integrated position sensor. `PositionerBaseGUI` inheriting Qt user interface base class `QWidget`, is presented in Figure E.1

Figure E.1 MiCo GUI component `PositionerBaseGUI`

`PositionerBaseGUI` contains several input and output fields, which are implemented as Qt GUI components. Description of each the fields are given in Table E.1.

Table E.1 Input and output fields of `PositionerBaseGUI`

Identifier	Category	Description
MiCoOL.1.	**Output** QTextLabel	Status indicator for MiCo actuators. Three possible states: OK………………………………..... BUSY…………………………..……. ERROR…………………………….
MiCoOL.2.	**Output** QTextLabel	Name label for actuator. Can be changed through configuration file
MiCoOL.3.	**Input** QPushButton	Moves actuator forwards relatively to current position. Moved distance in determined by fields MiCoOL.5., MiCoOL.6. and MiCoOL.7.
MiCoOL.4.	**Input** QPushButton	Moves actuator backwards relatively to current position. Moved distance in determined by fields MiCoOL.5., MiCoOL.6. and MiCoOL.7.
MiCoOL.5.	**Input** QSpinBox	Number of steps performed while moving the actuator. Valid range 0-30000 steps *Note:* See [20]
MiCoOL.6.	**Input** QSpinBox	Actuator's control frequency that the steps described in MiCoOL.5. are performed with. Valid range 1-18500 Hertz
MiCoOL.7.	**Input** QDoubleSpin Box	Actuator's amplitude that the steps described in MiCoOL.5. are performed with. Valid range 15-100 Volts

Fields MiCoOL.1 and MiCoOL.2 are output fields, which provide information regarding the actuator's state and name. The remaining field, MiCoOL.3 – MiCoOL.7 are input fields. Each input field can be invoked by the operator, response of the GUI component is described in Table E.1. More detailed information regarding each Qt GUI component type is available in [26].

`PositionerBaseGUI` can be specialized to extend functionality of the provided GUI component. `ClosedLoopGUI`, presented in Figure E.2, is used as an example of specializing `PositionerBaseGUI`. `ClosedLoopGUI` extends the functionality of the base class with additional features, which are required to control SmarAct actuators with in-built position sensor.

Figure E.2MiCo GUI component `ClosedLoopGUI`

The input and output fields of `ClosedLoopGUI` are presented in Table E.2. Fields MiCoCL.1 – MiCoCL.5 are output fields that work as indicators. These fields are responsible for providing the operator with information such as actuator's status, current position and name. Fields MiCoCL.6 – MiCoCL.13 are input fields. Purpose and description of each field is described in Table E.2.

Table E.2 Input and output fields of `ClosedLoopGUI`

Identifier	Category	Description
MiCoCL.1.	**Output** QTextLabel	Status indicator for MiCo actuators. Three possible states: OK……………………………….…... BUSY…………………………..…….. ERROR……………………………..
MiCoCL.2.	**Output** QTextLabel	Collision detection indicator. Two possible states: Collision detection enabled……….. Collision detection disabled………
MiCoCL.3.	**Output** QTextLabel	Synchronous movement indicator. Two possible states: Synchronous movement enabled...... Synchronous movement disabled….
MiCoCL.4.	**Output** QTextLabel	Name label for actuator. Can be changed through configuration file
MiCoCL.5.	**Output** QLineEdit	Displays actuator's current position in micrometers
MiCoCL.6.	**Input** QPushButton	Moves actuator forwards relatively to current position. Moved distance in determined by field MiCoCL.8. The button's text can be replaced through configuration file. In addition, a shortcut key indicated within brackets can be changed through the configuration file.
MiCoCL.7.	**Input** QPushButton	Moves actuator backwards relatively to current position
MiCoCL.8.	**Input** QDoubleSpin Box	Describes the moved distance in micrometers. The type of movement is determined by pressing MiCoCL.7., MiCoCL.8., or MiCoCL.9.
MiCoCL.9.	**Input** QPushButton	Moves actuator to absolute position described in field MiCoCL.8.
MiCoCL.10.	**Input** QCheckBox	Determines whether movements are performed a one single movement or a series of smaller steps. The size of maximum step is given through field MiCoCL.11. *Note:* This feature is currently disabled
MiCoCL.11.	**Input** QSpinBox	Describes maximum step size for sequential movement, which can be enabled with field MiCoCL.10 *Note:* This feature is currently disabled
MiCoCL.12.	**Input** QPushButton	Enables and disables synchronous movement. The state of this option is indicated through field MiCoCL.3.
MiCoCL.13.	**Input** QPushButton	Enables and disables usage of collision detection for the particular actuator. The state of this option is indicated through field MiCoCL.4.

