

In collaboration with



# Intuitive Programming of AIMM Robot

**Christian Carøe**

**Mikkel Hvilshøj**

**Casper Schou**



**AALBORG UNIVERSITY**

**Master's Thesis in Manufacturing Technology**

**Department of Mechanical and Manufacturing Engineering**



**Title:**

Intuitive Programming of  
AIMM Robot

**Semester Themes:**

Technological Innovation (3<sup>rd</sup>)  
Manufacturing Technology (4<sup>th</sup>)

**ECTS:**

60

**Project Period:**

September 2, 2011 - June 1, 2012

**Project Group:**

VT - 3<sup>rd</sup> and 4<sup>th</sup> semester  
Manufacturing Technology

**Group Members:**

---

Christian Carøe

---

Mikkel Hvilshøj

---

Casper Schou

**Supervisor:**

Professor Ole Madsen

**Number printed:** 5

**Report:** 87 pages

**Appendix:** 53 pages

**Annex:** 3 pages

**Submitted:** June 1, 2012

**Department of Mechanical &  
Manufacturing Engineering**

Fibigerstræde 16,  
DK-9220 Aalborg Ø  
Phone: +45 9940 8934  
Fax: +45 9815 3040  
<http://www.m-tech.aau.dk>

**Synopsis:**

In this project the design of the mobile manipulator Little Helper ++ is described. Little Helper ++ consists of a Neobotix platform, an electric gripper and the innovative KUKA Light Weight Robot. The work of this project contributes to the EU-funded research project TAPAS, in which the purpose is to develop a flexible industrial production assistant to meet the demand for a more flexible production.

The project describes the development of an intuitive software system, *Little Helper System*, controlling the manipulator and the gripper. *Little Helper System* is based on task-level programming consisting of three layers, devices primitives, skills and tasks, respectively. To control *Little Helper System* two different interfaces have been developed, a terminal based user interface (TUI) and a graphical user interface (GUI). GUI is accessible from an iPad which creates a user-friendly and comprehensible platform for controlling *Little Helper System*, by which the programming and the execution of an industrial task are brought to a level in which robotics expertise is no longer needed. GUI and the skill-based approach have been tested by a non robotics expert who subsequent to a short introduction has accomplished an industrial task replicated from Grundfos.

*Little Helper System* is verified by programming and executing two complex industrial tasks replicated from Grundfos. From the success of these tasks it is concluded that the skill-based approach is beneficial in programming and executing a complex industrial task.

*By signing this document, each member of the group confirms participation on equal terms in the project. Thus, each member of the group is responsible for the content of the report.*



# Preface

---

This project is an extended master's thesis including the 3<sup>rd</sup> and 4<sup>th</sup> semesters of Manufacturing Technology at Aalborg University, Department of Production and Mechanical Engineering, from September 2, 2011 to June 1, 2012. The themes of the 3<sup>rd</sup> and 4<sup>th</sup> semesters are *Technological Innovation* and *Manufacturing Technology*, respectively. The project is conducted in close collaboration with the EU-funded research project TAPAS.

The report is divided into 12 chapters, 6 appendices, and 1 annex. Each chapter is introduced by a text in italics, describing the purpose and content of the chapter. Tables, formulas, and figures are numbered by an index number corresponding to the respective chapter number e.g. figure 1.2 is the second figure found in chapter 1. Supplementary written material of this report is indexed as appendices. Next to the appendix an annex is included.

Videos captured during the project are included on an enclosed Video DVD and furthermore on (Youtube, 2011). When referring to a video, the report includes a QR-code linking to the respective video on (Youtube, 2011). The QR-code is scanned by using a smart phone or a tablet. To scan the QR-code by means of an iPhone the application *Scan* may be used and by means of an Android phone, *NeoReader* may be used. Please note that when the QR-code is scanned, the Internet is accessed. Files created during the project are included on an enclosed Appendix CD. References to files on the Appendix CD are shown as <path/filename.type>. The content of the enclosed Appendix CD is found on the back page of the report. Both the Video DVD and the Appendix CD are attached to the report. Furthermore the report contains a list of definitions which is found in chapter 12.

The group has participated in a study trip to Germany visiting KUKA Roboter GmbH, DLR, Bielefeld University, and the Technical University in Berlin. Furthermore the group has attended a guest lecture by Doctor Rodney Brooks at the University of Southern Denmark, the Hans Christian Andersen Academy, in Odense. In connection with the trip to Odense the group paid a visit to the company Universal Robots. A description of the study trip is found on the enclosed Appendix CD in </Documents/Study Trip.pdf>. The group had a booth at the Danish Robot Networks road trip "Robotterne kommer" on November the 14<sup>th</sup> 2011 at Aalborg University. The group carried out a demonstration at an event at Aalborg University as part of the *European Robotics Week* on November the 29<sup>th</sup> 2011 and furthermore at an event for primary school students in January 2012. An article about this project published on Videnskab.dk is included in the annex.

The project group want to thank the supervisor Professor Ole Madsen, the Ph.D. students Simon Bøgh, Mads Hvilshøj, Mikkel Rath Pedersen and former Ph.D. student Oluf Skov Nielsen, at the Department of Production and Mechanical Engineering at Aalborg University. Besides the group addresses a thank to Grundfos for good cooperation.



## Dansk resumé

---

Dette projekt er udarbejdet i tæt samarbejde med EU-projektet TAPAS. Projektet omhandler udviklingen af et softwaresystem, der gør anvendelsen og programmeringen af en mobil robot mulig for en operatør. Visionen for projektet er at udvikle en mobil robot, der kan indgå som en fleksibel produktionsressource til at assistere operatørerne i produktionen. Robotten instrueres nemt og intuitivt af produktionspersonalet til at varetage manuelle produktionsopgaver uden større omstrukturering af produktionsmiljøet. Den mobile robot skal ikke ses som en erstatning for operatørerne, men som en assistent ved gentagelsesprægede eller fysisk belastende arbejdsopgaver. Det følgende citat illustrerer visionen for dette projekt.

*"Fremtidens industrivirksomheder beskæftiger ikke kun mennesker af kød og blod. De giver også fuld-tidsarbejde til små robotter, der styrter rundt og går til hænde, hvor de ansattes kroppe og hjerner kommer til kort. De små teknologiske hjælpere får produktionen til at glide og sørger for, at opgaverne løses hurtigt og effektivt, så produkterne kan komme ud på markedet i en fart."* (Hildebrandt, 2012)

I projektet er den mobile robot "Little Helper ++" blevet designet og konstrueret. Den er baseret på en Neobotix platform, en elektrisk parallel griber og den innovative KUKA Light Weight Robot (LWR). Samtidig er et softwaresystem, kaldet *Little Helper System*, blevet udviklet til opsætning og afvikling af industrielle opgaver. Dette softwaresystem er bygget op omkring en række udviklede robotfærdigheder kaldet *skills*. Ved anvendelse af disse robotfærdigheder bliver en robotopgave ikke længere en specificering af individuelle robotbevægelser, men en specificering af operationer knyttet til den enkelte opgave. Robotfærdighederne består af softwareblokke, som er vigtige for den intuitive programmering. På den måde forsimples programmeringen af en kompleks opgave.

For at gøre robotprogrammering mulig for en operatør er der udviklet et intuitivt og let forståeligt brugerinterface. Dette interface består af en grafisk brugerflade, der kan tilgås fra en iPad, samt en fysisk interaktion med robotten. Under programmering af en opgave sættes selve sekvensen op i den grafiske brugerflade, hvorefter operatøren indlærer koordinater ved fysisk at flytte robotten rundt. Operatøren kan løbende under indlæringen af koordinater læse instruktioner på iPad'en om, hvordan der skal interageres med robotten.

Det udviklede interface kombineret med den færdighedsbaserede tilgang gør en operatør i stand til simpelt og intuitivt at programmere en industriel opgave, hvilket er blevet verificeret af en testperson uden større robotekspertise.

For at verificere det udviklede softwaresystem og det udviklede brugerinterface er der udført to komplicerede industrielle montageopgaver replikeret fra Grundfos. Programmeringen af disse montageopgaver er foretaget gennem det udviklede softwaresystem og det intuitive interface. Der er udviklet 10 forskellige robotfærdigheder, ud fra hvilke de to montageopgaver er blevet udført succesfuldt af Little Helper ++. Herudfra konkluderes det, at den robotfærdighedsbaserede tilgang med fordel kan anvendes til at løse industrielle opgaver.

# Table of Contents

---

<b>Preface</b>	<b>v</b>
<b>Dansk resumé</b>	<b>vii</b>
<b>Table of Contents</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Vision</b>	<b>5</b>
2.1 Vision . . . . .	5
2.2 Goal of this Project . . . . .	7
<b>3 Building Little Helper ++</b>	<b>9</b>
3.1 Previous Versions of Little Helper . . . . .	9
3.2 Little Helper ++ . . . . .	10
3.3 The Schunk Gripper . . . . .	14
3.4 Budget Summary . . . . .	14
<b>4 Industrial Verification</b>	<b>15</b>
4.1 KUKA LWR . . . . .	15
4.2 SQFlex Rotor Assembly Task . . . . .	16
<b>5 Definition of Skills</b>	<b>19</b>
5.1 Definition . . . . .	19
5.2 Implementation of Classic Robot Macros . . . . .	21
<b>6 Little Helper System</b>	<b>25</b>
6.1 Manipulator DCN . . . . .	26
6.2 Gripper DCN . . . . .	30
6.3 Platform DCN . . . . .	30
6.4 Vision DCN . . . . .	31
6.5 Skill-Based System . . . . .	31
6.6 Terminal User Interface - TUI . . . . .	36
6.7 Graphical User Interface - GUI . . . . .	41
6.8 Summary . . . . .	47
<b>7 Library of Skills</b>	<b>49</b>
7.1 Pick Skill . . . . .	49
7.2 PegInHole Skill . . . . .	52
7.3 More Skills . . . . .	55



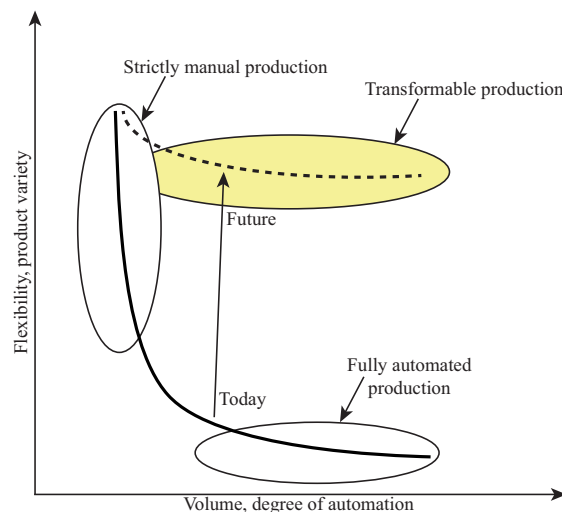
<b>8 Industrial Assembly Tasks</b>	<b>59</b>
8.1 Programming Approach . . . . .	60
8.2 Workstation 1 - Rotor Shaft Assembly . . . . .	60
8.3 Workstation 2 - Rotor assembly . . . . .	65
8.4 Workstation 3 - Rotor Cap Collection . . . . .	69
8.5 Conclusion . . . . .	71
<b>9 Discussion</b>	<b>73</b>
<b>10 Conclusion</b>	<b>77</b>
<b>11 Reflection</b>	<b>79</b>
11.1 TAPAS Month 24 Demonstration . . . . .	79
11.2 Commercialisation . . . . .	80
<b>Bibliography</b>	<b>83</b>
<b>12 Definitions</b>	<b>85</b>
<b>Appendix A Programmed Tasks</b>	<b>91</b>
A.1 LEGO DUPLO Assembly Task . . . . .	91
A.2 SQFlex Rotor Assembly . . . . .	97
A.3 Other Programmed Tasks . . . . .	99
<b>Appendix B Robot Operating System</b>	<b>103</b>
B.1 ROS File System Level . . . . .	103
B.2 ROS Computation Graph Level . . . . .	103
B.3 ROS Community Level . . . . .	104
<b>Appendix C Analysis of the KUKA LWR</b>	<b>105</b>
C.1 Inaccurate Force . . . . .	105
C.2 Interrupt - KRL . . . . .	106
C.3 Interrupt - External PC . . . . .	109
C.4 Latency of the Communication . . . . .	112
<b>Appendix D Budget for Building Little Helper ++</b>	<b>115</b>
<b>Appendix E Device Control Nodes</b>	<b>119</b>
E.1 Gripper DCN . . . . .	119
E.2 Vision DCN . . . . .	123
E.3 Platform DCN . . . . .	127
<b>Appendix F Skills and Device Primitives</b>	<b>129</b>
F.1 Skills . . . . .	129
F.2 Device Primitives . . . . .	142



# Introduction 1

This project is an extended master's thesis composed on the 3<sup>rd</sup> and 4<sup>th</sup> semester on Manufacturing Technology at Aalborg University's Department of Production and Mechanical Engineering. The project contributes to the EU-funded research project TAPAS in which the Department of Mechanical and Manufacturing Engineering participates. The basis of this project is the verification of a skill-based approach on industrial assembly tasks.

Today industrial robots are widely used in many different production systems throughout the world. The majority of these robots are inflexible, as they are fixed to a dedicated workstation. This inflexibility makes it difficult and time consuming to reprogram and/or move the robot to another task. A solution to this inflexibility could be a mobile robot. A mobile manipulation system can be described as an autonomous industrial mobile manipulator (AIMM) which aims at developing integrated robotic systems capable of performing work and assistance in industrial manufacturing environments. (M. Hvilshøj, S. Bøgh, O. S. Nielsen & O. Madsen, 2011) Figure 1.1 illustrates the vision of a new production segment, which offers a flexible mass production. In order to obtain this vision the need for flexible automation equipment emerges, to which an AIMM is considered as a part of the solutions.



**Figure 1.1:** The vision of a new production segment, which offers a flexible mass production. (M. Hvilshøj, S. Bøgh, O. S. Nielsen & O. Madsen, 2011)

Production-wise flexibility today means "to produce reasonably priced customized products of high quality that can be quickly delivered to customers." (Dr. Viorica Frunza, 2012) A mobile manipulator with the capability to be intuitive and easily programmed will have flexibility towards that of a human but still have the repeatability and accuracy of a robot. Mobile robots in the industry will not alone solve the

demand for flexible production systems but should be seen as a production resource similar to a human since the production capacity of the mobile robot easily and quickly can be moved from one production line to another. Thereby a mobile manipulator can increase or decrease the capacity of a production line if the demand changes.

### **AIMM**

AIMMs are intended as both a flexible and autonomous production equipment but also as an assistance to the operator. This implies that the AIMMs must co-exist with the operator, yet be independent systems capable of taking care of them selves for longer periods of time. In order to operate in an industrial environment the AIMM must be able to gather information about the environment and react upon this. This requires the ability to cope with a dynamic environment and the ability to avoid harming the surrounding equipment, people or the AIMM itself.

AIMMs consist of four main systems:

- a mobile platform
- a robotic manipulator
- a tooling system
- a sensor system, often a vision system

Furthermore a sophisticated IT-system capable of controlling and uniting these systems is an essential part of an AIMM.

Today research within the field of AIMM is being conducted throughout the world, including at Aalborg University.

### **Little Helper**

In 2007/2008 a group of four master students at Aalborg University developed and constructed an AIMM at the Department of Mechanical and Manufacturing Engineering. A picture of the constructed mobile robot is shown in figure 1.2. The fundamental vision was to create a "little helper" to assist the manual production personal by attending simple repetitive tasks in the production. From this vision the mobile robot and the project itself got the name "Little Helper". (S. Bøgh, M. Hvilshøj, C. Myrhøj & J. Stepping, 2008)

The main components of the mobile robot were a Neobotix platform, an Adept manipulator and a custom frame structure containing several electronic, pneumatic tooling system, and vision system. For further information about the components of Little Helper, see chapter 3 and (S. Bøgh, M. Hvilshøj, C. Myrhøj & J. Stepping, 2008).

In continuation of the Little Helper project both master projects and Ph.d projects have been conducted. In 2011 a second version of Little Helper was developed. This version was called Little Helper +. It still features the Neobotix platform but instead of the Adept manipulator a KUKA Light Weight Robot (LWR) manipulator was fitted. The KUKA LWR is currently only intended for research purpose and features torque sensors in each joint, a weight of only 16 kg, a payload of 7 kg, and a 7th axis. Because the controller for the KUKA LWR is considerably larger than the Adept controller, a new custom



*Figure 1.2: Little Helper mobile robot.*

frame structure was developed. To avoid increasing the size of the frame structure Little Helper + does not feature a pneumatic system. Since no electric tools were available at AAU until December 2011, Little Helper + was only used with passive tools until then. In chapter 3 and (Pedersen, 2011) further information about the components of Little Helper + is found.

### **The TAPAS Project**

The work related to the Little Helper project was the foundation for AAU's participation in a EU-funded project called TAPAS (TAPAS Community, 2009, page 4). The TAPAS project started in October 2010 and finishes in March 2014, a period of 42 months. The vision of the TAPAS project is:

*"TAPAS aims at paving the ground for a new generation of transformable solutions to automation and logistics for small and large series production, economic viable and flexible, regardless of changes in volumes and product type."*

This vision takes its basis in the demand for more flexible production systems, but also the fundamental vision of the Little Helper project. Apart from the Little Helper project the main focus is no longer the hardware and construction of the mobile robot, but on the development of the skill-based software and the industrial integration of the entire system. Further information about the TAPAS project is found in (TAPAS Description, 2010) and (TAPAS Community, 2009). (TAPAS Description, 2010) contains a detailed description of the work packages within the TAPAS project. Aalborg University is mainly focused on work package 3 and 4, please refer to (TAPAS Community, 2009, page 32).

### **Device Primitives and Skills**

As seen from the vision of the original Little Helper project and the vision of the TAPAS project the need for an intuitive and simple programming interface emerges. One of the key elements in achieving this, and furthermore simplifying the autonomous operation of the robot, is skills. Device primitives are the most basic layer. These are simple motions and functionalities bound to the devices, for an example

*Move linear* on the manipulator. Skills are actions corresponded to the manipulation of an object, for an example *Pick* or *Place*. Skills are composed of device primitives, thus creating a layer above these. A more comprehensive description of device primitives and skills is found in chapter 5.

### **This Project**

This project has been planned in correlation to the TAPAS project in order to achieve a close collaboration. The project will both contribute to the work in TAPAS but also benefit from the research and discussion within the TAPAS community. From an AIMM point of view this master's thesis will focus on the robotic manipulation, hence the manipulator and the tool. A main topic will be the development of the sufficient device primitives and skills for performing industrial assembly tasks. Furthermore a complete system featuring an intuitive programming interface will be developed incorporating these skills and device primitives.

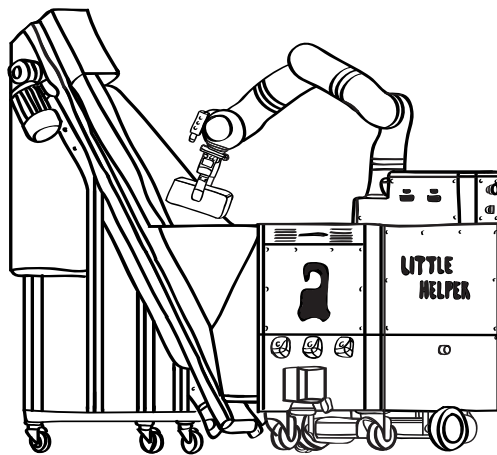
# Vision 2

---

*In this chapter a vision for an industrial implemented version of the Little Helper robot is proposed. The purpose of this vision is to clarify the general intentions of an fully developed industrial mobile robot. Based on this vision a goal for this master project is defined. The intention is to define the goal at the beginning of the project to help organizing the work and content of the project.*

## 2.1 Vision

It has been chosen to present the vision for a fully developed industrial mobile robot as a description of a daily use case of an fully developed industrial mobile robot. It is chosen to describe the use case in prose in order to tell the "story" from the operator's point of view.



**Figure 2.1:** Sketch of AIMM performing a part feeding task.

It is assumed that the mobile robot is operating in the area of rotor assembly in the SQ factory at Grundfos A/S. The mobile robot has a mission, where it primarily assembles the rotor of a SQFlex pump, but when needed the robot can also assist the operators in the assembly of the rotor shaft. It also handles part feeding, both to its own assembly task, but also to a spinning cell located near by. The mobile robot operates autonomously without human interaction making decisions based upon an assigned mission and the signals received from the surrounding production equipment. Figure 2.1 shows a sketch of the mobile robot performing a machine tending task.

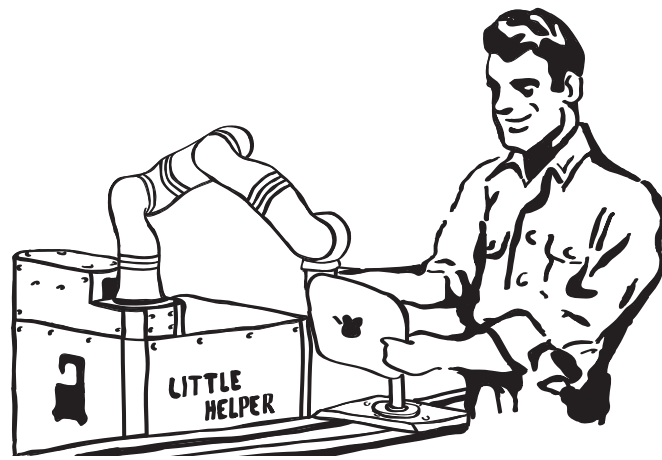
*John is an operator at the SQ plant at Grundfos A/S, working in the area of motor assembly. John has attended a simple instruction course in using one of the factory's mobile robots. A machine breakdown*

*has put extra pressure on the motor assembly line. To regain the lost production capacity a mobile robot from the area of rotor assembly has been set at John's disposal. Using his iPad John connects to the mobile robot, instructs it to finish its current assembly job and then drive to the motor assembly line.*

*A few minutes later the robot arrives.*

*Using an intuitive interface on his iPad combined with touch, vision, and voice gestures John programs two different assembly operations on the mobile robot. He also programs the robot to handle a part feeding task and a machine tending task. Finally he sets up the mission stating batch sizes, priorities and other relevant information.*

Figure 2.2 below is a sketch illustrating the fundamental idea of an operator intuitively programming the mobile robot.



**Figure 2.2:** Sketch of John interacting with the AIMM to program a new task.

*A few hours later the mobile robot has made up for the lost production capacity at one of the given workstations. John now wants to assign the robot to a new assembly task. During the programming of this assembly task John finds himself limited by some of the variable options in a skill, so he calls Robert.*

*Robert is the engineer responsible for the mobile robots in the SQ factory. He gets a call from the motor assembly line, where John is trying to program a task requiring a functionality beyond the current capabilities of the available skills. At the motor assembly line Robert and John agree on how to incorporate this new option. With this information Robert returns to his office, where he can access the entire system, including the underlying libraries, system variables etc. which are not available to the production operators. By using this interface he is able to develop, create, and change skills. He programs the option into the skill as requested by John and then returns to John. He assists John in programming the assembly to make sure that the new option is functioning as intended.*

The essential part of the vision is that the operator is able to intuitively and relatively fast to program the mobile robot to a new task. Furthermore the system is in constant development due to the close collaboration between the operators and the engineers maintaining the system.



## 2.2 Goal of this Project

The goal of this master project is to give a step towards realizing the vision for a fully developed mobile robot. The purpose is to demonstrate that the vision is achievable, and to give an idea of how a fully developed system would operate. Furthermore it should provide important "hands on" experience and testing, which can be used in the further development of Little Helper and TAPAS.

The main goal of the project is the development of a system based on skills and device primitives capable of programming and executing industrial tasks. In addition to this a library of skills and device primitives has to be established. To verify the developed system it is chosen to perform actual industrial assembly tasks programmed using skills and device primitives. Even though pick and place operations are a main part of most assemblies the industrial assembly operations performed must incorporate some operations with a complexity level beyond simple pick and place in order to demonstrate the flexibility of the system and the KUKA LWR.

To demonstrate a functioning system and to present the vision of an operating mobile robot, it is chosen to perform a demonstration incorporating the developed functionalities of the system.

It is desired that the demonstration incorporates as many aspects of a mobile robot operating in an industrial environment as possible in order to give an impression of the vision. In order to do this, it is the goal to use a mobile robot instead of the stationary set-up of the KUKA LWR available at Aalborg University at the initiation of this project<sup>1</sup>. It also demands use of actual industrial components and workpieces for the tasks performed by the mobile robot.

To give the impression of a fully developed mobile robot that can be programmed by a non robotics expert it is the goal to develop an intuitive man-machine-interface for programming a simple assembly. This is tested and verified by having a non robotics expert perform an actual programming situation of an assembly task.

The following itemized list summarizes the goals for the conclusion in 2012 of this master project.

1. Construct a mobile robot with a KUKA LWR.
2. Develop a robot programming system based on skills and device primitives.
3. Verify the developed system on complex industrial assembly tasks.
4. Develop a simple and intuitive interface for programming a new task.

Below follows an elaboration of the four goals from the list above.

### **Construct a mobile robot with a KUKA LWR**

At Aalborg University's Department for Mechanical and Manufacturing Engineering there is no longer a mobile robot available because Little Helper (first version) has been dismantled and Little Helper + (second version) has been relocated to Aalborg University's Department in Ballerup. In order to have a mobile robot available in Aalborg it is chosen to build the 3rd generation of Little Helper, Little Helper ++. It will be build from the platform of Little Helper, the KUKA LWR in Aalborg and a Schunk WSG50 electric parallel gripper, see chapter 3.

---

<sup>1</sup>Little Helper + features the KUKA LWR, but it is situated at Aalborg University in Ballerup.

### **Develop a robot programming system based on skills and device primitives**

First step in the development of a programming system based on skills and device primitives is to establish a system architecture. This is essential for structuring skills and device primitives. The objective is to both establish an architecture for the hardware, the software, and the communication which is described in chapter 6.

In order to perform several industrial tasks using the skill-based approach a library of relevant skills and device primitives must be developed which is described in chapter 7. The development of skills will be motivated by the programming of industrial tasks.

### **Verify the developed system on complex industrial assembly tasks**

In order to use the KUKA LWR as the manipulator on a mobile robot it is essential to verify its performance on a real industrial assembly, and consequently a simple verification will be done using the KRL language on the KRC prior to developing the control system which is described in chapter 4. Furthermore a simple verification of the skill-based approach implementing simple low-practice macros is described in chapter 5.

To verify the developed system, industrial assembly tasks with complexity exceeding simple pick and place operations will be performed. In conclusion of this project a demonstration will be performed, where the industrial assembly tasks will be included. This requires the set up, programming and testing of the developed system. The demonstration will serve as a validation of the entire system which is described in chapter 8.

### **Develop a simple and intuitive interface for programming a new task**

In order to have a non robotics expert to program a simple assembly task, a simple user interface for programming the system must be developed which is described in section 6.7. This interface must enable the user to use the functionality of the system but it must still be intuitive and simple to use. The purpose of the interface is also to present the vision for an easy programmable system. As the user interface is not a main focus of this project no specific demands for the interface are made.

# Building Little Helper ++ 3

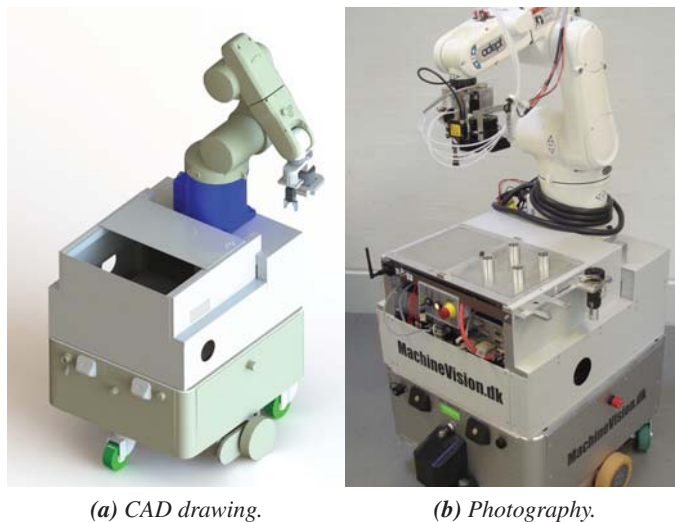
*This chapter describes the design and construction of the third generation of the Little Helper AIMM, Little Helper ++. This robot will be used in this master project and within the TAPAS project. The construction takes its basis in Little Helper +. The purpose of this chapter is to document the construction phase and the hardware components' implementation. The construction of Little Helper ++ is conducted in January 2012. A fast forward video of the physical construction of Little Helper ++ is obtained by scanning the QR-code (Building Little Helper ++ fast). A budget for all the components needed to construct Little Helper ++ is found in appendix D.*



*Building Little Helper ++ fast*

## 3.1 Previous Versions of Little Helper

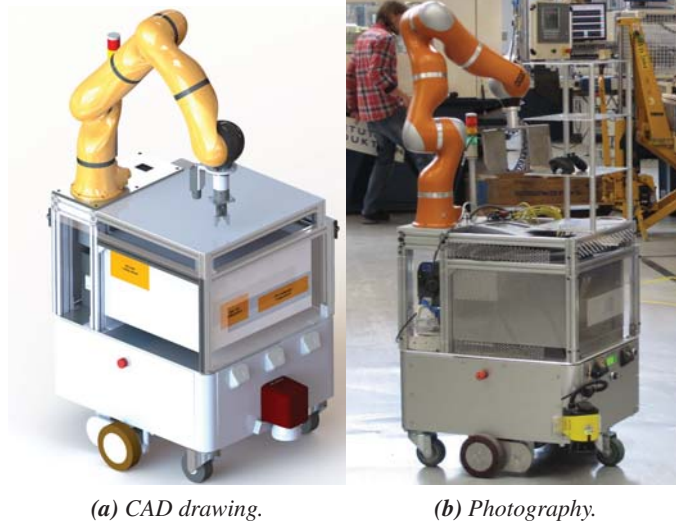
The first version of Little Helper is illustrated in figure 3.1 and was constructed using a Neobotix platform with integrated control software, an Adept manipulator etc. The control software on the platform makes it capable of map construction and collision avoiding. To avoid collisions the platform uses two laser scanners and five ultrasonic sensors. The platform contains a battery pack of eight 12 VDC batteries which makes the platform operational for approximate eight hours. On top of the platform a frame structure for the manipulator PSU and controller is mounted. Furthermore a power inverter, a compressor, an air reservoir, and controllers for the vision system are mounted inside the frame structure. The compressor and air reservoir are used for both the tool changing system and to operate the different tools. On top of the frame structure the Adept Viper s560 manipulator, a tool magazine, and a top plate are mounted.



**Figure 3.1:** Little Helper constructed in 2007/2008 by (S. Bøgh, M. Hvilshøj, C. Myrhøj & J. Stepping, 2008).

The second version is named Little Helper +, see figure 3.2, which is used at Aalborg University in

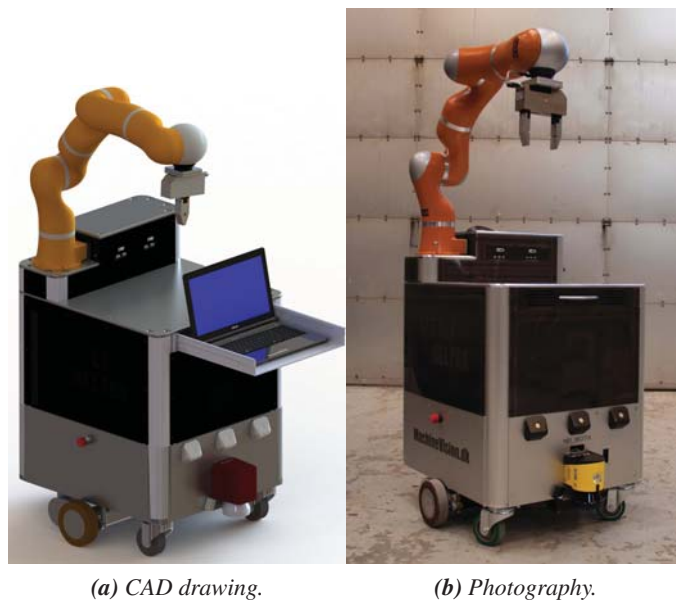
Ballerup. The main difference from the first version is that the Adept manipulator is replaced by a KUKA LWR. The controller of the KUKA LWR is rather large, and consequently the pneumatic tooling system is not implemented. Little Helper + is therefore designed with only a passive tooling option.



*Figure 3.2: Little Helper + constructed in 2011 by (Pedersen, 2011).*

## 3.2 Little Helper ++

Little Helper ++ is the newest version and the design is shown in figure 3.3. Little Helper ++ is designed and constructed in this project. The first version of Little Helper, figure 3.1, has been dismantled and the Neobotix platform is reused. The Neobotix platform has an accuracy of  $\pm 10$  mm and  $\pm 5$  degrees.

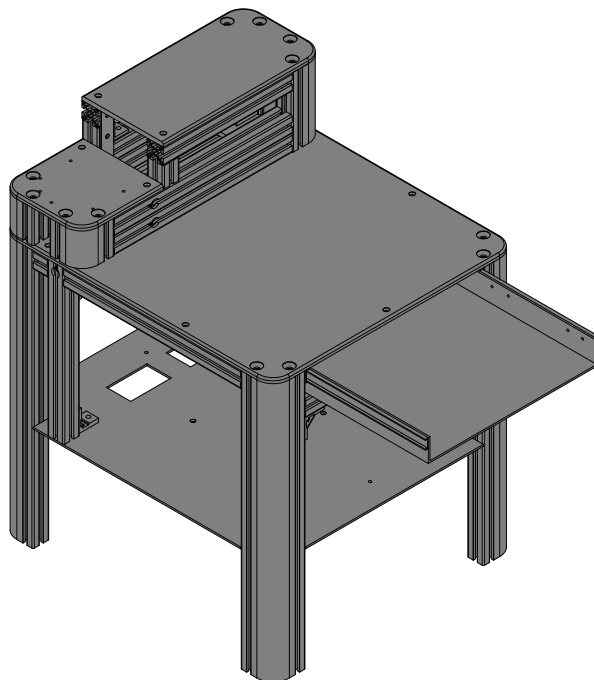


*Figure 3.3: Little Helper ++ constructed in this project.*

The frame structure of Little Helper ++ is designed to contain a circuit breaker, a fuse, a contactor, and an inverter to deliver 230 V which afterwards is converted to 24 V and 12 V through two different transformers. The frame structure furthermore contains the controller for the KUKA LWR, a 14 inch notebook, a wireless router, four ventilation fans to cool the electronics, a variety of cables and the teach pendant of the KRC.

### 3.2.1 The Platform Frame

To maintain the platform's ability to avoid collisions it is important to design the top of Little Helper ++ within the platform's dimensions. The Neobotix platform is constructed of MISUMI aluminium profiles, and it is chosen to use the same aluminium profiles for constructing the top frame to ensure a continues design. The frame is constructed as shown in figure 3.4. The height in which the manipulator is mounted is obtained from Little Helper due to studies of the optimal geometric design of a mobile robot. (S. Bøgh, M. Hvilshøj, C. Myrhøj & J. Stepping, 2008) The four through-going aluminium corner profiles, illustrated in figure 3.4, replace the corner profiles of the Neobotix platform to make the construction more rigid and in order to simplify the design. A 4 mm thick plate is placed on top of the platforms battery pack. This enables the KRC to slide into the frame without disassembling the frame structure. Beneath the top plate a drawer for a notebook is mounted, see figure 3.5. On each side of the frame plexiglass plates are mounted to cover the hardware as shown in figure 3.3. These cover plates have ventilation holes designed as the Little Helper logo and name. To assemble the MISUMI aluminium profiles dedicated types of nuts and brackets are used.



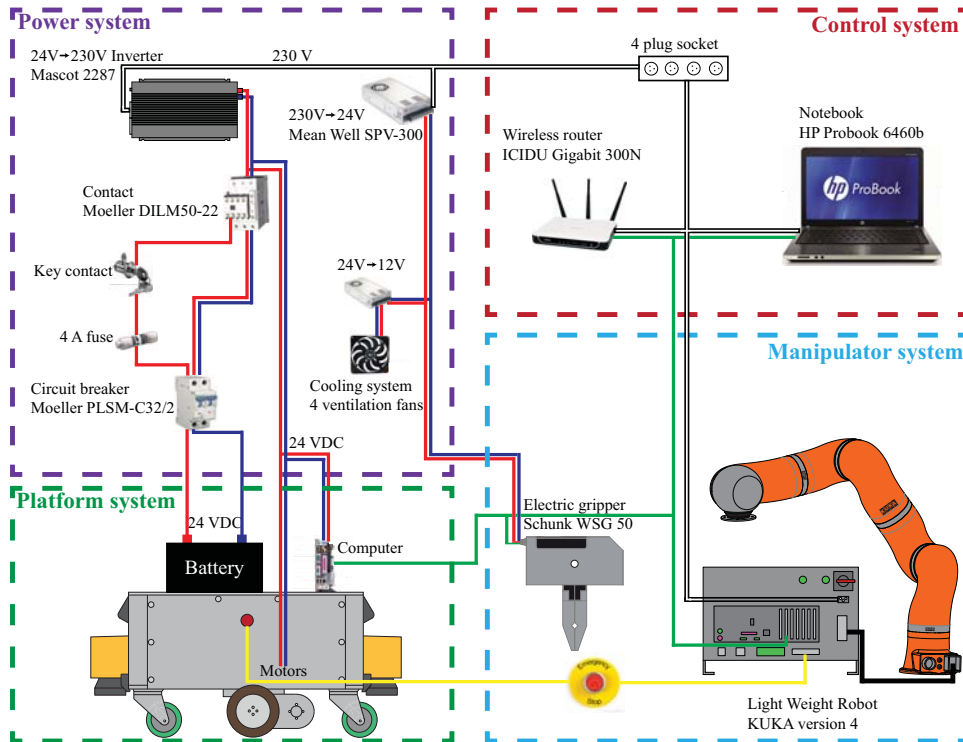
**Figure 3.4:** The aluminium frame designed for Little Helper, ++ using the MISUMI aluminium profiles.



*Figure 3.5: The laptop accessed from the drawer.*

### 3.2.2 The Electrical System

The eight 12 VDC batteries in the Neobotix platform are inverted into 230 VAC which is received by the KRC. For safety reasons a 32A circuit breaker, a contactor, a 4 A fuse, and a key contact are used. The wiring system is illustrated in figure 3.6.

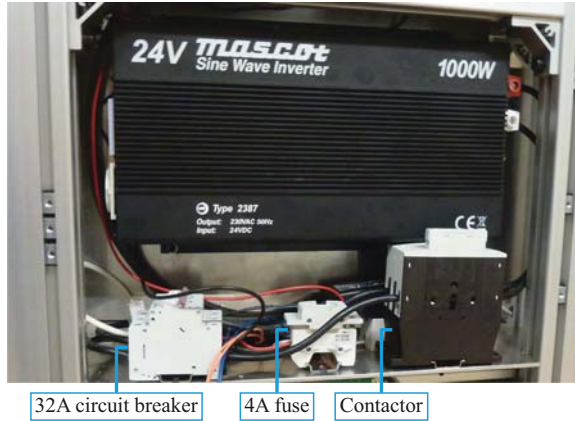


*Figure 3.6: Connections between the components of Little Helper ++.*

The electrical components are mounted inside the frame behind one of the end cover plates, which makes them easily accessible, see figure 3.7a. Four ventilation fans are mounted at the ventilation holes in the cover plates to reduce the heat inside the platform's frame structure. An emergency stop is mounted on the exterior of Little Helper ++ together with a key contact, see figure 3.7b, which turns on the power



supply from the batteries. The emergency stop button on the exterior is serial connected to the two emergency stop buttons on the platform. If one of the emergency stop buttons is pushed, both the platform and the LWR are stopped.



(a) The 24-230V inverter, 32A circuit breaker, 4A fuse and contactor inside Little Helper ++.



(b) Screen and USB connectors to the KRC and the Neobotix platform. Emergency stop and key contact to the battery power supply.

**Figure 3.7:** The electrical and emergency stop systems.

On top of the platform a connector for an external power supply (230 V) is fitted, see figure 3.8. Using this supply it is possible to operate Little Helper ++ except for the platform. It is possible to change between the power supplies using a small switch seen in figure 3.8.



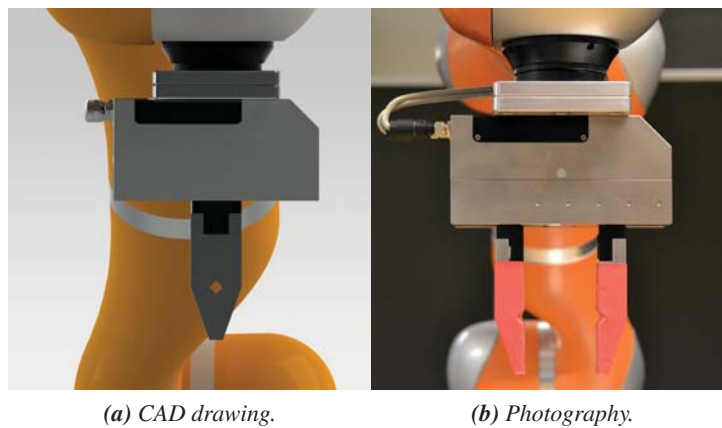
**Figure 3.8:** The 230V input and the switch to change the power supply.

### 3.2.3 The Computer System

The laptop is connected to a wireless gigabit router which is connected to the gripper, the KRC and the Neobotix platform. Furthermore the laptop has access to remote desktops on both the platform computer and the KRC. The connectors in figure 3.7b can be used to access the platform computer and the KRC. The embedded laptop is easy accessible from the drawer, see figure 3.5

### 3.3 The Schunk Gripper

The TAPAS project has financed a Schunk WSG50 electric gripper which is shown in figure 3.9. The figure shows a set of adapter plates for mounting the gripper onto the manipulator. Furthermore jaws for the gripper has been designed. The jaws are designed narrower at the tip to ensure that the manipulator can operate in narrow spaces. Furthermore a slot in each jaw is designed for better grasping of cylindrical objects. The Schunk WSG 50 gripper has a maximum stroke of 110 mm, a maximum grasping force of 120 N, and a maximum speed of 420 mm/s. The gripper is connected to 24 VDC and is controlled by an Ethernet UDP connection. The wiring to the gripper uses the build in wires in the KUKA LWR intended for components mounted on the TCP. The wiring diagram is found on the enclosed Appendix CD in </Documents/SchunkInstall.pdf>.



**Figure 3.9:** The Schunk WSG50 gripper with adapter plates for mounting it on the KUKA LWR and specially designed jaws.

### 3.4 Budget Summary

A budget for building Little Helper ++ is shown in table 3.1. A complete budget is found in appendix D. A number of different screws, cables, working hours etc. have been used which have not been included in the price.

Description	Price
Platform	46,800 EUR
Electrical system	1,412 EUR
Computer system	842 EUR
Schunk gripper	4,621 EUR
KUKA LWR	100,000.00 EUR
Total	153,732 EUR

**Table 3.1:** The budget for building Little Helper ++. All prices are excluding VAT.



# Industrial Verification 4

---

*The following chapter is based on practical work performed from the beginning of September to the end of November 2011. The purpose of this chapter is to document that it is possible to accomplish an industrial assembly task with the KUKA LWR. This chapter begins with an introduction to the KUKA LWR followed by a description of the chosen industrial assembly task.*

## 4.1 KUKA LWR

The KUKA LWR is a manipulator with seven degrees of freedom. The manipulator was originally developed by the German Aerospace Center (DLR) in the early nineties for use in aerospace programs. This project takes its starting point in the KUKA LWR 4+, figure 4.1.