APPENDIX F DAQCO USER INTERFACE

The graphical user interface of DAQCo consists of three GUI components implemented in `UDaqMainWidget`, `UDaqPlotter` and `UDaqFileWidget`. `UDaqMainWidget` is a composite object hosting the two other objects. `UDaqMainWidget` itself provides only minimum functionality through three menus called *File*, *Acquisition* and *View*, which are shown in Figure F.1. The first menu is provides the functionality required to stop DAQCo, the second opens configuration dialog, and the third shows or hides the `UDaqFileWidget`.

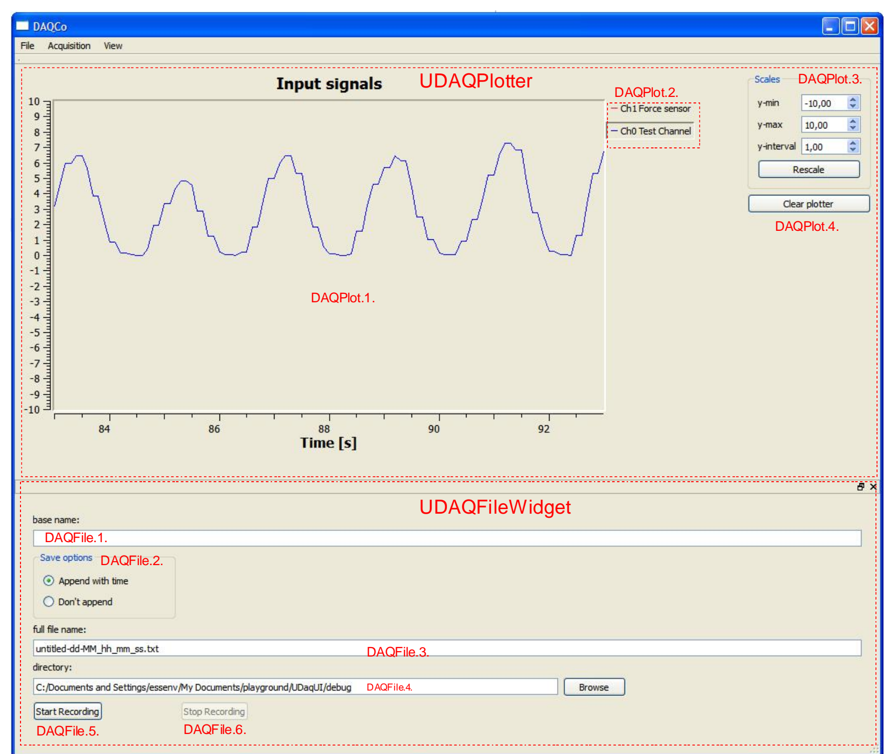

Figure F.1 DAQCo user interface

`UDaqPlotter` is responsible for providing the functionality required in visualization of the acquired data. `UDaqPlotter` provides few features that allow customizing of the visualized data. Table F.1 presents the functionality provided through `UDaqPlotter` and describes types of the used Qt GUI components. More detailed information regarding the Qt GUI components is available in [26].