*Figure 4.1: KUKA LWR 4+. (METAL SUPPLY, 2012)*

The LWR distinguishes itself from other manipulators by having a load to weight ratio close to 1:1, this means that the manipulator weighs approximately 16 kg and is able to handle loads up to 14 kg for a short period of time. Under normal conditions the manipulator's payload is limited to 7 kg to avoid overheating the electrical motors. The LWR uses both position and compliance control, which means that the manipulator is able to operate compliantly by measuring the torque in each joint and from these measurements calculate the exterior force. By the introduction of the compliance control benefits regarding safety are achieved, while programming and instruction of new tasks is eased by physical interaction with the manipulator, hence the ability to move the manipulator by hand. As a result of the compliance control it is possible to model the system as a mass-spring-damper system. Because of the compliance control and the integrated torque sensors complex industrial assembly tasks may be carried out more effectively. The compliance control and the low weight make the LWR suitable for use in human environments and for integration on a mobile robot. The manipulator is a modular system

consisting of five identical parts, a base and a head. The introduction of a seventh axis leads to an infinite number of different ways to reach the same coordinate and orientation in the workspace compared to a traditional six axis manipulator, which can only reach the same coordinate and orientation in maximum eight different ways. By the introduction of the seventh axis, benefits regarding flexibility and collision avoidance are achieved. The first generation of the KUKA LWR was made from carbon-fiber while the second generation is made from aluminium as a result of better heat emission. (KUKA Roboter GmbH, 2010)

### 4.1.1 Use of the KUKA LWR in an Industrial Assembly Task

An assembly task is described by contact between two or more objects and is often a straightforward process for a human because of a human's ability to use the senses of touch and vision. A traditional manipulator using position control is only able to move to an exact coordinate, but the use of the KUKA LWR enables the ability to sense contact by the introduction of torque sensors. The implementation of torque sensors enables the LWR to solve more complex tasks due to the LWR's ability to react upon and search for contact. Furthermore the LWR can act like a mass-spring-damper system which is beneficially for pick and place operations. In pick and place operations the object may become wedged in the fixture. To avoid this problem the manipulator is given low spring constants in the exposed directions. This property is beneficially for e.g. peg-in-hole operations because the manipulator can be given a low spring constant in the given direction, and the peg is able to slide into the hole while simply piloting the manipulator.



*Rotor Assembly in KRL*

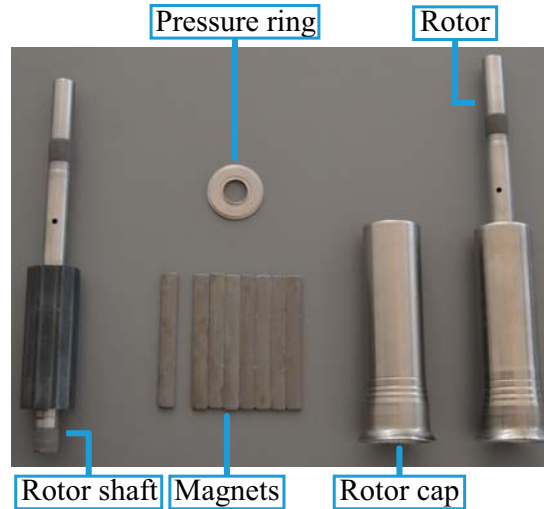
## 4.2 SQFlex Rotor Assembly Task

A complete description of the assembly task is found in appendix A.2 and a video of the assembly task is obtained by scanning the QR-code (*Rotor Assembly in KRL*).

Through the TAPAS project a Grundfos assembly task has been chosen to demonstrate that it is possible to accomplish an industrial assembly task with the KUKA LWR. The assembly task is the assembly of the SQFlex rotor which is performed manually at the Grundfos factory. The rotor consists of a rotor shaft, a pressure ring, a rotor cap and eight magnets, see figure 4.2.

The rotors are assembled in quantities of approximately 80 per day and the cycle time for one rotor is approximately 30 seconds excluding the operation time of a hydraulic press. A Grundfos description of the assembly task is found on the enclosed Appendix CD </Documents/Instruction- SQFlex production.pdf>. The manual assembly task at Grundfos is conducted in the following way:

1. The pressure ring is placed onto the rotor shaft.
2. The rotor shaft including the pressure ring is centred in the fixture.
3. Eight magnets are placed around the rotor core.
4. The rotor cap is placed on top of the rotor.
5. The rotor is pressed.
6. The assembled rotor is removed.



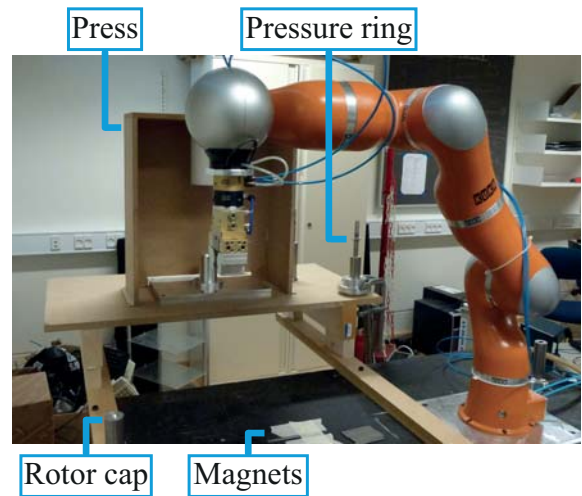
**Figure 4.2:** The components used for the assembly of the SQFlex rotor.

#### 4.2.1 Set-up

At Grundfos the SQFlex rotor assembly is carried out in a narrow environment inside a cabinet containing a hydraulic press, see figure 4.3a. To ensure the feasibility of the assembly task it has been chosen to make a replication of the hydraulic press cabinet environment at AAU, see figure 4.3b. Since the system does not feature a vision system for locating the objects and both magnets and pressure rings are supplied in a disorderly fashion at Grundfos, it is chosen to structure the environment in order to accomplish this assembly task. It has been necessary to feed the magnets and the pressure ring individually at known locations. The replication environment is shown in figure 4.3b. An assembled rotor is used to hold the pressure ring and the magnets are placed manually in a marked area on the table. Furthermore the rotor cap is placed on a peg.



(a) Workspace at Grundfos.

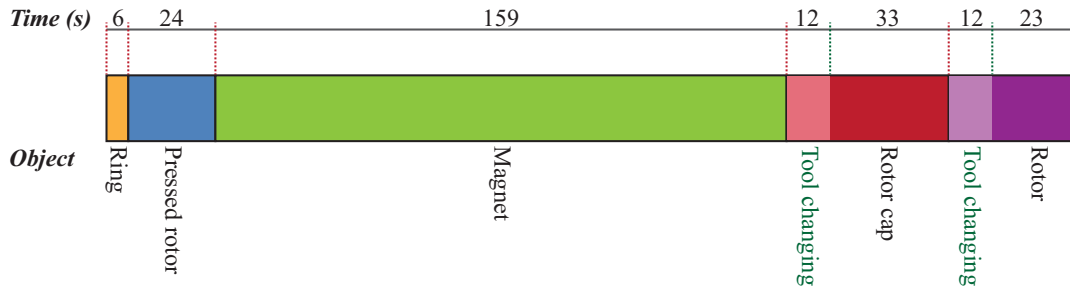


(b) Replication of the workspace at AAU.

**Figure 4.3:** Set-up for the SQFlex rotor assembly.

### 4.2.2 Results

The cycle time for the assembly of one rotor conducted by the KUKA LWR is 4:29 minutes without any actions towards refining or optimizing the programmed task. No robustness test has been carried out since the focus is to document that it is possible to accomplish an industrial assembly task. Figure 4.4 shows the cycle time of the assembly task conducted by the KUKA LWR. The most time consuming part is inserting the magnets.



**Figure 4.4:** Cycle time for the SQFlex assembly performed by the KUKA LWR.

### Errors

During the assembly task different errors occurred. These errors were misplacing the magnets in the fixture, incorrect rotation of the rotor core when inserting the magnets and misplacing the rotor cap on the rotor shaft. These errors occurred primarily as a result of the used pneumatic tools due to their short stroke and uncontrollable grasping force. Furthermore the jaws mounted on the tools did not provide a firm grip on all objects due to wear.

During the assembly task a problem regarding a varying and unstable measurement of the force as a result of movement occurred. To examine this phenomenon several experiments are carried out, see appendix C.1. The experiments document that the force varies as a result of movement of the manipulator. This has an impact on the ability to sense collision with an object given that it is hard to separate data from an actual collision with the noise from the varying force. No action towards minimising the variation of the force has been taken since a hardware problem in joint three could be the cause and the KUKA LWR is scheduled for repair in July 2012.

### Conclusion

From the SQFlex assembly task it has been demonstrated that it is possible to program the KUKA LWR to perform a complicated industrial assembly task. The assembly task has been performed in a replication of the environment from Grundfos where the only changes are structured feeding of the magnets, the pressure ring, and the rotor cap. The cycle time for the task is 4:29 minutes, while the manual cycle time is 30 seconds. It is estimated that better structuring of the objects' locations and the implementation of an electric gripper could reduce the cycle time.

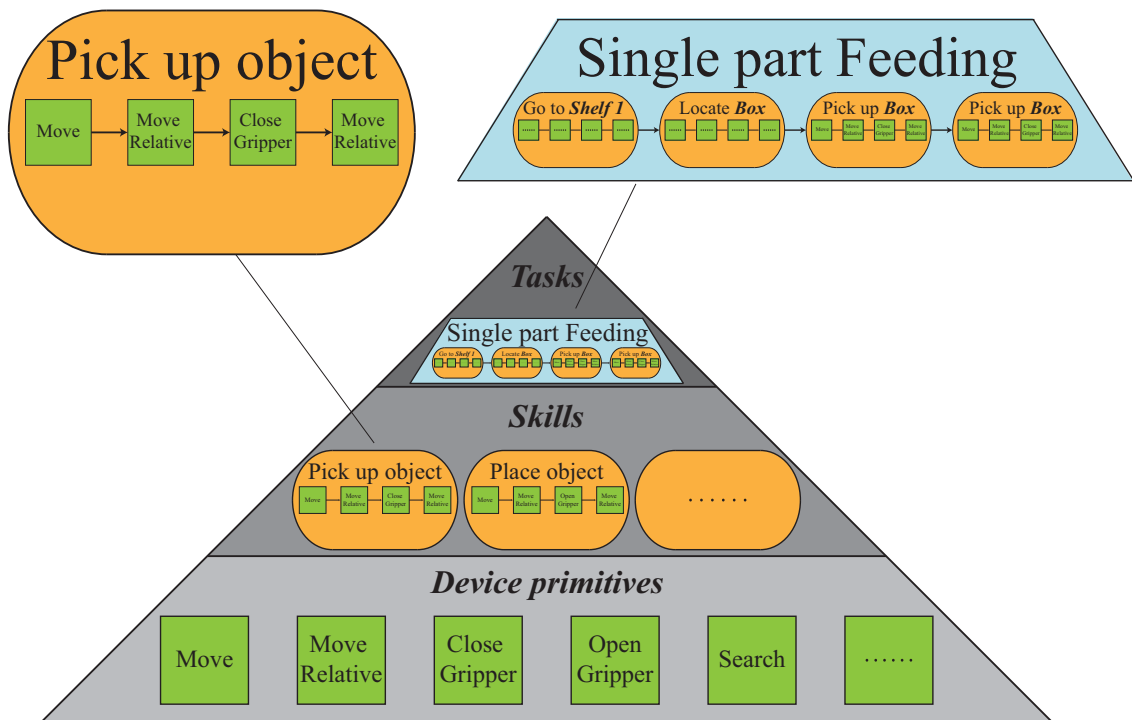
By using the integrated torque sensors to search for contact more widely it is estimated that some of the errors due to inconvenient tools could be eliminated.

# Definition of Skills 5

*This chapter presents a general definition of device primitives, skills and tasks. The purpose is to establish a stringent definition prior to the development of a system based on device primitives, skills and tasks. Besides the definition of device primitives, skills and tasks a practical approach for the implementation of conventional robot macros is documented.*

## 5.1 Definition

The following definition of skills and tasks is established by TAPAS and is based on (S. Bøgh, O. S. Nielsen, M. R. Pedersen, V. Krüger & O. Madsen, 2011). The introduction of an autonomous industrial mobile manipulator (AIMM) and the increased need for flexible production give rise to time-consuming issues regarding programming. In order to ease and expedite the programming, task-level programming is chosen. The approach of task-level programming is divided into three layers consisting of devices primitives, skills, and tasks. Figure 5.1 shows the setting and the interaction between the individual layers. In the following sections tasks, skills, and device primitives are described.



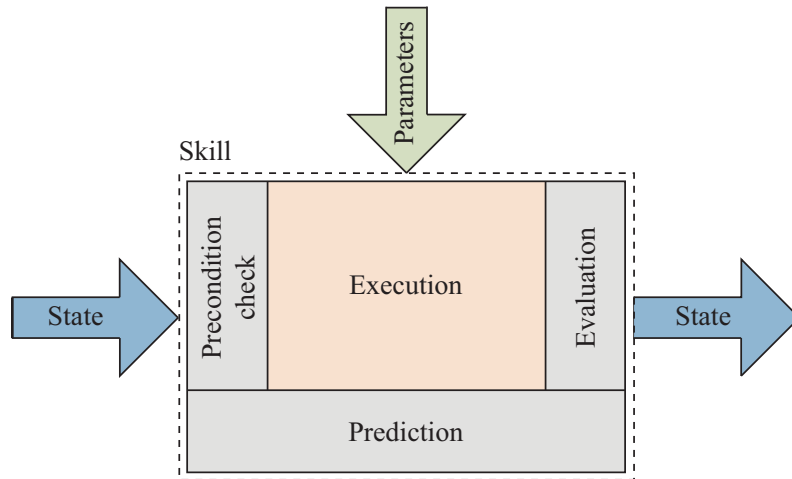
**Figure 5.1:** Relations between tasks, skills, and device primitives.

### 5.1.1 Tasks

A task is described as a sequence of skills and contains an overall goal e.g. *pick up the rotor at station A*. In this way a task is established on the basis of a library of skills. A task is defined by measurable state variables while skills change the condition of the state variables during the sequence. The desired state variables are specified by the user and are measured by vision, tactile, torque, and other sensors. The implication of this is that the user is able to establish a task only by specifying the desired state variable while the skills are used to change the state variable to the desired conditions. By implementing this approach an operator with no experience in robot programming should be able to program a task using a user interface.

### 5.1.2 Skills

Skills represent the foundation of a task and are described as an intelligent sequence of device primitives. Skills are attached to objects while classic robot macros are based on 3D coordinates. To exemplify the differences between classic robot macros and skills a *Pick Up* operation is taken as an example. Using conventional robot macros the object is attached to a prespecified 3D location while skills are applied to the object. This means that the object is located by sensing devices and once the 3D location is found the execution is conducted in a similar way as a classic robot macro for picking up an object. The prerequisite for proper application of a skill is ensured and verified by pre- and postcondition checks. This means that all preconditions must be fulfilled before the execution of a skill while a postcondition check verifies the outcome of a skill. In accordance to the *Pick Up* example the reachability of the object is a precondition while holding the right object in the gripper is a postcondition. Based on the skill definition a general skill model is established in figure 5.2.



**Figure 5.2:** Skill definition. (S. Bøgh, O. S. Nielsen, M. R. Pedersen, V. Krüger & O. Madsen, 2011)

The input values are divided into two groups consisting of state variables and parameters. The precondition check is conducted by examination of the given input values. If the precondition check is performed successfully the execution part is conducted. In other words, the precondition check serves as a safety net. The parameters are a specification of the given assignment containing information such as the location or the object, e.g. *move to station A* or *place Magnet1*. The postcondition check is conducted based on a prediction of the desired goal and an evaluation of the outcome. If the comparison between

the evaluation and the predicted outcome is consistent within a given range, the postcondition check is successful. As outcome the state variables are changed and updated based on the accomplished skill.

### 5.1.3 Device Primitives

Device primitives are described as basic motions and functionalities e.g. *Open the gripper*, *Move to XYZ*, *Search in x-direction* or *Set stiffness*. The motion or functionality of a skill is conducted as a result of the device primitives. A device primitive is declared as a command conducting a motion or functionality while the level below is described as the hardware embedded driver.



LEGO DUPLO Assembly

## 5.2 Implementation of Classic Robot Macros

To obtain competences in using the KUKA KRL and furthermore to understand the advantages of task-level programming it is chosen to implement device primitives and classic robot macros on the KRC. The implementation is based on the LEGO DUPLO Assembly Task, see appendix A.1. A video of the LEGO DUPLO Assembly Task is obtained by scanning the QR-code (*LEGO DUPLO Assembly*). The execution of the task is first conducted by conventional programming in KRL and then converted into device primitives and robot macros. The main issues of the task are to find the location of the brick and furthermore to obtain a correct assembly of the brick onto the plate. In appendix A.1 the task sequence for conventional programming in KRL is formulated in words to provide a simplified overview of the actual KRL code. The actual KRL code requires 246 lines of code to complete the task. From this, conventional programming in KRL is considered time consuming and inefficient for reuse of code. By analysing the task sequence, a number of device primitives and robot macros have been identified. In table 5.1 and 5.2 a description of the identified device primitives and robot macros are shown.

Device primitive	Description
Move XYZ	Moving to a specified coordinate
Move Rel	Moving relative to the current position in a given direction
Search Rel	Searching for collision relative to the current position in a given direction
Close Gripper	Closing the gripper
Open Gripper	Opening the gripper

**Table 5.1:** Description of the identified device primitives.

Robot macro	Description
Pick up	Picking up an object
Place	Placing an object
Find brick	Finding the coordinate of a brick using a searching algorithm
Assembly place	Searching for correct assembly

**Table 5.2:** Description of the identified classic robot macros.

The classic robot macros are parametric functions consisting of a sequence of device primitives. A *Pick Up* macro contains five input variables. A description of the five variables is listed in table 5.3. The execution of a *Pick Up* macro is arranged in four steps and is either executed in #tool or #base frame (x5):

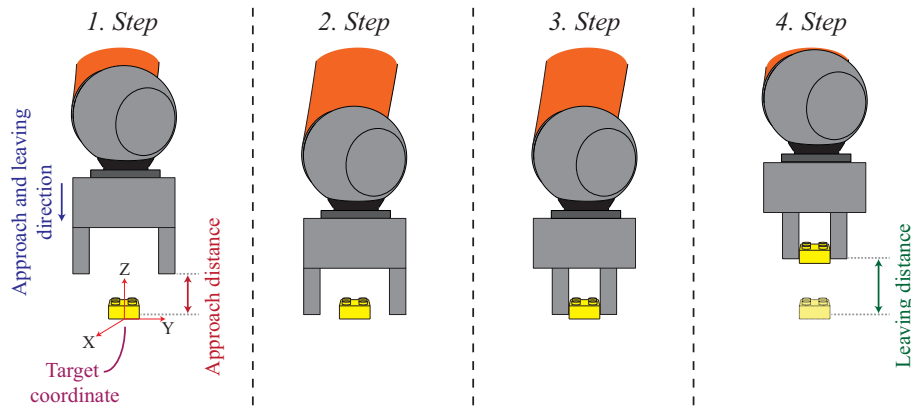


Variable	Type	Input options	Description
x1	frame	coordinate	Target coordinate
x2	char	"x", "y" or "z"	Direction of approaching and leaving the target coordinate
x3	real	number	Distance of approaching the target coordinate
x4	real	number	Distance of leaving the target coordinate
x5	char	"b" or "t"	#base or #tool frame

**Table 5.3:** Description of a Pick up macro.

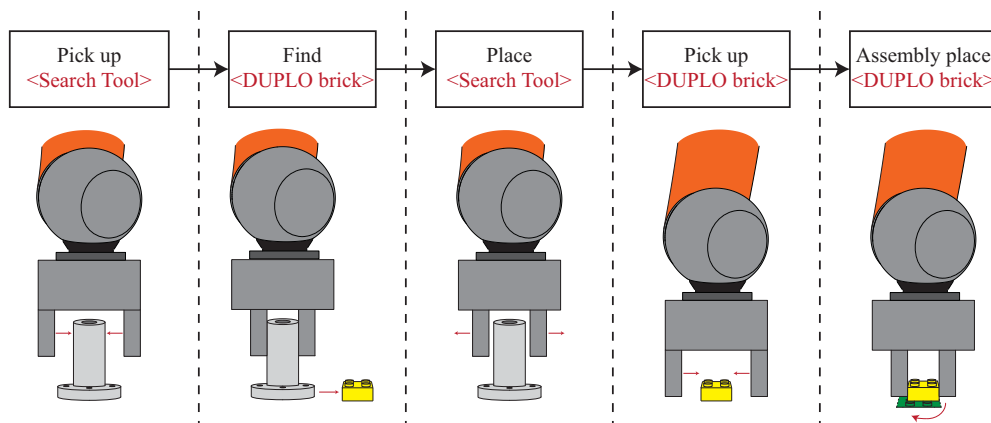
1. Move to a distance (x3) above the target coordinate (x1).
2. Move linear down to the target coordinate (x1).
3. Close the gripper.
4. Move linear to a distance (x4) above the target coordinate.

An illustration of the execution is shown in figure 5.4.



**Figure 5.3:** The execution sequence of a Pick Up macro.

A description of the remaining individual robot macros and device primitives is found on the enclosed Appendix CD in </Documents/Robot Macros.pdf>. As a result of the analysis the task sequence is simplified to contain 5 robot macros. The task sequence is shown in figure 5.4.



**Figure 5.4:** An illustration of each robot macro in the task sequence.



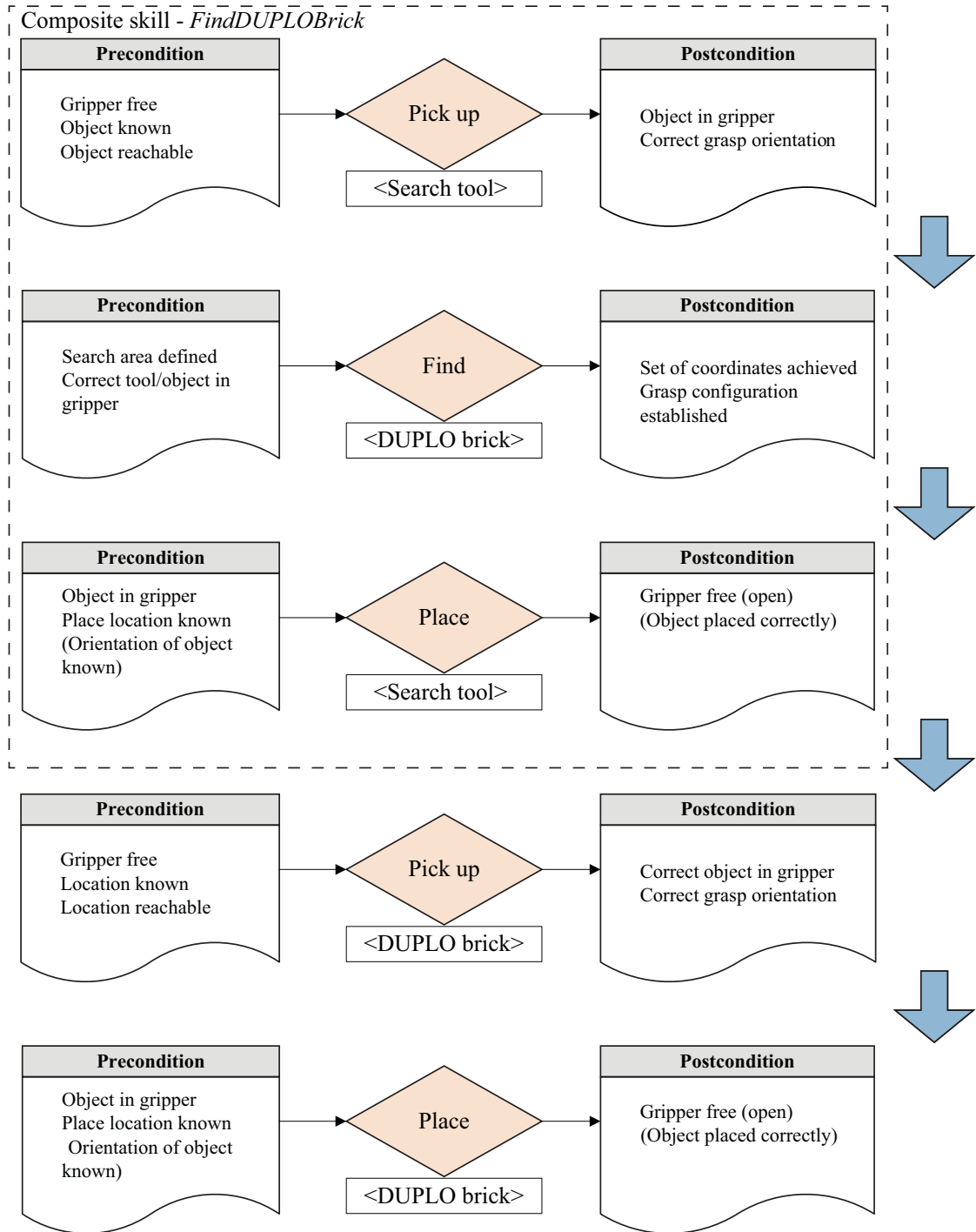
The introduction of robot macros eases and expedites the programming and the actual code in KRL is reduced from 246 lines of code to 10 lines of code of which only 5 lines are the actual robot macro sequence.

### 5.2.1 Reflection on Classic Robot Macros

The identified classic robot macros are unintelligent functions based on 3D coordinates. According to the definition of skills from section 5.1.2 the assumption of an object-oriented function verified by pre- and postcondition check is not met. To reflect on the robot macros pre- and postcondition checks are established for the individual robot macros in the task sequence while it is assumed that they are object-oriented. The established pre- and postcondition checks for the individual robot macros are illustrated in figure 5.5.

It is possible to combine skills, and thus create composite skills, as illustrated in figure 5.5. The skills used to locate the DUPLO brick can be combined to one skill *FindDUPLOBrick*. Furthermore the *Pick* and *Place* skills can be combined to a composite *Pick-and-Place* skill. The combining of skills creates much more specific skills which leaves small programming freedom for the user yet faster programming. In this project it has been chosen not to develop composite skills.

From the definition of skills in section 5.1.2 a skill should be object-oriented, but without a vision camera for object recognition the need for 3D coordinates is inevitable. As this project will not incorporate a vision camera fully object-oriented skills cannot be implemented. Skills in this project will though incorporate pre- and postcondition checks. In chapter 6.5 the implementation of device primitives and skills in the developed system on an external PC is described and in chapter 8 the use of these device primitives and skills is evaluated.

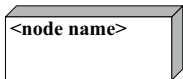
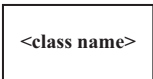
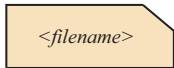
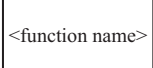


**Figure 5.5:** Pre- and postcondition checks of the individual robot macros for the LEGO DUPLO task. The pre- and postcondition checks are not implemented in the robot macros.

# Little Helper System 6

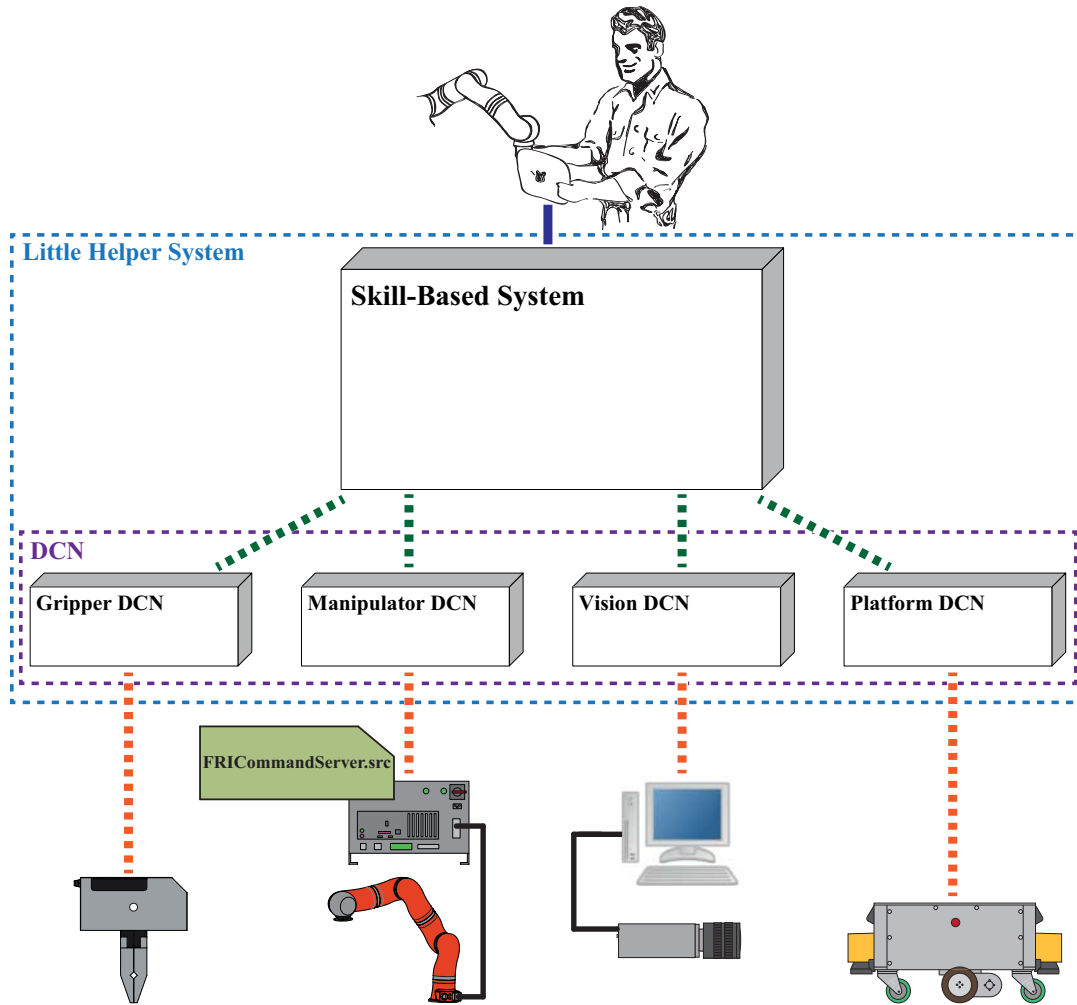
In this chapter the development of a control system called *Little Helper System*. An architecture of this system is presented with focus on the software architecture and implementation. *Little Helper System* consists of four *Device Control Nodes (DCN)* for communicating with the different hardware devices of *Little Helper ++* and a central node containing both skills, device primitives, tasks, and user interfaces. The purpose is to document the architecture of the entire system and furthermore to document the software structure of skills, device primitives, and tasks.

Through this chapter several elements recur in the figures. These elements are defined in figure 6.1.

Data objects		Interaction
 <b>Node</b>	 <b>Class</b>	■ ■ ■ ■ ■ <b>ROS Action communication</b>
 <b>File</b>	 <b>Function</b>	■ ■ ■ ■ ■ <b>UDP communication</b>
		■ ■ ■ ■ ■ <b>C++ include</b>

**Figure 6.1:** Definition of elements that recur in the figures of this chapter.

*Little Helper System* is defined as the developed software system running on the embedded laptop in *Little Helper ++*. The system consists of four *Device Control Nodes (DCN)* for communication with the devices outside the embedded laptop and the central node called *Skill-Based System*. Each of the DCNs and *Skill-Based System* are run separately, hence as separate executable programs. Communication between the DCNs and *Skill-Based System* is conducted using the Robot Operating System (ROS) architecture. ROS is chosen as it offers a tool for controlling and standardising communication between nodes (sub-programs) and furthermore ROS offers online sharing of developed software. An introduction to ROS is found in appendix B. Figure 6.2 illustrates the general architecture of *Little Helper System*. All the DCNs and *Skill-Based System* have been developed exclusively during this master project, except for the *Manipulator DCN*, which has been developed in collaboration with (Nielsen, 2011). To document the developed software the architecture of each of the DCNs and *Skill-Based System* are described in section 6.1 to 6.5. This is done both to document the structure of each of the nodes, but also to document the communication from the DCNs to the hardware. *Skill-Based System* contains several classes including device primitives and skills, see section 6.5.



**Figure 6.2:** General view of the architecture of Little Helper System. The figure illustrates the four separate DCN nodes and the Skill-Based System node. Furthermore the figure shows the communication method between the nodes. The source code and architecture of each of the five nodes are described in this chapter.

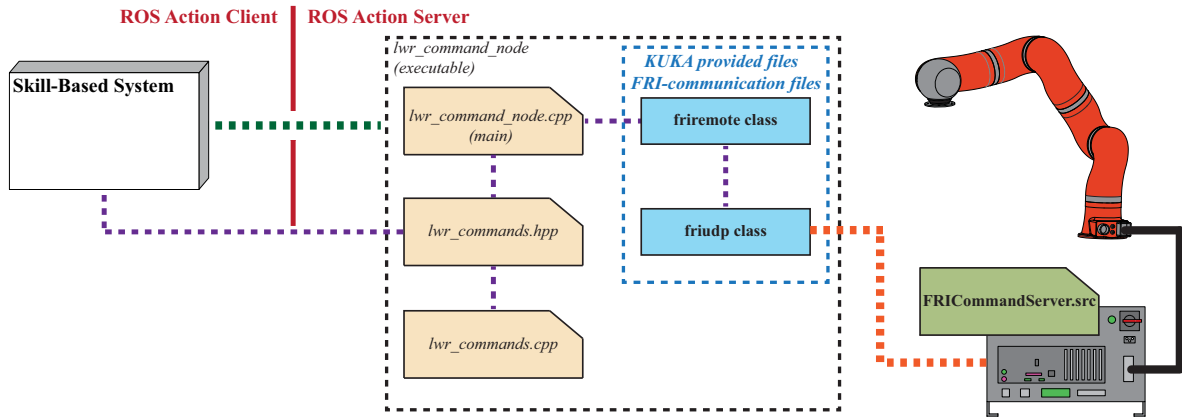
## 6.1 Manipulator DCN

The communication between the KUKA LWR and an external computer is possible via the Fast Research Interface (FRI). The FRI-protocol is incorporated in the KRC and the basic FRI functionalities are controlled by the teach pendant. Along with the LWR, KUKA provides enclosed C++ classes to handle the basic communication to the KRC via FRI from an external computer. These classes make it possible to exchange data with the controller and to access the functionalities of the robot. The development of the Manipulator DCN takes its starting point in the software architecture ROS.

The name of the developed ROS Package is *fri\_communication* and it is developed in collaboration with (Nielsen, 2011). The executable ROS Node for controlling the KUKA LWR within the package is *lwr\_command\_node* and the code is written in the C++. The ROS Package *fri\_communication* is found on the enclosed Appendix CD in `</Software/ROS/fri_communication>`. The architecture and the principle of *fri\_communication* are described, but the actual code will not be documented within this report. For information about the code, please see the documentation within the source files found on the enclosed Appendix CD. (KUKA System Software, 2011) and (ROS Wiki, 2012) have been used in the development of the *fri\_communication* package.

### 6.1.1 Architecture of the `fri_communication` ROS Package

Figure 6.3 shows a visual representation of the architecture of the `fri_communication` ROS Package.



**Figure 6.3:** The architecture of the `fri_communication` ROS Package.

The ROS Package `fri_communication` is used to control the KUKA LWR, thus on one side it must communicate with the KRC via the FRI-protocol, and on the other side it must communicate with *Skill-Based System* via a ROS Action interface. In order to follow this architecture the `lwr_command_node` is developed as a ROS Action Server, and at the same time contains the FRI-communication files to communicate with the manipulator. *Skill-Based System* implements a ROS Action Client which makes requests to the ROS Action Server. The FRI-protocol enables the exchange of 16 integer-, 16 floating-, and 16 boolean values every 4 millisecond. According to this approach a case-based program is developed on the KRC, see `FRICommandServer.src` below. A short description of the objects from figure 6.3 follows.

#### `lwr_command_node.cpp`

`lwr_command_node.cpp` is the main file of the `lwr_command_node` executable. It consists of the code needed for the ROS Action Server set-up and control. Upon receiving a ROS Action Goal from the ROS Action Client it calls the requested function from the `lwr_commands.cpp` and the `friremote` class. `lwr_command_node.cpp` contains six different ROS Action Servers respectively a server for controlling movement, stiffness, tool, base, interrupt, and force applications of the manipulator. These servers define the most essential device primitive groups on the manipulator. Communication to these six servers are conducted from the `MotionPrimitives` class implemented in the *Skill-Based System* node, see section 6.5.1. Thus the `MotionPrimitives` class is used as a wrapper class for controlling the Manipulator DCN. The six device primitive groups, and thus the six functionality groups of the manipulator, are listed below:

- Move
- Stiffness
- Base
- Tool
- Force applications
- Interrupt

### **lwr\_commands**

*lwr\_commands.cpp* contains functions controlling the assignment of the 16 integer-, 16 boolean-, and 16 floating values corresponding to the case structure of the *FRICCommandServer.src*. *lwr\_commands.hpp* is a shared file between the ROS Action Server and the ROS Action Client. This file contains enumerators defining data being transferred between the ROS Action Server and the ROS Action Client.

### **firemote class and friudp class**

The *firemote* class contains functions by which it is possible to access the functionalities of the manipulator without using *FRICCommandServer.src* and functions for constructing the desired package. The UDP communication set-up and functions for sending, receiving, and decoding UDP packages to and from the KRC are implemented in the *friudp* class.

### **FRICCommandServer.src**

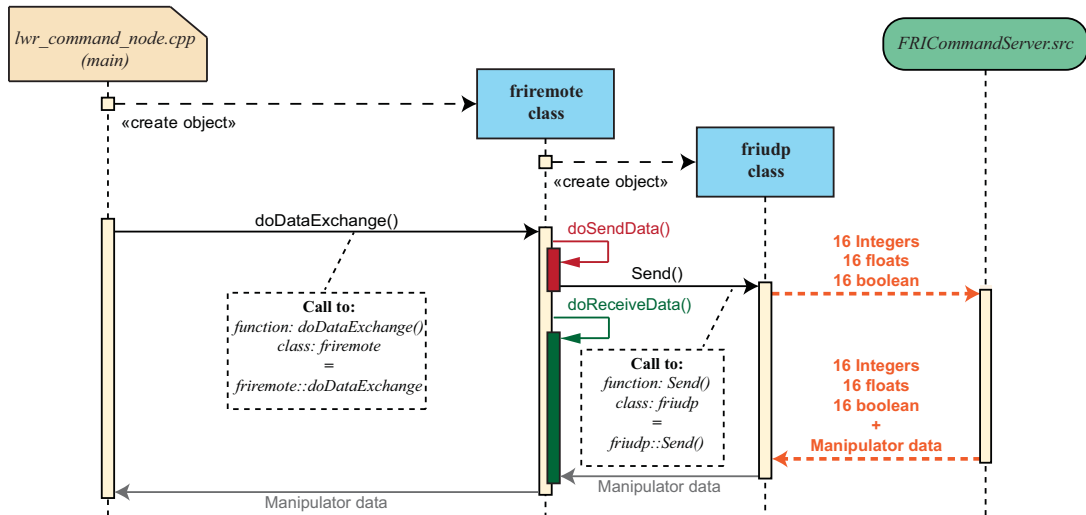
*FRICCommandServer.src* is a case-based program written in KRL-language. The exchanging of the 16 integer-, 16 boolean-, and 16 floating values controls the action on the KUKA LWR by switching between the different cases of the case structure in the program. A sequence number keeps track of new actions and a state value defining the outcome of the given assignment is returned. *FRICCommandServer.src* consists of a loop with no end condition in which an overall case structure separates the individual actions. To obtain the opportunity to interrupt a manipulator action an Interrupt function is specified, see appendix C.2. The overall case structure separates the six different manipulator actions which corresponds to the same division of the ROS Action Servers in *lwr\_command\_node.cpp*. In this way a movement action is indicated by a commandID corresponding to the proper case in the overall case structure followed by a declaration of a motion type, a 3D location, and a velocity. For further information please see the documentation within the source files found on the enclosed Appendix CD in </Software/KRC/FRICCommandServer.src>.

#### **6.1.2 System sequence**

In addition to the architecture description, system sequence diagrams have been created. The purpose is to document how the communication with the KRC is conducted and to illustrate how a command is passed through the code and thus document the dynamic correlation between the different objects of the ROS Package. The system sequence diagrams have been formulated in the Unified Modelling Language (UML) standard. On all sequence diagrams the parameters passed along with the function calls are omitted to decrease complexity.

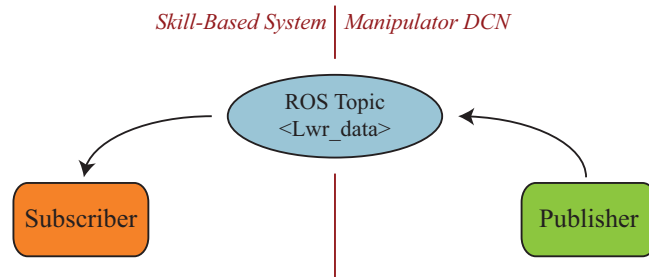
Figure 6.4 shows the data exchange between the KUKA LWR and *lwr\_command\_node.cpp* and the creation of two classes.

The data exchange is conducted every four millisecond and within each data exchange the buffer containing the 16 floating-, 16 boolean-, and 16 integer values is transferred to the KRC. This means that if no new action is requested from the ROS Action Client, hence the data in the buffer has not been changed, the same data is transmitted from *lwr\_command\_node.cpp* to the KRC. As long as the sequence number in *FRICCommandServer.src* remains unchanged the KRC makes no further actions. *lwr\_command\_node.cpp* and *Skill-Based System* contain a ROS Publisher and a ROS Subscriber re-



**Figure 6.4:** The system sequence diagram shows the creation of two classes and the data exchange between the KUKA LWR and *lwr\_command\_node.cpp*.

spectively, see appendix B. In this way the received manipulator data is published to a given ROS Topic and can be accessed from *Skill-Based System*. A graphical approach of the data exchange from *lwr\_command\_node.cpp* to *Skill-Based System* via a ROS Subscriber and a ROS Publisher is shown in figure 6.5. A description of ROS Topic, Subscriber, and Publisher is found in appendix B.



**Figure 6.5:** The interaction between a ROS Subscriber and a ROS Publisher.

Figure 6.6 shows how the command for moving the manipulator is passed through the system. *lwr\_command\_node.cpp* receives a ROS Action Goal containing a number of parameters such as 3D location, motion type, velocity etc. *lwr\_command\_node* calls the *move()* function in *lwr\_commands.cpp* as a result of the received ROS Action Goal. In this function the parameters from the ROS Action Goal are assigned to a memory buffer located in the *friremote* class. This is illustrated using a pink colour in figure 6.6. The data of this buffer is transmitted to the KRC through the UDP connection at the next data exchange.

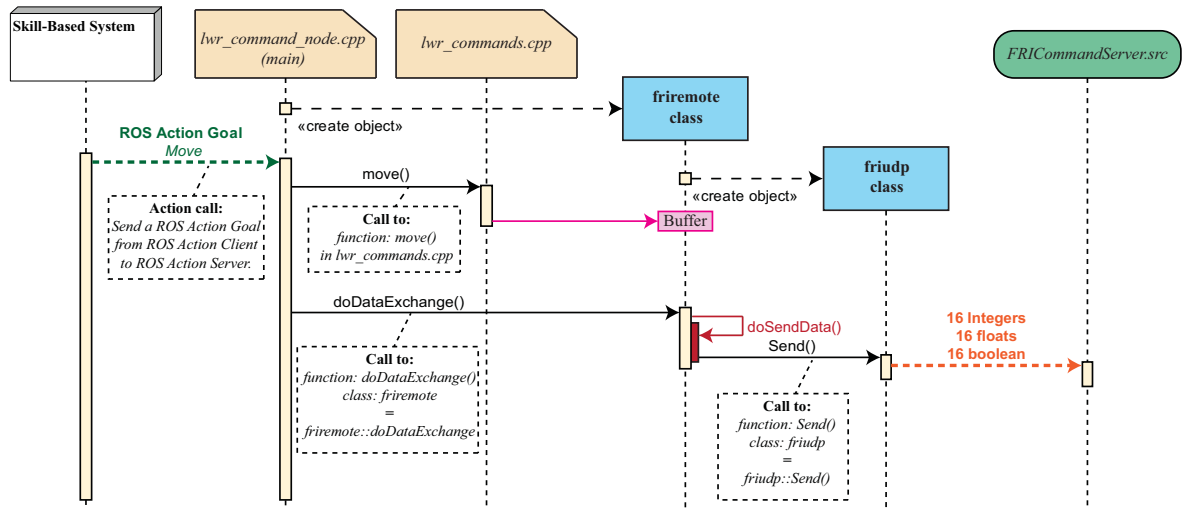


Figure 6.6: The system sequence shows how the command for moving the manipulator is passed through the system.

## 6.2 Gripper DCN

In section 3.3 the hardware implementation of a Schunk WSG50 electric gripper is described. A Device Control Node to handle the communication to the gripper has been developed and is thoroughly described in appendix E. The developed DCN is based on the same fundamental architecture as the Manipulator DCN and thus uses a ROS Action communication towards *Skill-Based System* and an UDP communication towards the gripper.

The Gripper DCN contains a wrapper class to handle the ROS Action communication from the client, and consequently this class is intended for inclusion in the *Skill-Based System* node. The most essential device primitives of the gripper are:

- Home
- Move
- Grasp
- Release
- Get grasp state
- Get width

A full list of the device primitives of the gripper is found in appendix F.

## 6.3 Platform DCN

In chapter 3 the hardware implementation of the Neobotix MP-L655 platform is described. The platform is controlled by an incorporated computer and the navigation system is configured by enclosed software. The software enables configuration of maps, workstations, and trajectory planning. As the platform has to be integrated with *Little Helper System* a platform DCN must be developed according to the architecture illustrated in figure 6.2.

This development has not been initiated in this project although the establishment of a general architecture is presented in appendix E.



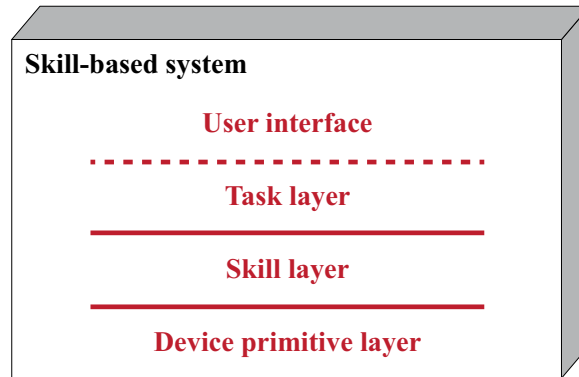
## 6.4 Vision DCN

The motivation for implementing a vision system and thus developing a Vision DCN is in consequence of Workstation 3 described in chapter 8. The Vision DCN is currently under development, and consequently it has not been implemented yet. Workstation 3 in chapter 8 has been used as a basis of the work, yet the vision system will accommodate other objects. The vision system is supplied by Grundfos and consists of a vision camera, a windows based PC and a vision software developed by (Bigum, 2012).

The purpose of the Vision DCN is to handle the communication between the Grundfos vision system and *Skill-Based System*. The communication between the systems requires the establishment of a communication protocol which has been done as a collaboration between the project group and the vision department at Grundfos, (Bigum, 2012). The development of the communication protocol to the vision system is a work in progress, and consequently only the fundamentals of the communication protocol are established. A description of the architecture of the Vision DCN and the established protocol are found in appendix E.

## 6.5 Skill-Based System

*Skill-Based System* is the central control system for operating Little Helper ++. On one side it has the ROS Action interface towards the DCNs and on the other side it has the user interface for programming and operating Little Helper ++. Between these interfaces tasks, skills, and device primitives are implemented. Figure 6.7 visualizes the content of the *Skill-Based System* node divided into four different layers.



**Figure 6.7:** The content of the *Skill-Based System* node visualized in four layers.

The division into these layers is done in correspondence to figure 5.1. In the *Skill-Based System* node the user interface defines the main routine and thus implements the layers beneath it. As *Skill-Based System* is aimed at operating on task level, the task layer is integrated in the user interface, see to figure 6.7. Two different user interfaces have been developed and even though these interfaces are part of the *Skill-Based System* node it is chosen to document the interfaces in separate sections, see section 6.6 and 6.7. This is done as the interfaces are quite complex and they are described in a less code-wise approach.

In appendix F the developed skills and device primitives are described.

### 6.5.1 Device Primitive Layer

The device primitive layer is the bottom layer in the *Skill-Based System* node as it handles the communication to the DCNs. It currently consists of two classes, *MotionPrimitives* and *WsgClient*, as the Vision DCN and the Platform DCN are currently under development. When the DCNs to the vision system and the platform are developed, classes to handle the communication to these DCNs will also be included in the device primitive layer. The *MotionPrimitives* class and the *WsgClient* class both implement a ROS Action Client for communicating with the DCNs via ROS Action protocols. The *WsgClient* class contains device primitives of the gripper, hence functions defining simple motions and functionalities of the gripper, and the *MotionPrimitives* class contains device primitives of the manipulator, hence functions defining simple motions and functionalities of the manipulator. The rest of this section will focus on the *MotionPrimitives* class as the *WsgClient* class is described in section 6.2.

The implementation of the device primitive layer is included in figure 6.8.

The functions in the *MotionPrimitives* class are generalized and thus defined by the input parameters. The device primitive layer does not incorporate any checks or supervision of the feasibility of the motion. Thus supervision of the environment and the system itself is carried out in the skill and task layer.

An example of a function in the *MotionPrimitives* class defining a simple motion of the manipulator is *MotionPrimitives::MoveCart(...)*<sup>1</sup>. The *MoveCart(...)* function moves the manipulator's TCP to a cartesian coordinate in the workspace. The motion is defined by a cartesian coordinate (including orientation), a velocity, and a motion type, by which the last mentioned parameter defines whether to move linear or point-to-point to the coordinate.

Another example of a function in the *MotionPrimitives* class is *MotionPrimitives::SearchRel(...)*, which searches a relative distance from the current position of the TCP in a given direction. This device primitive creates the foundation for all skills dependent on contact. Contact is found along the path if the force exceeds a limit defined by an input parameter. If contact is found *MotionPrimitives::SearchRel(...)* calls the function *MotionPrimitives::Interrupt(...)*, which stops the manipulator in its current motion.

An example of a function in the *MotionPrimitives* class defining a functionality is *MotionPrimitives::SetBase(...)*. This function does not initiate any motion of the manipulator but defines a new workspace and is typically used when operating in different workspaces, hence working at different workstations. The only input for the function is the cartesian coordinate of the new reference coordinate system.

The *MotionPrimitives* class implements a ROS Subscriber to the manipulator data published by the Manipulator DCN. The sharing of the manipulator data via a ROS Topic is illustrated in figure 6.5. The manipulator data received by the ROS Subscriber in the *MotionPrimitives* class is placed in public variables to make them available to both the device primitives and to the rest of *Skill-Based System*.

Both the *MotionPrimitives* class and the *WsgClient* class implement a *singleton* pattern. The *singleton* pattern is a way of structuring the class so that no more than one instance of the class can exist at a time. This is used as several classes from the skill layer require instances of the *MotionPrimitives* class and the

---

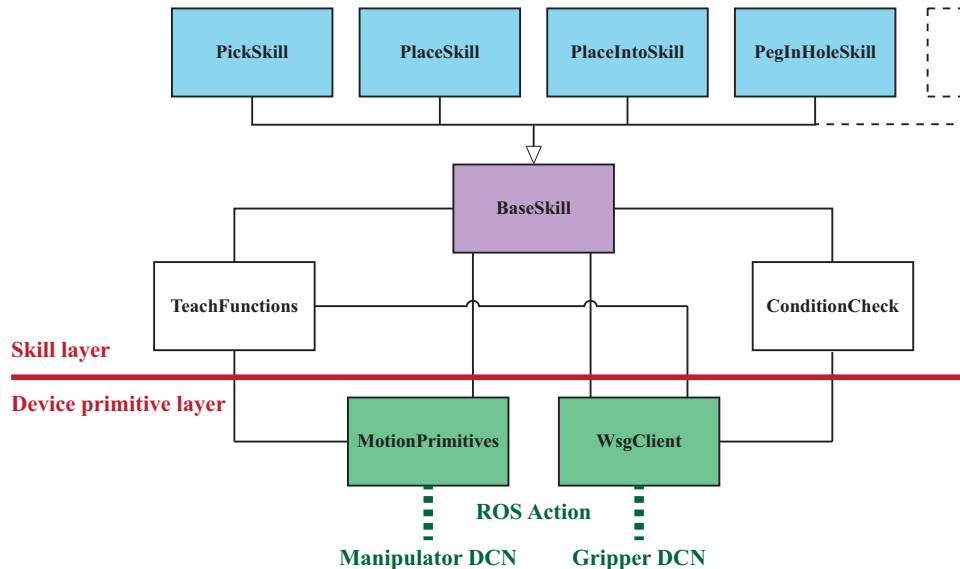
<sup>1</sup>The input parameters have been omitted from the function name to decrease complexity.

*WsgClient* class, see figure 6.8. Whenever an object of the class is needed a special function in the class is called, which returns a pointer to the current instance of the class. If no instance has been created, it creates a new instance. The restriction of only one instance of the class at a time is beneficial as only one object communicating with the DCNs is desired. Furthermore this makes the single instance global and thus data of the instance is shared.

For further documentation of the content of the *MotionPrimitives* class, please refer to the *MotionPrimitives.hpp* file on the enclosed Appendix CD in `</Software/Skill_Based_System/src/>`.

### 6.5.2 Skill Layer

The skill layer is the central part of the skill-based approach. Figure 6.8 is a class diagram showing the relations between the classes in the skill layer. The classes of the device primitive layer are also included in figure 6.8 (marked in green) as these classes are essential in the class relation of *Skill-Based System*.



**Figure 6.8:** Class diagram of the classes in the skill layer and device primitive layer in Skill-Based System.

The skills are implemented as separate classes (marked in blue) all inheriting from a parent class (marked in purple). The skill layer also consists of a class handling the pre- and postcondition checks and a class containing functions used in the teaching phase. These different objects in the skill layer are described below.

#### Skills

The skills shown in figure 6.8 are only a selection of the skills within the system. As shown in figure 6.8 the different skills are implemented in separate classes all inheriting from the *BaseSkill* class. Chapter 7 describes the developed skills.

The *BaseSkill* class is used as the basis for all the individual skills, hence it is a parent to all the skills. It links the individual skills to the classes below, by creating an object of *ConditionCheck* and *Teach-*

*Functions* and gets the instance<sup>2</sup> of the *MotionPrimitives* class and the *WsgClient* class. Thus whenever an object of a skill is created this object has access to the functions of the classes below. The *BaseSkill* class contains functions that are used in many of the skills. An example of such a function is the *BaseSkill::CalcApproachLeaving* which calculates the approach and leaving coordinate of a skill.

The individual skills follow the same formalism and they all contain a function called *<Skill>::Teach(...)* and a function called *<Skill>::Execute(...)*. *<Skill>::Teach(...)* defines the teaching routine of the skill and *<Skill>::Execute(...)* defines the execution routine of the skill. By using the same structure in all skills, the development of a new skill is carried out systematic and easily. Furthermore by having the skills in separate classes the development of a new skill does not interfere with the already implemented and operating skills. This also makes it easy to separate the skills and to create individual selections of skills for a specific robot or a specific operating environment.

### ConditionCheck

In reference to figure 5.1, what distinguishes skills from classic robot macros is among other things the precondition check and postcondition evaluation. These checks have been implemented as a separate class, *ConditionCheck*, containing a function for each possible condition check.

When executing a skill, hence calling the *<Skill>::Execute(...)* function, pre- and postcondition checks are performed. This is done by calling functions in the *ConditionCheck* class from the skill. If a precondition check returns false, the skill is skipped.

From figure 6.8 please note that the *ConditionCheck* class only is connected to the *WsgClient* class from the device primitive layer. This is due to the fact that the only condition checks currently implemented are based on information from the gripper. A future development could be condition checks based on direct sensing of the environment through a vision system embedded on Little Helper ++.

The condition checks based on the gripper are to check whether the gripper is empty or currently grasping, and to check the width of the grasped object. The obtained width is compared to the specified width and tolerance of the expected object. Though it is not possible to verify the object itself based on the width of an object and the tolerance, it is possible to verify whether the grasped object is within the tolerance of the expected width.

### TeachFunctions

During the teaching phase the user interacts with the manipulator by e.g. applying a force to the TCP in a given direction or storing a coordinate by holding the TCP steady. The functions for handling the user interaction during the teaching phase are placed in the *TeachFunctions* class. Many of these functions require the command of both the manipulator and the gripper and consequently the *TeachFunctions* class uses an instance of both the *WsgClient* class and the *MotionPrimitives* class.

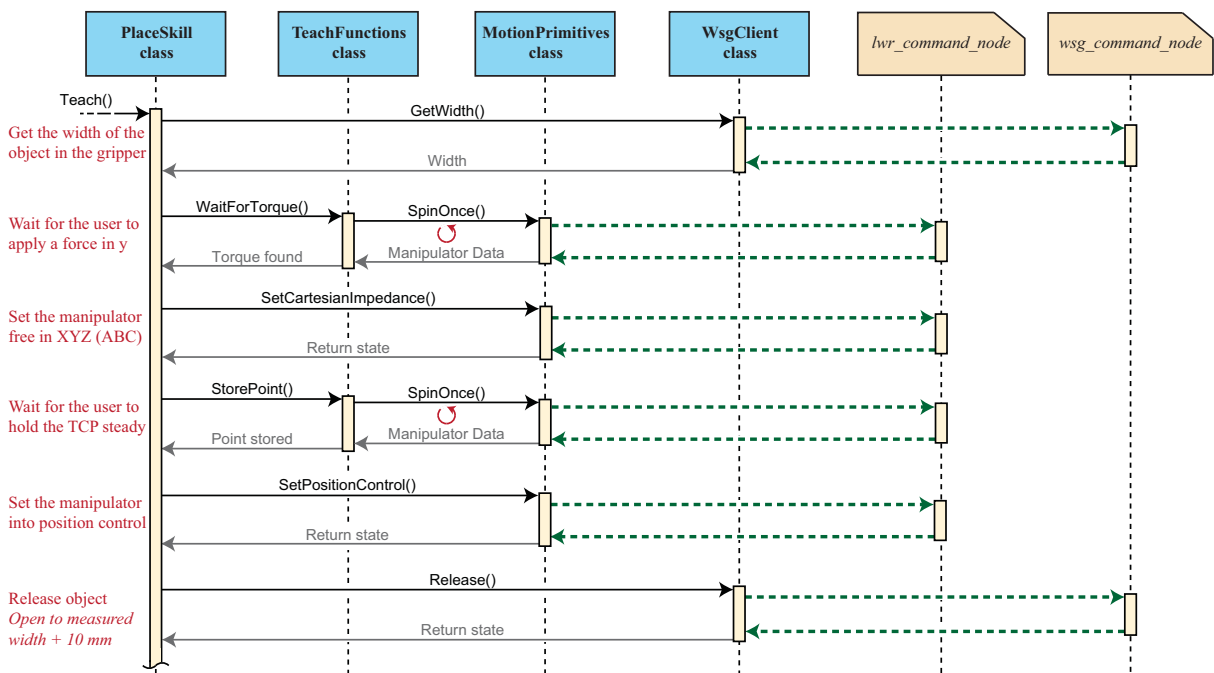
---

<sup>2</sup>*MotionPrimitives* and *WsgClient* have been implemented with a *singleton* pattern and thus a new object cannot be created. Instead the single instance is obtained.

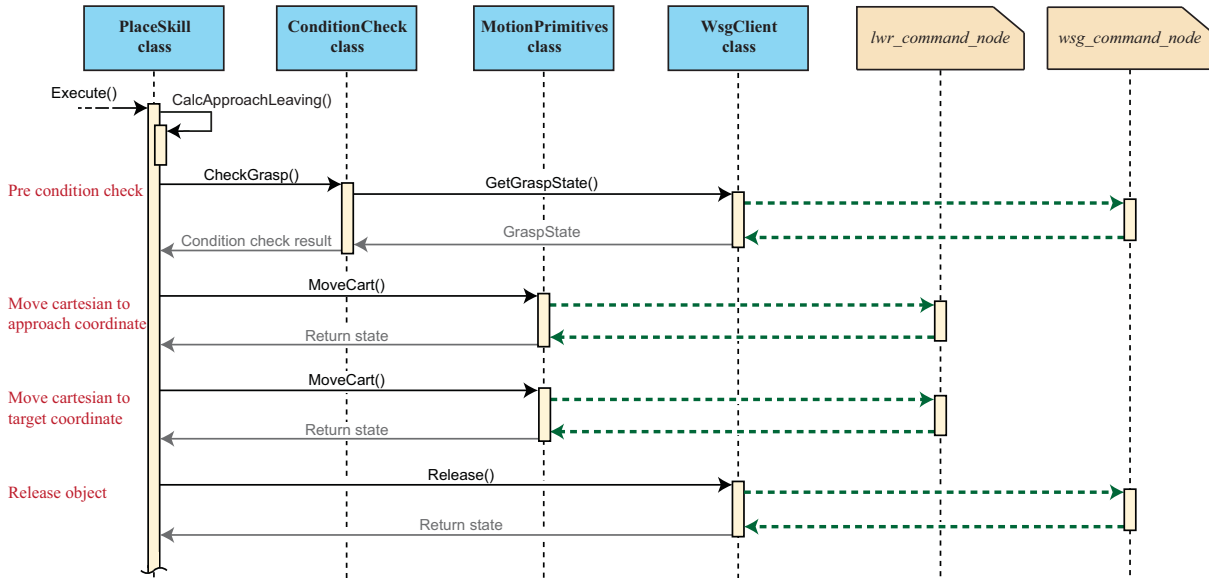
## System Sequence

The class diagram in figure 6.8 shows the static relation of the skill layer and device primitive layer. To further elaborate the relation of the different classes and the objects of these, two system sequence diagrams are included, see figure 6.9 and 6.10. These system sequence diagrams illustrate the dynamic behaviour of the classes in the skill layer and the device primitive layer.

Figure 6.9 shows a system sequence diagram based on the teaching of a *Place* skill and figure 6.10 shows a system sequence diagram based on the execution of a *Place* skill. In both diagrams the parameters passed along with the function calls are omitted to decrease complexity. The ROS Action communication to the DCNs is illustrated by the dashed green lines, but the data passed to and from the DCNs is not visualised. Neither of the diagrams illustrate the creation of objects and the obtaining of instances and only part of the *PlaceSkill::Teach(...)* and *PlaceSkill::Execute(...)* respectively are illustrated.



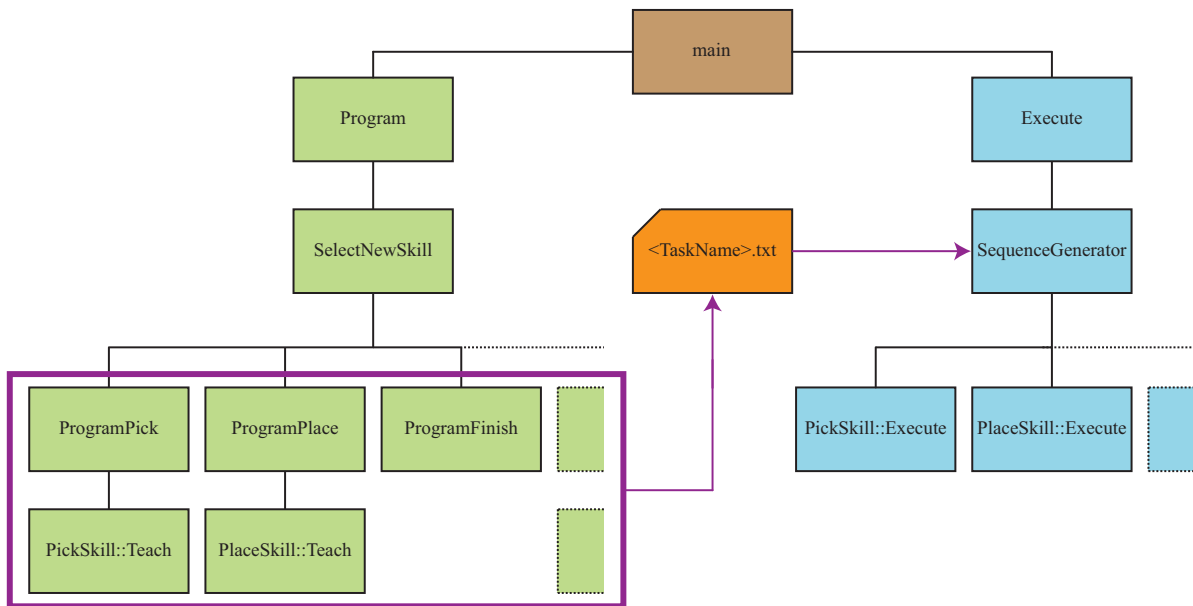
**Figure 6.9:** System sequence diagram illustrating internal function calls in Skill-Based System during part of the *PlaceSkill::Teach(...)* function. The purpose is to illustrate the internal relation of the classes in the skill layer and device primitive layer. The red cycle symbol illustrates the repetition of this communication until a certain condition is fulfilled. The *WaitForTorque()* function will keep requesting the manipulator data until a given force is measured in the y-direction and the *StorePoint()* function will keep requesting the manipulator data until the TCP is held relatively steady. The *SpinOnce()* function simply retrieves the manipulator data from a ROS Topic.



**Figure 6.10:** System sequence diagram illustrating internal function calls in Skill-Based System during part of the *PlaceSkill::Execute(...)* function. The purpose is to document the internal relation of the classes in the skill layer and device primitive layer.

## 6.6 Terminal User Interface - TUI

The Terminal User Interface (TUI) is a sequential terminal based system in which the task layer is integrated, confer figure 6.7. In chapter 5 the task layer is defined as a sequence of skills selected on the basis of a library of skills. The sequence of skills is established by a programming phase where the skills are sequentially programmed one by one. During the execution phase the established skill sequence is executed. A graphical representation of the architecture of TUI is shown in figure 6.11.



**Figure 6.11:** The function architecture of TUI. TUI is divided into an execution and a programming part connected by a text file.

TUI consists of two parts, an execution and a programming part. The interaction between the two parts

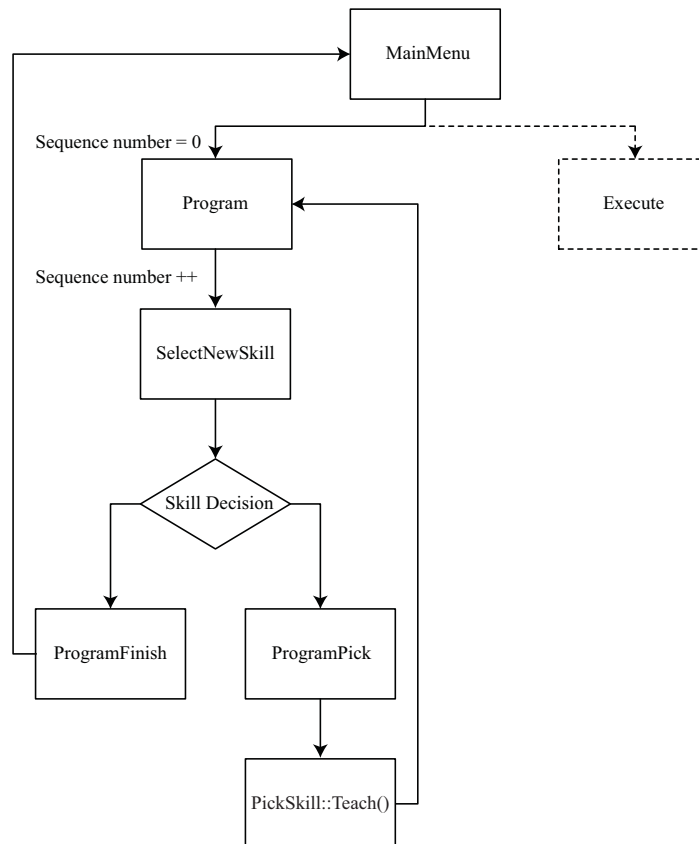
is a text file consisting of the skill sequence and parameters from the programming part. In figure 6.11 the programming part is indicated with a green colour, the execution part is indicated with a blue colour, while the text file is indicated by an orange colour. The name of the text file is established by the user during the programming of a new task.



Pick n Place

### 6.6.1 Program

The programming part consists of two sub parts. A specifying part in which the user enters a number of parameters in TUI and a teaching part in which a number of parameters are established as a result of the physical interaction with the manipulator. A video of a teaching routine is obtained by scanning the QR-code (*Pick n Place*). In this way a *Pick* skill is established by user input in the function *ProgramPick* followed by a teaching routine in *PickSkill::Teach(...)*. All parameters are attributed to the text file which is symbolised by a purple box in figure 6.11. The generation of a skill sequence is illustrated by a flow chart in figure 6.12. To decrease the complexity of the flow chart only functions corresponding to the *Pick* skill are included. The *Program* function controls the sequence number and the overall skill sequence while *SelectNewSkill* establishes the function call to the corresponding skill as a result of the given user input. The skill sequence is increased as long as the user input differs from *ProgramFinish*. This approach is exemplified in the flow chart, see figure 6.12.



**Figure 6.12:** The flowchart shows an excerpt from the generation of a skill sequence. To decrease the complexity of the flow chart only functions corresponding to the *Pick* skill are included.

### 6.6.2 Execute

The execution of a skill sequence is performed by loading and interpreting the text file in the function *SequenceGenerator*. Figure 6.13 shows a notional skill sequence written in the text file. The skill sequence consists of a *MoveTo* skill, a *Pick* skill, and an end condition. In order to separate the individual skills each line is commenced with a number specifying the sequence number.

<TaskName>.txt

1#SkillType 7 MoveTo	MoveTo Skill
1#Velocity 1.0	Parameter for MoveToSkill::Execute()
1#FrameType c	
1#MotionType p	
1#MoveFrameJoint1	47 26 -74 -101 49 104 3
1#MoveFrameCartesian1	118 -519 568 24 -89 -23
1#MoveFrameJoint2	0 0 0 0 0 0 0
1#MoveFrameCartesian2	0 0 0 0 0 0
2#SkillType 2 Pick	Pick Skill
2#FrameType t	Parameter for PickSkill::Execute()
2#Velocity 0.7	
2#Stiffness 0	
2#ObjectName Magnet1	
2#ObjectTolerance 2	
2#GraspForce 70	
2#ApproachDirection z	
2#ObjectWidth 2	
2#MoveFrame 2	-339 554 -90 1 70
2#ApproachDistance 56	
2#LeavingDirection y	
2#LeavingDistance -10	
3#SkillType 0 Finish	End of sequence

**Figure 6.13:** The text file shows a notional skill sequence consisting of a *MoveTo* skill, a *Pick* skill and an end condition. Parameters marked by a purple text colour are user input, parameters marked by a green text colour are accessed from other text files, see section 6.6.3, while parameters marked by a black text colour are established as a result of the teaching routine of the chosen skill.

A skill is described by an initial skill type number, marked by a red text colour, followed by a number of parameters. The skill type number is used to indicate the loading and interpretation process in the case structured function *SequenceGenerator* while the parameters are used for the execution of the relevant skill. The interpretation of the skill parameters is described in appendix F.

### 6.6.3 Object-oriented skills

According to the definition of skills from chapter 5 it follows that a skill is applied to objects. The vision of object-orientation is implemented by supplying information about the object to the skill. This is applied for all pick and place operations. The implementation is conducted by the establishment of two text files, *ObjectTypes.txt* and *ReferencedObjects.txt*.

#### ObjectTypes.txt

*ObjectTypes.txt* contains general information about the different object types, which are listed below.

- Object width
- Tolerance on object width
- Grasp force



An object type defines a set of general parameters for an object, but not specific parameters bound to a specific instance of the given object. For example rotor core is an object type and thus the object type defines information about the width, tolerance, and grasping force of a rotor core. Information about a specific instance of the rotor core, hence locational information, is handled by the *ReferencedObjects.txt*. The object type information is used for pre- and postcondition checks and in the programming and execution phase of the individual skill. An additional parameter indicating the type of object is added to the configuration of the *Pick* skill and the *PickFromStack* skill, confer appendix F. When programming a *Pick* skill, unless a new object type is selected, grasping force, tolerance, and object width are used from the *ObjectTypes.txt*. When selecting a new object type the tolerance and grasp force is specified while the object width is measured during the teaching phase. These information are stored in *ObjectTypes.txt*.

### ReferencedObjects.txt

*ReferencedObjects.txt* contains information about a specific object, e.g. *RotorCore1*. These information are listed below.

- 3D location
- Approach distance
- Approach direction
- Leaving distance
- Leaving direction

When programming a *Pick* skill an object name based on the object type is ascribed to a specific object. When the object is placed the above listed information is furthermore ascribed to the object by the *Place* skill, hence written to the specific object name in *ReferencedObjects.txt*. From this information it is possible to grasp the object again without teaching the information above. In this way when specifying to pick a given object again during the programming of the task sequence the manipulator will simply pick the object based on the information in *ReferencedObjects.txt*.

## 6.6.4 Interface

The interface of TUI is represented as a simple terminal running on the embedded laptop of Little Helper ++. The main menu contains three options, respectively closing the program, programming a new task or executing a known task, see figure 6.14.

```

tapas@ubuntu:~$ rosrun lwr_system TUI
Welcome to Little Helper System

Main menu - select an action followed by enter
*****

Value: Name:      Description:
-----
0      Exit       End program
1      Program    Program a new task
2      Execute    Execute a task

Please select an action: 1

```

**Figure 6.14:** The main menu of the user interface of TUI. The user is able to program a new task, execute a known task or close the program.

When selecting *Program* a task name is requested. The task name corresponds to the name of the text file from figure 6.11. Figure 6.15 shows the task name entry and the skill menu. The skill menu consists of 10 different skills. The option to finish the sequence (value 0) does not appear until after programming the first skill. In this case a *Pick* skill is selected.

```
*****
Task filename: Task.txt

Skill menu - to select a skill type its value followed by enter
*****

Value:  Name:      Description:
-----
1      Home        Go to home
2      Pick         Pick up an object
3      Place        Place an object
4      PlaceOnto     Place an object onto a surface
5      Calibrate     Calibrate to workstation
6      PlaceInto     Place an object into a hole
7      MoveTo        Move to a point
8      PegInHole     Place peg into hole
9      Rotate        Rotate around axis
10     PickFromStack Pick up an object from a stack

Please select a skill: 2
```

**Figure 6.15:** The skill menu of the user interface of TUI. The skill menu consists of 10 skills.

Figure 6.16 shows an excerpt from the user input of the *Pick* skill, hence the selection of the object type, the assignment of the object name and the selection of the velocity.

```
Pick menu - please fill in the data for Pick skill
*****

Please enter object type
-----
1      RedBox
2      RotorAxle
3      Cranfield_square
4      RotorCap
5      RotorCore
6      PressureRing
7      PressureRingSide
8      Magnet
9      new object type
Object type: 8

New object name: Magnet1

Please enter velocity (0.01 - 1.00)
-----
Velocity: 0.3
```

**Figure 6.16:** The user input of the *Pick* skill. The user is guided through the selection of the individual parameters.

Once all user inputs are obtained the user is instructed through the teaching routine. After completing the teaching routine the program returns to the skill selecting menu in order to select the next skill in the sequence.

### 6.6.5 Reflection on TUI

TUI presents a simple terminal based interface. It has been used in the development of *Little Helper System* and the process of implementing new skills is relatively quick. The sequential construction of TUI makes it easy to assess the system and to implement new aspects in the software. TUI is used as a

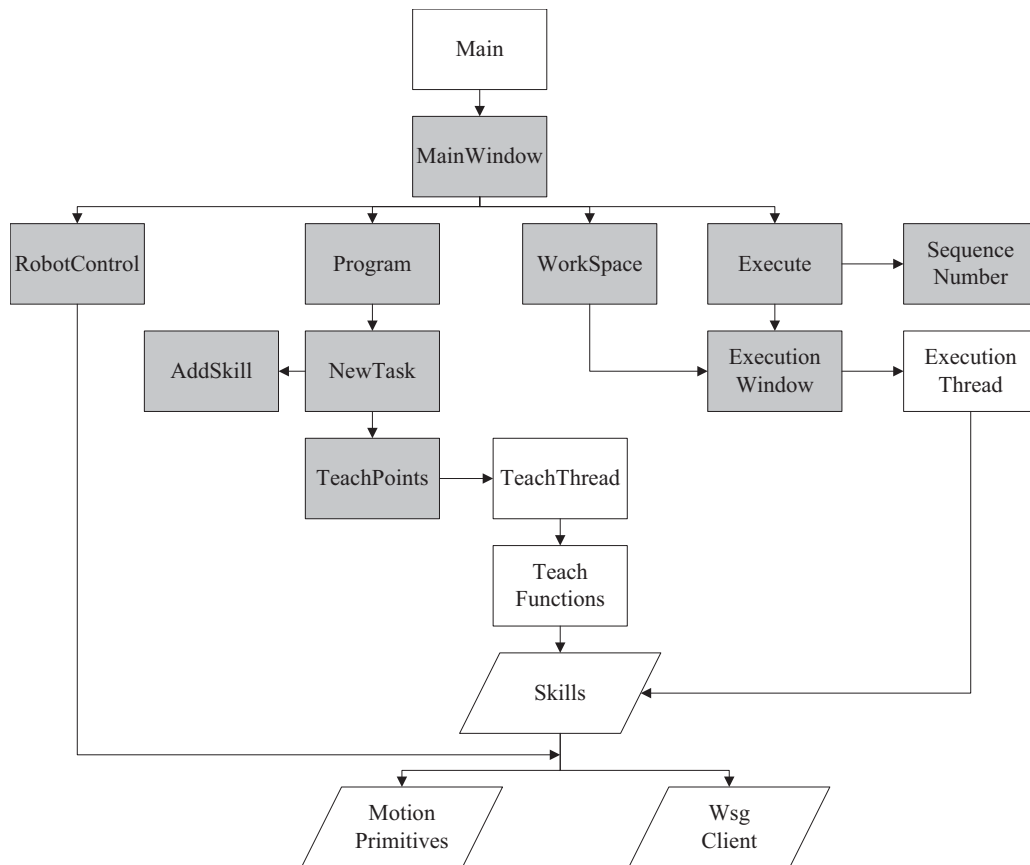
programming interface for the SQFlex assembly tasks, see chapter 8, because of its sequential approach to programming a task sequence.

## 6.7 Graphical User Interface - GUI

The motivation to create the graphical user interface, named GUI, is to develop a more user-friendly interface in contrast to TUI. Instead of terminal based menus and simple keyboard inputs GUI presents the user with graphical menus in which the input is based on clicking, selecting, and dragging.

GUI implements the same functionalities as TUI, hence programming and executing tasks. Besides these functionalities the option to manage workspaces and to manually control the gripper and the manipulator have been implemented. In the execution phase GUI presents a primitive mission set-up, in which it is possible to create a sequence of tasks.

GUI has been developed in QtCreator<sup>3</sup> (Nokia, 2012), in which each window is implemented as a separate class. Figure 6.17 illustrates the software architecture and window hierarchy of GUI.



**Figure 6.17:** Software architecture of GUI. Grey boxes illustrate windows while white boxes illustrate other classes essential to the software architecture.

Each window appears to the user as a new window and is called by input from the user. The following description of GUI illustrates an excerpt of the interface and the architecture.

<sup>3</sup>QtCreator is developed by Nokia and is intended for development of graphical software.

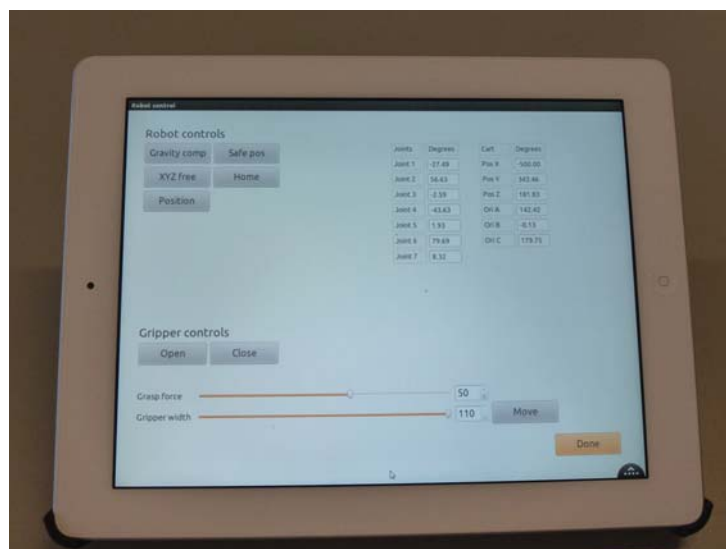
### 6.7.1 Interface

Figure 6.18 shows the main window of GUI. The main window contains five options, respectively closing the program, creating a new task, executing an existing task, managing workspaces, and manually controlling the gripper and manipulator.



**Figure 6.18:** The MainWindow of GUI. The user can choose to e.g. program or execute a task.

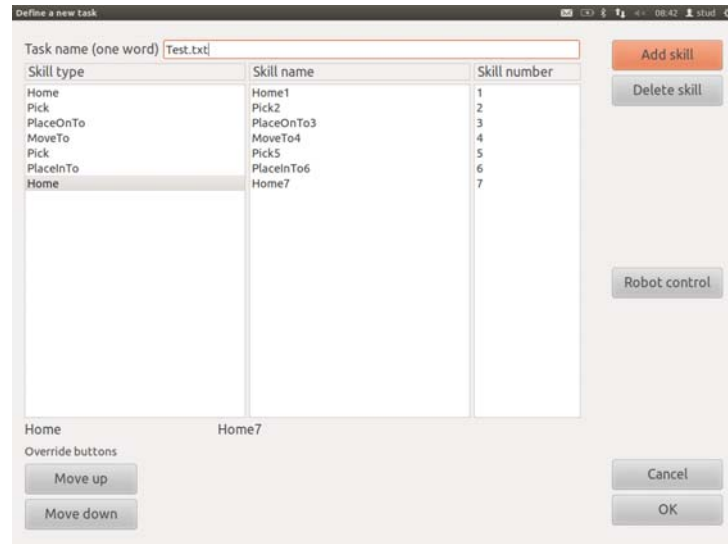
In order to increase the user-friendliness an iPad has been integrated from which GUI can be accessed. This presents the user with a portable platform by which the application becomes familiarised. The integration of an iPad is conducted by using the iPad application Splashtop HD. Splashtop HD enables the configuration of a remote desktop through which the laptop on Little Helper ++ is accessed. Figure 6.19 shows GUI accessed from an iPad.



**Figure 6.19:** GUI accessed from an iPad.

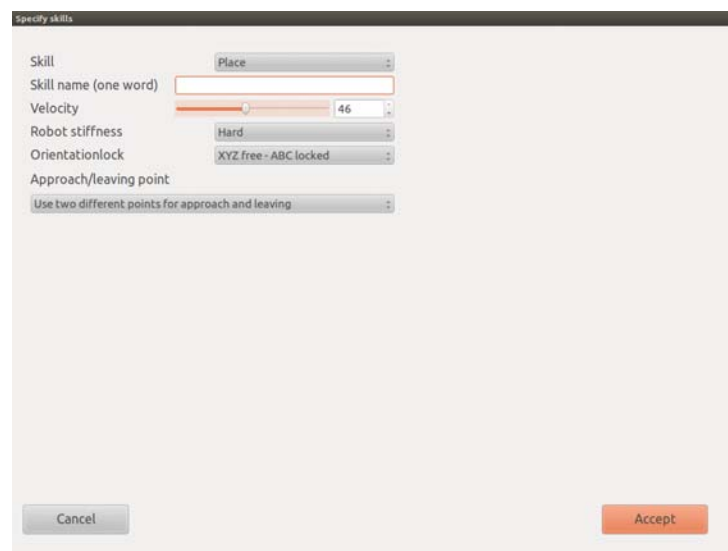
### 6.7.2 Programming

The programming of a new task consists of a part in which the entire skill sequence is configured including specifying a number of user inputs. Once the entire task sequence is specified the teaching of the individual skills follows. Figure 6.20 shows the configuration of a skill sequence.



**Figure 6.20:** Configuration of a skill sequence.

The specification of input parameters of a skill is primarily done by clicking, selecting, and dragging by which the option to change selected input parameters are presented. Depending on the selected skill different specifications are required. Several user input have been standardised in order to ease the specification, e.g. the standardisation of the stiffness containing four modes, *Low*, *Medium*, *Hard*, and *Full*. To exemplify the configuration of a skill the configuration of a *Place* skill is shown in figure 6.21.

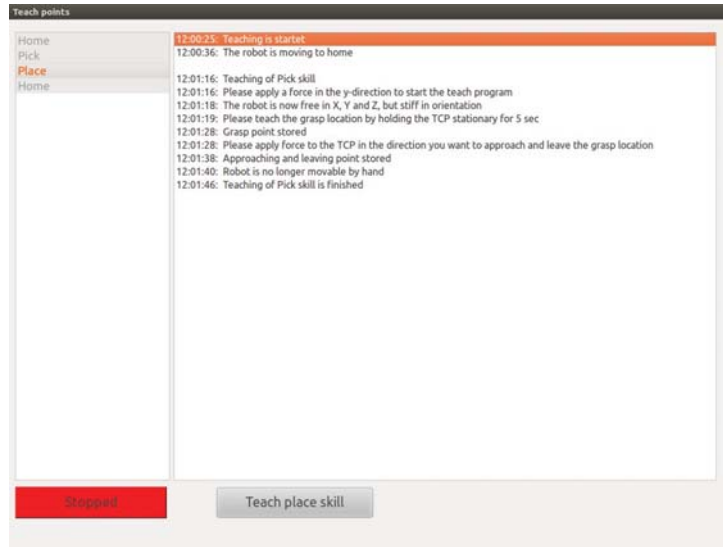


**Figure 6.21:** Configuration of a *Place* skill.

Once the skill sequence has been configured the teaching phase is commenced, during which the user is instructed by informative text. The teaching of the task sequence is conducted by the teach routine implemented in each skill, see section 6.5.2. Once the teaching routine in a skill is executed the windows

of GUI would halt and await the termination of the teaching routine. This is inconvenient as the GUI interface cannot be updated and print instructions to the user. To accommodate this issue the teaching routines are handled by the *TeachThread*, see figure 6.17, which is run as a separate thread, thus this has required the implementation of multithreading.

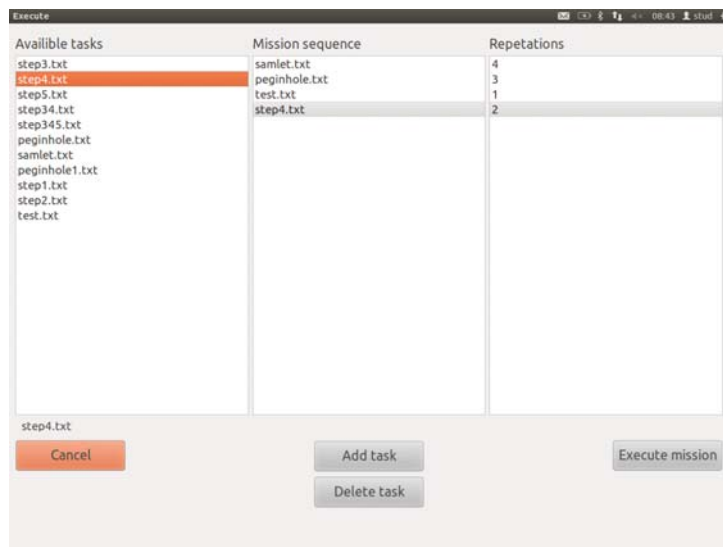
The instruction of teaching the individual skills is shown in figure 6.22.



*Figure 6.22: Informative text instructing the user when teaching the individual skills.*

### 6.7.3 Execution

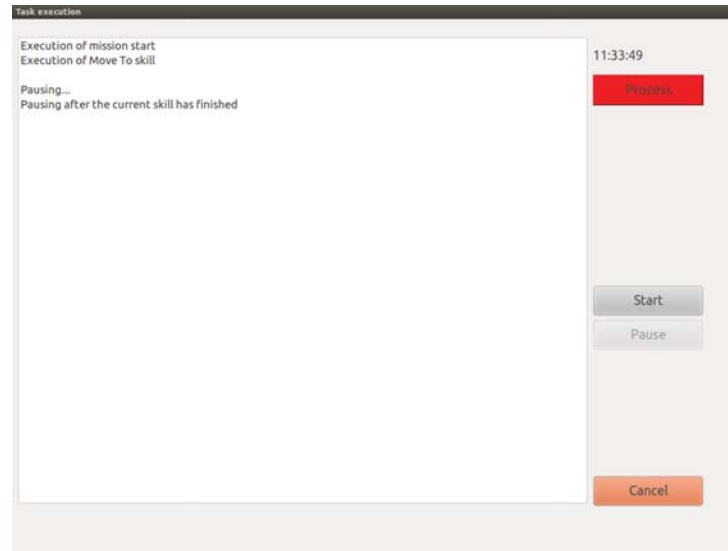
The execution of an existing task is similar to the approach from TUI, though in the execution menu a primitive mission set-up is implemented. The mission set-up enables the composition of several tasks by which an extensive task can be divided into shorter sequences, see figure 6.23.