Table F.1 Functionality of `UDaqPlotter`

DAQPlot.1.	**Output** QwtWidget	Input signals	Visualization of acquired signals
DAQPlot.2.	**Input** QwtLabel	N/A	List of acquired signals. Operator may show/hide signals shown in DAQPlot.1
DAQPlot.3.	**Input** QDoubleSpinBox: y-min y-max y-interval. QPushButton: rescale	Scales	Scaling options for DAQPlot.1 y-axis. *y-min*: y-axis minimum value *y-max*: y-axis maximum value *y-interval*: interval between y-axis major ticks *Rescale*: applies the modification made to DAQPlot.3. fields
DAQPlot.4.	**Input** QPushButton	Clear plotter	Clears the history data of DAQPlot.1.

`UDaqFileWidget` is responsible for providing a GUI component, which allows storing of the acquired data into a file. The functionality is provided through several text input fields, which determine filename, directory and possible usage of timestamp. `UDaqFileWidget` features are presented more in details in Table F.2.

Table F.2 Functionality of `UDaqFileWidget`

DAQFile.1.	**Input** QLineEdit	Base name	Base for naming the recorded data.
DAQFile.2.	**Input** QRadioButton	Save options	Time stamp option for file name entered to field DAQFile.1. Options are: *Append with time*, which appends the file name with a time stamp containing date, hour, minutes and second. *Don't append*, which maintains to original file name
DAQFile.3.	**Output** QLineEdit	Full file name	Shows whole file name, including format of the possible time stamp
DAQFile.4.	**Input** QLineEdit	Directory	Directory where the recorded data will be stored with file name indicated by field DAQFile.3.
DAQFile.5.	**Input** QPushButton	Start Recording	Starts recording incoming signals to file indicated by field DAQFile.3. located in directory of DAQFile.4. *Note1:* Button is disabled when pressed. *Note2:* Pressing enables button DAQFile.6. *Note3:* If a file indicated by the fields DAQFile.3. and DAQFile.4. already exists, the overwriting is confirmed with a dialog.
DAQFile.6.	**Input** QPushButton	Stop Recording	Stops recording incoming signals and closes the file, where the data has been stored *Note1:* Button is disabled when pressed. *Note2:* Successful closing of measurement file enables button DAQFile.5.

DAQCo User Interface

DAQCo user interface includes an additional dialog called `UDaqConfigureDialog`, which allows the operator to alter the channel configuration of the DAQ units attached to DAQS. Illustration of `UDaqConfigureDialog` is presented in Figure F.2.