*Figure 6.23: The configuration of a mission.*

During the actual execution the user is continuously informed about the progress and furthermore has the option to stop, pause, and start the execution is presented, see figure 6.24. As the execution of the

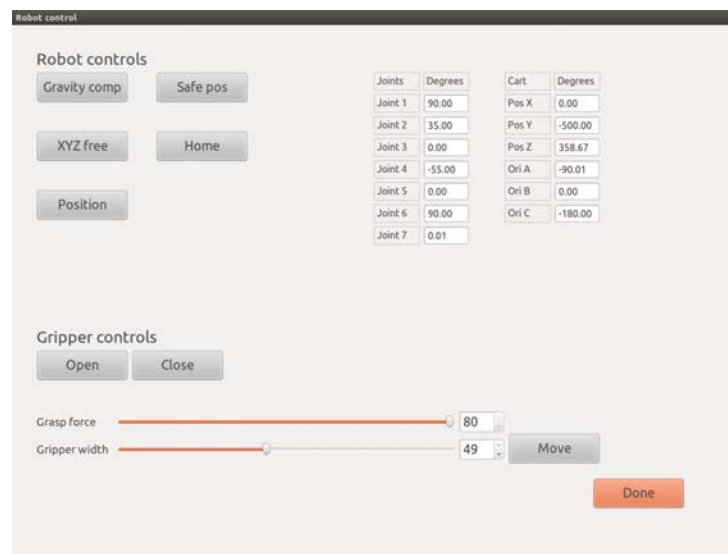
individual skills is performed by the execution routine in each skill this would make the windows of GUI halt and await termination of the execution routine. Like the teaching phase this issue has been accommodated using multithreading and thus the execution routines are run from the separate thread *ExecutionThread*, see figure 6.17.



**Figure 6.24:** The execution of a mission.

#### 6.7.4 Robot control

In *Robot control* manual control of the gripper and manipulator is presented. In this way interaction with the robot when not programming or executing a task is facilitated. These functionalities are furthermore beneficial in situations involving overruling of the system, e.g. moving the manipulator to a safe position or releasing an incorrectly grasped object. *Robot control* enables the application of compliance mode, moving to a predefined location and control of the gripper, see figure 6.25.

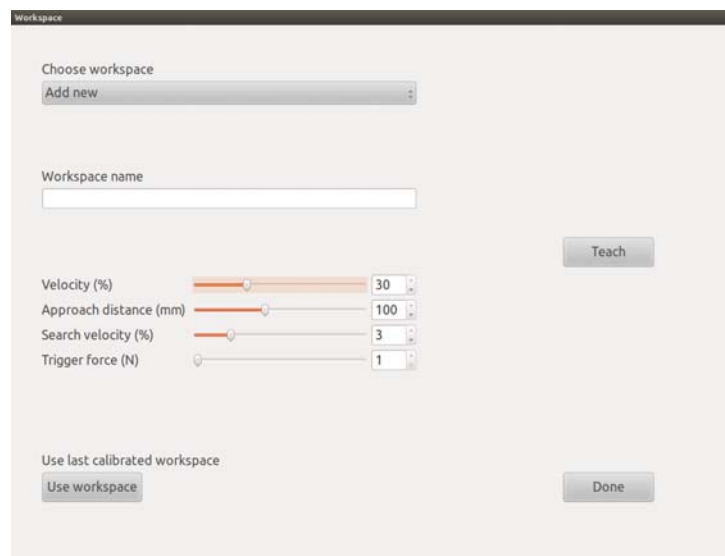


**Figure 6.25:** Manual control of simple functionalities of the manipulator and the gripper in *Robot control*.

### 6.7.5 Workspace

In chapter 8 *Little Helper System* is verified on three industrial tasks during which the robot is moved between different workstations. As the positioning of Little Helper ++ relative to the workstation when arriving is not sufficiently accurate it is necessary to calibrate to the workstation. In chapter 7 the development of a skill to accommodate this calibration is described. Consequently of Little Helper ++ being a mobile robot the coordinates of each workstation are stored referenced to the workstation, hence a workspace, and thus the calibration defines the manipulator relative to the workspace.

The managing of workspaces is presented in the *Workspace* window. From this window new calibration routines can be programmed and already taught calibration routines can be executed to calibrate the robot to a given workstation. Figure 6.26 shows the interface for managing the workspaces.



**Figure 6.26:** Configuration of workspaces.

### 6.7.6 Test of GUI

It is chosen to test GUI and thereby *Little Helper System* on a non robotics expert. Henrik Wiberg, assistant engineer at Aalborg University, has been challenged to program Workstation 3 from chapter 8 using GUI via an iPad. Prior to programming the task Henrik Wiberg is given a 10 minute introduction to both the KUKA LWR, the gripper, the developed skills, GUI, and the task at Workstation 3. The configuration of the task sequence has been done independently by Henrik Wiberg and he has chosen to use a sequence of 7 skills to complete the task:

1. Home
2. MoveTo - Approach coordinate to the conveyor belt.
3. Pick <Rotor cap> from the conveyor belt.
4. MoveTo - From the conveyor belt to the fixture.
5. Place <Rotor cap> on the fixture on Little Helper ++.
6. MoveTo - Leaving coordinate from the fixture.
7. Home



A video of the test of GUI is obtained by scanning the QR-code (*Operator Programming an Industrial Task*).



*Operator Programming an Industrial Task*

## Results

During the test Henrik Wiberg successfully programmed the task in the first attempt. From the following execution the programmed sequence is verified and although it is less refined than the work of a robotics expert, the task is executed with success.

Through the test issues regarding the remote desktop interface used to transfer the GUI interface to the iPad was experienced and consequently the intuitive interface was effected by this. In the long term an independent iPad application handling the interface should be developed to accommodate the transfer issue, but in the short run the embedded laptop of Little Helper ++ is used to configure the task and the iPad is used during the teaching part where the issue is counterbalanced by the portable benefits of the iPad.

In conclusion of the test Henrik Wiberg is quoted:

*"Using this system for a few hours, I reckon that I would master it sufficiently to configure a more complex task."*

*"It wouldn't take long before I don't need the instructions on the iPad in order to interact with the manipulator."*

### 6.7.7 Reflection on GUI

GUI is an intuitive and user-friendly interface that allows the user to program a task from a library of skills without extensive knowledge about conventional robot programming. In GUI the user is guided by instructions during the configuration and teaching of a new task. The fact that GUI can be used from an iPad makes the programming convenient due to the familiarised interface of an iPad. In GUI the entire skill sequence is specified prior to the teaching routine. This approach is inexpedient for an extensive skill sequence as the task may be difficult to predict. The implementation of a primitive mission set-up may solve this problem, though implementation of the approach from TUI is recommended. Furthermore the implementation of a new skill is rather complicated compared to TUI as new skills must be implemented in several windows.

In section 6.7.6 GUI is tested by a non robotics expert, who found GUI easily comprehensible and succeeded in programming an industrial task in the first attempt. Though it is recommended that users of *Little Helper System* receive a short introduction in operating the entire system.

## 6.8 Summary

In this chapter the software architecture of *Little Helper System* has been presented. In chapter 8 the developed system is used to teach and execute industrial tasks, which will serve as a verification of the developed system. The software implementation of the skills has been presented in this chapter, though the identification and development of the individual skills have been motivated by the teaching of the three industrial tasks. In chapter 7 the developed skills are described.



# Library of Skills 7

---

*This chapter presents a description of the skills developed during the project. The purpose is to describe the programming and execution of a skill and thus the usage of the skill. A complete list of the developed skills including input parameters is found in appendix F.*

In chapter 5 a skill is defined as an intelligent sequence of device primitives including pre- and postcondition checks. During the project 10 different skills have been developed based on the abilities needed to solve the two industrial assembly tasks described in chapter 8. The skills are intended to be general and able to accommodate any industrial task, and thus the only connection to a specific task is the input parameters.

The programming of a task is divided into two parts, a specification part conducted on the iPad or the computer and a subsequent teaching part where the user interacts with the manipulator. The teaching part is defined by the teaching routine implemented in each skill, see chapter 6. The specification part is conducted in either the interface of TUI or GUI, during which the user specifies e.g. velocity, manipulator stiffness, search velocity etc.

The execution of a task is conducted by the execution routine defined in each skill. As a skill is executed different pre- and postcondition checks are performed in the beginning and in the end of a skill. All precondition checks must be fulfilled prior to executing the task, while a postcondition check verifies the outcome of the skill, see chapter 5.

The *Pick* and the *PegInHole* skills are described in detail in section 7.1 and 7.2 respectively, and the rest of the developed skills follow. A complete description of all skills is found in appendix F.

## 7.1 Pick Skill



*Pick*

The *Pick* skill is used to pick up an object. A video of the *Pick* skill is obtained by scanning the QR-code (*Pick*).

### 7.1.1 Programming

The object-oriented approach described in chapter 5 is implemented in the *Pick* skill, and thus during the specification part of the programming the user must specify information about the object to be grasped.

### 7.1.2 Specification

In reference to section 6.6 first the user must specify an object type followed by a specific object. Figure 7.1 and figure 7.2 both illustrate the specifying part of the *Pick* skill conducted in GUI.

The 'Specify skills' dialog box shows the 'Pick' skill selected. The 'Skill name (one word)' field is empty. The 'Velocity' slider is set to 61. The 'Robot stiffness' dropdown is set to 'Medium'. The 'Object type' dropdown is set to 'RedBox'. The 'Orientationlock' dropdown is set to 'XYZ free - ABC locked'. The 'Approach/leaving point' dropdown is set to 'Use two different points for approach and leaving'. A text box on the right says 'Add new RedBox1'. At the bottom, it says 'The new object will be named: RedBox2'. There are 'Cancel' and 'Accept' buttons.

**Figure 7.1:** Specifications of a *Pick* skill with known object type.

The 'Specify skills' dialog box shows the 'Pick' skill selected. The 'Skill name (one word)' field is empty. The 'Velocity' slider is set to 61. The 'Robot stiffness' dropdown is set to 'Medium'. The 'Object type' dropdown is set to 'Add new'. The 'Orientationlock' dropdown is set to 'XYZ free - ABC locked'. The 'Approach/leaving point' dropdown is set to 'Use two different points for approach and leaving'. The 'Object type (one word)' field is set to 'RotorAxle1'. Under 'Advanced settings', the 'Object tolerance' slider is set to 3 and the 'Grasping force' slider is set to 50. At the bottom, it says 'The new object will be named: RotorAxle1'. There are 'Cancel' and 'Accept' buttons.

**Figure 7.2:** Specifications of a *Pick* skill with new object type.

Figure 7.1 shows the specification interface when selecting a known object type and figure 7.2 shows the specification interface when selecting a new object type. The specifications needed in the *Pick* skill are:

- Velocity
- Object type
- Stiffness level
- Approach/Leaving
- Orientation lock

The *Stiffness level* defines the stiffness mode of the manipulator when leaving the grasping coordinate and contains pre-specified stiffness settings, see section 6.6. By using specified levels instead of an arbitrary value the user input becomes less complicated. Setting a stiffness less than *Full* is beneficial when e.g. picking an object from a hole as the object otherwise could become wedged in the hole.

The *Approach/Leaving* parameter defines whether to use the same coordinate for approaching and leaving the object or two separate coordinates.

The *Orientation lock* input defines whether the TCP should be locked in orientation, ABC, during the subsequent teaching part.

If a new object type is selected the grasp force, object tolerance, and object name must furthermore be specified. If an existing object type is specified a referenced object must be chosen, see figure 7.1.

## Teaching

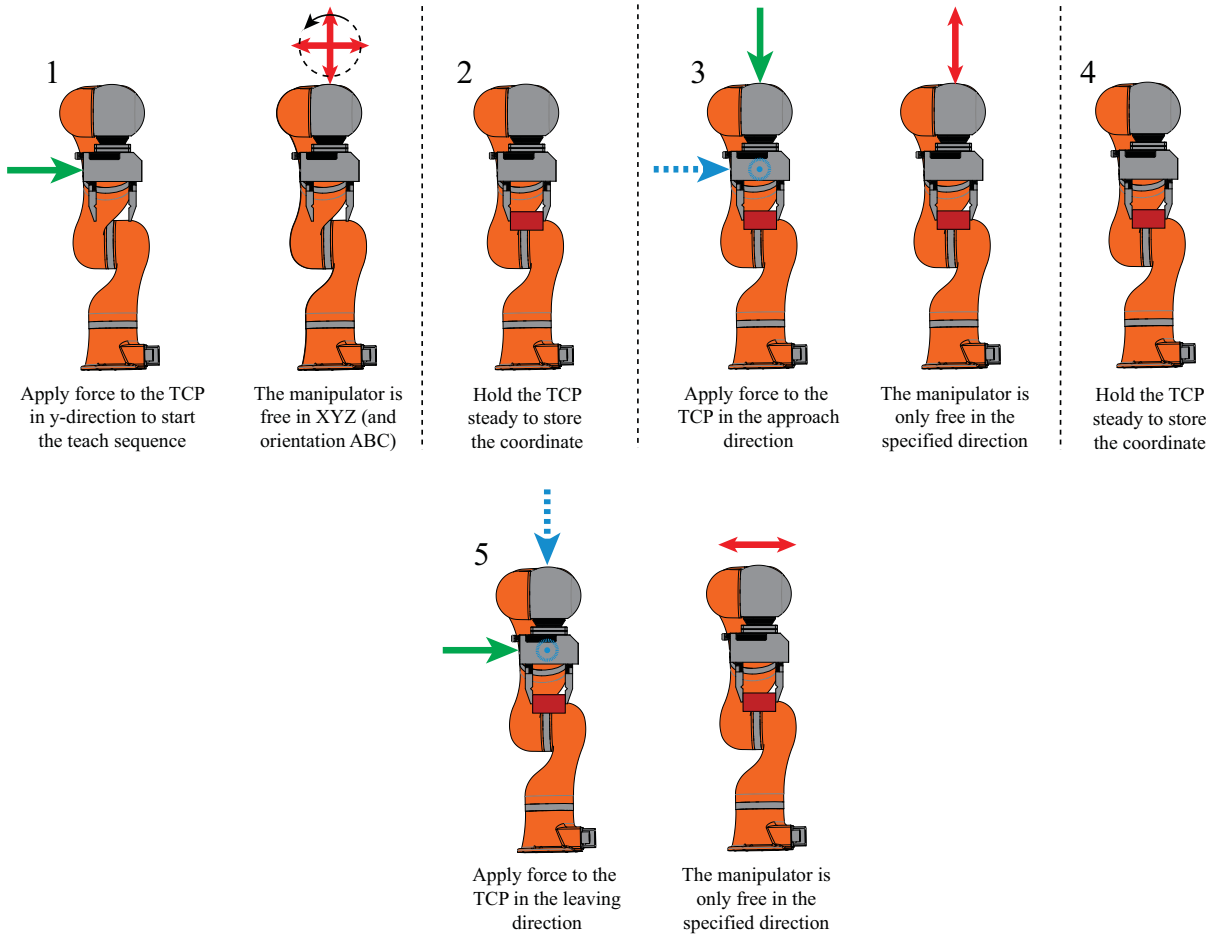
The sequence in figure 7.3 illustrates the physical input during the teaching of a *Pick* skill. Below each manipulator in the figure an instruction to the user is listed. The numbers in the figure refer to the listed enumeration below:

1. Apply a force in the y-direction of the TCP. Subsequent the manipulator becomes free in all directions, XYZ, and free in orientation, ABC, depending on the user specification.
2. By holding the TCP stationary until a beep sounds the manipulator is set to *Position Control*. Subsequently the gripper will grasp the object while the manipulator has a low stiffness in the direction of the gripper movement. During this the TCP is centred to the object as the object is grasped. The coordinate is stored and the object is released to enable force input of the approach direction.
3. The direction of the approaching the object is taught by applying a force in the given direction and subsequently the TCP becomes movable only in that given direction.
4. By holding the TCP stationary until a beep sounds the manipulator is set to *Position Control* and the approach distance is stored.
5. If the *Approach/Leaving* parameter is specified to use two separate coordinates for approaching and leaving the object, the manipulator returns to the grasp coordinate and the enumerators 3 and 4 are repeated for the leaving direction and distance.
6. The manipulator returns to the object coordinate, grasps the object and moves to the leaving coordinate.

### 7.1.3 Execution

From the programmed input the *Pick* skill is executed. Prior to executing the sequence the *BaseSkill::CalcApproachLeaving*, described in section 6.5.2, is used to calculate an approach and a leaving coordinate. The execution of the *Pick* skill is listed below:

- *Check that the gripper is empty.*
- *Move to the calculated approach coordinate.*



**Figure 7.3:** The figure shows the input parameters of the Pick skill taught by physical interaction with the manipulator. Blue and green arrows indicate the input options whereas the green arrows indicate the chosen direction. The red arrows indicate in which directions the manipulator is movable.

- Open gripper.
- Move linearly to the target coordinate.
- Set low stiffness in the tool y-direction.
- Grasp object.
- Check width of grasped object.
- Set specified stiffness.
- Move linearly to the leaving coordinate.
- Set position control.
- Check that the object is still in the gripper.

If the gripper is not empty prior to executing, the *Pick* skill is skipped.

## 7.2 PegInHole Skill

The *PegInHole* skill is used to place a peg into a hole hence an object into another object. The skill differentiates from the *PlaceInto* skill by angling the peg relative to the hole and thus increasing the

probability of correctly finding the hole. The *PlaceInto* skill does not increase probability of finding the hole correctly, but instead implements an active search algorithm to find the hole if not initially found. A video of the *PegInHole* skill is obtained by scanning the QR-code (*PegInHole*).



*PegInHole*

### 7.2.1 Programming

During the specification part of the programming the user must specify the parameters listed below:

- Velocity
- Search velocity
- Trigger force

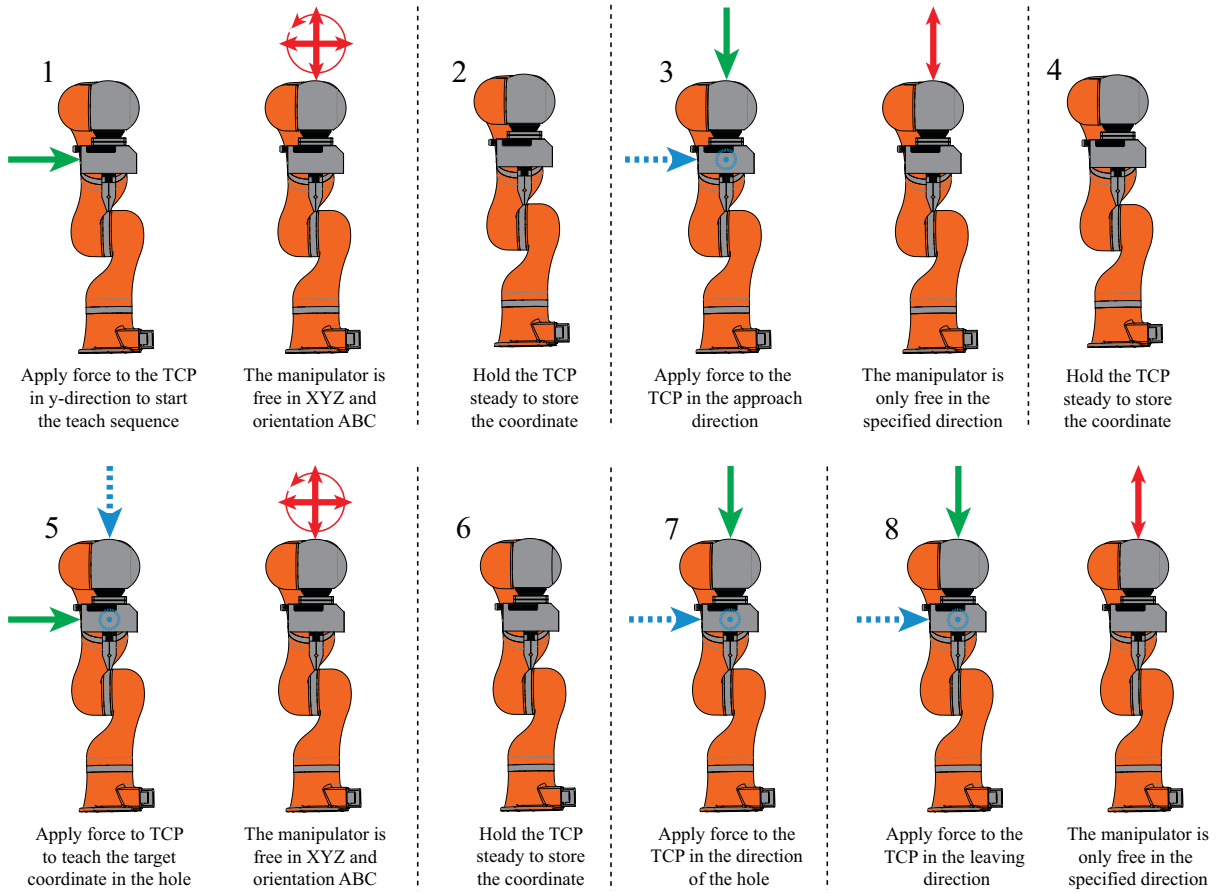
The *Search velocity* defines the search velocity when searching for the hole, please see the listed sequence in section 7.2.2. The *Trigger force* defines the force increase needed to interrupt the search.

The sequence in figure 7.4 illustrates the physical input from the user during the teaching of the *PegInHole* skill. Below each manipulator in the figure an instruction to the user is listed. The numbers in the figure refer to the listed enumeration below:

1. Apply a force in the y-direction of the TCP to start. Subsequently the manipulator becomes free in all directions and orientation.
2. The grasped object is placed angled and in contact with the edge of the hole. By holding the TCP steady until a beep sounds the manipulator is set to *Position Control* and the coordinate is stored.
3. The direction of approaching this coordinate is taught by applying a force in the given direction and subsequently the TCP becomes movable only in that given direction.
4. The TCP is moved to the desired approach distance and by holding the TCP steady until a beep sounds the manipulator is set to *Position Control* and the approach distance is stored.
5. Apply a force in the y-direction of the TCP to proceed. Subsequently the manipulator becomes free in all directions and orientation.
6. The grasped object is placed in the hole and by holding the TCP steady until a beep sounds the manipulator is set to *Position Control* and the coordinate is stored. The gripper releases the object.
7. A force is applied in the direction of the hole.
8. The direction of leaving is taught by applying a force in the given direction and subsequently the TCP becomes movable only in that given direction.
9. The TCP is moved to the desired leaving distance, and by holding the TCP steady until a beep sounds the manipulator is set to *Position Control* and the leaving distance is stored.

### 7.2.2 Execution

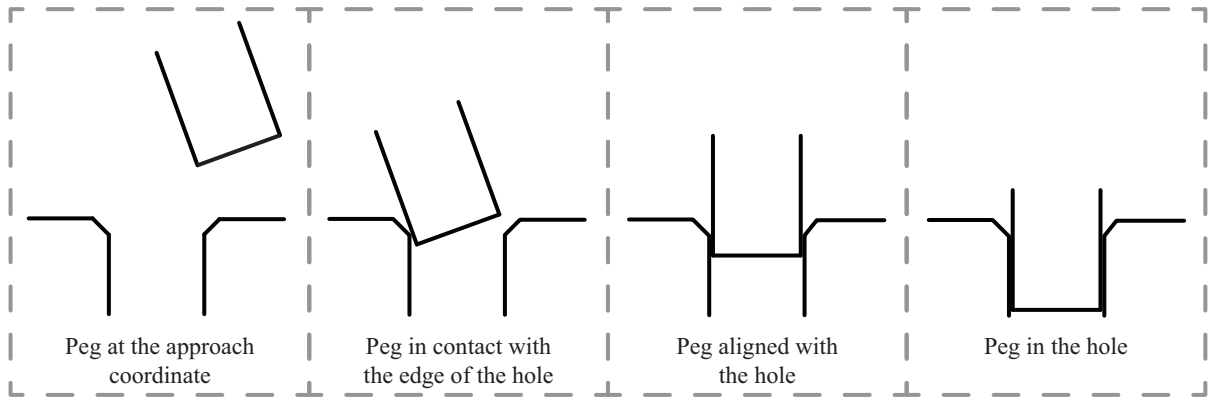
The execution sequence of the *PegInHole* skill is listed below, and figure 7.5 illustrates the main sequence. Prior to executing the sequence the *BaseSkill::CalcApproachLeaving*, described in section 6.5.2, is used to calculate an approach and a leaving coordinate. The execution of the *PegInHole* skill is listed below:



**Figure 7.4:** The figure shows the input parameters of the *PegInHole* skill taught by physical interaction with the manipulator. Blue and green arrows indicate the input options whereas the green arrows indicate the chosen direction. The red arrows indicate in which directions the manipulator is movable.

- Check that the gripper is holding an object.
- Move to the approach coordinate.
- Move object linearly towards the hole and stop at a distance of 10 mm from hole.
- Search for contact with hole.
- Calculate length of the grasped object.
- Align the object to the orientation of the hole by rotating the peg around the end in contact with the hole.
- Move to target coordinate inside the hole.
- Release object in gripper.
- Move linearly to the leaving coordinate.
- Check that the gripper is empty.





**Figure 7.5:** Execution sequence of the *PegInHole* skill.

## 7.3 More Skills

In this section the developed skills are presented one by one, and short descriptions of the usage of each skill are included. A more thorough description of the programming and execution of the individual skills is found in appendix F.

### Home

The *Home* skill moves the manipulator to a specific coordinate located above the platform. The coordinate is specified in the joint space. The *Home* skill does not need any physical input from the user during the teaching sequence as the manipulator automatically moves to home position at a safe velocity during the teaching routine.

### Place



*Place*

The *Place* skill is used to place or drop an object at a given coordinate. The manipulator moves to a calculated approach coordinate, from which it moves linearly to the target coordinate. Subsequently the object is released and the manipulator moves linearly to a calculated leaving coordinate. During the programming phase the *Place* skill ascribes location information to the placed object, see section 6.6. A video of the *Place* skill is obtained by scanning the QR-code (*Place*).



*PlaceOnto*

### PlaceOnto

The *PlaceOnto* skill is used to place an object onto a surface. The skill differentiates from the *Place* skill by searching for contact with the surface, and thus the skill is used if the surface or the object height is not known precisely. Furthermore it can be used to carefully place an object. The execution sequence is similar to that of the *Place* skill described above, but as the manipulator moves from the approach coordinate towards the target coordinate, it will stop at a given distance from the target coordinate and search for contact. A video of the *PlaceOnto* skill is obtained by scanning the QR-code (*PlaceOnto*).

### PlaceInto

The *PlaceInto* skill is used to e.g. fit an object into a hole. The execution sequence is similar to that of the *Place* skill described above, but as the manipulator moves from the approach coordinate towards the

target coordinate, it will stop at a given distance from the top of the hole and search for contact. If contact is found, hence the object has missed the hole, a search pattern is conducted orthogonally to the hole while keeping a force towards the hole. Eventually the object will align with the hole and a drop is measured. Subsequently the manipulator moves linearly to the target coordinate inside the hole and the object is released.

If no contact is found when searching towards the hole, hence the object slides into the hole, the manipulator is moved to the target coordinate and the object is released. A video of the *PlaceInto* skill is obtained by scanning the QR-code (*PlaceInto*).



*PlaceInto*

### MoveTo

The *MoveTo* skill is used to store via coordinates. This is an advantage as the manipulator often operates in a narrow environment and no external motion planner is available. The *MoveTo* skill allows the user to store several coordinates in a row in order to move the manipulator through a discrete trajectory. In the *MoveTo* skill furthermore it is possible to align the TCP relative to the base coordinate system. This is beneficial when e.g. picking up an object from a horizontal surface. A video of the *MoveTo* skill is obtained by scanning the QR-code (*MoveTo*).



*MoveTo*

### Rotate

The *Rotate* skill rotates an object around its axis. To use the *Rotate* skill a *Pick* skill must precede in order to grasp the object first. The *Pick* skill is used with no leaving distance. The angle of the rotation is taught by physically moving the TCP during the teaching phase or by specifying an arbitrary rotation angle. Furthermore, it is possible to repeat a given angular rotation, hence rotate the object, release and move back and then grasp and rotate the object again, until an external signal is received. A video of the *Rotate* skill is obtained by scanning the QR-code (*Rotate*).



*Rotate*

### PickFromStack

The *PickFromStack* skill is used to search for and to pick up the upper object in a stack. The manipulator moves to an approach coordinate. It moves to a starting search coordinate whereas it searches towards the stacked objects until a force is measured. As the upper object is found the gripper opens and on the basis of a specified height of the object, only the upper object is picked up. The manipulator moves to the approach coordinate. A video of the *PickFromStack* skill is obtained by scanning the QR-code (*PickFromStack*).



*PickFromStack*

### Calibrate

The *Calibrate* skill performs a calibration between the robot and a given workstation using the manipulator's torque sensors. The calibration is necessary due to the accuracy of the platform. To perform the calibration the gripper is closed and the jaws are used for calibration, hence touching the workstation. The user teaches four coordinates in three orthogonal planes, from which a coordinate system at the workstation is calculated.

In the beginning of the execution of the calibration routine the TCP moves to a coordinate at a specified distance away from the workstation. The manipulator searches for contact with the workstation, and when contact is found, the coordinate is saved. The manipulator moves to a via coordinate in between

each of the four coordinates. This minimizes the risks for collisions as the manipulator moves in joint space. A new workstation coordinate system is calculated and it is set as the base of the manipulator. To perform the calibration four coordinates in three planes are sufficient as it is assumed that the robot only can be orientated around a vertical axis. Furthermore, it is assumed that the horizontal axis of the robot and the workstation are parallel. The calibration can only be performed to a workstation where three perpendicular planes are available. A video of the *Calibrate* skill is obtained by scanning the QR-code (*Calibrate*). Information about the calculation of a workspace is found in (Craig, 2005).

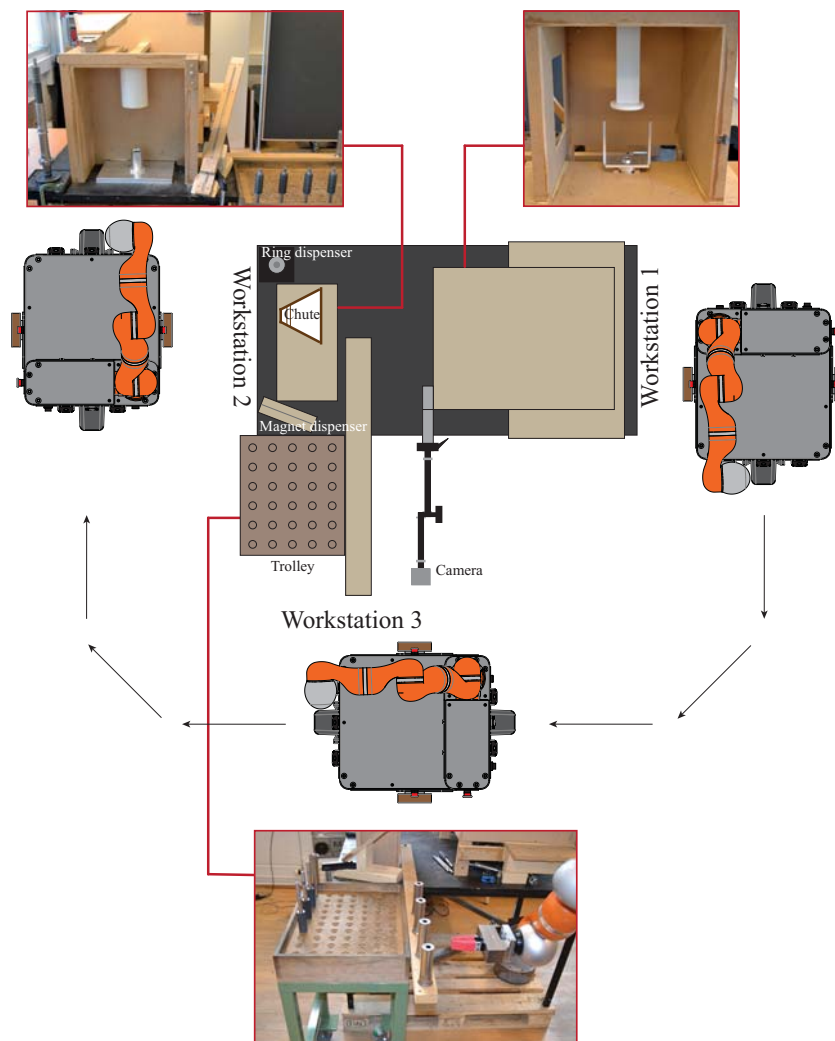


*Calibrate*



# Industrial Assembly Tasks 8

*This chapter describes the programming and execution of tasks from the SQ production facility at Grundfos. These tasks correspond to the tasks planned for the month 24 demonstration in the TAPAS project. The three tasks are the rotor cap collection, the rotor shaft assembly, and the rotor assembly. The purpose of programming and completing these tasks is to verify Little Helper System and furthermore to test the skill-based approach. This is done in reference to the vision in chapter 2. Furthermore, teaching, and programming these tasks have helped identify new skills to be developed.*



**Figure 8.1:** An illustration of the replication set-up at Aalborg University viewed from the top. All replications are manufactured in wood with the same dimensions as the real workstations at Grundfos.

Figure 8.1 illustrates the set-up of the replications mounted on a table in the Robot laboratory at AAU. For each workstation Little Helper ++ has a dedicated parking location. For the platform to move autonomously between the workstations a map of the laboratory floor at AAU has been created in the Neobotix platform software. It has been necessary to set up boards to define the working envelope of the platform. Due to the accuracy of the platform, a calibration to each workstation is necessary before initiating the task. At Workstation 1 and Workstation 2 the calibration is carried out using the manipulator. This is done by performing a previously taught calibration routine specific for each workstation. The *Calibrate* skill is described in chapter 7. At Workstation 3, the rotor cap collection, a vision system supplied by Grundfos is used to locate the rotor caps. At this workstation it is sufficient to calibrate to the workstation using the vision system.

## 8.1 Programming Approach

The programming of the tasks is performed using TUI described in section 6.6. TUI is used instead of GUI as it is possible to specify and teach the individual skills step by step. This is an advantage when programming a complex and comprehensive task consisting of several skills, as it can be difficult to predict the entire sequence in a complicated task. The programming of Workstation 1 and Workstation 2 was completed in 7 days combined. During this time several new skills were developed. An estimation of the combined programming time at Workstation 1 and Workstation 2, given that all needed skill are available, is one working day.

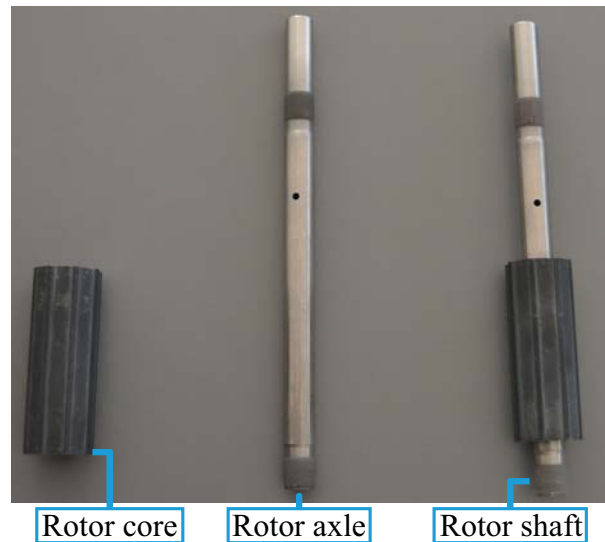
The rotor cap collection at Workstation 3 has not been programmed as the Vision DCN still is under development, see section 6.4. This task is the least complicated of the three tasks with respect to the programming of the manipulator. Even though only the approach to this task is described in this chapter, the task is used in verification of GUI, refer to section 6.7.

## 8.2 Workstation 1 - Rotor Shaft Assembly

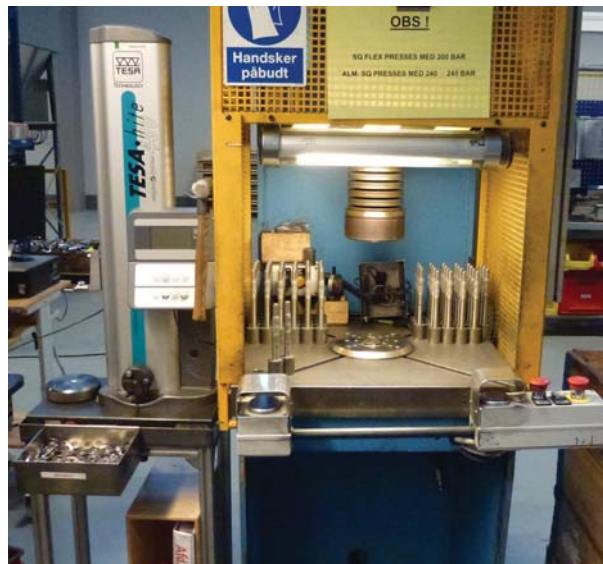


*Rotor Shaft Assembly*

At Workstation 1 the assembly of the rotor shaft is conducted, see figure 8.1. In figure 8.2 the components used in this task are shown. The purpose of Workstation 1 is to assemble the rotor core and the rotor axle into a semi-finished product referred to as the rotor shaft. The components are fixed together by a hydraulic press located inside a cabinet at Workstation 1. Figure 8.3 shows the environment of Workstation 1 at Grundfos. A video of the rotor shaft assembly is obtained by scanning the QR-code (*Rotor Shaft Assembly*).



*Figure 8.2: Components used in the rotor shaft assembly.*



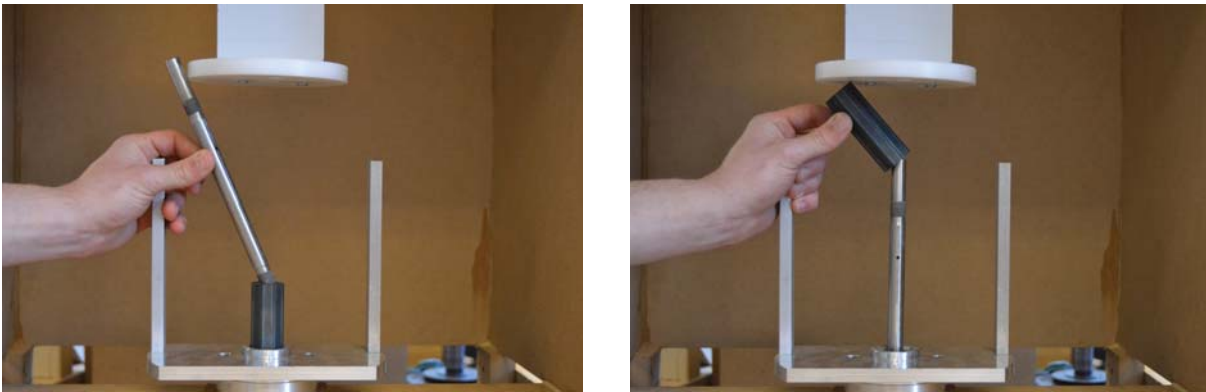
*Figure 8.3: The environment at Workstation1 at Grundfos.*

A detailed description of the manual assembly of the rotor shaft is found on the enclosed Appendix CD in [/Documents/Instruction- SQFlex production.pdf](#). A short enumerated sequence of the manual work follows:

1. Place a rotor axle in the rotor core.
2. Place the rotor axle and the rotor core in the fixture.
3. Rotate the rotor axle until a signal is indicated.
4. Use both hands to activate the press.
5. Remove the pressed rotor.

### 8.2.1 Analysis

The manual work sequence requires modification to be performed by a manipulator. In the manual sequence the rotor core and rotor axle are placed assembled in the fixture. Placing them in the fixture requires a grasp fixating both the rotor core and the rotor axle. This is not possible using the Schunk WSG50 gripper without manufacturing specialised jaws. Furthermore it is impossible to place the parts individually as a result of insufficient space, see figure 8.4.



*Figure 8.4: The figures illustrate the impossibility in placing the parts individually.*

To solve this problem the rotor core is placed on a magnet fixed on the side of the press cabinet. The rotor axle is placed into the fixed rotor core and the rotor axle is used to lift the rotor core. This sequence is shown in figure 8.5.

Once the rotor core and rotor axle are inside the fixture a hole in the rotor axle must be aligned correctly due to a later magnetization. This is checked by a laser equipment that senses a small hole in the rotor axle. The rotor axle is rotated until the laser beam passes through the hole and the operator is informed by a digital display. In the set-up at AAU this signal is emulated.

As a result of the analysis the robot sequence is established:

1. Place a rotor core on the magnet.
2. Place a rotor axle into the rotor core.
3. Place the rotor axle and the rotor core into the fixture.
4. Rotate the rotor axle until a signal is received.
5. Emit signal to active hydraulic press and wait.
6. Remove the rotor shaft.

### Applied Skills

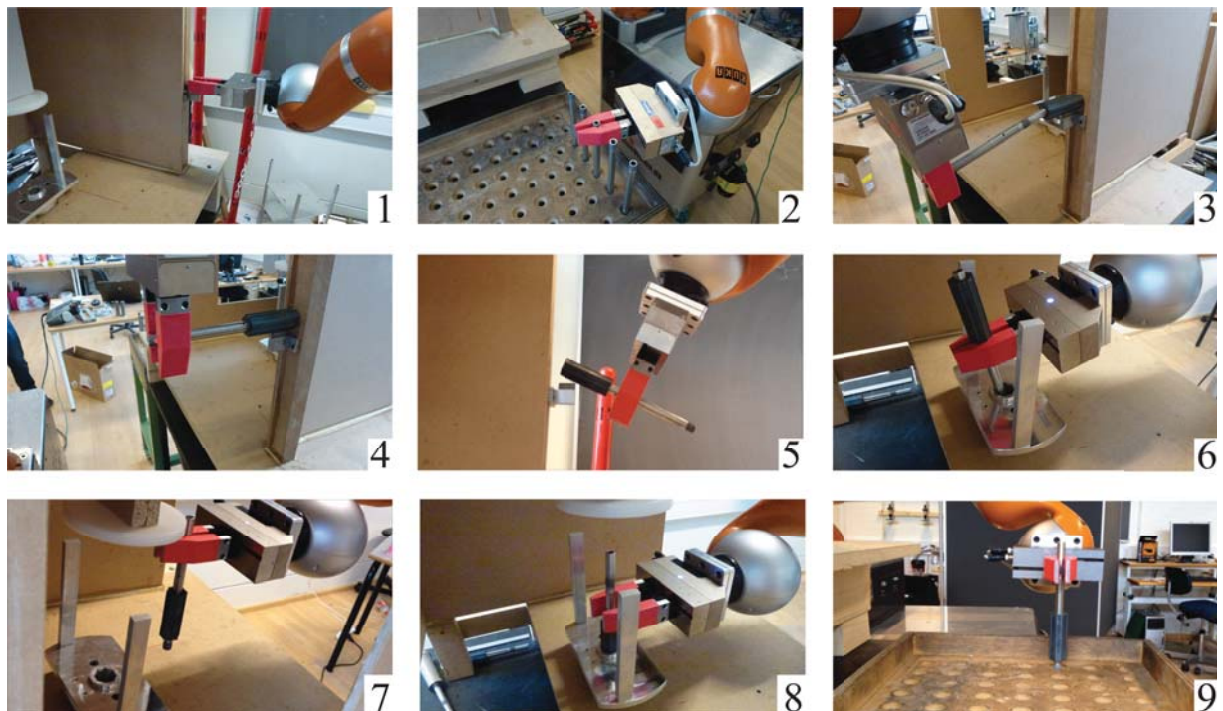
The sequence described above is transformed into a skill sequence listed below. *MoveTo* skills have been used between the individual skills in order to avoid collisions, joint limits, and singularities. The skill sequence consists of 1 *Calibrate* skill, 10 operational skills, 14 *MoveTo* skills, and 1 *Home* skill.

- *Calibrate* <Workstation 1>.
- *Pick* <rotor core> from the table.



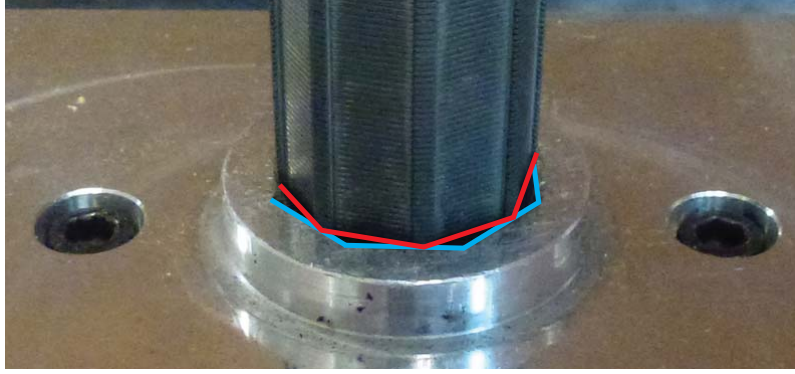
- *PlaceOnTo* <rotor core> on the fixed magnet.
- *Pick* <rotor axle> from the trolley.
- *PegInHole* <rotor axle> in the rotor core.
- *Pick* <rotor axle with core> from the fixed magnet.
- *PegInHole* <rotor axle with core> in the press fixture.
- *MoveTo* align the rotor core.
- *Pick* <rotor axle> in the press fixture.
- *Rotate* <rotor axle>.
- **Pressing**
- *Pick* <rotor shaft> from the press fixture.
- *PlaceInTo* <rotor shaft> into the trolley.
- *Home*

Figure 8.5 shows the sequence of the task.



*Figure 8.5: Sequence of the rotor shaft assembly task.*

As the rotor axle and rotor core are placed into the fixture the rotor core is randomly orientated on top of the gripper jaws. Subsequent to the placement the gripper is withdrawn and consequently the rotor core either drops into the fixture or onto the fixture depending on its orientation, see figure 8.6. To ensure correct alignment a *MoveTo* skill is conducted. This induces a collision between a gripper jaw and the rotor core which aligns the rotor core and thereby makes it drop into the fixture.

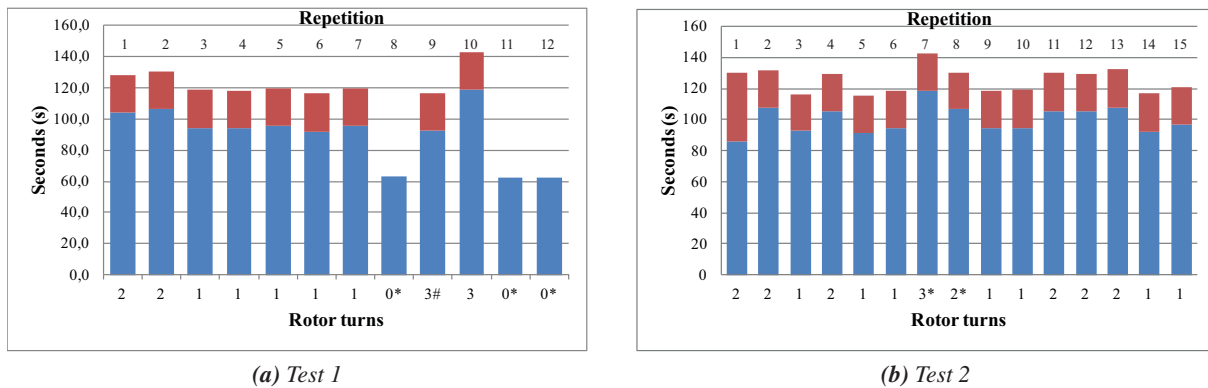


**Figure 8.6:** Wrong alignment of the rotor core when dropped from the gripper jaws.

### 8.2.2 Robustness

In order to validate the robustness of the programmed sequence, it is chosen to execute the task 15 times. Figure 8.7a shows the first attempt to execute the task 15 times. The diagram illustrates the time consumed by the manipulator. The first part (blue) is before the hydraulic press is activated and the second part (red) is after.

During the first attempt an error due to an inaccurately stored coordinate occurred. The problem was solved and 15 new repetitions were conducted, see figure 8.7b. During the second attempt the manipulator failed to set position control. As the problem was not due to a task related issue a new attempt to conduct 15 repetitions was not initiated.



**Figure 8.7:** Robustness test of the rotor shaft assembly task. The lower x-axis, marked Rotor turns, indicates the number of rotations needed in order to align the rotor axle, see appendix F. \* indicates an error while # indicates an abnormal observation which could have caused an error. Data from the conducted task is found on the enclosed Appendix CD in </Documents/Assembly Tasks Data.xlsx>.

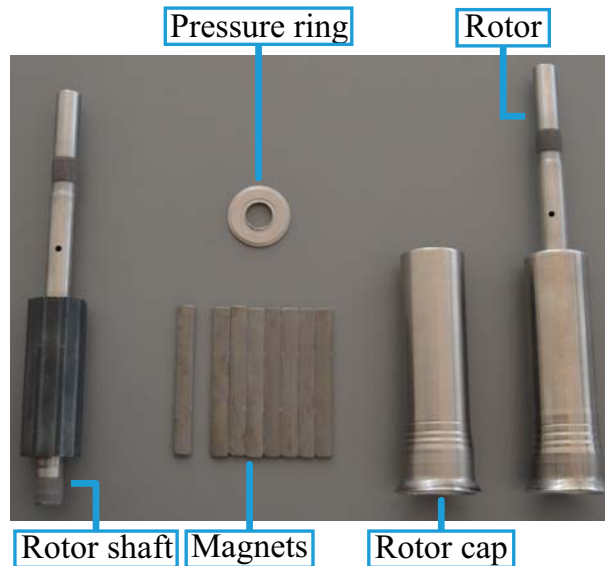
The average time for all successful repetitions is 99.6 seconds for the first part and 24.8 seconds for the last part. This gives a total average time of 124.4 seconds. For comparison a human operator is able to complete the assembly task in approximately 14 seconds. The standard deviation is 8.0 seconds for all repetitions which primarily is due to the rotation of the rotor axle. A description of the *Rotate* skill is found in appendix F. For the task to be considered sufficiently robust for an industrial production 15 repetitions are far from sufficient, but the success of the test indicates that the system is robust.

### 8.3 Workstation 2 - Rotor assembly

At Workstation 2 the assembly of the SQFlex rotor is conducted. The environment consists of a cabinet containing a fixture and a hydraulic press, see figure 8.1. Figure 8.9 illustrates the environment at Grundfos while the used components are shown in figure 8.8. A video of the rotor assembly is obtained by scanning the QR-code (*Rotor Assembly*).



*Rotor Assembly*



*Figure 8.8: Components used in the rotor assembly.*



*Figure 8.9: The environment at Workstation 2 at Grundfos.*

A detailed description of the manual assembly of the task is found on the enclosed Appendix CD in [</Documents/Instruction - SQFlex production.pdf>](#). A short enumerated sequence of the manual work follows:

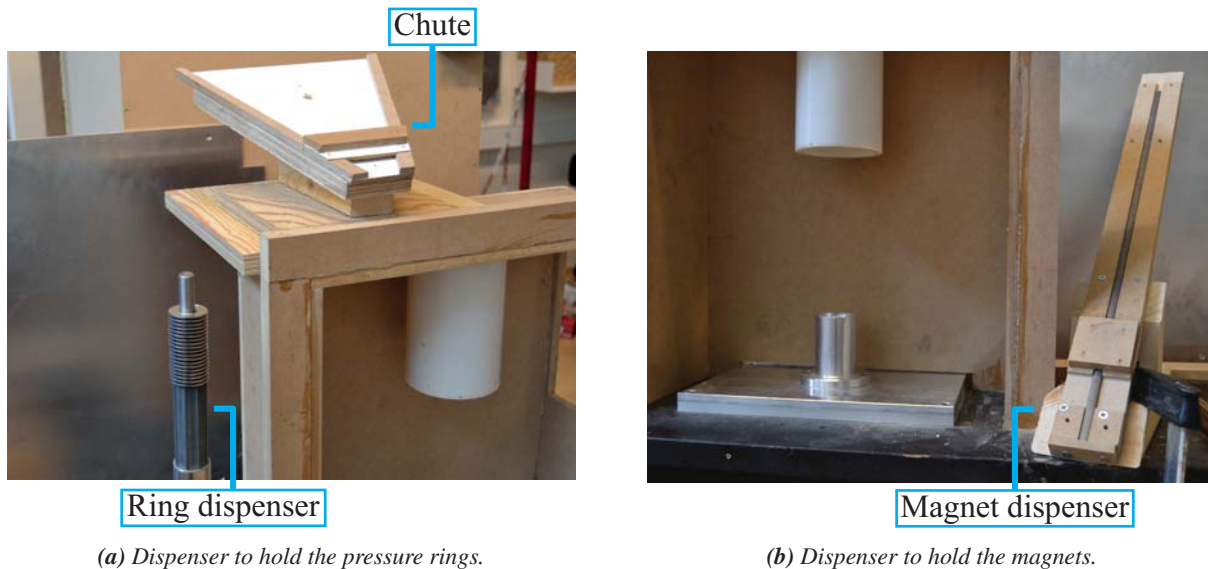
1. Place the pressure ring on the rotor shaft
2. Place the rotor shaft with the ring in the fixture
3. Place the eight magnets on each side of the octagonal rotor core in the fixture
4. Place the rotor cap on top of the fixture
5. Use both hands to activate the press

### 6. Remove the pressed rotor

It should be noted that the assembled rotor still needs trimming of the excess cap afterwards. The rotor assembly task has been programmed using robot macros in KRL, see appendix A.2, though this time the task is programmed using *Little Helper System*.

### 8.3.1 Analysis

The manual work sequence requires modifications to be performed by a manipulator. The pressure rings, the magnets, and the rotor caps are placed in boxes at Grundfos and in order to avoid bin picking it is chosen to construct simple dispensers. The pressure rings are stacked on a vertical steel rod from which the pressure rings can be picked ordered. After picking a pressure ring is dropped into a chute in order to obtain the desired grasp configuration as shown in picture 2 in figure 8.12. The constructed dispenser to hold the pressure rings and a dispenser to hold the magnets are shown in figure 8.10a and figure 8.10b respectively.



**Figure 8.10:** The constructed dispensers.

In the Grundfos environment the pressure ring and the rotor shaft are assembled and afterwards placed together in the fixture. This operation is separated into two steps respectively a step for placing the pressure ring into the fixture and a step for placing the rotor shaft inside the pressure ring in the fixture. When placing the rotor shaft inside the fixture the orientation of the rotor core is unknown. In order to obtain a correct alignment of the rotor core prior to placing the magnets a *MoveTo* skill is applied. Due to the narrow environment in the press, see figure 8.11, the placing of the rotor cap and the removing of the pressed rotor are conducted by complex motions using a *MoveTo* skill.

As a result of the analysis the robot sequence is established:

1. Place the pressure ring in the fixture.
2. Place the rotor shaft in the fixture.
3. Orient the rotor shaft.
4. Place the eight magnets on each side of the rotor core in the fixture.



**Figure 8.11:** The narrow environment at Workstation 2 when placing the rotor cap.

5. Place the rotor cap on top of the rotor shaft.
6. Emit a signal to the hydraulic press and wait.
7. Remove the pressed rotor.

### 8.3.2 Applied Skills

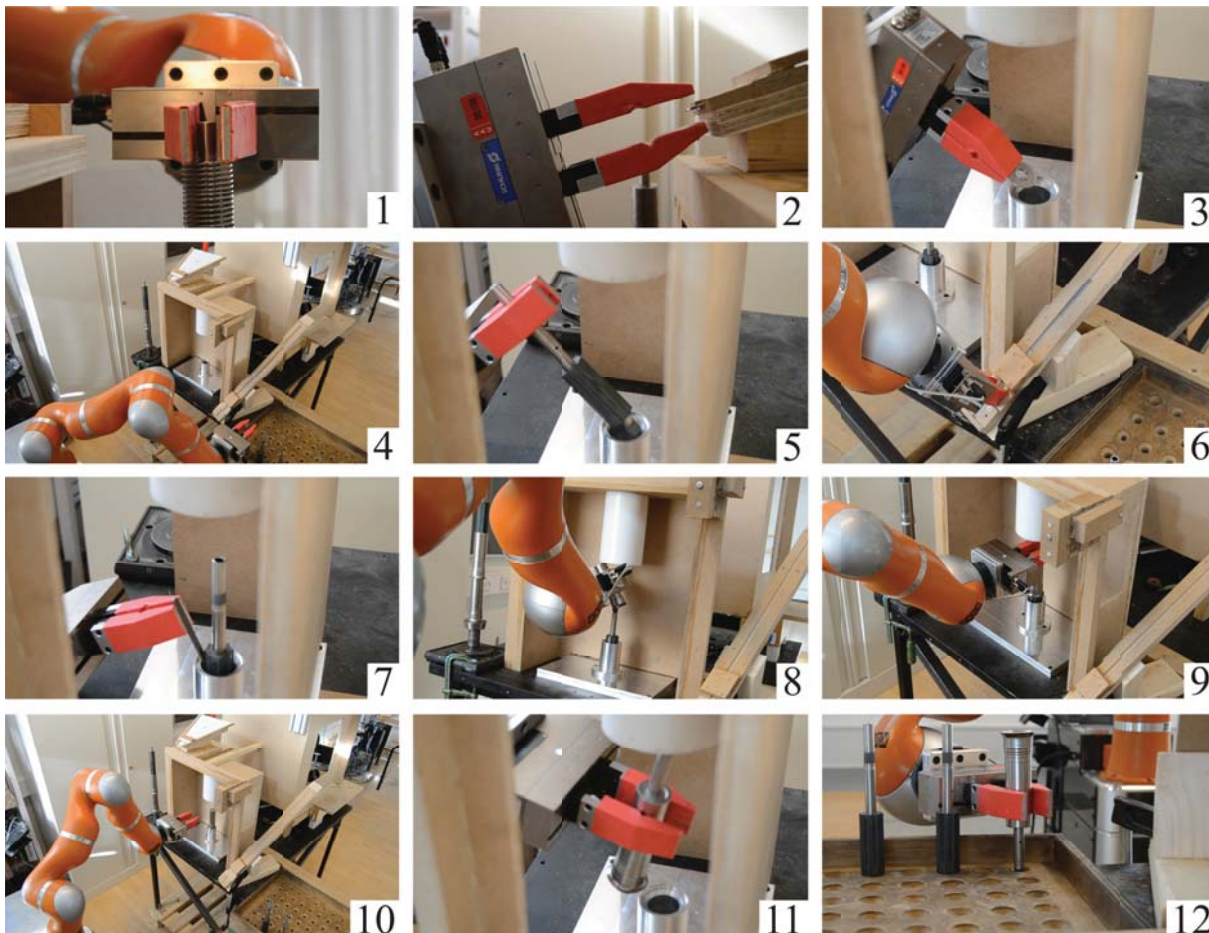
The sequence described above is transformed into a skill sequence listed below. *MoveTo* skills are used between the individual skills in order to avoid collisions, joint limits, and singularities. The skill sequence consists of 1 *Calibrate* skill, 45 operational skills, 31 *MoveTo* skills, and 1 *Home* skill.

- *Calibrate* <Workstation 2>
- *PickFromStack* <pressure ring> from the vertical rod
- *Place* <pressure ring> into the chute
- *Pick* <pressure ring> from the chute
- *Place* <pressure ring> into the press fixture
- *Pick* <rotor shaft> from the trolley
- *PegInHole* <rotor shaft> into the press fixture
- *PlaceInto* <rotor shaft> into the pressure ring inside the press fixture
- *MoveTo* align the rotor shaft
- Placing the magnets (Repeated 8 times)
  - *Pick* <magnet> from the magnet dispenser
  - *PegInHole* <magnet> into the rotor core
  - *Pick* <rotor shaft>
  - *Rotate* <rotor shaft> 45 degrees
- *Pick* <rotor cap> from the platform fixture
- *MoveTo* <rotor cap> onto the rotor shaft



- *Place* <rotor cap> onto the rotor core
- *Pick* <rotor cap>
- *PlaceInto* <rotor cap> down over the rotor axle
- **Pressing**
- *Pick* <rotor>
- *MoveTo* <rotor> out of the press fixture
- *PlaceInto* <rotor> in the trolley
- *Home*

Figure 8.12 shows the sequence of the task.

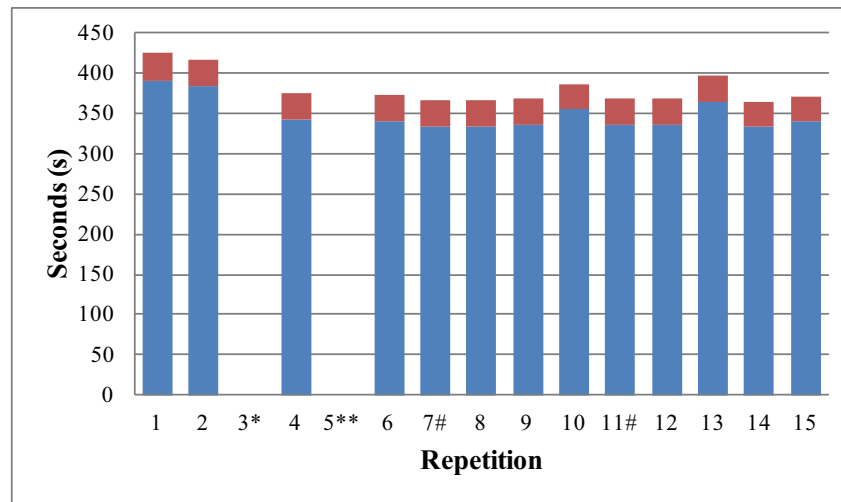


*Figure 8.12: The sequence of the rotor assembly task.*

### 8.3.3 Robustness

In order to validate the robustness of the rotor shaft assembly it is chosen to execute the task 15 times, see figure 8.13. The diagram illustrates the time consumed by the robot to execute the task. The first part (blue) is before the hydraulic press is activated and the second part (red) is after.

During the robustness test an error due to an inaccurately specified object thickness in the *PickFromStack* skill occurred. Furthermore, the manipulator failed to set position control once.



**Figure 8.13:** Robustness test of the rotor assembly task performed with 15 repetitions. \* indicates an error while # indicates an abnormal observation which could have caused an error. Data from the conducted task is found on the enclosed Appendix CD in </Documents/Assembly Tasks Data.xlsx>.

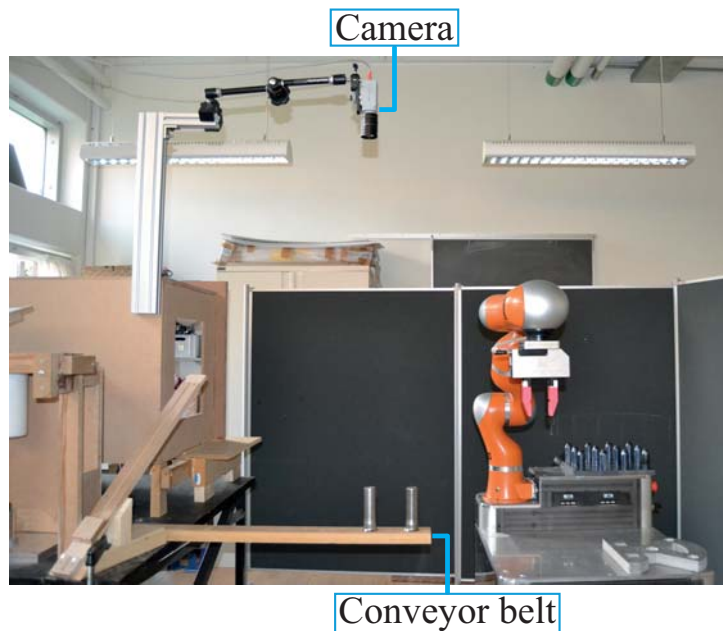
The average time for all successful repetitions is 347.7 seconds for the first part and 32.9 seconds for the second part. This gives a total average time of 380.6 seconds. For comparison, a human operator is able to complete the whole assembly task in approximately 30 seconds. The standard deviation is 19.6 seconds for all successful repetitions which primarily is due to the searching algorithms used in several skills. For the task to be considered sufficiently robust for an industrial production 15 repetitions are far from sufficient, but the success of the test indicates the robustness of the system.

## 8.4 Workstation 3 - Rotor Cap Collection

At Workstation 3 the rotor caps are collected. The environment at Grundfos is shown in figure 8.14. The rotor caps are produced in a spinning cell and afterwards exit on a conveyor belt. The conveyor belt is replicated by a wooden lath and a vision system supplied by Grundfos is used to locate the rotor caps, see figure 8.15. As the Vision DCN still is under development this section only describes the approach to the task.



*Figure 8.14: The spinning cell environment at Grundfos.*

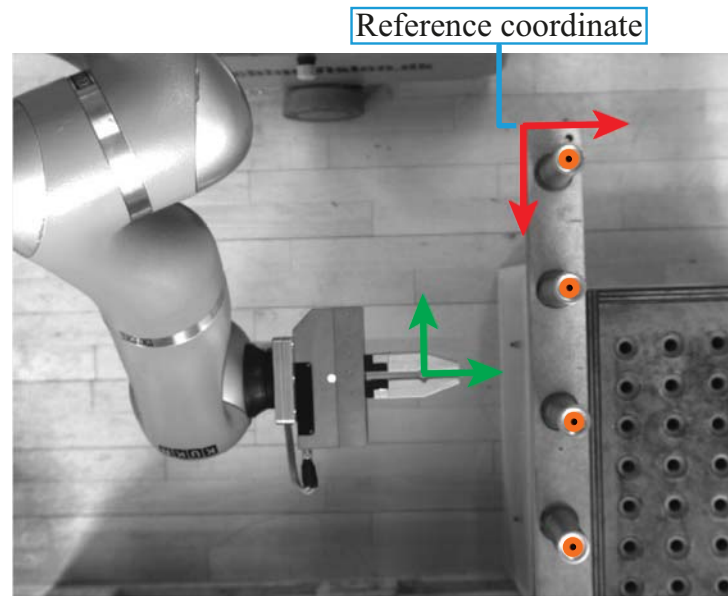


*Figure 8.15: The rotor cap, the conveyor belt and the vision camera in the replication at AAU.*

### 8.4.1 Analysis of Workstation 3

At Grundfos the rotor caps are placed on a conveyor belt from which they are dropped into a pallet. In order to avoid bin picking it is chosen to collect the rotor caps directly from the conveyor belt. The rotor caps are not sufficient uniformly distributed on the conveyor belt and consequently the vision system is used to locate the rotor caps and furthermore to calibrate the gripper to the conveyor belt. The calibration is conducted by locating the gripper relative to a fixed reference coordinate, see figure 8.16. Based on this information a new base coordinate system coinciding with the reference coordinate of the vision system is set on the manipulator.





**Figure 8.16:** Rotor cap and gripper recognition.

Once the rotor caps are collected they are placed on a dedicated fixture mounted on Little Helper ++, see figure 8.17.



**Figure 8.17:** A dedicated fixture to hold the rotor caps. The fixture has been constructed at AAU from i.a. an old broom shaft.

## 8.5 Conclusion

Within this chapter it has been verified that it is possible to program two complex industrial assembly tasks using *Little Helper System*. *Little Helper System* has been used to both program and execute the tasks. The programming of the tasks has been eased significantly compared to traditional robot programming due to the skill-based approach.

To solve the tasks it has been necessary to develop dispensers to hold both the magnets and the pressure rings. Furthermore it has been necessary to mount a magnet to hold the rotor core as it is assembled with the rotor axle and a fixture to hold the rotor caps on Little Helper ++'s platform. In reference to the vision from chapter 2 the manufactured dispensers and fixtures are constructed in such a way, that an operator quickly could have created these fixtures and dispensers using materials found in the production area or from simple wood boards and laths.

The programmed tasks have been verified through robustness tests where two types of errors occurred.

The errors were failing to set position control on the manipulator and inaccurately stored coordinates. The position control failure is considered to be due to an error on the KRC or the Manipulator DCN, and no actions towards identifying the course of the problem have been taken as a defect in joint three has been found, see appendix C. In spite of the errors the execution of the tasks is considered relative robust even though only 15 repetitions have been carried out.

The cycle time at Workstation 1 is approximately 14 seconds for an operator and 124 seconds for the manipulator, while the cycle time at Workstation 2 is approximately 30 seconds for an operator and 380 seconds for the manipulator. The cycle time of the manipulator is more than 12 times longer than that of an operator. It should be noted, that the cycle time of the manipulator could be decreased by optimizing the sequence of especially the *MoveTo* skills, though it is not expected to significantly decrease the ratio. Furthermore it should be noted that the cycle time listed for an operator does not include breaks and other interruptions.

# Discussion 9

---

*In this chapter the process and the results of the project are discussed. The purpose is to give a critical evaluation of the choices, assumptions and results that have been established during the project.*

## **The Construction of Little Helper ++**

During the project several issues with the KUKA LWR have been experienced, and in the beginning of 2012 in collaboration with KUKA it was determined that joint 3 had an unknown defect. As these issues continued, and others occurred, during the project it is debatable whether the manipulator should have been returned to KUKA for repair in the beginning of 2012 instead of waiting until July 2012. An early repair might have improved the performance of the conducted industrial tasks as the torque sensor might have been more accurate. A repair in the beginning of 2012 would have resulted in several weeks without the manipulator, which would have induced a setback of the project.

## **Little Helper System**

*Little Helper System* aims at implementing the skill-based approach in programming industrial tasks. Chapter 6 describes the software architecture of this system and the implementation of skills. The architecture is characterized by the process of the system development and the progress of establishing a skill implementation architecture. Due to this it is debatable whether the architecture could have been optimized or made more comprehensive.

In reference to the architecture a stringent naming convention has not been implemented. This is due to the fact, that none of the students of this project possessed any C++ or object-oriented programming experience<sup>1</sup>, and consequently the developed code is characterized by the learning process. It is debatable whether a stringent naming convention should have been implemented later in the development process to make the code more comprehensible. This has been omitted as the further development of the system has had higher priority than refining the software.

In chapter 6 it is described how the task layer is integrated into the two user interfaces, hence TUI and GUI. In relation to a future development of a more intelligent task layer or the development of a new interface, it is debatable whether the integration of the task layer into the user interfaces is an ill-fitted approach. Alternatively, the interfaces should have been implemented as a separate layers. Given this approach the GUI could be developed as a profound iPad application by which the performance of the interface on the iPad is considered to be better. The iPad interface is implemented in order to verify the advantage in having a portable device during the programming of a task.

As described in section 6.5.2 each skill is implemented as a class containing a teaching and an execution routine. The purpose of having each skill in a separate class is to ease the implementation and devel-

---

<sup>1</sup>At the time of the initiation of the project in September 2011.

opment of a new skill, yet the menu set-up of the skill in either GUI or TUI is still integrated in the user interfaces. It can be discussed whether the menu set-up should have been implemented in the skill classes, while only a menu template should have been implemented in the user interfaces. In this way everything connected to a skill is collected in the skill class.

### **Industrial Assembly Tasks**

The developed system is tested in two industrial assembly tasks replicated from assembly tasks at Grundfos. In reference to the selected tasks it can be discussed whether the two industrial assembly tasks contain too similar components. Given a bigger variation in the objects further skills might have been identified. These two tasks have been used as they will be part of the TAPAS month 24 demonstration. The third task, the collection of the rotor caps at workstation 3 described in chapter 8, has not been programmed as the DCN to the Grundfos vision system is currently under development, see section 6.4. It is debatable whether this DCN could have been finished if the establishment of a communication protocol had been initiated earlier on in the project<sup>2</sup> The implementation of the vision system has been given a low priority as it is not considered to be an essential part of the skill-based approach and as the implementation of the vision system will not induce new challenges.

The vision of a Little Helper is presented in chapter 2. In this vision it is implied that setting up a new task should be intuitive and simple and thus it can be discussed whether the development of dispensers for magnets and pressure rings, see chapter 8 is within the scope of the vision. The ring dispenser is built from simple objects related to the SQFlex production, while the magnet dispenser is a simple construction in wood. It is estimated that the dispensers could have been built by the production operator and thus implementation of the dispensers is considered within the scope of the vision. Furthermore without the implementation of these dispensers a bin picking problem arises. Alternatively, the supply chain of the components could be altered to supply the components in an ordered fashion.

### **Graphical User Interface - GUI**

GUI is aimed at being an intuitive interface making the programming of industrial tasks available to an operator. Even though the implementation of a new task is configured in a user-friendly interface (GUI) it should be noticed that the teaching of a skill requires general knowledge about robotics. In this way the operator should be able to predict singularities, speed limits, torque limits, collisions, and joint limits. In order to simplify user input in GUI default values and standardisation of applications are established. The user inputs are essential to the robustness of a skill sequence and given that the user is uncomprehending to the given input an incorrect output appears. A potential solution to this problem could be a more thorough description of the user input or simply to create a system which itself could handle the assignment of values.

The configuration of a task in GUI is disadvantageous for a complicated task as it is difficult to predict the individual skills in the total sequence. Given this the industrial assembly tasks described in chapter 8 have not been programmed using GUI. In this way the user should be able to choose between the current configuration approach and the configuration approach used in TUI.

---

<sup>2</sup>Work on the Vision Control Node was initiated in April 2012.

---

## **Safety**

The implementation of a system controlling the safety has been disregarded in this project. It is debatable whether this should have had higher priority as it constitutes an essential part of a fully developed industrial system. The reason that safety has not been a main priority in the project is due to the research based progress of the project in which safety implementation prior to the system development would have been inconvenient. Furthermore the robot is operated in a controlled laboratory environment with full human supervision of its movements in contrast to autonomous operation in an industrial environment. Little Helper ++ has three emergency stops mounted on the exterior of the robot, but when the platform is moving it would be desirable to have a wireless emergency stop. A survey of existing products on the market has been made, yet a purchase has not been made as operating the platform autonomously has not been a key issue in this project.



# Conclusion 10

---

Throughout this project a mobile robot has been constructed, a skill-based system for operating the robot has been developed, and this system has been verified on two industrial assembly tasks. Furthermore an intuitive and user-friendly interface has been implemented. Below a conclusion of the goals listed in chapter 2 is given.

## **Construct a mobile robot with a KUKA LWR**

In chapter 3 it is described how the mobile robot Little Helper ++ has been designed and constructed. Little Helper ++ is based on the same main components as those of Little Helper +, yet it is designed with a more aesthetic exterior. The main components are the KUKA LWR manipulator, Neobotix platform, an aluminium frame structure, and the Schunk WSG50 electric gripper.

## **Develop a robot programming system based on skills and device primitives**

One of the key objectives through this project has been the development of a system incorporating the skill-based approach, *Skill-Based System*, for programming and executing tasks using Little Helper ++. A software system has been developed in C++ implementing the ROS architecture, and it consists of two control nodes, one for the manipulator and one for the gripper, and the central node called *Skill-Based System*. The definition of the system architecture and the development of *Little Helper System* are documented in chapter 6.

The development of this system has led to a standardisation of the software implementation of skills. Each skill is implemented with both a teaching routine and an execution routine, in which each skill also implements pre- and postcondition checks.

## **Verify the developed system on complex industrial assembly tasks**

The developed *Little Helper System* and the skill-based approach have been verified by programming and executing two complex industrial assembly tasks replicated from Grundfos using only the developed skills. A total of ten different skills has been developed, driven by the programming of the two tasks. This has served as a verification that industrial assembly tasks can be accommodated using a skill-based approach and that the developed teaching interface is sufficiently robust to set up an actual industrial task. Furthermore from the performed tasks the work of this master project will contribute to the TAPAS month 24 demonstration.

## **Develop a simple and intuitive interface for programming a new task**

To control *Skill-Based System* two different user interfaces have been developed, a terminal based interface and a graphical interface. The graphical user interface (GUI) creates an easy comprehensible

platform for using *Skill-Based System*, through which the programming and execution of an industrial task are brought to a level where robotics expertise is no longer needed. This is both due to GUI's intuitive and menu based interface and to the skill-based approach, which generalises task programming into object operations instead of robot motions.

An iPad has been integrated from which GUI can be accessed. This presents the user with a known platform by which the use becomes familiarised, and furthermore the iPad makes the software portable easing the teaching phase.

### Summary

In conclusion of this project the skill-based approach is found applicable. It presents a generalized approach to task programming which both benefits the software architecture of the system and forms a basis of moving the robot programming from the experts to the general operators. The skill-based approach has been verified in this project through the development of *Skill-Based System* used for both programming and executing industrial tasks. This system and the skill-based approach have proven to be applicable in the programming of complex industrial tasks, and furthermore the intuitive user interface has proven that programming complex tasks is comprehensible to people other than robotics experts.



*In this chapter a recommendation of the work toward the TAPAS month 24 demonstration is described, followed by a reflection on a future commercialisation of a Little Helper product.*

## 11.1 TAPAS Month 24 Demonstration

In October 2012 the TAPAS project must perform a month 24 demonstration at Grundfos. The focus of this demonstration will be on the three assembly tasks described in chapter 8, and consequently the work of this master project will form the basis of the month 24 demonstration. In this section the recommendations of the work towards the month 24 demonstration are described.

### Little Helper ++

Throughout this project various unknown issues regarding the control of the manipulator have been experienced. At the beginning of 2012 in cooperation with KUKA it was ascertained that joint 3 of the KUKA LWR had an unknown defect. To avoid interfering with the work of this project the manipulator is scheduled for repair in July 2012. After this repair it is suggested that a number of experiments are carried out to determine whether the performance has been enhanced.

During the project problems regarding the sensors installed on the platform have arisen. The problems are not described further, but must be attended prior to the month 24 demonstration.

During the month 24 demonstration Little Helper ++ will drive autonomously between the three workstations and therefore it is recommended to implement a wireless emergency stop.

### Little Helper System

In section 6.3 an expected architecture of the Platform DCN is presented and in section 6.4 an expected architecture of the Vision DCN is presented. As neither of these DCNs have been fully developed and integrated they must be attended prior to the month 24 demonstration in order to obtain a fully integrated control system.

In chapter 8 the communication to the Grundfos equipment is described, e.g. activating the hydraulic presses is only emulated. As the month 24 demonstration is conducted at Grundfos using the actual production equipment an interface towards the Grundfos OPC server is needed. Such an interface could be implemented as a DCN, but as the mission planner already is connected to the Grundfos OPC server it could be used as a "gateway" for sending and receiving simple signals.

### **Programming the Tasks**

Even though the workstations at AAU are replications of the workstations at Grundfos it is not likely that the programmed tasks can be executed directly at Grundfos. Instead it is recommended to reprogram both tasks. In order to accommodate this programming the current approach of GUI is considered inconvenient as specification of the entire task is required prior to teaching the skills. It is recommended to alter the GUI interface to implement an approach similar to that of TUI, where the sequence is extended concurrently with the teaching of the skills. Otherwise TUI could be used for the reprogramming of the tasks.

During the programming of the tasks it is recommended to implement *MoveTo* skills between the object manipulating skills in order to ensure that the manipulator is moved collision freely.

To ensure a robust execution of the tasks at Grundfos it is recommended to be thorough and accurately in the teaching of the *Calibrate* skill.

### **Mission Planner**

During the month 24 demonstration a mission planner will control the work of Little Helper ++ on task level. The development of this mission planner is conducted by (Dang, 2012). As the mission planner is being developed as a separate system an interface between *Little Helper System* and the mission planner must be established and implemented prior to the month 24 demonstration.

### **Error Handling**

In this project error handling during the execution of a task has not been implemented. This is not considered essential to implement prior to the month 24 demonstration, although some error feedback to the mission controller is expected in order to keep track of the production progress. This should be implemented in the task layer, see figure 6.7. As the task layer is currently implemented as a simple macro of skills the implementation of error handling might require the development of a more active and intelligent task layer.

### **Environment**

In order to operate the Neobotix platform in the production environment at Grundfos boards to enclose the working area must be incorporated. Furthermore it is important to ensure a clean surface of the floor to ensure that the wheels of the platform can obtain traction.

The three workstations at Grundfos must be altered to incorporate dispensers and fixtures similar to those used in this project, see chapter 8.

## **11.2 Commercialisation**

A future commercialisation of Little Helper ++ or a descendant of it gives rise to several issues that are found essential to the industrial implementation of Little Helper ++.

### **Intelligent Motion Planning**

In chapter 6 the development of an intuitive interface is described. This interface, along with the skill-based approach, brings industrial task programming to a level where it no longer requires robotics exper-

tise, though issues regarding collisions, software limits, and singularities still emerge during the teaching of a task. As these issues complicate the teaching phase and are considered out of scope for an industrial operator it is recommended that an intelligent motion planner is implemented. This motion planner should be integrated as a functionality in the skill layer or device primitive layer. Supplying the motion planner with an environment model, e.g. CAD models or models obtained using a stereo vision camera, is considered necessary in order to avoid collisions.

### **Robustness**

During this project several issues regarding the reliability and accuracy of the KUKA LWR have been experienced. Whether these issues are due to the unknown defect in joint 3 or due to the fact that the KUKA LWR is a manipulator in development are uncertain. The reliability and robustness of a commercial robot rely upon the reliability and robustness of the individual components. It is recommended to verify the general robustness and reliability of a Little Helper robot prior to release, as the reliability and robustness experienced throughout this project are found insufficient.

### **Vision System**

It is considered desirable to implement a vision camera mounted on the gripper. A vision camera could be used both to check pre- and postcondition checks but also in the execution and teaching of a skill. In the future the specification and teaching of a skill might be expanded to include simple voice or vision gestures.

### **Mission Layer**

In extend of the layers described in chapter 6 a commercialised mobile robot should furthermore include a mission layer. The main functionality of the mission layer is the mission set-up, planning, and control. Furthermore integration and interaction with the surrounding environment and equipment are controlled through this layer.

### **Safety**

Safety is considered to be an essential part of a mobile robot operating autonomously, although safety has not been addressed in this project. The platform uses the embedded laser scanners and ultrasonic sensors to avoid collision with obstacles including humans, but when operating the manipulator, no safety is implemented. Two independent solutions to this safety issue have been devised. Using the sensors of the platform the system monitors any moving obstacles and thus a potential human. If any moving obstacle comes within a given outer perimeter of the robot a *safe mode* is activated, and thus the manipulator's velocity is decreased. If a moving obstacle comes even closer and within the work envelope of the manipulator the operation is stopped or paused.

In continuation of the described approach, the *safe mode* could implement a surveillance of the force measurements of the manipulator. If any measured force exceeds a given maximum not expected according to the task, this could be a potential collision and thus the manipulator should stop.

### User Interface

The user interface GUI described in chapter 6 creates an intuitive interface and thus it is considered to significantly ease the programming of an industrial assembly task. This interface is aimed at task level programming and with minor aspects of the mission control. In a commercial product a complete interface would have to be developed, where managing mission, device set-up, navigation, environment interaction etc. is possible. In such an interface the developed *Little Helper System* is considered to be a central part, however in a further developed version.

Having a complete system interface while still being dependent on the teach pendant to control the KRC is considered inconvenient. It is recommended to integrate a new controller for the manipulator or to remove the dependency of the teach pendant. The purpose is to integrate the entire control of the system into a single interface as the JAVA-API under development by KUKA.

The interfaces developed in this project, hence GUI and TUI, are based on physical interaction to teach a skill. The interface of a commercial robot would incorporate several options to set up a task, e.g. the option to use vision gestures, voice gestures or a not even identified interaction method.

# Bibliography

---

- Bigum, J. (2012). *Vision Engineer - M.Sc.E.E.* GRUNDFOS Management A/S.
- Craig, J. J. (2005). *Introduction to Robotics - Mechanics and Control* (3rd ed.). Pearson Education International. ISBN: 0-13-123629-6.
- Dang, V. Q. (2012). *Ph.d.-student*. AAU. Department of Production and Mechanical Engineering.
- Dr. Viorica Frunza (Used: January 04, 2012). *FLEXIBLE MANUFACTURING SYSTEMS (FMS)*. University of Kentucky. URL: <http://www.uky.edu/~dsianita/611/fms.html>.
- Hildebrandt, S. (Used: May 16, 2012). *iPad styrer vaks lille industrirobot*. Videnskab dk. URL: <http://videnskab.dk/teknologi/ipad-styrer-vaks-lille-industrirobot>.
- KUKA Roboter GmbH (2010). *Light Weight Robot 4+*. On the enclosed Appendix CD </Documents/Assembly\_instructions\_LWr4-plus.pdf>.
- KUKA System Software (2003). *KRC2/KRC3 Expert Programming* (Release 5.2 ed.). KUKA Robotics GmbH.
- KUKA System Software (2011). *KUKA.FastResearchInterface* (1.st ed.). KUKA Robotics GmbH.
- M. Hvilshøj, S. Bøgh, O. S. Nielsen & O. Madsen (2011). *Autonomous Industrial Mobile Manipulation (AIMM) - Past, present and future*. On the enclosed Appendix CD </Documents/Autonomous Industrial.pdf>.
- METAL SUPPLY (Used: April 13, 2012). URL: <http://www.metal-supply.com/announcement/view/7193/kuka-roboter-is-presenting-a-new-robot-generation-at-automatica-2010>.
- Montgomery, D. C. (2005). *Design and Analysis of Experiments* (6th ed.). John Wiley & Sons inc.
- Nielsen, O. S. (2011). *Software developer*. Universal Robots ApS.
- Nokia (Used: April 4, 2012). *Qt Creator* (2.4.1 ed.). Nokia Corporation. URL: <http://qt.nokia.com/>.
- Pedersen, M. R. (2011). *Integration of the KUKA Light Weight Robot in a mobile manipulator*. Aalborg University. Master's Thesis.
- Pedersen, M. R. (2012). *Ph.d.-student*. AAU. Department of Production and Mechanical Engineering.
- R. E. Walpole, R. H. Myers, S. L. Myers & K. Ye (2007). *Probability & Statistics for Engineers & Scientists* (8th ed.). Pearson Education International. ISBN: 0-13-204767-5.
- ROS Wiki (Used: November 2011 - May 2012). *Documentation - ROS Wiki*. URL: <http://www.ros.org>.

- S. Bøgh, M. Hvilshøj, C. Myrhøj & J. Stepping (2008). *Fremtidens produktionsmedarbejder - udvikling af mobilrobotten lille hjælper* (1st ed.). Aalborg University. Master's thesis.
- S. Bøgh, O. S. Nielsen, M. R. Pedersen, V. Krüger & O. Madsen (2011). *Does Your Robot Have Skills?* Dept. of Mechanical and Manufacturing Engineering, Aalborg University.
- TAPAS Community (2009). *TAPAS in a nutshell*. KUKA Roboter GmbH. On the enclosed Appendix CD </Documents/TAPAS in a nutshell.pdf>.
- TAPAS Description (2010). *TAPAS Description Annex I - "Description of Work"*. Official TAPAS letter of application. On the enclosed Appendix CD </Documents/TAPAS letter of application annex I.pdf>.
- Weiss Robotics GmbH (2011). *WSG Command Set Reference Manual* (Firmware 2.2.0 ed.). Weiss Robotics GmbH. On the enclosed Appendix CD </Documents/WSG50\_Command\_Set\_Reference\_Manual.pdf>.
- Youtube (Created: September 26, 2011). *VT3AAU2011*. Christian Carøe, Mikkel Hvilshøj, Casper Schou. URL: <http://www.youtube.com/user/VT3AAU2011?feature=mhee>.

# Definitions 12

---

*This chapter presents a list of abbreviations and terms used in this report.*

## **AIMM**

Autonomous Industrial Mobile Manipulator.

## **Compliance control**

The KUKA LWR's ability to act like a mass-spring-damper system. This control mode is also used to make the manipulator movable by hand by setting all spring coefficients to zero.

## **DCN**

Device Control Node: Software node for handling the communication to a device.

## **Device primitive**

A basic motion or functionality of a device.

## **FRI**

Fast Research Interface to communicate with the KUKA LWR.

## **Gripper**

The Schunk WSG 50 electric gripper.

## **GUI**

The developed Graphical User Interface to *Little Helper System*.

## **KRC**

KUKA Robot Controller.

## **KRL**

KUKA Robot Language.

## **KUKA LWR**

The KUKA Light Weight Robot.

### **Little Helper**

1<sup>st</sup> generation developed in 2007 by (S. Bøgh, M. Hvilshøj, C. Myrhøj & J. Stepping, 2008).

### **Little Helper +**

2<sup>nd</sup> generation developed in 2011 by (Pedersen, 2011).

### **Little Helper ++**

3<sup>rd</sup> generation developed in this project in 2012.

### **Little Helper System**

The entire software system of Little Helper ++, hence *Skill-Based System* and the DCNs.

### **Manipulator**

The KUKA Light Weight Robot.

### **MMI**

Man-Machine-Interface.