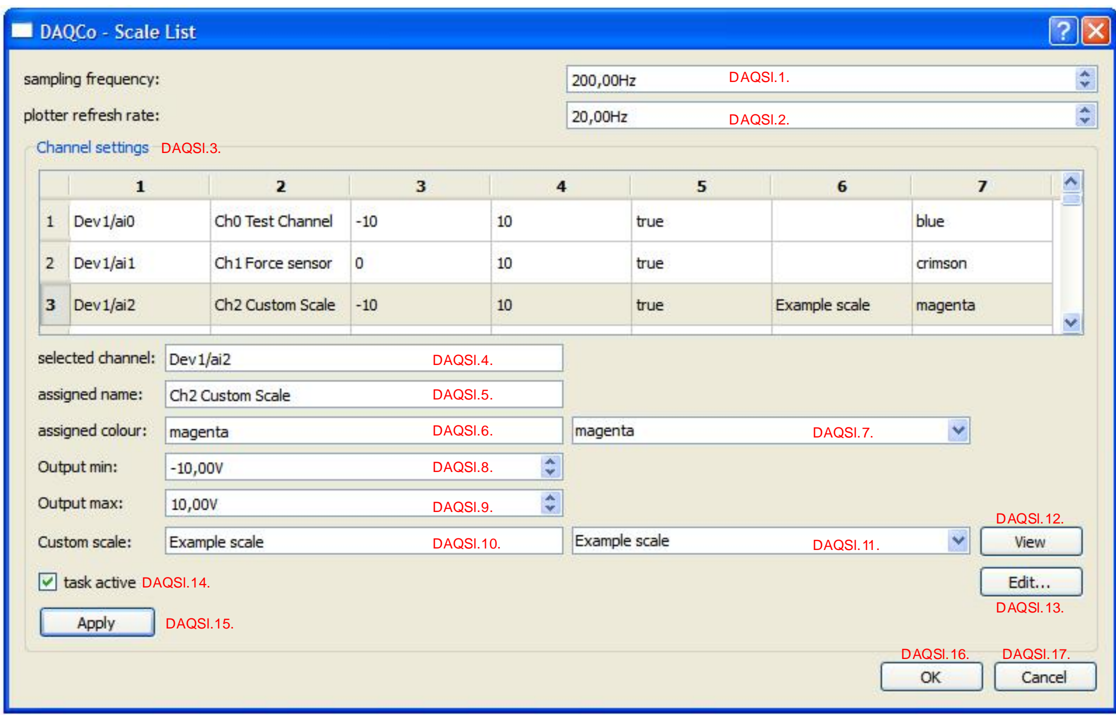

Figure F.2 Configuration dialog `UDaqConfigureDialog`

`UDaqConfigureDialog` dialog allows operators to configure each channel of a DAQ unit separately. Operator may assign each channel with different output range and visualization colour. Channel can be removed from visualization and data storing by deactiving them. In addition, each channel may have its own custom conversion scale, which converts the input signal into user defined unit. The custom conversion scales are given as polynomials. The detailed functionality of *`UDaqConfigureDialog`* is presented in Table F.3.

Table F.3 Functionality of `UDaqConfigureDialog`

DAQSl.1.	**Input** QDoubleSpinBox	Sampling frequency	Data acquisition sampling frequency in Hertz. Valid range 10-15000 Hertz.
DAQSl.2.	**Input** QDoubleSpinBox	Plotter refresh rate	Refresh rate for `UDaqPlotter` in Hertz. Valid range 1-300 Hertz.
DAQSl.3.	**Output** QTableWidget	Channel settings	Presents lists of available channels together with user defined parameters. Column 1: Physical address Column 2: Assigned name Column 3: Output minimum value in Volts Column 4: Output maximum value in Volts Column 5: Channel active in data acquisition Column 6: Name of assigned custom conversion scale Column 7: Assigned visualization colour *Note:* Prior to editing fields DAQSl.4-DAQSl.15, the channel must be selected by pressing the respective row of DAQSl.3.
DAQSl.4.	**Output** QLineEdit	Selected channel	Indicates selected channel's physical address
DAQSl.5.	**Input** QLineEdit	Assigned name	Assigns a name for selected channel
DAQSl.6.	**Output** QLineEdit	Assigned colour	Indicates selected channel's visualization colour
DAQSl.7.	**Input** QComboBox	N/a	Assigns a new visualization colour for selected channel
DAQSl.8.	**Input** QDoubleSpinBox	Output min	Assigns a new minimum output value for selected channel. Valid range -10-10 Volts
DAQSl.9.	**Input** QDoubleSpinBox	Output max	Assigns a new maximum output value for selected channel. Valid range -10-10 Volts
DAQSl.10.	**Output** QLineEdit	Custom scale	Active channel's custom conversion scale
DAQSl.11	**Input** QComboBox	N/A	Assigns a new custom conversion scale for selected channel
DAQSl.12	**Input** QPushButton	View	Displays the structure of selected custom conversion scale
DAQSl.13	**Input** QPushButton	Edit…	Opens *Custom Scale Editor* dialog
DAQS1.14	**Input** QCheckBox	Task active	Activates or deactivates selected channel
DAQSl.15	**Input** QPushButton	Apply	Applies the modifications made to selected channel
DAQSl.16	**Input** QPushButton	OK	Applies the modifications made to selected channel and closes the dialog
DAQSl.17	**Input** QPushButton	Cancel	Cancels the modifications made to selected channel and closes the dialog