### **Operator**

A production employed person with no robotics knowledge.

### **Position control**

Traditional control mode of a manipulator used for moving to specific coordinates.

### **Pre- and postcondition check**

Verifies the input state and the output state of a skill.

### **Programming**

Specifying and teaching of a skill.

### **Robot**

The entire Little Helper ++.

### **Robot macro**

A sequence of device primitives based on 3D coordinates.

### **ROS**

Robot Operation System.



---

**Skill**

An intelligent sequence of device primitives applied on objects.

**Skill-Based System**

Main control system of Little Helper ++, hence it controls the DCNs to the hardware devices.

**Specifying**

The part of the programming, where the user specifies a number of parameters.

**Task**

A sequence of skills or robot macros.

**TCP**

1. Tool Center Point of the manipulator.
2. Transmission Control Protocol (Ethernet).

**Teaching**

The part of the programming, where the user interacts physically with the manipulator.

**TUI**

The developed Terminal User Interface to *Little Helper System*.

**UDP**

User Datagram Protocol (Ethernet).

**Wrapper**

A simple convenient function which wraps the ROS Action call.



# Appendix



# Programmed Tasks A

---

*The purpose of this chapter is to describe the programmed tasks that have been conducted throughout the project. The conducted tasks are the LEGO DUPLO assembly task, see appendix A.1, the SQFlex rotor assembly task, see appendix A.2 and other programmed tasks which are not described in such detail, see appendix A.3. The tasks are conducted with the KUKA LWR mounted on a table and using pneumatic tools.*



*LEGO DUPLO Assembly*

## A.1 LEGO DUPLO Assembly Task

A thoroughly description of the DUPLO task is found on the enclosed Appendix CD in </Task descriptions/DUPLO Description.pdf>. A video of the LEGO DUPLO Assembly Task is obtained by scanning the QR-code (*LEGO DUPLO Assembly*). The task has been conducted in September 2011.

### A.1.1 Purpose of the Task

The purpose of the LEGO DUPLO Assembly is to attain competences in using the KUKA LWR for simple tasks and furthermore to attain competences in programming the task by means of the KRL programming language. Specific competences in using the compliance modes build into the KUKA LWR are needed. Another purpose is to program the task in conventional robot programming in KRL and then convert the programming into classic robot macros and device primitives.

### A.1.2 Competences

It is expected to achieve the following competences through programming the LEGO DUPLO Assembly Task:

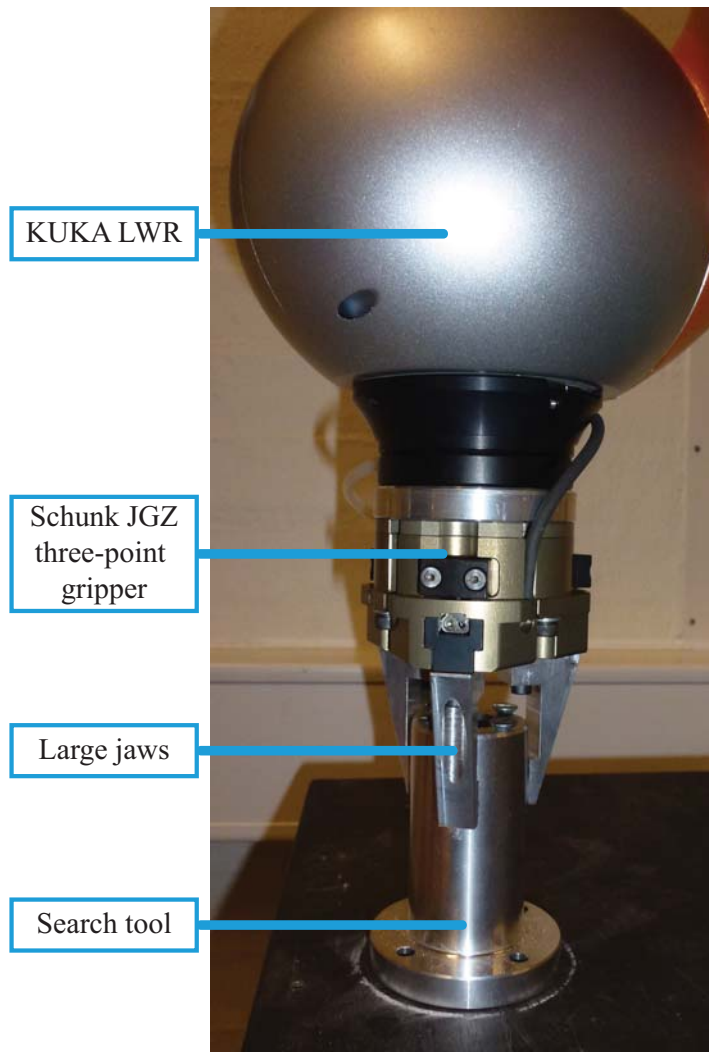
- Programming in KUKA Robot Language (KRL)
- Define and program device primitives
- Define and program robot macros
- Use the KUKA LWR's torque sensors
- Use the compliance control of the KUKA LWR
- Verify the fundamental idea of the skill-based approach

### A.1.3 Description

The objective of the task is to search for a DUPLO brick in a certain search area. Once the brick's center point is known, the brick is picked up and placed on a LEGO DUPLO plate. The brick must be placed correctly according to the knobs on the plate.

Since a three-point-gripper is mounted on the manipulator, a search tool is used to locate the brick more accurately. A circular search tool is used as shown in figure A.1, which is picked up at a specific location. After picking up the search tool the LWR searches for the randomly placed DUPLO brick on the DUPLO plate as shown in figure A.3.

The search is carried out using the torque sensors in the manipulator to "sense" contact. The manipulator moves to a specified starting coordinate. The LWR searches in the y-direction until a specified force is exceeded, which is interpreted as contact with the brick. The location is stored and the manipulator starts a search in the x-direction. The dimensions of the brick is known, so the search in the x-direction is started from the center of the brick. As a specified force is exceeded, contact with the brick is made, and the x-coordinate is stored. The same approach is used to locate the height of the brick.



**Figure A.1:** The three-point gripper on the KUKA LWR holding the circular search tool.

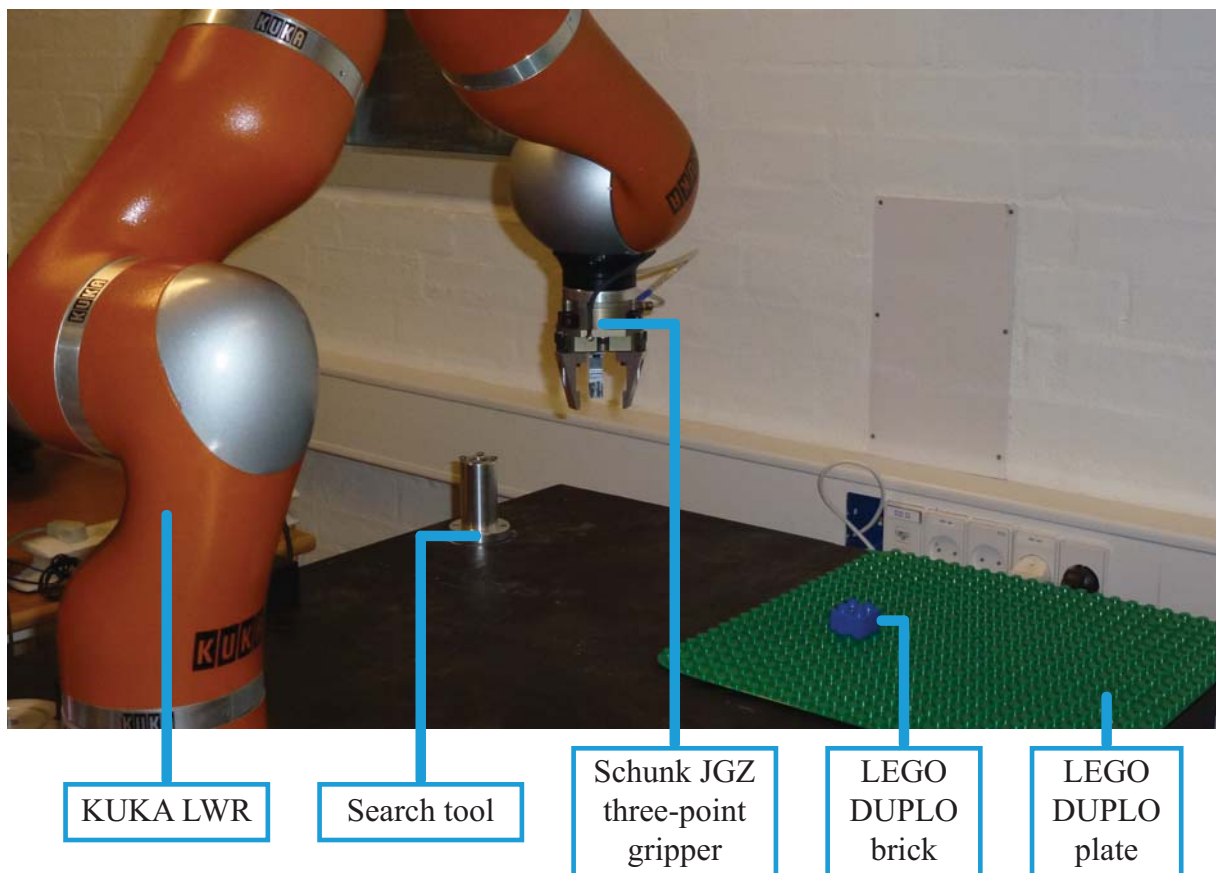
As a result of the searching algorithm the coordinate of the center point of the brick is found. The search tool is placed at the same prespecified location. The brick is picked up and the manipulator moves to a predefined position above the DUPLO plate. A search downwards in the z-direction is carried out. Once contact is made with the plate, the manipulator is searching for correct alignment between the knobs on the plate and the brick. The pattern for this search is a square movement that increases in size for every

revolution. During this search the contact force between the plate and brick is measured. Once it drops, the brick is pushed down into place. Again a search algorithm is used in order to make sure that the brick is pushed correctly into place.

#### A.1.4 Set-up and Equipment

In figure A.2 the set-up of the LEGO DUPLO Assembly Task is shown. The DUPLO brick is randomly placed on the plate. The following equipment is used:

- KUKA Light Weight Robot
- Schunk JGZ 64-1 308920 three-point gripper
- LEGO DUPLO brick in dimensions 2 x 2 studs
- LEGO DUPLO plate in dimensions 24 x 24 studs
- Circular search tool with search diameter 63 mm

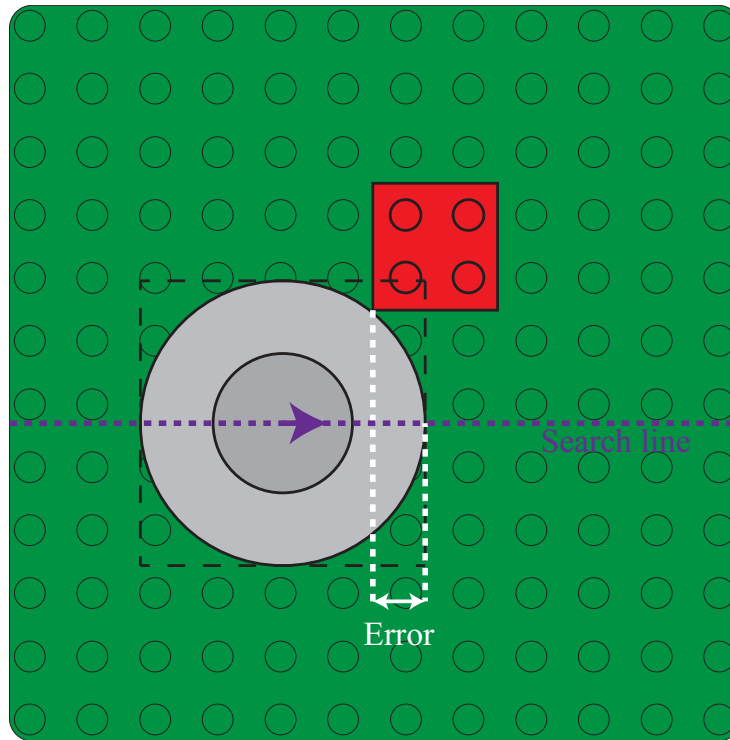


*Figure A.2: The set-up of the LEGO DUPLO Assembly Task.*

Since the search tool is circular the manipulator searches with a circular search face. Unless contact with the brick is made in line with the search line, the stored coordinate is inaccurate. This is shown in figure A.3. A square or rectangular search tool would eliminate this problem<sup>1</sup>, but was not available at

<sup>1</sup>As long as the contact area is parallel to the brick

the time. A square search tool could be made, but since it is not critical for the purpose of achieving the competences no further action is taken.



**Figure A.3:** The disadvantage of the circular search tool contra a rectangular search tool if the brick does not lie directly on the search line.

### A.1.5 Task - Conventional Programming

The sequence of conventional programming of the task is listed below. The sequence is described in words instead of the actual KRL code. This is done to make it easier to read and comprehend. The actual KRL code requires 246 lines to complete the task. The KRL sequence has been described in words which follows below:

Go to home  
 Move to search tool location + 50 mm in the z-direction  
 Move 50 mm down in the z-direction  
 Close gripper  
 Move 20 mm up in the z-direction  
 Move to predefined search start coordinate in the y-direction  
 Activate interrupt Y and move 200 mm in the y-direction  
 At force contact above 3 N stop and store the position  
 Move 5 mm back in the y-direction  
 Move 75 mm in the x-direction  
 Move 52.5 mm in the y-direction  
 Activate interrupt X and move 200 mm in the x-direction  
 At force contact above 3 N stop and store the position  
 Move 5 mm back in the x-direction  
 Move 100 mm up in the z-direction



Move to the calculated center of the LEGO brick in the x- and y-direction  
Activate interrupt Z and move 200 mm in the z-direction  
At force contact above 3 N stop and store the position  
Move 20 mm up in the z-direction  
Move to search tool location + 50 mm in the z-direction  
Move 50 mm down in the z-direction  
Open gripper  
Move 100 mm up in the z-direction  
Move to home  
Move to stored location of the LEGO brick + 20 mm in the z-direction  
Move 84 mm down in the z-direction and wait 0.2 sec  
Close gripper and wait 0.2 sec  
Move 100 mm up in the z-direction  
Move to a predefined coordinate on the LEGO plate + 50 mm in the z-direction  
Activate interrupt Z and move 200 mm in the z-direction  
At force contact above 5 N stop and store the position  
Move 20 mm up in the z-direction  
While the force has not dropped more than 5 N  
Move  $0.1 \cdot n$  mm in the y-direction  
Move  $0.1 \cdot n$  mm in the x-direction  
Move  $-0.2 \cdot n$  mm in the y-direction  
Move  $-0.2 \cdot n$  mm in the y-direction  
Update n  
End while  
Move 6 mm down in the z-direction and wait 0.2 sec  
Open gripper  
Move 50 mm up in the z-direction  
Move to home

### A.1.6 Task with Classic Robot Macros

In order to understand the definition of skills from chapter 5 it is chosen to implement device primitives and classic robot macros on the KRC. In this way the actual KRL code has been analysed in order to identify similar motions and operations.

#### Device Primitives

The analysis has revealed the use of five different motion types or device primitives as shown below.

- MoveXYZ
- MoveRel
- SearchRel
- OpenGripper
- CloseGripper

The device primitives are programmed in individual programs in KRL, which can be called from other programs such as classic robot macros. The device primitives are generalised so that all parameters are variable and thus is given as input to the device primitive.

### Classic Robot Macros

The analysis of the KRL code has identified four different robot macros. These are listed below.

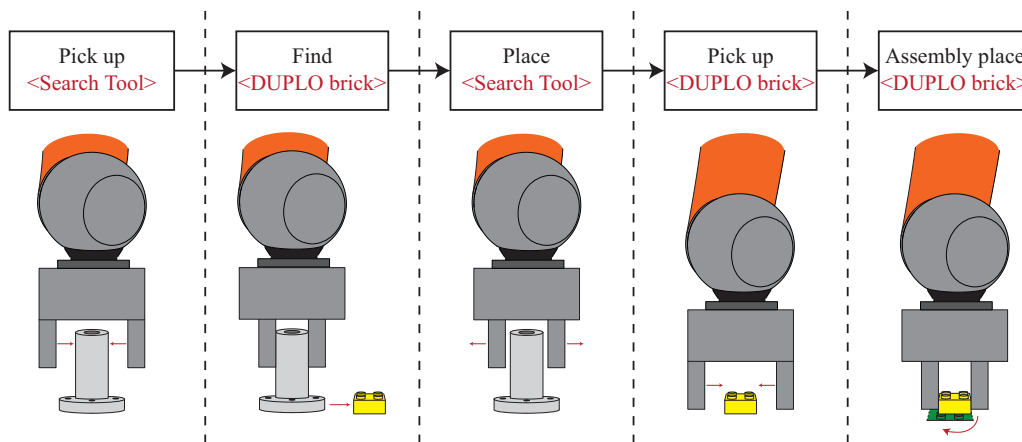
- Pick up
- Find
- Place
- Assembly place

A description of the individual classic robot macros is found in chapter 5. The classic robot macros consist of a number of device primitives and are implemented on the KRC.

### The task

The task sequence is established from the identified robot macros. The sequence of the robot macros is listed below and is shown in figure A.4.

1. Pick up <Search Tool>
2. Find <LEGO DUPLO brick>
3. Place <Search Tool>
4. Pick up <LEGO DUPLO brick>
5. Assembly place <LEGO DUPLO brick>



*Figure A.4: The task sequence containing the identified robot macros.*

The actual code in KRL contains 10 lines of code of which only 5 lines are the actual sequence of robot macros. Furthermore it is necessary that the task contains five predefined coordinates which are stored in the task program in KRL.

#### A.1.7 Conclusion of the LEGO DUPLO Assembly Task

The programming of the LEGO DUPLO Assembly Task has given competences in programming in KRL. Competences have been achieved in using the integrated torque sensors to interrupt a motion as the manipulator collides with an obstacle. In order to understand the definition of skills from chapter 5 it has been chosen to implement device primitives and classic robot macros. By analysing the task sequence a number of device primitives and robot macros has been identified. The introduction of robot macros eases and expedites the programming and the task specific code is reduced from 246 lines of code to 10 lines of code of which only 5 lines are the actual

sequence. By doing this competences in building and using general parameter based device primitives and classic robot macros have been achieved.



*Rotor Assembly in KRL*

## A.2 SQFlex Rotor Assembly

The KRL code is found on the enclosed Appendix CD in </Task descriptions/SQRotorAssemblySRC.pdf>. A video of the SQFlex rotor assembly is obtained by scanning the QR-code (*Rotor Assembly in KRL*). The task was conducted in November 2011. A replication of the workstation from Grundfos has been made, see figure A.5.

### A.2.1 Purpose of the Task

The purpose of this task is to use robot macros to program an industrial assembly task. The goal is to document that the assembly of a SQFlex rotor is actually possible by using the KUKA LWR without changing the workstation or assembly operation. Furthermore, it is the purpose to document and test the preprogrammed robot macros and device primitives on an industrial task.

### A.2.2 Abstract

In this chapter the programming of the assembly of a SQFlex rotor from Grundfos is described. The set-up includes a reconstruction of the workspace at Grundfos and a copy of the fixtures used in the production. The sequence of the assembly has been slightly changed to accommodate the use of a manipulator and likewise have the feeding of two of the components. The task has been programmed using device primitives and robot macros, resulting in 67 lines of code in the main program. The assembly task takes 30 seconds for an operator, while it takes 4:47 minutes for the robot. It is possible to reduce the assembly time of the manipulator by e.g. placing the components more appropriate. The programming of this task demonstrates that it is possible to do a manual assembly task using the KUKA LWR.

### A.2.3 Competences

It is expected to achieve the following competences through the SQFlex rotor assembly:

- Solve an industrial assembly task
- Programming in KUKA Robot Language (KRL)
- Define and program new and more complex device primitives
- Define and program new and more complex robot macros
- Use of the KUKA LWR's compliance mode

### A.2.4 Set-up and Equipment

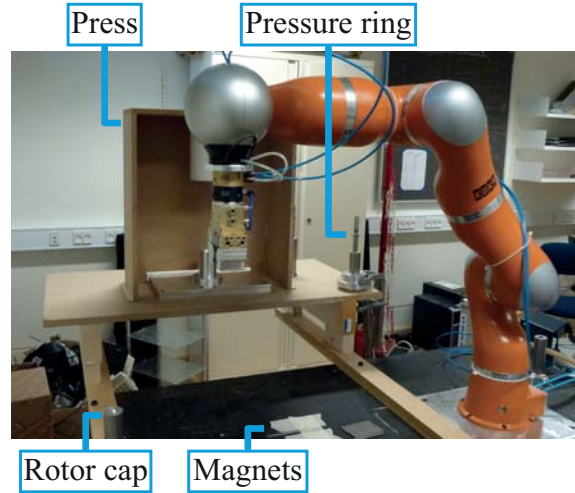
The set-up for this task is shown in figure A.5.

The equipment used is:

- KUKA Light Weight Robot
- Sommer pneumatic parallel gripper
- SMC pneumatic parallel gripper with 120 mm long custom made jaws.
- Schunk SWA pneumatic tool changer
- Reconstruction of the workstation at Grundfos including fixtures from Grundfos.
- Components of the rotor assembly, see figure A.6.

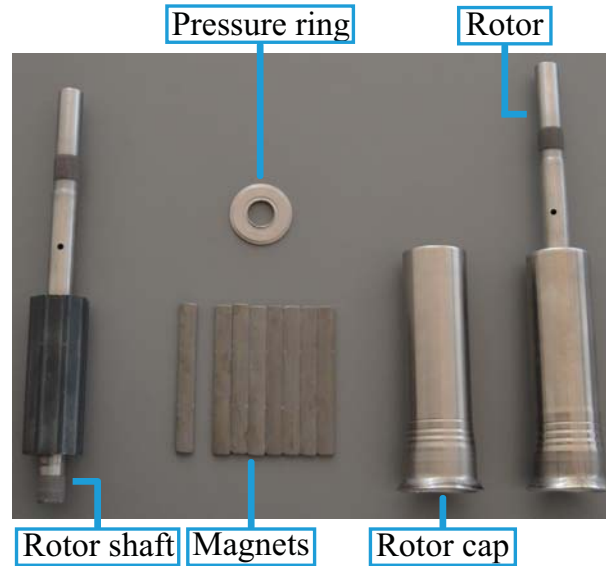


(a) Actual workspace at Grundfos.



(b) Reconstruction of the actual workspace at Grundfos.

**Figure A.5:** Setup for the SQFlex rotor assembly.



**Figure A.6:** Components used in the rotor assembly.

### A.2.5 Task description

The components of the rotor are assembled inside the fixture and subsequently the components are pressed. Afterwards a measurement to validate the tolerances of the rotor is performed. As a result of this measurement the rotor is either discarded or accepted. The manual assembly sequence is found on the enclosed Appendix CD in [Documents/Instruction- SQFlex production.pdf](#). The manual assembly sequence is as follows:

1. Place the pressure ring on the rotor shaft.
2. Place the rotor shaft with the ring in the fixture.
3. Place the eight magnets on each side of the octagonal rotor core in the fixture.
4. Place the rotor cap on top of the fixture.
5. Use both hands to activate the press.
6. Remove the rotor.
7. Measure the rotor from the rotor end to the rotor cap using a vernier caliper.

Once fitted to the rotor shaft the pressure ring is not held into place and the operator must fixate both the pressure ring and the rotor shaft when placing them on the fixture. As this is not possible with a manipulator without specialised jaws, it is chosen to place the ring in the fixture first, and then afterwards place the rotor shaft in the fixture. This means that the sequence of the rotor assembly is as follows:

1. Place the pressure ring in the fixture.
2. Place the rotor shaft in the ring in the fixture.
3. Place the eight magnets on each side of the octagonal rotor core.
4. Place the rotor cap on top of the fixture.
5. Send a signal to activate the press.
6. Remove the pressed rotor
7. Measure the rotor from the rotor end to the rotor cap using the gripper.

In the assembly of the rotor by a Grundfos operator the magnets and pressure rings are placed in either piles or bins. As the current set-up does not involve a vision system, picking from these piles or bins is not possible. Therefore the ring has been placed at a known location, and the magnets are fed one at a time at a known location.

During the programming of this task several techniques for programming have been tried. It has been found that the fastest and easiest way of programming the task is to use the gravity compensation mode of the KUKA LWR to quickly pilot the manipulator to a given coordinate. If better precision for a coordinate is needed, the manipulator is set in position control and jogged to the exact coordinate. After all coordinates have been stored the sequential building of the program using robot macros and device primitives is carried out. This method requires an extensive description and sequential plan of the task.

### A.2.6 Conclusion of the SQFlex Rotor Assembly

Programming this task demonstrates that it is possible to conduct a manual industrial assembly task using the KUKA LWR. It has been necessary to feed the magnets and the pressure ring individually at known locations. Even though a vision system might be added to the set-up later on, picking objects from random orientations would increase the cycle time. Instead a specialised non actuated dispenser could be integrated. Implementing such a device would still make the assembly possible for an operator.

The task has been programmed using device primitives and robot macros, which has proven to ease the programming. The main program consists of 67 lines of code build from 5 different robot macros and 12 different device primitives. The programming time of the task has not been measured.

The cycle time of the task is measured to 4:47 minutes, while an operator at Grundfos completes the task in 30 seconds which is nine time faster. It should be noted, that no actions towards refining or optimizing the programmed task have been taken, which could lead to a decrease in the cycle time of the manipulator.

## A.3 Other Programmed Tasks

This section gives a short description of the other tasks that have been conducted during the project. The general purpose of these tasks has been to obtain competences in using the KUKA LWR.

### Opening of Beer Can



Opening of Beer Can

The beer task has been programmed using the function *RecordTrajectory*, which records the TCP's coordinate every given time interval. When the task is executed the manipulator moves through the recorded trajectory. The programming time is approximate two minutes. The source code for the task is found on the enclosed Appendix CD

in </Task descriptions/PlayTrajectorySRC.pdf> and in </Task descriptions/RecordTrajectorySRC.pdf>. A video of the opening of beer can is obtained by scanning the QR-code (*Opening of Beer Can*). The task was conducted in September 2011.



*Figure A.7: The set-up for the conducted beer task. The plastic cup functions as a fixture for the beer can.*



*Picture Drawing*

### Picture Drawing

The drawing task has been programmed using the function *RecordTrajectory*. In this task the recorded trajectory draws a picture on a piece of paper. The source code for the task is found on the enclosed Appendix CD in </Task descriptions/PlayTrajectorySRC.pdf> and in </Task descriptions/RecordTrajectorySRC.pdf>. A video of the picture drawing is obtained by scanning the QR-code (*Picture Drawing*). The task was conducted in September 2011.



*Figure A.8: The set-up for the drawing task.*



*Cranfield Benchmark*

### Cranfield Benchmark

The Cranfield benchmark is conducted in order to verify the *pick* and *place* robot macros. A sequence of 28 *pick* and *place* robot macros have been programmed in order to solve this task. The source code for the task is found



on the enclosed Appendix CD in [Task descriptions/CranfieldBenchmarkSRC.pdf](#). A video of the execution of the programmed task is obtained by scanning the QR-code (*Cranfield Benchmark*). The task was conducted in November 2011.



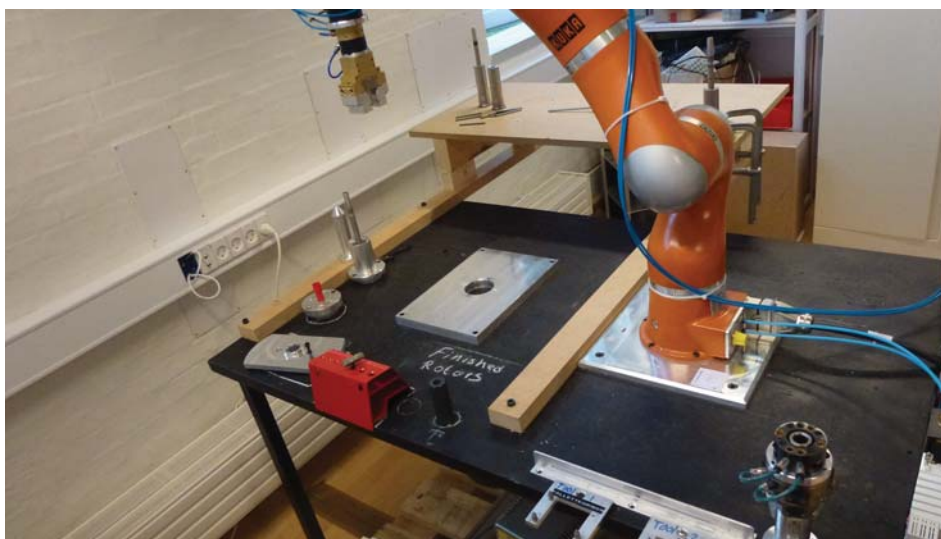
*Figure A.9: The set-up of the Cranfield benchmark.*



*SQ Fixture Assembly*

### SQFlex Fixture Assembly

In this task the fixture for the SQFlex press is assembled using device primitives and robot macros in KRL. The purpose of this task is to verify the KUKA LWR and the skill-based approach on another task than the SQ Rotor Assembly, see appendix A.2. To accomplish this task the compliance mode and torque sensors are used widely. There is not taken any actions towards time optimisation during this task. The source code of the task is found on the enclosed Appendix CD in [Task descriptions/SQPressSet-upAssemblySRC.pdf](#). A video of the assembly of the SQ fixture is obtained by scanning the QR-code (*SQ Fixture Assembly*). The task was conducted in November 2011.



*Figure A.10: The set-up for the SQ Fixture Assembly.*

### Intuitive Teaching of Pick and Place

A simple task containing a pick and a place operation is programmed through an intuitive teaching interface, where the user interacts with the manipulator physically. The purpose of this task is to verify a simple push-gesture based interface in order to obtain experiences aimed at the development of a user interface. This task has been programmed from an external computer. The source code of the task is found on the enclosed Appendix CD in </Task descriptions/Interactive teaching of pick and place SRC.pdf>. A video of the intuitive teaching of a pick and a place operation is obtained by scanning the QR-code (*Pick n Place*) and QR-code (*Pick n Place 2*). The task was conducted in December 2011.



*Pick n Place*



*Pick n Place 2*

**Figure A.11:** A user physically interacting with the KUKA LWR.



# Robot Operating System B

---

*The following appendix is a general introduction to the Robot Operating System (ROS) that is used in this project. This appendix is based on (ROS Wiki, 2012).*

ROS is a free software framework intended for the robotics domain. It is open source and developed ROS software is easily shared via the ROS webpage, (ROS Wiki, 2012). ROS is primarily used within the robotics research throughout the world.

The main purpose of ROS is to support reuse of code in robotics research and development. ROS is designed to be easy to use with other robot software frameworks and is language independent so it is easy to implement with e.g. C++ or Python. The fundamental idea is, that software is organized in relevant packages and each piece of software is executed as a separate runtime process. The ROS platform supplies an easy comprehensive interface for communication and data transfer between these separate processes.

In ROS there is three levels of concepts, the ROS file system level, the ROS computation graph level and the ROS community level.

## B.1 ROS File System Level

The ROS file system level are ROS resources used on files and folders. Using the ROS framework software is organized in separate packages. A package contains the source files of the specific software, the compiled executable files (ROS runtime processes (nodes)), a ROS-dependent library, datasets, configuration files, message files or anything else that is usefully organized together. An example of a ROS Package is found on the enclosed Appendix CD in `</Software/ROS/wsg_communication>`.

Besides the ROS Package the ROS file system level includes:

- Package: A package contains a certain software collection often containing an executable node, e.g. *wsg\_communication*.
- Manifest: A manifest provides metadata (data about data) about packages, e.g. license information, dependencies, and language information about the compiler.
- Stack: Stacks are collections of packages that gives a combined functionality, e.g a collection of all relevant packages of *Little Helper System*.
- Message file: A message file define the data structure of a messages used in ROS, this could be in a ROS Topic.
- Service files: A service file defines the data structure associated with a service in ROS.

## B.2 ROS Computation Graph Level

The ROS computation graph level is the level that makes communication and data transfer between separate processes possible. It is a peer-to-peer network where the communication and data processing between processes are controlled by the ROS Core, which is the main controller of the ROS computation graph level.

In order to communicate and process data between processes via the ROS peer-to-peer network the implementation of ROS specific code into the source code of the software is required. It is also due to this generalized interface between processes, that sharing and implementing software (packages) are rather straight forward.

Below the different types of concepts that are used in the basic computation graph is listed:

- **Node:** Nodes are processes that can perform computation, hence an executable node. A node could e.g. control the vision system or the movable platform.
- **ROS Core:** The ROS Core monitors and facilitates the data exchange between running nodes and thus it is the backbone of the ROS computation graph level.
- **Parameter server:** A parameter server is used by the nodes to store and retrieve static parameters at runtime such as configuration parameters.
- **Message:** A message is defined by the message file from the ROS File System Level. A message is a collection of data being transferred between nodes.
- **Topic:** A ROS Topic is used to route messages between nodes. The Topic acts like a "cloud" or buffer to which one node will publish messages and another node will subscribe. A topic can have many different publishers and subscribers.
- **Service:** A service defines an interaction between nodes. One node will offer a service, hence the server, which another node will use, hence the client.
- **Service server:** A server offers a given service and awaits the request from any service client.
- **Service client:** A client makes use of a service offered by a service server.

### B.2.1 ROS Action

The ROS Action service is specific ROS Service, and thus it has a server and a client. The ROS Action Service is defined by a goal, a feedback, and a result. A service file, see appendix B.1, defines the data structure of the Action service and thus defines the data structure of the goal, feedback and result.

- **Goal:** A goal is a message send from the ROS Action Client to the ROS Action Server containing the data needed for the service.
- **Feedback:** The feedback is used to give the ROS Action Client repeated information about a service in progress e.g. where the platform is located to a given time.
- **Result:** A result defines the outcome of a service and thus it is only send once from the ROS Action Server to the ROS Action Client.

### B.3 ROS Community Level

The ROS Community Level is used to share new software and knowledge, thus it is primarily the ROS webpage (ROS Wiki, 2012). Developed software can be downloaded and new software can be added onto this webpage. The fact that software is run as separate runtime processes and that communication between them are generalised in the ROS framework makes implementing a downloaded software package rather simple.

*The following chapter is based on practical work performed in January 2012. Initially the chapter introduces a documentation of an inaccurate force measurement in the TCP due to motion of the manipulator followed by an experiment of the performance of the interrupt function in KRL and on an external computer. Raw data and processed data are available on the enclosed Appendix CD in </Experiment/>. Please note that the experiments conducted and the results obtained in this appendix are not aimed at a thorough statistical analysis but are simply to illustrate the problems in question.*

## C.1 Inaccurate Force

The KUKA LWR has integrated torque sensors in every joint as mentioned in chapter 4. In this section the use of the compliance control will be verified for use in real-time problems. Through different tasks a problem regarding an inaccurate and unstable force measurement as a result of movement has occurred. An experiment will be carried out to document the force inaccuracy.

### C.1.1 Purpose

The purpose of this section is to document that the force inaccuracy is a function of translation and thus has impact on the ability to sense collision with objects.

### C.1.2 Abstract

In this section the use of the compliance control has been verified. The experiment shows that movement of the manipulator has a critical impact on the measurement of the force in the TCP. The implication of this is that the ability and the standardization of detecting collision with an object are deteriorated.

### C.1.3 Competences

Through the experiment it is expected to achieve the following competences:

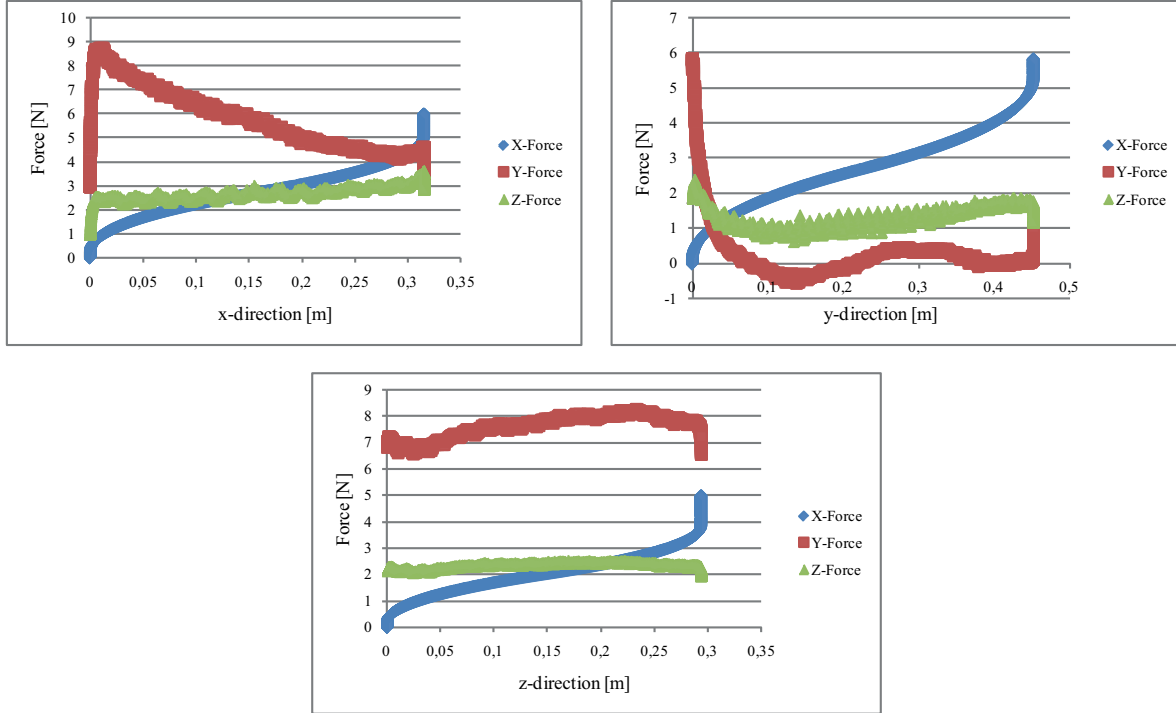
- Use the FRI communication.
- Use a logger program to keep track of the data.

### C.1.4 Description

The manipulator is moved between two coordinates while the force in the TCP is logged. In this way a logger program keeps track of the force in the three directions x, y, and z as the manipulator moves in the specified trajectory. The experiment will be conducted for different trajectories along which no contact to the environment is made.

### C.1.5 Result

The manipulator is moved linearly in three directions x, y, and z. Figure C.1 shows that the force measurement is instable and varies as a function of movement in all directions. It is hard to find a clear pattern in the variation but figure C.1 indicates that the acceleration could have impact on the force measurement, even though the KUKA LWR should compensate for the acceleration in the calculation of the forces.



*Figure C.1: Force as function of translation in x-, y-, and z-direction.*

### C.1.6 Conclusion

Through experiments it has been documented that the force varies as a result of movement of the manipulator. This has a decisive impact on the ability to sense collisions with objects given that it is hard to separate data from an actual collision with the noise from the varying force. Furthermore, it is hard to draw up a standardization of the ability to sense an object because the force seems dependent on the manipulator's acceleration, position, and orientation. The KUKA LWR measures an angular momentum in each joint. It is from these measurements and the specified inertia and mass of the tool the forces in the TCP are calculated, and thus any measurement instability or error in a given joint are accumulated in the calculated TCP force. This makes the calculated force in the TCP relatively sensitive, and thus the varying and inaccurate force in the TCP could simply be due to the mechanical dynamics of the manipulator.

After having conducted these experiments it has been proved in collaboration with KUKA that joint 3 on the manipulator has an unknown defect. The manipulator has been booked for repair in July 2012. This defect may affect the force measurements consequently.

## C.2 Interrupt - KRL

The ability to sense collision with an object is programmed by an *Interrupt function* and an appertaining *Interrupt program*. The interrupt function is declared in the main program while the interrupt program is developed separately from the main program. If the specified condition is fulfilled the work in progress is stopped and the interrupt

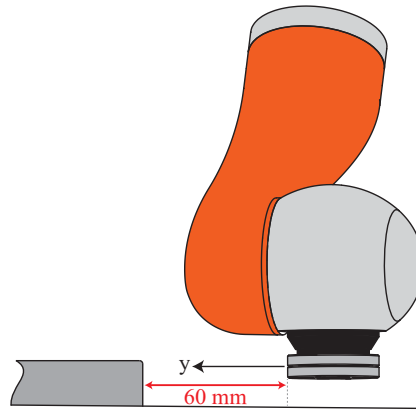
program is conducted. The interrupt program consists of a *Brake* command which produces a hard deacceleration of the manipulator and a *Resume* command which cancels all running interrupt programs. For further informations see (KUKA System Software, 2003).

### C.2.1 Purpose

The purpose of this section is to verify the ability to detect collision with an object in real-time problems. The interrupt function and the appertaining interrupt program are used to brake the manipulator when the manipulator collides with the object. Different influential factors are changed during the experiment to achieve a minimum force and overshoot.

### C.2.2 Description

The manipulator is moved until it collides with an object and meanwhile the force in the TCP is logged. The distance to the object is 60 mm and the manipulator is moved in the y-direction as illustrated in figure C.2 .



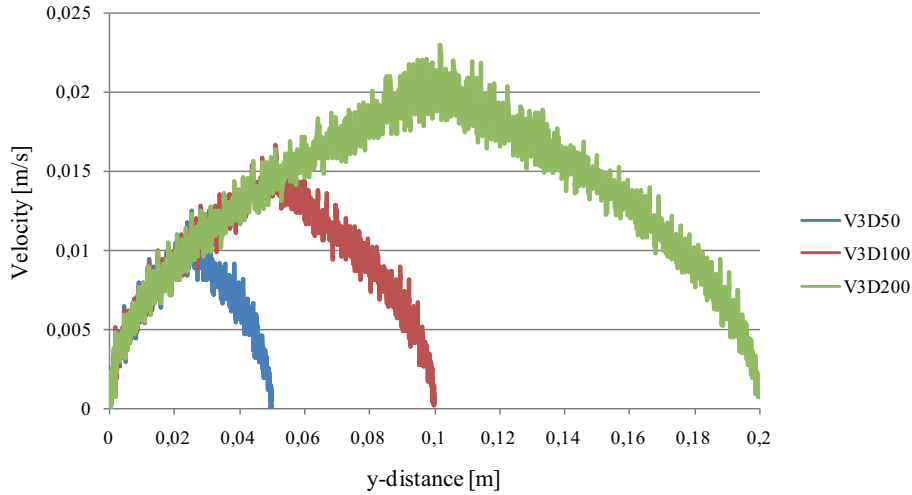
**Figure C.2:** The interrupt setup - the manipulator is moved in the y-direction until it collides with the block.

Through preliminary experiment the influential factors and their limit values have been identified. The influential factors and their minimum permissible values are shown in table C.1.

Factor	Minimum value
Trigger Force	3 N
Velocity	3%
Search distance	60 mm

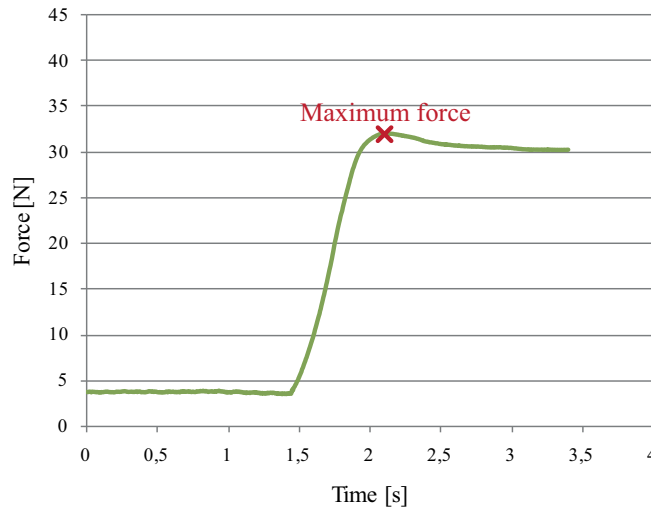
**Table C.1:** The influential factors and their limit values. The limit values are identified for this experiment only as a result of the varying force measurement.

A trigger force below 3 N causes interrupt without collision as a result of the inaccurate and varying force. It has been found that the search distance has a dominant influence on the velocity. Figure C.3 shows three different search distances with the same velocity profile, hence the velocity specified on the KRC. This velocity profile is declared in KRL, but as figure C.3 indicates the velocity profile never obtains a constant velocity, and it seems more likely to be a scaling of both the acceleration and the velocity. The results in figure C.3 show how moving the manipulator 200 mm instead of 50 mm doubles the maximum velocity. This makes the actual search velocity at contact dependent upon the search distance, and consequently a search distance of approximately half of the actual distance is inadmissible. Based on figure C.3 the chosen search distance of 70 mm for the interrupt experiment is a good choice as contact is made during the deacceleration phase.



**Figure C.3:** Velocity profile 3% with three different search distances. The velocity is dependent on the search distance.

A velocity profile below 3% is undesirable in real-time problems because it is simply too time-consuming. On the otherhand a velocity profile of 30% or more is also inadmissible as it generates an undesireably high force. The experiment is conducted for different velocities and trigger forces with five repetitions for each. It is chosen to evaluate the maximum force arising from the collision between the manipulator and the object. The maximum force is illustrated in figure C.4.



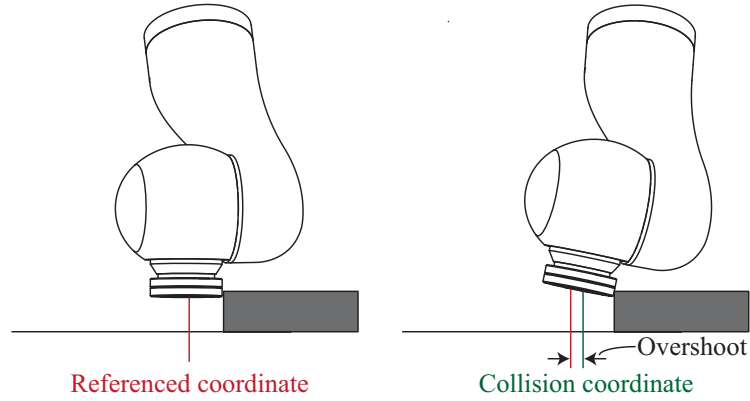
**Figure C.4:** The figure illustrates the maximum force.

Furthermore, it is chosen to evaluate the overshoot caused by the twist between TCP and the object. An illustration of the overshoot is shown in figure C.5.

### C.2.3 Result

Table C.2 shows the mean value of the maximum force and the maximum overshoot for each profile. The profile name KRLV3F6 refers to a velocity profile of 3% and a trigger force of 6 N. As expected it turns out that profile KRLV3F3 achieves the best performance.

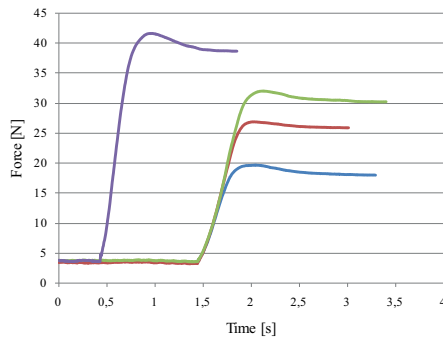
A graphical representation of the best repetition from each profile is shown in figure C.6.



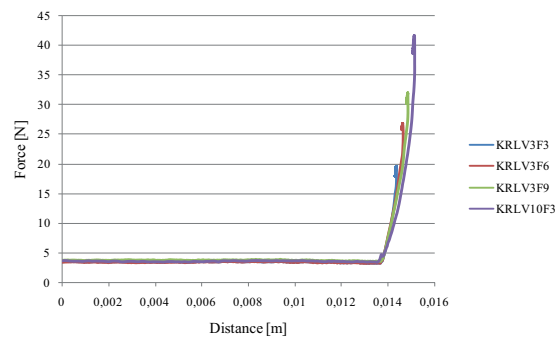
**Figure C.5:** The figure illustrates the overshoot. The overshoot is determined by subtracting the collision coordinate from a reference coordinate.

Profile	Maximum force [N]	Maximum overshoot [mm]
KRLV3F3	20.81	0.77
KRLV3F6	27.99	1.04
KRLV3F9	33.15	1.24
KRLV10F3	42.17	1.61

**Table C.2:** The mean values of maximum force and maximum overshoot.



**(a)** Force as function of time.



**(b)** Force as function of distance.

**Figure C.6:** The best repetition from each profile. The profile KRLV3F3 achieves the best performance.

## C.2.4 Conclusion

Through the experiments the performance of the interrupt function has been documented. Using the minimum admissible values of the velocity and the trigger force the best performance is achieved. A search velocity of 30% or higher is inadmissible as it generates too high forces. It is also documented that the actual velocity of the TCP is dependent of the length of the trajectory and never becomes constant. Furthermore, it can be concluded that the varying force has a decisive impact on the trigger force and thus complicates the standardization of a general interrupt setup.

## C.3 Interrupt - External PC

An external computer is used to control the interrupt function. The force is measured on the external computer and when it exceeds the specified trigger force a signal is transmitted to a KRL program on the KRC. The condition

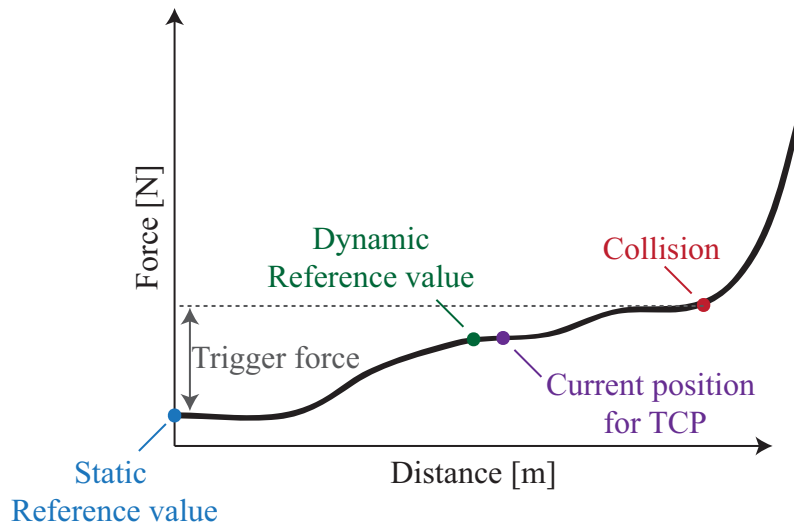
in the declared interrupt function is fulfilled upon the receipt of the signal from the external computer and the interrupt program leads to a hard deacceleration of the manipulator.

### C.3.1 Purpose

The purpose of this experiment is to compare the real-time performance of interrupting the manipulator from the external computer and directly from KRL. The performance of the external computer is compared to the profile KRLV3F3 from appendix C.2. Furthermore it is chosen to establish an improved interrupt function on the external computer in order to reduce the influence of the inaccurate force measurement, see appendix C.1.

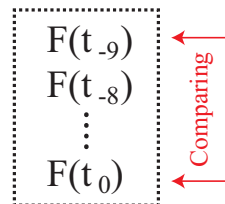
### C.3.2 Improved interrupt function

The inaccurate force gives rise to a problem concerning the trigger force as mentioned in appendix C.2.4. In the experiment in appendix C.2 the reference value of the force is static and thereby takes no account of the varying force. The improved interrupt function updates the reference value during the searching process which is designated as a *dynamic reference value*. Figure C.7 illustrates the advantage of using a dynamic reference value.



**Figure C.7:** The static reference value takes no account of the unstable force measurement. The implication of this is that a high value of the trigger force may be necessary.

The measured force is stored in an array, and the most recent measurement is compared to a previously stored measurement as illustrated in figure C.8, hence the dynamic reference trails the actual TCP measurement.



**Figure C.8:** The most recent measurement of the force is compared with a previously stored measurement of the force.



### C.3.3 Description

The experiment is conducted in the same way as described in appendix C.2.2. This means that the experiment is conducted for different velocities and trigger forces with 5 repetitions each. Furthermore, it is chosen to evaluate the maximum force and the overshoot. Analysis of variance and multiple comparison are used to compare the two systems. It is desirable to establish a hypothesis test to determine whether there is a significant difference between the two systems or not. This is done by testing the similarity between the systems mean value of the maximum force. The null hypothesis is established in equation C.1, in which  $\mu$  indicates each group's mean value. It is noticed that it is a stronger argument to reject a null hypothesis because a null hypothesis never will be proven. All calculations are carried out in MATLAB and underlying theory is not described. For further information on the theory, see (Montgomery, 2005) and (R. E. Walpole, R. H. Myers, S. L. Myers & K. Ye, 2007).

$$\begin{aligned} H_0 : \mu_1 &= \mu_2 = \dots = \mu_n \\ H_1 : &\text{At least two mean values are dissimilar} \end{aligned} \quad (\text{C.1})$$

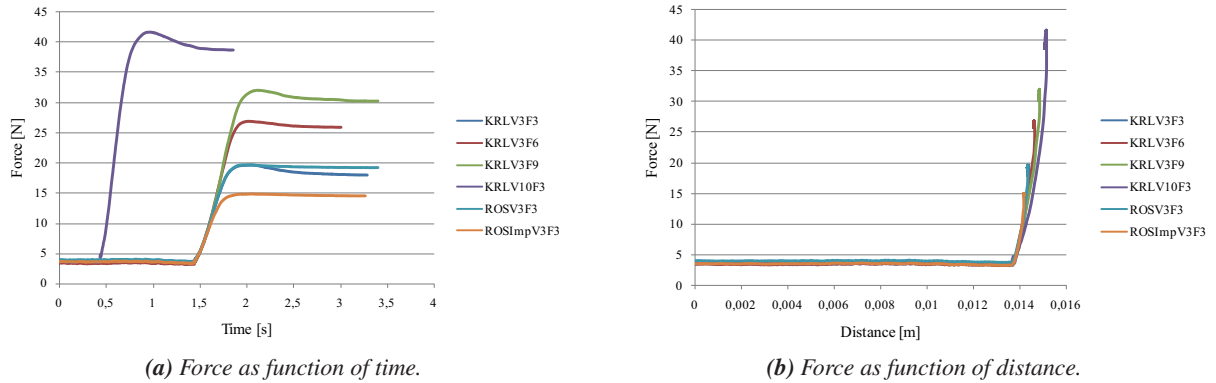
### C.3.4 Result

Table C.3 shows the mean values of the maximum force and maximum overshoot for each profile.

Profile	Maximum force [N]	Maximum overshoot [mm]
KRLV3F3	20.81	0.77
ROSV3F3	20.58	0.73
ROSImpV3F3	16.11	0.54

**Table C.3:** Mean values of maximum force and maximum overshoot.

A graphical representation of the best repetition from each profile is illustrated in figure C.9.



**Figure C.9:** The best repetition from each profile.

The result of the analysis of variance is shown in table C.4.

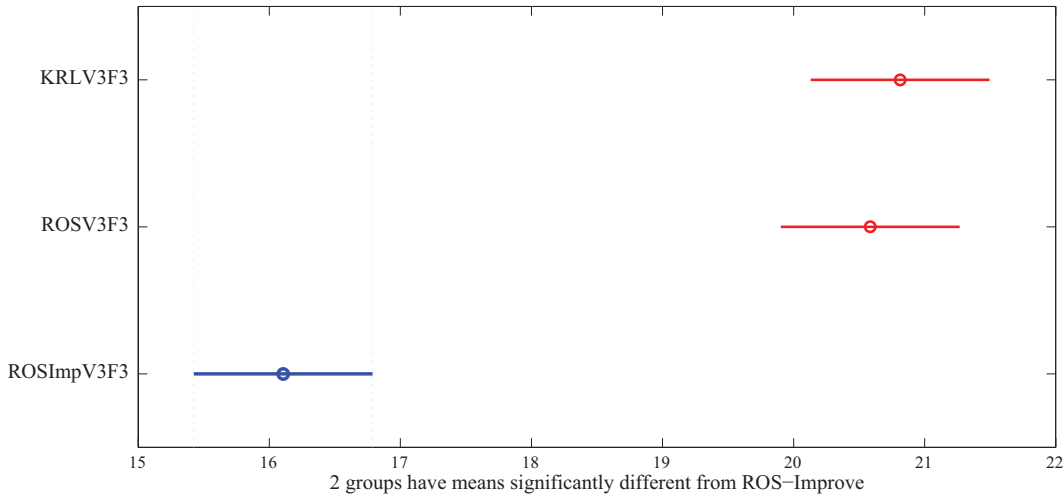
	SS	dF	MS	$f_0$	$f_{0.05}(2, 12)$	P
Group	70.45	2	35.23	-	-	-
Error	7.85	12	0.6538	-	-	-
Total	78.30	14	-	-	-	-
-	-	-	-	54.88	3.88530	$1.01 \cdot 10^{-2}$

**Table C.4:** F-distribution of comparison between the two systems.

The calculated critical value,  $f_0$ , exceeds the critical value of the 0.05 level of significance,  $f_{0,05}(2, 12)$  and furthermore the 0.05 level of significance,  $\alpha$ , exceeds the p-value, see equation C.2. This implies that the null hypothesis is *rejected* and thereby the groups' mean values are different.

$$\begin{aligned} f_0 &> f_{0,05}(2, 12) \\ \alpha &> P \end{aligned} \quad (C.2)$$

While the *analysis of variance* examines if the mean values are the same between the groups, the *multiple comparison* examines which mean values differ from each other. The multiple comparison shows that the mean value of ROSImpV3F3 is significantly different from the mean value of KRLV3F3 and ROSV3F3 while a significant difference between the mean value of KRLV3F3 and ROSV3F3 is not present, figure C.10.



**Figure C.10:** Multiple comparison - the mean value of the ROSImpV3F3 is significantly different from KRLV3F3 and ROSV3F3 while a significant difference between the mean value of KRLV3F3 and ROSV3F3 is not present.

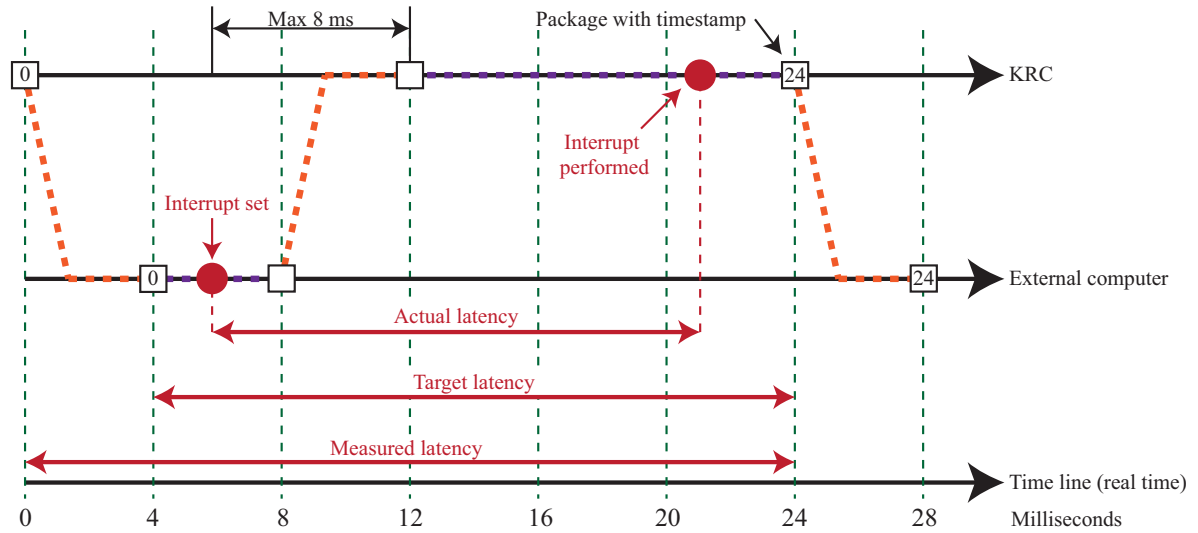
### C.3.5 Conclusion

The conducted experiment has documented that there is no significant difference between interrupt used in KRL and on an external computer. This is of great importance for a final system. The improved interrupt function has a positive effect on reducing the influence of the varying force and the overshoot and clears the way for further improvements to better the performance of interrupt.

## C.4 Latency of the Communication

Even though no significant difference between interrupting the KUKA LWR from an external computer and from the KRC has been found, it is chosen to investigate the latency of the communication between the KRC and the external computer. In this section the latency of using the interrupt function on an external computer is examined. The latency is examined in order to establish its influence on the ability to detect collision with an object. Figure C.11 illustrates how the latency is measured.

The communication between the KRC and the external PC is on both devices carried out every 4 ms, yet this continuous data exchange has been omitted from the figure to decrease complexity. The intension of the experiment is to measure the actual latency marked in figure C.11, though this is quite comprehensive as measurements based on two different time lines, hence the KRC and the external PC are inaccurate. Instead it is chosen to use the time line from the KRC, which is done by using a time stamp included in all packages going from the KRC back to the external PC. By using the time stamp of the packages it becomes clear, see figure C.11, that the measurements are now based on a discrete time line and thus rounded to 4 ms. Furthermore as illustrated in figure C.11 the time



**Figure C.11:** Illustration of the latency experiment. The orange dashed lines illustrate UDP data transfer, while the purple dashed lines illustrate device operating time, hence the software routines and internals of the KRC and the external PC.

recorded upon activating the interrupt on the external PC is based on the last received package, which has a 4 ms old time stamp. This can however, easily be addressed by subtracting this inaccuracy and thus the measured latency becomes the target latency, as illustrated in figure C.11. From figure C.11 it is evident that the target latency can exceed the actual latency by up to 8 ms.

The average latency is measured to 30.27 ms but this corresponds to the *measured latency* in figure C.11 and consequently 4 ms is subtracted to obtain the *target latency*. This gives an average latency of 26.27 ms, yet please note that the actual latency could be up to 8 ms less. From the latency experiments it is observed that the measured latency does not seem to have a significantly effect on the performance of the interrupt function, hence the performance of setting interrupt from the external computer and the KRC does not differ significantly as described in C.3. From figure C.11 it becomes clear, that the latency caused by the UDP data transfer and wait time associated with the discrete data exchange is maximum 8 ms, see figure C.11. This is given as the time from the interrupt is set until the KRC receives the package. From the raw data it is concluded that the data exchange is in fact conducted every 4 ms.

Compared to the physical braking of the manipulator the latency is considered of low influence, and setting the interrupt from an external computer does not differ significantly from setting it on the KRC.



# Budget for Building Little Helper ++

# D

*This Appendix contains the budget for building Little Helper ++. All prices are excluding VAT and parts marked with a \* was already available parts at AAU or parts without an invoice. Some of the parts marked with \* is an estimated price.*

## Budget for the frame structure

Item no.	Model	Description	QTY	Price
1	HNTE8-6	Nuts for aluminium profiles	81	42.93 EUR
2	HNTE8-8	Nuts for aluminium profiles	60	31.80 EUR
3	HBLTS8	Brackets for aluminium profiles	12	16.44 EUR
4	HBLSS8	Brackets for aluminium profiles	10	12.20 EUR
5	HBLBS8	Brackets for aluminium profiles	10	42.90 EUR
6	SHFBS12-20	Bolts for aluminium profiles	20	34.00 EUR
7	SHFBS6-15	Bolts for aluminium profiles	15	6.75 EUR
8	SHFBS6-20	Accessories for aluminium profiles	15	0.96 EUR
9	SHFBS5-8	Accessories for aluminium profiles	4	1.32 EUR
10	HFS8-4040-4000	Aluminium profile	1	60.40 EUR
11	HFS8-4040-2000	Aluminium profile	1	30.20 EUR
12	HFSR8-808040-3200	Aluminium profile	1	136.32 EUR
13	MISUMI Europe	Delivery	1	15.00 EUR
14	Ems 1, 2 parts	Computer tray rail	1	17.58 EUR
15	4 mm	Computer tray	1	10.00 EUR*
16	10 mm	Top plates	3	275.00 EUR
17	5 mm	Cover plates	10	EUR 50.00*
18	Neobotix MP-L655 V3.0	Platform	1	46,000.00 EUR*
19		Total		46,783.80 EUR

**Table D.1:** Budget for the frame structure of Little Helper ++. All prices are excluding labour costs.

**Budget for the electrical system**

Item no.	Model	Description	QTY	Price
20	Moeller DIL M50	24 V Relay	1	272.48 EUR
21	Moeller PLSM-C32/2	Circuit-breaker	1	35.07 EUR
22	Mean Well SPV-300	230V -> 24V	1	50.00 EUR*
23		24 V -> 12 V adapter	1	50.00 EUR*
24	Mascot 2287	DC/AC inverter - pure sine wave	1	505.00 EUR
25		16 mm <sup>2</sup> cable	10 m	59.61 EUR
26		80 mm ventilator fan	2	20.00 EUR*
27		120 mm ventilator fan	2	30.00 EUR*
28		Labour costs	5	261.74 EUR
29		Miscellaneous		128.14 EUR
30		Total		1,412.04 EUR

*Table D.2: Budget for the electrical system of Little Helper ++.***Budget for the computer system**

Item no.	Model	Description	QTY	Price
31	HP Probook 6460b	Laptop	1	735.00 EUR
32	ICIDU 300n	Wireless gigabit router	1	58.00 EUR*
33	Microconnect Mini	VGA adapter	2	8.00 EUR
34	Trust EasyConnect	4 Port USB2 Mini Hub	2	10.75 EUR
35		Monitor cable	2	20.00 EUR*
36		USB extender	2	4.00 EUR*
37		Total		841.75 EUR

*Table D.3: Budget for the computer system of Little Helper ++.***Budget for the gripper system**

Item no.	Model	Description	QTY	Price
38	Phoenix SACC-M 8MS-4QO-0,25-M	Ethernet connector	1	EUR 7.92
39	Phoenix SACC-M 8FS-4QO-0,25-M	Power connector	1	EUR 7.70
40	Schunk WGS 50	Electrical gripper	1	EUR 4,545.00*
41	Custom made	Adapter plates	2	EUR 30.00*
42	Custom made	Jaws	2	EUR 30.00*
43		Total		4,620.62 EUR

*Table D.4: Budget for the gripper system of Little Helper ++. All prices are excluding labour costs.*

---

### Total budget

Item no.	Description	Price
19	Platform and frame structure	EUR 46,783.80
30	Electrical system	1,412.04 EUR
37	Computer system	EUR 841.75
43	Schunk gripper	EUR 4,620.62
44	KUKA LWR	EUR 100,000.00
	Total	153,732.46 EUR

*Table D.5: Total budget for Little Helper ++.*

### Budget within this project

Description	Price
Frame structure	EUR 723.80
Electrical system	1,412.04 EUR EUR
Computer system	753.75 EUR
Schunk gripper	15.62 EUR
Total	2,905.21 EUR

*Table D.6: Budget for building Little Helper ++ within this project.*





*The following appendix contains a description of the Gripper DCN, the Vision DCN and the Platform DCN. The purpose is to describe the architecture and the implementation of these DCNs as they are not thoroughly described in chapter 6.*

## E.1 Gripper DCN

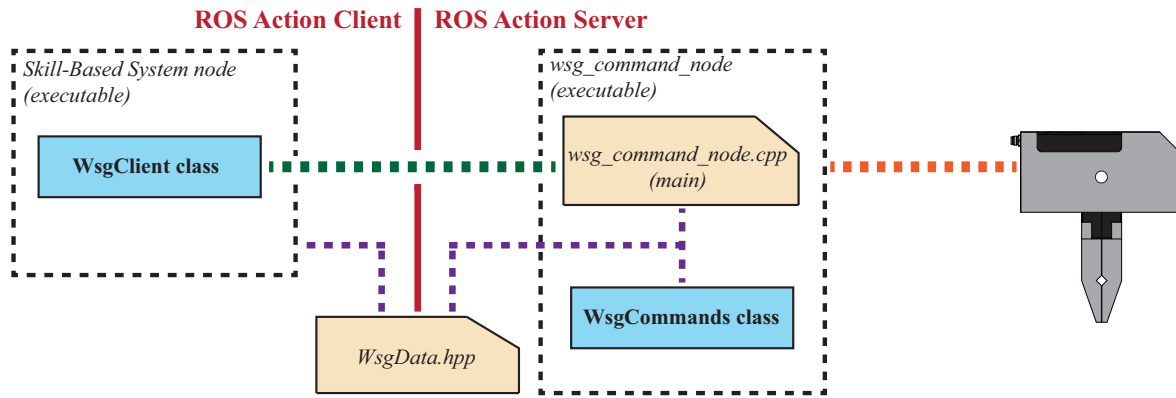
In section 3.3 the hardware implementation of a Schunk WSG50 electric gripper is described. The communication to the gripper is conducted via an Ethernet connection using either a TCP (Transmission Control Protocol), an UDP (User Datagram Protocol), or a CANbus (Controller Area Network bus) connection. With the gripper two different software interfaces are included, a webpage, and a win32 program. Both interfaces have simple graphical user interfaces but they only option manual control of the gripper. In industrial applications manual control is inconvenient as the gripper has to be integrated with other equipment. For automatic control it is possible to run scripts directly on the gripper or to develop user software to communicate directly with the gripper via the TCP, UDP or CANBUS connection. Developing user software is found more convenient, since it allows the gripper to be implemented into an existing software architecture, which in this project is ROS.

Due to this a ROS Package to command the WSG50 gripper has been developed, since no such was available on (ROS Wiki, 2012). The developed ROS Package is called *wsg\_communication* and it will be used within the TAPAS project. The executable ROS Node for controlling the gripper within the ROS Package is *wsg\_command\_node*, which uses an UDP connection to communicate with the gripper. The code is written in C++. The ROS Package *wsg\_communication* is found on the enclosed Appendix CD in `</Software/ROS/wsg_communication>`. The architecture and the principle of *wsg\_communication* are described, but the actual C++ code will not be documented within this report. For information about the code, please refer to the documentation within the source files found on the enclosed Appendix CD. (Weiss Robotics GmbH, 2011) and (ROS Wiki, 2012) have been used in the development of the *wsg\_communication* package.

### E.1.1 Architecture of the *wsg\_communication* ROS Package

Figure E.1 shows a visual representation of the architecture of the *wsg\_communication* ROS Package.

As shown in figure E.1 the *wsg\_command\_node* is a ROS Node used to control the gripper, hence it works as a DCN to the gripper. On one side it communicates with the gripper via an UDP connection and on the other side it works as a ROS Action Server, see (ROS Wiki, 2012), which will receive commands from a ROS Action Client. On the enclosed Appendix CD in `</Software/ROS/wsg_communication>` the package also includes a file named *test\_wsg\_node*. This is an executable file, the purpose of which is to demonstrate the use of the *wsg\_command\_node* and the *WsgClient* class, hence it replaces *Skill-Based System* in figure E.1. A short description of the objects from figure E.1 follows.



**Figure E.1:** The architecture of the `wsg_communication` ROS Package.

### `wsg_command_node.cpp`

`wsg_command_node.cpp` is the main file of the `wsg_command_node` executable. It consists of the code needed for the ROS Action Server configuration and control. Upon receiving a ROS Action Goal from the ROS Action Client it calls the requested function from the `WsgCommands` class.

### WsgCommands Class

The `WsgCommands` class contains functions corresponding to the functions of the gripper and functions for constructing, sending, receiving, and decoding UDP packages.

The most essential functions implemented in the `WsgCommands` class are:

- Home
- Move
- Grasp
- Release
- GetGraspState
- GetWidth

A full list of the functions in the `WsgCommands` class, including definition of input parameters and return data, is found in appendix F.

### WsgClient Class

The `WsgClient` class facilitates the communication to the Gripper DCN, hence the `wsg_command_node`, and thus it is intended for inclusion in any program that communicates with the `wsg_command_node`. In *Little Helper System* the `WsgClient` class is included in *Skill-Based System*, see figure 6.7. It simply wraps the ROS Action calls to the ROS Action Server into private functions for more convenient implementation in the user program.

As the `WsgClient` class simply wraps the communication to the Gripper DCN it implements functions corresponding to those of the `WsgCommands` class and consequently the essential functions of the `WsgClient` class are similar to those of the `WsgCommands` class.

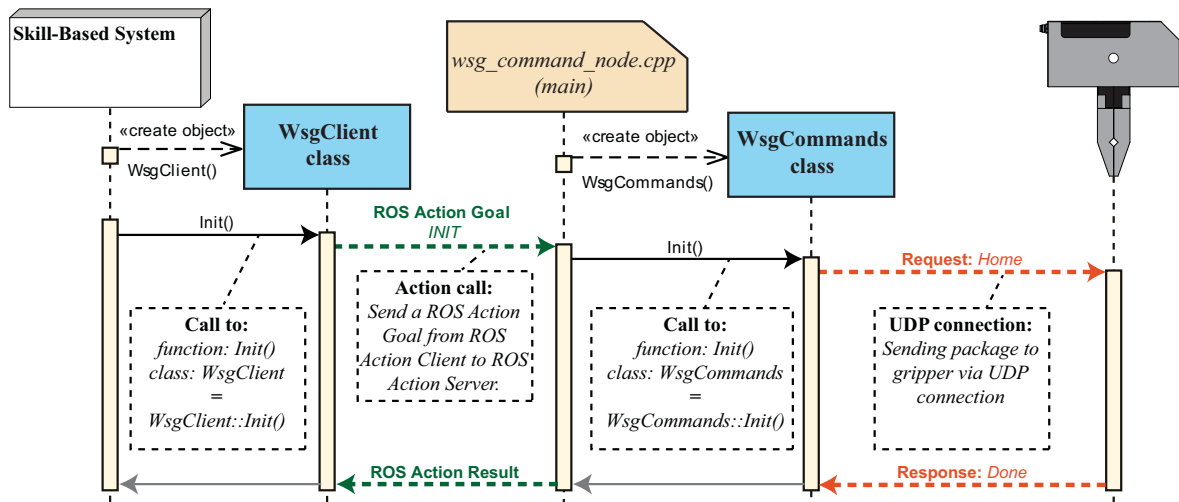
## WsgData.hpp

*WsgData.hpp* is a shared file between the ROS Action Server and the ROS Action Client, hence the *wsg\_command\_node* and *Skill-Based System*. This file contains enumerators defining data being transferred between the ROS Action Server and the ROS Action Client.

### E.1.2 System Sequence

In extend of the architecture description two system sequence diagrams have been created. The purpose is to document how a command is passed through the code and thus illustrate the correlation between the different objects of the ROS Package. The system sequence diagrams have been formulated in the Unified Modeling Language (UML) standard and show three examples of a command being processed.

The first diagram is shown in figure E.2. It shows the creation of the two classes and the initialisation of the gripper by calling the *Init(...)* command.



**Figure E.2:** System sequence diagram showing the creation of the two classes and the call to the *Init(...)* command which initialises the gripper.

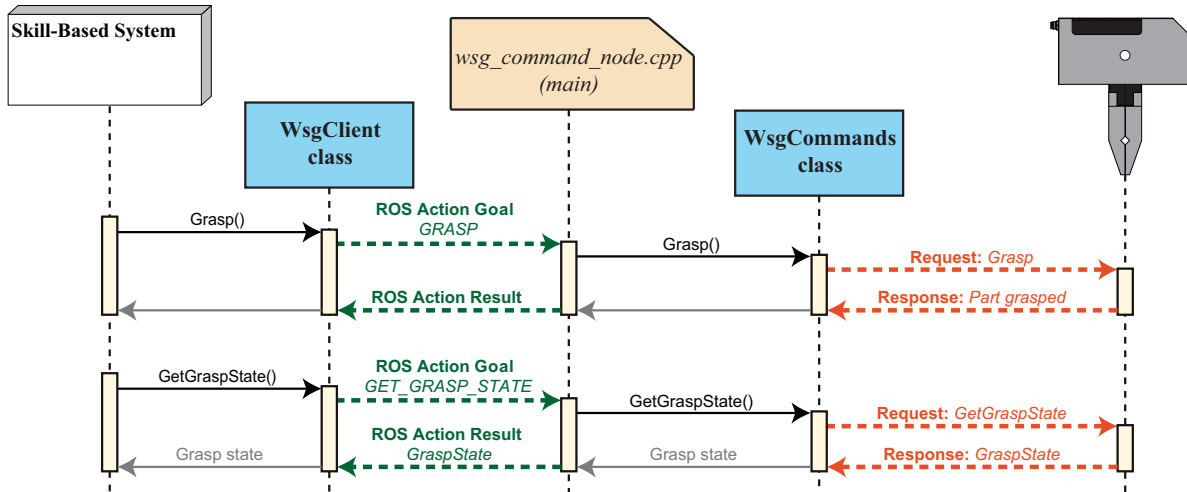
Whenever sending a package to the gripper the *WsgCommands* class will call several private classes within itself to build and send the command and to receive a response. This has been omitted in figure E.2 and figure E.3 to decrease the complexity and disorder as these private functions are of no influence to the comprehension of the command passing between the objects. However, these private function calls are described later in appendix E.1.3. On both sequence diagrams the parameters passed to a function along with the function call are left out to decrease complexity.

When the gripper receives a command it returns an UDP-package from which a state variable is passed back through the system.

Figure E.3 shows how the command for grasping an object and afterwards requesting the grasp state on the gripper is passed through the system. Requesting the grasp state identifies how the grasp went. If an object is lost after it has been grasped the grasp state will change on the gripper, but this will only be apparent to the software upon requesting the grasp state again.

### E.1.3 UDP Communication

In this section the method of passing and receiving UDP packages to and from the gripper is described more thoroughly. This is done using the *Grasp(...)* command as shown in figure E.3 as an example. The basis of the communication between the gripper and the *wsg\_command\_node* is a simple protocol defined by Schunk in

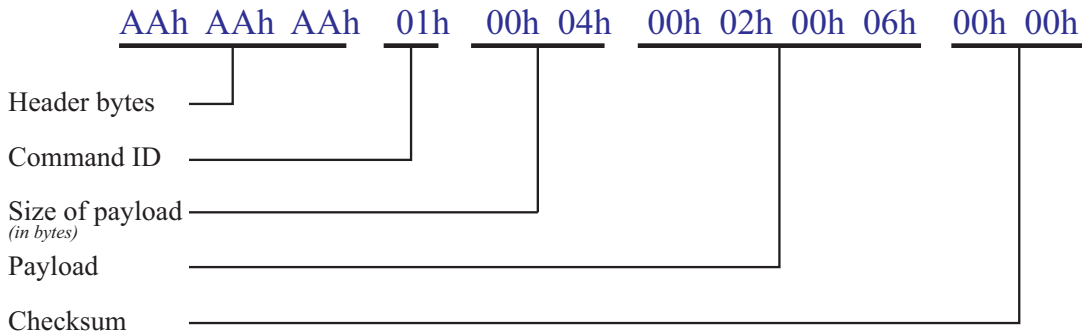


**Figure E.3:** Sequence diagram showing how grasping an object and afterwards requesting the grasp state is passed through the system as commands.

the Command Set Reference Manual (Weiss Robotics GmbH, 2011). This protocol defines the structure of the packages transferred to and from the gripper. Figure E.4 shows the configuration of a package sent to the gripper and figure E.11 shows the configuration of a package received from the gripper.

*Gripper DCN* → *Gripper*

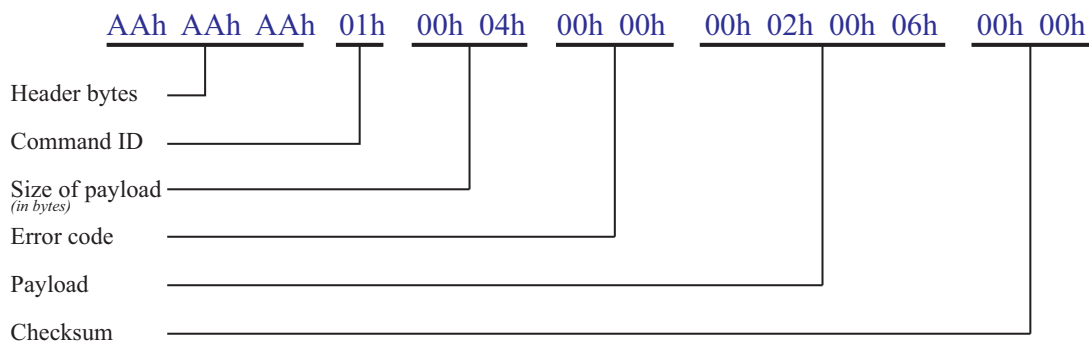
*Request*



**Figure E.4:** Description of a package going from the Gripper DCN on the computer to the gripper. The lower case "h" after each byte indicates that the bytes are in hex format.

*Gripper* → *Gripper DCN*

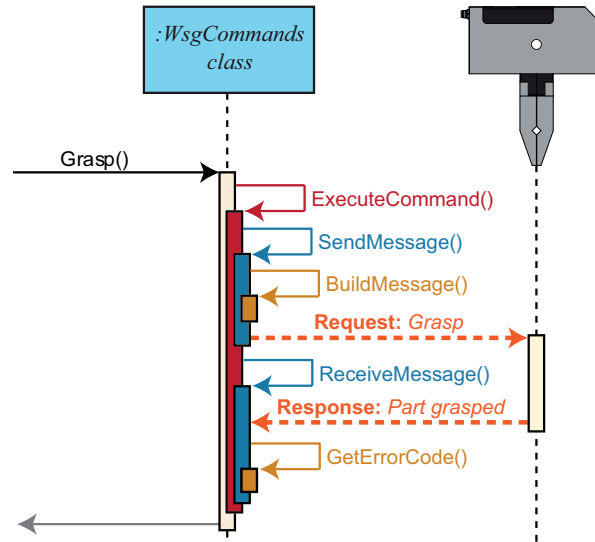
*Response*



**Figure E.5:** Description of a package going from the gripper to the Gripper DCN on the computer. The lower case "h" after each byte indicates that the bytes are in hex format.

A description of each part of the packages is thoroughly described in (Weiss Robotics GmbH, 2011). However, it should be mentioned that the checksum evaluation has been disabled and thus the values of the checksum bytes are of no influence. When enabled, the checksum is calculated based on the specific bytes in the package and is used to verify the integrity of a package.

When a ROS Action Goal is received in the *wsg\_command\_node* the function *Grasp(...)* from the *wsg\_commands* class is called requesting the gripper to grasp an object. Regardless the requested function, the methodology for creating and sending a package to the gripper and afterwards receiving an answer is the same. This methodology is illustrated in figure E.6.



**Figure E.6:** This sequence diagram shows the call to several private functions within the *WsgCommands* class when sending and receiving UDP packages. The purpose is to illustrate the procedure of the UDP communication with the gripper. Please note, that the *Grasp(...)* function is only used as an example, hence the procedure is the same for all communication with the gripper.

As shown in figure E.6 the *Grasp(...)* function calls the *ExecuteCommand(...)* function, which essentially first sends the command and afterwards awaits a response from the gripper. The *SendMessage(...)* function does the actual sending of the package to the gripper, but first it calls the *BuildPackage(...)* function. This function builds the package from the payload data, a predefined header, a checksum, and the command ID. Even though the checksum evaluation has been disabled the checksum bytes must still be included in the package and thus they are simply set to 00h 00h in all packages. The built package is placed in the memory until it is passed on to the gripper. Once the message is sent, the *ExecuteCommand(...)* function calls the *ReceiveMessage(...)* function, which receives a package via the UDP connection. Once a package is received, an error code is decoded. This error code is essentially the response from the gripper and is used to determine whether the execution of a command went well or not. The received data from the gripper are translated from bytes into the appropriate data type and returned backwards through the system. Finally this data is returned to the ROS Action Client in a ROS Action Result, see figure E.3.

## E.2 Vision DCN

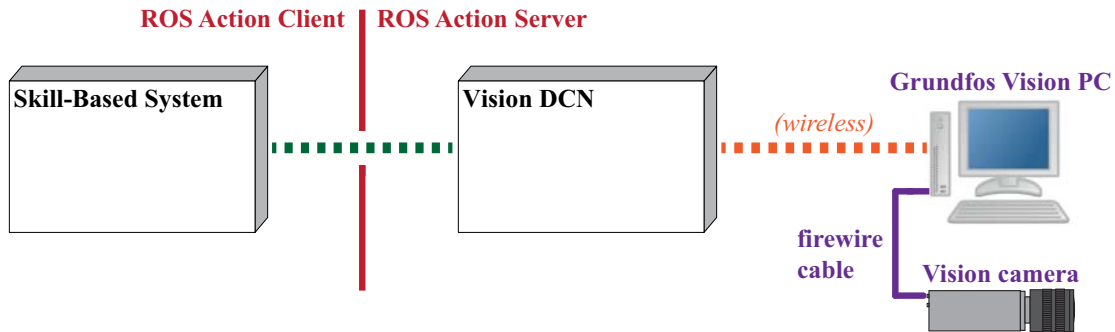
The motivation for implementing a vision system and thus developing a Vision DCN is Workstation 3 described in chapter 8. This Vision DCN is currently under development, and consequently it has not been implemented yet. Workstation 3 in chapter 8 has been used as a basis of the work on the Vision DCN, yet the vision system will accommodate other objects. The vision system is supplied by Grundfos and consists of a vision camera connected to a Windows based PC. This PC contains a vision software developed by the vision department at Grundfos,

(Bigum, 2012), based on the National Instruments LabVIEW software.

The purpose of the Vision DCN is to handle the communication between the Grundfos vision system on a Windows based PC and *Skill-Based System* on the Ubuntu based embedded laptop. The communication is conducted via a wireless Ethernet connection handled by the Ethernet router incorporated in Little Helper ++. The communication between the systems requires the establishment of a communication protocol which has been done as a collaboration between this project group and the vision department at Grundfos, (Bigum, 2012). The protocol is based on the communication protocol for the Schunk WSG50 gripper, refer to section 6.2, and the PLC communication protocol used at Grundfos, found on the enclosed Appendix CD in </Documents/Grundfos Vision-PLC communicationV12>. The development of the communication protocol to the vision system is an ongoing work, and consequently this section only illustrates the fundamentals of the communication protocol. Thus the presented protocol could be subjected to future changes and the software construction of the DCN is currently to be initiated.

### E.2.1 Architecture of the Vision Implementation

Figure E.7 shows a visual representation of the implementation of the vision system.



**Figure E.7:** The architecture of the implementation of the vision system.

As shown in figure E.7 an UDP-protocol is used for communication between *Skill-Based System* and the Grundfos vision PC. This has been chosen in order to keep a stringent communication to all the DCNs.

### E.2.2 System Sequence

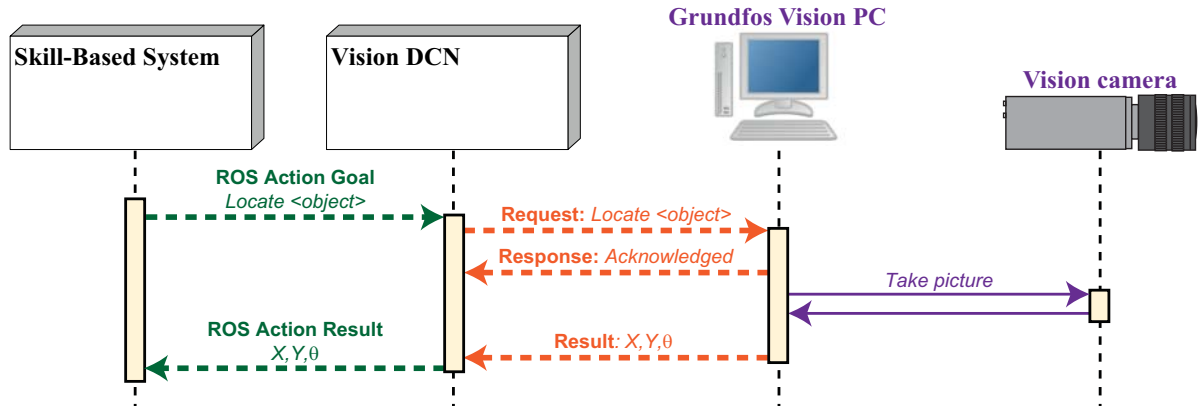
The first step in the establishment of the communication protocol has been the definition of the sequence of communication. Figure E.8 shows a system sequence diagram.

In figure E.8 it is illustrated how three general package types are used in the UDP communication. The *Request* is a package sent from the Vision DCN to the Grundfos vision PC. Upon receipt of a request the vision PC returns a *Response* package to acknowledge the request. Once a picture is taken and has been processed the Grundfos vision PC returns a *Result* package containing the result data.

The system sequence diagram in figure E.8 shows how the communication related to identifying an object is conducted. In section 8.4 it is described how the vision system should not only be used to locate the rotor cap, but also to calibrate the gripper. This localisation is conducted similar to any other object.

### E.2.3 UDP Communication

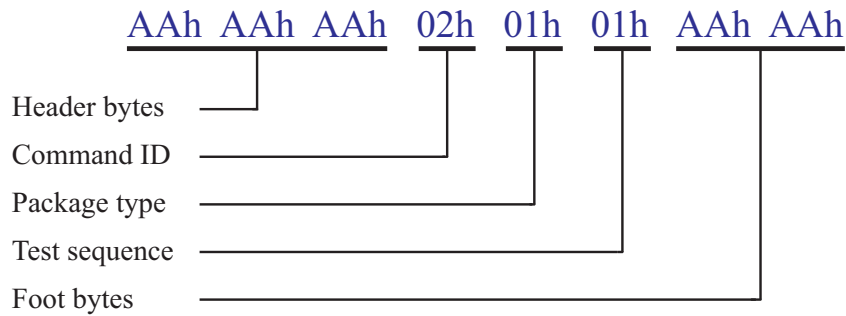
The establishment of an UDP communication protocol is the foundation of the Vision DCN. As the establishment of this protocol is part of this project the protocol is more thoroughly described than in the other DCNs. In this section the package configuration of the *Request*, *Response*, and *Result* package from figure E.8 are described. The



**Figure E.8:** System sequence diagram showing the established communication sequence associated with locating an object. Please note that the communication sequence is currently under development and thus the diagram is for illustration purpose only. The purple line illustrates the firewire cable connection to the camera. This has been included to show when the picture is actually taken.

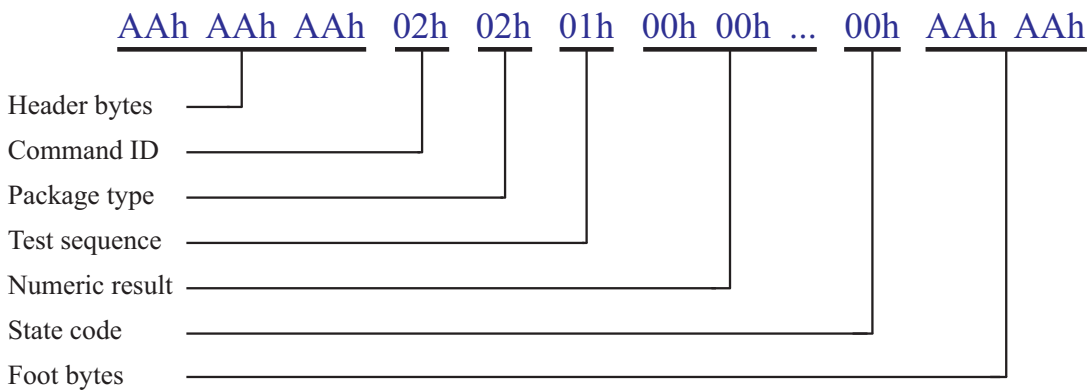
*Request* package is the package sent from the Vision DCN to the Grundfos vision PC whereas the *Response*, and *Result* packages are sent from the vision PC to the Vision DCN.

*Vision DCN* → *Grundfos vision PC*  
Request package



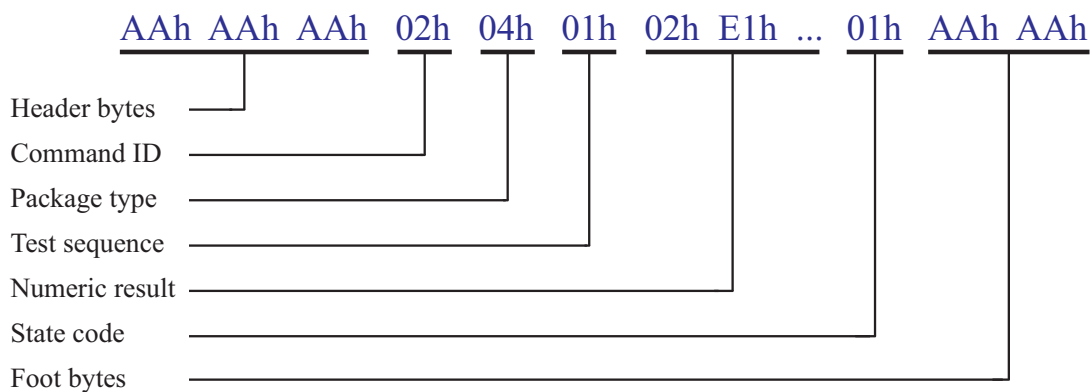
**Figure E.9:** Description of a Request package sent from the Vision DCN, hence Little Helper ++, to the Grundfos vision PC. The lower case "h" after each byte indicates that the bytes are in hex format.

*Grundfos vision PC* → *Vision DCN*  
Response package



**Figure E.10:** Description of a Response package sent to the Vision DCN, hence Little Helper ++, from the Grundfos vision PC. The lower case "h" after each byte indicates that the bytes are in hex format.

*Grundfos vision PC* → *Vision DCN*  
*Result package*



**Figure E.11:** Description of a Result package sent to the Vision DCN, hence Little Helper ++, from the Grundfos vision PC. The lower case "h" after each byte indicates that the bytes are in hex format.

The different parameters of the packages are described in the following sections where the options for the parameters are listed in tables. Please note that the tables only list the currently identified options, and that these tables will be subject to additions as further vision tasks are developed.

### Header bytes and Foot bytes

The *Header bytes* and *Foot bytes* are used to define the beginning and ending of a package and thus they always have the value "AAh".

### Command ID

The *Command ID* defines which command is requested from the vision software on the Grundfos vision PC. This parameter is also included in the *Response* and *Result* package from the vision PC where it carries the value from the preceding request from the Vision DCN. Table E.1 shows the *Command IDs* currently identified.

Command ID	Action	Description
01h	Check ready	Check vision system is ready
02h	Trig	Start vision test

**Table E.1:** List of Command IDs.

### Package type

The *Package type* determines the type of the package in question, hence whether it is a *Request*, a *Response* or a *Result* package. Table E.2 shows the *Package types* currently identified.

Package type	Action	Description
01h	Request	Request from Vision DCN to vision PC
02h	Response	Acknowledgment from vision PC
03h	Result (no numeric data)	Result from vision PC excluding numeric data
04h	Result (numeric data)	Result from vision PC including numeric data

**Table E.2:** List of Package types.



### Test sequence

The *Test sequence* parameter defines which test algorithm or sequence of test algorithms the vision software should perform. It is this parameter that defines which camera to use, what to locate and which type of algorithms to perform in the recognition. This parameter is also included in the *Response* and *Result* package from the vision PC where it carries the value from the preceding request. This is done as a verification of the current *Test sequence*. Table E.3 shows the *Test sequences* identified in Workstation 3 in chapter 8.

Test sequence	Action	Description
01h	Locate rotor cap	Locate the rotor cap in Workstation 3
02h	Calibrate gripper	Locate gripper in Workstation 3

**Table E.3:** List of Test sequences.

### Numeric result

The *Numeric result* contains numeric data being transfered back to the Vision DCN. This numeric data is included in all packages sent from the vision PC, yet it is set to zero in other packages than *Result* packages including numeric data, hence Package type "04h". Typically the numeric data consists of a 2D coordinate (X,Y) and an angle. The *Numeric result* is not included in the *Request* package.

### State code

The *State code* is used to determine the state of the vision software on the Grundfos vision PC. It is both used to determine the outcome of a vision test and to determine the state of the vision software itself. The outcome of the vision test is either a true or false condition, while the vision system itself can declare several different error conditions. In table E.4 a selection of *State codes* is illustrated. The *State code* is not included in the *Request* package.

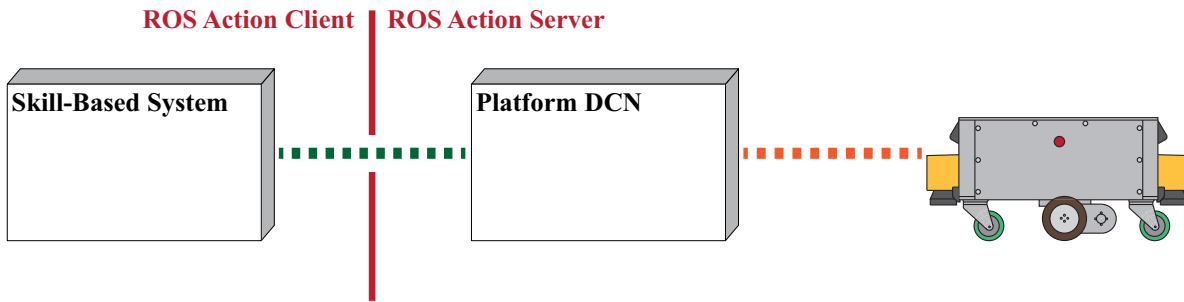
State code	Action	Description
00h	Acknowledge	Acknowledge of a request
01h	Test success	Vision test sequence returned "success", hence condition fulfilled
02h	Test failed	Vision test sequence returned "fail", hence condition not fulfilled
03h	Not ready	Vision system not ready
04h	Unknown command	Command ID unknown
05h	Unknown package	Package type unknown
06h	Unknown test	Test sequence unknown
07h	Incomplete package	Partial package received
08h	Vision system fail	Failure somewhere in the vision system

**Table E.4:** List of State codes.

## E.3 Platform DCN

In chapter 3 the hardware implementation of the Neobotix MP-L655 platform is described. The platform is controlled by an incorporated computer and the navigation system is configured by enclosed software. The software enables configuration of maps, workstations and trajectory planning. As the platform has to be integrated with *Little Helper System* a Platform DCN must be developed according to the architecture illustrated in figure 6.2. The

communication between *Skill-Based System* and the Platform DCN follows a ROS architecture while the communication between the platform DCN and the Neobotix platform is conducted via an Ethernet connection using an UDP protocol. Figure E.12 shows a visual representation of the architecture.



**Figure E.12:** The architecture of the platform ROS Package.

The development of a ROS Package for controlling the platform in TAPAS is ascribed (Pedersen, 2012) but is still under development. In the beginning of 2012 Neobotix released a ROS Package for controlling the platform. Since the implementation of the platform is of low priority in this project it is chosen to await the work of (Pedersen, 2012) in order to compare his work to that of Neobotix. In chapter 8 the system is verified by solving three industrial tasks. During this project the platform is controlled through the enclosed software and thus an interaction between the platform and *Skill-Based System* is not obtained. In the TAPAS month 24 demonstration an interaction between the platform and *Skill-Based System* is needed and hence must be implemented.

# Skills and Device Primitives

# F

*In this appendix the developed skills and device primitives are described. For each skill the usage, programming sequence, execution sequence, input parameters and pre- and postcondition checks are described. The purpose of this appendix is to present each skill, which is intended as a set of instructions on how to use the skill. The developed device primitives of the gripper and the manipulator are listed.*

## F.1 Skills

From chapter 5 a skill is defined as an intelligent sequence of device primitives including pre- and postcondition checks. A skill is separated into two parts, a programming and an execution part. The programming is furthermore separated into a teaching and a specifying part where the specifying part is conducted in either TUI or GUI. After the specification part of a skill the user physically interacts with the manipulator during the teaching sequence, thus the user teaches the locational input parameters. Most of the developed skills incorporate pre- and postcondition checks.



*Pick*

### F.1.1 Pick

The *Pick* skill is used to pick up an object. A video of the *Pick* skill is obtained by scanning the QR-code (*Pick*).

#### Execution

1. Move to an approach coordinate calculated from the *ApproachDistance* (x3), *ApproachDirection* (x2), and *MoveFrame* (x1) with *Velocity* (x9).
2. Open the gripper to the *ObjectWidth* (x6) + 20 mm.
3. Move linear to the *MoveFrame* (x1) with *Velocity* (x9).
4. Grasp object of *ObjectWidth* (x6) using *GraspForce* (x8).
5. Move to a leaving coordinate calculated from the *LeavingDistance* (x4), *LeavingDirection* (x5), and *MoveFrame* (x1) with *Velocity* (x9). During this movement use *Stiffness* (x10).

**Input variables**

	Name	Type	User input	Description
x1	MoveFrame	Frame	Teach	Target coordinate
x2	ApproachDirection	Char	Teach	Direction of approaching the target
x3	ApproachDistance	Float	Teach	Distance of approaching the target
x4	LeavingDistance	Float	Teach	Distance of leaving the target
x5	LeavingDirection	Char	Teach	Direction of leaving the target
x6	ObjectWidth	Float	Teach	Expected width of the object
x7	ObjectTolerance	Float	Specify	Tolerance of the object width
x8	GraspForce	Float	Specify	Grasping force
x9	Velocity	Float	Specify	Velocity of the manipulator
x10	Stiffness	Int	Specify	Cartesian stiffness of the manipulator

**Table F.1:** Input parameters of the execution of the *Pick* skill.

**Precondition**

Check that the gripper is empty.

**Postcondition**

Check that the gripper is holding an object and that this object has the correct width (within specified tolerance).

**Programming**

The object-oriented approach described in chapter 5 is implemented in the *Pick* skill and thus during the specification part of the programming the user must input information about the object to be grasped. In reference to section 6.6 the user must first specify an object type followed by a specific object.

The programming sequence of the *Pick* skill is listed below. In this sequence a known object type, but a new instance of that object type is used as an example. Furthermore in the sequence two separate approach and leaving coordinates are taught.

1. Specify an existing or new object type (*ObjectWidth*(x6), *ObjectTolerance* (x7), and *GraspForce* (x8)). (A known object type is chosen)
2. Specify *Velocity* (x9) and *Stiffness* (x10).
3. Specify whether to use the same or two separate leaving and approach coordinates.
4. Specify an instance of the object type or a new instance. (A new instance of the object type is chosen)
5. Teach *MoveFrame* (x1).
6. Teach *ApproachDirection* (x2).
7. Teach *ApproachDistance* (x3).
8. Teach *LeavingDirection*(x5).
9. Teach *LeavingDistance*(x4).
10. The manipulator picks up the object and moves to the leaving coordinate



*Place*

**F.1.2 Place**

The *Place* skill is used to place an object at a specified coordinate, by which the *Place* skill can be used to drop an item instead of placing it. A video of the *Place* skill is obtained by scanning the QR-code (*Place*).

### Execution

1. Move to an approach coordinate calculated from the *ApproachDistance* (x3), *ApproachDirection* (x2), and *MoveFrame* (x1) with *Velocity* (x7).
2. Move linear to the *MoveFrame* (x1) with *Velocity* (x7). During this movement use *Stiffness* (x8).
3. Release object by releasing to *ObjectWidth* (x6) + 20 mm.
4. Move to a leaving coordinate calculated from the *LeavingDistance* (x4), *LeavingDirection* (x5), and *MoveFrame* (x1) with *Velocity* (x7).

### Input variables

	Name	Type	User input	Description
x1	MoveFrame	Frame	Teach	Target coordinate
x2	ApproachDirection	Char	Teach	Direction of approaching the target
x3	ApproachDistance	Float	Teach	Distance of approaching the target
x4	LeavingDistance	Float	Teach	Distance of leaving the target
x5	LeavingDirection	Char	Teach	Direction of leaving the target
x6	ObjectWidth	Float	Teach	Width of the grasped object
x7	Velocity	Float	Specify	Velocity of the manipulator
x8	Stiffness	Int	Specify	Cartesian stiffness of the manipulator

**Table F.2:** Input parameters of the execution of a *Place* skill.

### Precondition

Check that the gripper is holding an object and that this object has the correct width (within specified tolerance).

### Postcondition

Check that the gripper is empty.

### Programming

The programming sequence of the *Place* skill is given below where two separate approach and leaving coordinates are used.

1. Specify *Velocity* (x7) and *Stiffness* (x8).
2. Specify whether to use the same or two separate leaving and approach coordinates.
3. Teach *MoveFrame* (x1).
4. Teach *ApproachDirection* (x2).
5. Teach *ApproachDistance* (x3).
6. Teach *LeavingDirection* (x5).
7. Teach *LeavingDistance*(x4).



*PlaceInto*

### F.1.3 PlaceInto

The *PlaceInto* skill is used to place an object into a hole. Furthermore the skill can also be used to place an object with a hole down over a peg. A video of the *PlaceInto* skill is obtained by scanning the QR-code (*PlaceInto*).

**Execution**

1. Move to an approach coordinate calculated from the *ApproachDistance* (x3), *ApproachDirection* (x2), and *MoveFrameAbove* (x11) with *Velocity* (x9).
2. Move to a coordinate 5 mm from the hole defined by the *MoveFrameAbove* (x11) with *Velocity* (x9).
3. Search 10 mm with *SearchVelocity* (x7) in the *ApproachDirection* (x2), until the force exceeds *TriggerForce* (x8).

*If contact found:*

1. Apply a force of 10 N in the *ApproachDirection* (x2), hence towards the hole, while executing a *SearchPattern* (x12) until a drop of 1.5 mm is measured.
2. Move linear to *MoveFrame* (x1).
3. Release object by releasing to *ObjectWidth* (x6) + 20 mm.
4. Move to a leaving coordinate calculated from the *LeavingDistance* (x4), *LeavingDirection* (x5), and *MoveFrame* (x1).

*If no contact found:*

1. Move linear to the target coordinate (x1).
2. Release object by releasing to *ObjectWidth* (x6) + 20 mm.
3. Move to a leaving coordinate calculated from the *LeavingDistance* (x4), *LeavingDirection* (x5), and *MoveFrame* (x1) with *Velocity* (x9).

**Input variables**

	Name	Type	User input	Description
x1	MoveFrame	Frame	Teach	Target coordinate
x2	ApproachDirection	Char	Teach	Direction of approaching the target
x3	ApproachDistance	Float	Teach	Distance of approaching the target
x4	LeavingDistance	Float	Teach	Distance of leaving the target
x5	LeavingDirection	Char	Teach	Direction of leaving the target
x6	ObjectWidth	Float	Teach	Width of the grasped object
x7	SearchVelocity	Float	Specify	Velocity when searching for contact
x8	TriggerForce	Float	Specify	Force to trig contact
x9	Velocity	Float	Specify	Velocity of the manipulator
x10	Stiffness	Int	Specify	Cartesian stiffness of the manipulator
x11	MoveFrameAbove	Frame	Teach	Coordinate above the hole, hence the top of the hole
x12	SearchPattern	Char	Specify	Increasing search pattern (circular or rectangular)

*Table F.3: Input parameters of the execution of a PlaceInto skill.*

**Precondition**

Check that the gripper is holding an object and that this object has the correct width (within specified tolerance).

**Postcondition**

Check that the gripper is empty.

## Programming

The programming sequence of the *PlaceInto* skill is given below where two separate approach and leaving coordinates are used.

1. Specify *Velocity* (x9) and *Stiffness* (x10).
2. Specify whether to use the same or two separate leaving and approach coordinates.
3. Teach the coordinate above the hole, *MoveFrameAbove* (x11).
4. Teach the target coordinate, *MoveFrame* (x1).
5. Teach *ApproachDirection* (x2).
6. Teach *ApproachDistance* (x3).
7. Teach *LeavingDirection* s(x5).
8. Teach *LeavingDistance* (x4).



*PlaceOnto*

### F.1.4 PlaceOnto

The sequence of the *PlaceOnto* skill is similar to that of the *Place* skill, but when moving from the approach coordinate towards the target coordinate the *PlaceOnto* skill searches for contact with the surface. The skill is typically used to place an object gently onto a surface. A video of the *PlaceOnto* skill is obtained by scanning the QR-code (*PlaceOnto*).

## Execution

1. Move to an approach coordinate calculated from the *ApproachDistance* (x3), *ApproachDirection* (x2), and *MoveFrame* (x1) with *Velocity* (x9).
2. Move linear to a coordinate 5 mm from the *MoveFrame* (x1) with *Velocity* (x9).
3. Search with *SearchVelocity* (x7) in the *ApproachDirection* (x2), hence towards the hole, until the force exceeds *TriggerForce* (x8).
4. Release object by releasing to *ObjectWidth* (x6) + 20 mm.
5. Move to a leaving coordinate calculated from the *LeavingDistance* (x4), *LeavingDirection* (x5), and *MoveFrame* (x1) with *Velocity* (x9).

## Input variables

	Name	Type	User input	Description
x1	MoveFrame	Frame	Teach	Target coordinate
x2	ApproachDirection	Char	Teach	Direction of approaching the target
x3	ApproachDistance	Float	Teach	Distance of approaching the target
x4	LeavingDistance	Float	Teach	Distance of leaving the target
x5	LeavingDirection	Char	Teach	Direction of leaving the target
x6	ObjectWidth	Float	Teach	Width of the grasped object
x7	SearchVelocity	Float	Specify	Velocity when searching for contact
x8	TriggerForce	Float	Specify	Force to trig contact
x9	Velocity	Float	Specify	Velocity of the manipulator

**Table F.4:** Input parameters of the execution of a *PlaceOnto* skill.

**Precondition**

Check that the gripper is holding an object and that this object has the correct width (within specified tolerance).

**Postcondition**

Check that the gripper is empty.

**Programming**

The programming sequence of the *PlaceOnto* skill is given below where two separate approach and leaving coordinates are used.

1. Specify *Velocity* (x9).
2. Specify whether to use the same or two separate leaving and approach coordinates.
3. Teach *MoveFrame* (x1).
4. Teach *ApproachDirection* (x2).
5. Teach *ApproachDistance* (x3).
6. Teach *LeavingDirection*(x5).
7. Teach *LeavingDistance*(x4).



*MoveTo*

**F.1.5 MoveTo**

The *MoveTo* skill is used to store via coordinates which the manipulator will move to when executed. A sequence of coordinates can be stored to create a discrete trajectory. The manipulator moves to the coordinates in either joints or cartesian. Furthermore the TCP can be aligned relative to the base reference frame during the teaching of the *MoveTo* skill. This is beneficial if the subsequent skill requires the gripper to be horizontal or vertical. A video of the *MoveTo* skill is obtained by scanning the QR-code (*MoveTo*).

**Execution**

1. Move to the target coordinate(s) (x1) with *Velocity* (x2).

**Input variables**

	Name	Type	User input	Description
x1a	MoveFrameJoint	Frame	Teach	Target coordinate (joints)
x1b	MoveFrameCartesian	Frame	Teach	Target coordinate (cartesian)
x2	Velocity	Float	Specify	Velocity of the manipulator
x3	FrameType	Char	Specify	Choice of joint or cartesian motion
x4	MotionType	Char	Specify	If cartesian motion chosen: Linear or PTP motion

**Table F.5:** Input parameters of the execution of a *MoveTo* skill.

**Precondition**

None

**Postcondition**

None



## Programming

During the teaching of the *MoveTo* skill the user can teach several coordinates. After each stored coordinate the user is given the option to align the TCP relative to the base reference frame.

1. Specify *Velocity* (x2).
2. Specify *FrameType* (x3). *If cartesian: Specify MotionType* (x4).
3. Teach *MoveFrameJoint* (x1a) and *MoveFrameCartesian* (x1b). Teach alignment if needed.

### F.1.6 Home

The *Home* skill moves the manipulator to a specific coordinate located above the platform. The coordinate is specified in the joint space.

#### Execution

1. Move to the specific Home coordinate with *Velocity* (x1).

#### Input variables

	Name	Type	User input	Description
x1	Velocity	Float	Specify	Velocity of the manipulator

**Table F.6:** Input parameters of the execution of a *MoveTo* skill.

#### Precondition

None

#### Postcondition

None

## Programming

The *Home* skill does not need any physical input from the user during the teaching sequence as the manipulator automatically moves to home position at a safe velocity during the teaching routine.

1. Specify *Velocity* (x1).



*Rotate*

### F.1.7 Rotate

The *Rotate* skill is used to rotate an object, e.g. an axle in a hole. The *Rotate* skill presents the user with two options, to rotate an arbitrary angle once or to repeat the rotation until an external signal is received. If rotating until an external signal is received the object is rotated an arbitrary angle and subsequently the object is released, the manipulator rotates back, the object is grasped again and the rotation is repeated. Independent of which of the two above options is chosen the user has the option to retain the grasp or release the object after the rotation. A video of the *Rotate* skill is obtained by scanning the QR-code (*Rotate*).

**Execution**

Rotate an arbitrary angle:

1. Rotate an *Angle* (x2) in *Direction* (x4) with *Velocity* (x3).
2. If *GripperCondition* (x1) is specified to 1: Release object.

Rotate until external signal:

1. *Repeat until external signal*:
  - a) Rotate an *Angle* (x2) in *Direction* (x4) with *Velocity* (x3).
  - b) Release object.
  - c) Rotate a negative *Angle* (-x2) in *Direction* (x4) with *Velocity* (x3).
  - d) Grasp object.
  - e) Rotate an *Angle* (x2) in *Direction* (x4) with *Velocity* (x3).
2. If *GripperCondition* (x1) is specified to 1: Release object.

**Input variables**

	Name	Type	User input	Description
x1	GripperCondition	Int	Teach	Retain grasp or open gripper
x2	Angle	Int	Teach/specify	Angle of rotation
x3	Velocity	Float	Specify	Velocity of the manipulator
x4	Direction	Char	Teach	Rotation direction
x5	AngleSignal	Int	Specify	Defines if an external signal is used

*Table F.7: Input parameters of the execution of a Rotate skill.*

**Precondition**

None

**Postcondition**

None

**Programming**

The programming sequence of the *Rotate* skill differs depending on whether the angle is specified or taught. The sequence below exemplifies both cases.

1. Specify *Velocity* (x3).
2. Teach the direction of rotation, *Direction* (x3).
3. Teach or specify the angle of rotation, *Angle* (x2).

### F.1.8 PegInHole

The *PegInHole* skill is used to place a peg into a hole hence an object into another object. The skill differentiates from the *PlaceInto* skill by angling the peg relative to the hole and thus increasing the probability of correctly finding the hole. The *PlaceInto* skill does not increase probability of finding the hole correctly, but instead implements an active search algorithm to find the hole if not initially found. A video of the *PegInHole* skill is obtained by scanning the QR-code (*PegInHole*).



*PegInHole*

#### Execution

1. Move to an approach coordinate calculated from the *ApproachDistance* (x3), *ApproachDirection* (x2), and *MoveFrameAbove* (x10) with *Velocity* (x9).
2. Move to a coordinate 5 mm from the hole defined by the *MoveFrameAbove* coordinate (x10) with *Velocity* (x9).
3. Search with *SearchVelocity* (x7) in the *ApproachDirection* (x2), hence towards the hole, until the force exceeds *TriggerForce* (x8).
4. Calculate the length of the grasped object based on the current position and the target coordinate, *MoveFrame* (x1).
5. Align the peg to the orientation of the hole by rotating around end of the peg defined by the calculated length.
6. Move linear to *MoveFrame* (x1).
7. Release object by releasing to *ObjectWidth* (x6) + 20 mm.
8. Move to a leaving coordinate calculated from the *LeavingDistance* (x4), *LeavingDirection* (x5), and *MoveFrame* (x1) with *Velocity* (x9).

#### Input variables

	Name	Type	User input	Description
x1	MoveFrame	Frame	Teach	Target coordinate
x2	ApproachDirection	Char	Teach	Direction of approaching the target
x3	ApproachDistance	Float	Teach	Distance of approaching the target
x4	LeavingDistance	Float	Teach	Distance of leaving the target
x5	LeavingDirection	Char	Teach	Direction of leaving the target
x6	ObjectWidth	Float	Teach	Width of the grasped object
x7	SearchVelocity	Float	Specify	Velocity when searching for contact
x8	TriggerForce	Float	Specify	Force to trig contact
x9	Velocity	Float	Specify	Velocity for the manipulator
x10	MoveFrameAbove	Frame	Teach	Contact coordinate between hole and peg
x11	GripperCondition	Int	Teach	Retain grasp or open gripper
x12	HoleDirection	Char	Teach	Direction of hole opening
x13	CalculatedLength	Float	Calculated	Calculated length of the object

**Table F.8:** Input parameters of the execution of a *PegInHole* skill.

**Precondition**

Check that the gripper is holding an object and that this object has the correct width (within specified tolerance).

**Postcondition**

Check that the gripper is empty.

**Programming**

The programming sequence of the *PegInHole* skill is given below.

1. Specify *Velocity* (x9), *SearchVelocity* (x7), and *TriggerForce* (x8).
2. Teach the coordinate above the hole, *MoveFrameAbove* (x10), hence where the peg is angled relative to the hole and the end of the peg is in contact with the edge of the hole.
3. Teach *ApproachDirection* (x2).
4. Teach *ApproachDistance* (x3).
5. Teach the coordinate of the end position, *MoveFrame* (x1).
6. Teach *HoleDirection* (x12).
7. Teach *LeavingDirection* (x5).
8. Teach *LeavingDistance* (x4).



*PickFromStack*

**F.1.9 PickFromStack**

The *PickFromStack* skill is used to pick an object from a stack of similar objects, hence the same object type. The skill searches for the upper object in the stack and subsequent only the upper object is picked. A video of the *PickFromStack* skill is obtained by scanning the QR-code (*PickFromStack*).

**Execution**

1. Move to an approach coordinate calculated from the *ApproachDistance* (x3), *ApproachDirection* (x2), and *MoveFrame* (x1) with *Velocity* (x9).
2. Move linear to the *MoveFrame* (x1) with *Velocity* (x9).
3. Open gripper to *ObjectWidth* (x6) - 4 mm.
4. Search with *SearchVelocity* (x7) in the *ApproachDirection* (x2), hence towards the stack, until contact is found.
5. Open the gripper to the *ObjectWidth* (x6) + 20 mm.
6. Move a distance of *ObjectThickness* (x11) in the *ApproachDirection* (x2).
7. Grasp object of *ObjectWidth* (x6) using *GraspForce* (x8).
8. Move to the leaving coordinate (the calculated approach coordinate) with *Velocity* (x9).

## Input variables

	Name	Type	User input	Description
x1	MoveFrame	Frame	Teach	Target coordinate
x2	ApproachDirection	Char	Teach	Direction of approaching the target
x3	ApproachDistance	Float	Teach	Distance of approaching the target
x4	LeavingDistance	Float	Teach	Equal to approach distance
x5	LeavingDirection	Char	Teach	Equal to approach direction
x6	ObjectWidth	Float	Teach	Expected width of the object
x7	SearchVelocity	Float	Specify	Velocity when searching for contact
x8	Stiffness	Int	Specify	Cartesian stiffness of the manipulator
x9	Velocity	Float	Specify	Velocity of the manipulator
x10	GraspForce	Float	Specify	Grasping force for the gripper
x11	ObjectThickness	Float	Specify	Expected thickness of the object
x12	ObjectTolerance	Float	Specify	Tolerance of the object width

**Table F.9:** Input parameters of the execution of a *PickFromStack* skill.

## Precondition

Check that the gripper is empty.

## Postcondition

Check that the gripper is holding an object and that this object has the correct width (within specified tolerance).

## Programming

The programming sequence of the *PickFromStack* skill is given below.

1. Specify a *Velocity* (x9), *Stiffness* (x8), *GraspForce* (x10), and *ObjectThickness* (x11).
2. Teach *MoveFrame*(x1), hence the starting coordinate of the search.
3. Teach *ApproachDirection* (x2).
4. Teach *ApproachDistance* (x3).
5. The manipulator searches for the upper object and picks it up.



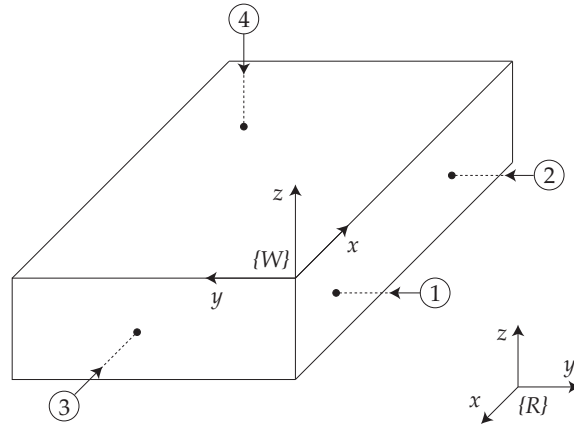
*Calibrate*

### F.1.10 Calibrate

The *Calibrate* skill performs a calibration between the manipulator and a given workstation. The calibration is necessary due to the platforms accuracy of  $\pm 10$  mm and  $\pm 5$  degrees. The user teaches the manipulator four coordinates in three planes, from which a reference frame of the workstation is calculated. The *Calibrate* skill needs four calibration coordinates as illustrated in figure F.1. To perform the calibration the gripper is closed and the jaws are used to search for contact in the four coordinates, hence touching the workstation.

Four coordinates in three planes are sufficient to calculate the reference frame as only the horizontal orientation, hence around a vertical axis, is compensated. By doing this it is assumed that the robot is parked horizontally parallel to the workstation. A video of the *Calibrate* skill is obtained by scanning the QR-code (*Calibrate*).

The calculation of the workstation reference frame is based on (Craig, 2005).



**Figure F.1:** Search strategy of the calibration. The search coordinates (numbers) starts a distance ( $x_6$ ) from the taught coordinates (dots). This distance may vary due to the inaccuracy of the platform. (Pedersen, 2011)

### Execution

1. Move to *ViaPoint* ( $x_5$ ) in joint space with *Velocity* ( $x_9$ ).
2. Move to *SearchPoint1* in joint space with *Velocity* ( $x_9$ ).
3. Move linearly (*ApproachDistance* ( $x_6$ ) - 25 mm) in the z-direction, hence toward the contact coordinate.
4. Search with *SearchVelocity* ( $x_7$ ) until the force exceeds *TriggerForce* ( $x_8$ ).
5. Move *ApproachDistance* ( $x_6$ ) back from the contact coordinate with velocity ( $x_9$ ).
6. Move to *ViaPoint* ( $x_5$ ) in joint space with velocity ( $x_9$ ).
7. Move to *SearchPoint2* in joint space with *Velocity* ( $x_9$ ).
8. Move linearly (*ApproachDistance* ( $x_6$ ) - 25 mm) in the z-direction, hence toward the contact coordinate.
9. Search with *SearchVelocity* ( $x_7$ ) until the force exceeds *TriggerForce* ( $x_8$ ).
10. Move *ApproachDistance* ( $x_6$ ) back from the contact coordinate with velocity ( $x_9$ ).
11. Move to *ViaPoint* ( $x_5$ ) in joint space with velocity ( $x_9$ ).
12. Move to *SearchPoint3* in joint space with *Velocity* ( $x_9$ ).
13. Move linearly (*ApproachDistance* ( $x_6$ ) - 25 mm) in the z-direction, hence toward the contact coordinate.
14. Search with *SearchVelocity* ( $x_7$ ) until the force exceeds *TriggerForce* ( $x_8$ ).
15. Move *ApproachDistance* ( $x_6$ ) back from the contact coordinate with velocity ( $x_9$ ).
16. Move to *ViaPoint* ( $x_5$ ) in joint space with velocity ( $x_9$ ).
17. Move to *SearchPoint4* in joint space with *Velocity* ( $x_9$ ).
18. Move linearly (*ApproachDistance* ( $x_6$ ) - 25 mm) in the z-direction, hence toward the contact coordinate.
19. Search with *SearchVelocity* ( $x_7$ ) until the force exceeds *TriggerForce* ( $x_8$ ).
20. Move *ApproachDistance* ( $x_6$ ) back from the contact coordinate with velocity ( $x_9$ ).
21. Move to *ViaPoint* ( $x_5$ ) in joint space with velocity ( $x_9$ ).

### Input variables

	Name	Type	User input	Description
x1	SearchPoint1	Frame	Teach	Target coordinate 1
x2	SearchPoint2	Frame	Teach	Target coordinate 2
x3	SearchPoint3	Frame	Teach	Target coordinate 3
x4	SearchPoint4	Frame	Teach	Target coordinate 4
x5	ViaPoint	Frame	Teach	Via point to minimize the risk of collision
x6	ApproachDistance	Float	Teach	Distance of approaching the target
x7	SearchVelocity	Float	Specify	Velocity when searching for contact
x8	TriggerForce	Float	Specify	Force to trig contact
x9	Velocity	Float	Specify	Velocity of the manipulator

**Table F.10:** Input parameters of the execution of a Calibrate skill.

### Precondition

Check that the gripper is closed.

### Postcondition

None

### Programming

The programming sequence of the *Calibrate* skill is given below.

1. Specify *Velocity* (x9), *TriggerForce* (x8), *ApproachDistance* (x6), and *SearchVelocity* (x7).
2. Teach *SearchPoint1* (x1), *SearchPoint2* (x2), *SearchPoint3* (x3), and *SearchPoint4* (x4).
3. Teach *ViaPoint* (x5).

Subsequent to the teaching part a workstation reference frame is not calculated. In order to do this the taught sequence must be executed.

## F.2 Device Primitives

In this section the developed device primitives of the gripper and the manipulator are listed.

	Description
MoveLinRel	Moves the TCP linear relative to the current coordinate
MoveCart	Moves the TCP to a cartesian coordinate
MoveCirc	Moves the TCP in a circle using the current-, a via- and an end coordinate
MoveJoint	Moves each joint to a specified angle
SearchRel	Searches for contact in a direction and distance relative to the TCP
SetBase	Defines the workspace of the manipulator
SetTool	Defines the tool of the manipulator
SetJointImpedance	Specifies a stiffness for each joint
SetCartesianImpedance	Specifies a stiffness in the cartesian space
SetGraspStiffness	Specifies the stiffness of the manipulator 0 in the y-direction
SetPositionControl	Sets the manipulator in position control
SetDesiredForce	Specifies a build in force application of the manipulator
LWRDataCB	Publish data from LWR
Interrupt	Stops the manipulator in its current motion

*Table F.11: Device primitives of the manipulator.*



	Description
InitCheck	Check that the gripper is online and initialized
Init	Check that the gripper is initialized, if not do initialization
Loop	Check communication to the gripper
Disconnect	Announce disconnect
Home	Home the gripper (initialize)
Move	Position the jaws
Stop	Stop the movement
FastStop	Emergency stop
AckStop	Acknowledge a fast stop (emergency stop)
Grasp	Grasp a part with a given width
Release	Release a part by opening to a given width
SetAcc	Set the acceleration of the gripper
GetAcc	Get the acceleration of the gripper
SetForceLimit	Set the grasp force of the gripper
GetForceLimit	Get the grasp force of the gripper
SetLimits	Set the software limits of the gripper (work envelope)
GetLimits	Get the software limits of the gripper
ClearLimits	Clear the software limits of the gripper
OverdriveOn	Activate overdrive mode to allow higher grasping force
OverdriveOff	Deactivate overdrive mode
TareForce	Zero the force measurement
GetSysState	Get the system state (device state of the gripper)
GetGraspState	Get the grasp state*
GetGraspStatis	Get the grasping statistics
ResetGraspStatis	Reset the grasping statistics of the gripper
GetWidth	Measure the current width of the gripper (the jaws' position)
GetSpeed	Measure the current speed of the gripper
GetForce	Measure the current force of the gripper

**Table F.12:** Device primitives of the gripper. \*(Idle, Grasping, No part found, Part lost, Holding, Releasing, Positioning)



# Annex



## iPad styrer vaks lille industrirobot

Af: Sybille Hildebrandt, Journalist  
11. maj 2012 kl. 10:12

**Tre ingeniørstuderende fra Aalborg Universitet har gjort højteknologisk robot interaktiv, så den i princippet kan styres af hvem som helst ved simpelt tryk på robotten eller en iPad.**

Fremtidens industrivirksomheder beskæftiger ikke kun mennesker af kød og blod. De giver også fuldtidsarbejde til små robotter, der styrer rundt og går til hænde, hvor de ansattes kroppe og hjerner kommer til kort. De små teknologiske hjælpere får produktionen til at glide og sørger for, at opgaverne løses hurtigt og effektivt, så produkterne kan komme ud på markedet i en fart.

Den fagre nye verden venter lige om hjørnet, for sådan en robot, kaldet Little Helper ++, er faktisk allerede ved at tage form på Aalborg Universitet. En stribe ph.d.-studerende og specialestuderende har gennem flere år løst hver deres afgrænsede opgaver, som langsomt, men sikkert har [bygget robotten op fra grunden](#) og har gjort den hurtigere, mere fleksibel og brugervenlig.

»Vi har opbygget et kompliceret softwaresystem, der gør det nemt at programmere robotten og styre den ved hjælp af direkte berøringer eller gennem en iPad.«

»Dette system gør en person helt uden robotkundskaber i stand til at oplære robotten i løsning af nye arbejdsopgaver,« fortæller Christian Carøe, som har udviklet softwaren i samarbejde med sine to entusiastiske kammerater Mikkel Hvilshøj og Casper Schou i forbindelse med deres specialeår på Maskin & Produktion på Aalborg Universitet.

### **Robotter skal være omstillingsparate**

Den første Lille Hjælper blev udviklet af en gruppe studerende på Aalborg Universitet tilbage i 2007. Tanken var, at det kun skulle være et afgrænset projekt, men det greb om sig. Lille Hjælper virkede som en magnet på nysgerrige studerende, der brændte for at nørkle med ny, avanceret robotteknologi.

Robottens potentiale er da også stort – så stort, at EU har valgt at tage udviklingen af den under sine vinger og gøre den til en del af det europæiske robotprojekt TAPAS, hvis visioner rækker langt ud over Danmarks grænser:

Overalt i Europa er industrien udfordret af, at brugerne forventer at få produkter med mange forskellige features, som kan målrettes til deres eget personlige behov. Dertil kommer, at de tit og ofte skifter produktet ud med et nyt.

Et produkts levetid er altså blevet meget kortere, hvilket stiller store krav til virksomhederne om hele tiden at poste nye produkter på markedet. Det kræver, at virksomhederne er meget fleksible og hurtigt kan tilpasse deres produktionsapparat til en ny situation, og det er her, at Little Helper ++ kommer ind i billedet.

## **Robotten er ikke en erstatning – kun en hjælp**

I Danmark arbejder man typisk med traditionelle robotter, som er designet til at løse faste opgaver uafhængigt af de ansatte, som de derfor også typisk er afskærmet fra. Men det gør det omstændeligt at ændre produktionen til fremstillingen af nye produkter.

»Visionen i TAPAS er at udvikle en mobil robot, der kan indgå som en fleksibel produktionsressource til at hjælpe de ansatte. Little Helper++ er et bud på sådan en robot, som ikke skal opfattes som en erstatning for de ansatte, men som en assistent. De ansatte skal nemt og hurtigt kunne instruere robotten, så den kan løse opgaver efter nye behov, uden at det kræver en total omstrukturering af produktionssystemet,« fortæller Mikkel Hvilshøj.

Den nye Little Helper er altså yderst brugervenlig og vaks til at lære nyt. Og den kan indgå i mange forskellige teams, fordi den af de tre specialestuderende hånd er udstyret med mange avancerede egenskaber.

## **Little Helper kan dreje og bøje sig**

Ud over at robotten kan føle sig frem ved hjælp af nogle særlige sensorer i alle samlinger, kan den også dreje, rotere og bøje sig rundt om syv forskellige akser, hvilket er én akse mere, end hvad der er normalt for en traditionel robot.

Det lyder alt sammen flot på papiret, men det har også vist sig at fungere fortræffeligt i praksis.

Robotten har med succes løst **to manuelle arbejdsopgaver**, som er typiske hos ventilationsvirksomheden Grundfos, men som er udført i et kopieret miljø på Aalborg Universitet:

1. [Samling af en rotor til dybvandspumpen SQ-Flex](#)
2. [Samling af dele til roteren til dybvandspumpen SQ-Flex](#)

Robottens næste udfordring bliver at løse selvsamme opgaver hos Grundfos til oktober i forlængelse af TAPAS-projektet.

## **Ny software er genbrug, så det basker**

Softwaren er baseret på det såkaldte Robot Operation System, som er open source-kode, der er stykket sammen af kode-stumper fra andre robotsystemer.

Mens de fleste traditionelle robotter også er sværvægtede, der vejer på den forkerte side af 50 kg, er robotarmen på Little Helper en lille spirrevip på kun 14 kg.

»Formålet er at give robotten en lang række færdigheder, som den kan bruge til at løse mange forskelligartede opgaver. Disse færdigheder er afgørende for robottens intuitive interface, fordi de gør det muligt at lave softwareblokke, som efterfølgende er simple for brugeren at sammensætte og tilpasse til et specifikt behov,« siger Casper Schou.

---

URL: <http://videnskab.dk/teknologi/ipad-styrer-vaks-lille-industrirobot>

© Ophavsretten tilhører Videnskab.dk







On the enclosed Appendix CD you will find:

- CAD models of Little Helper++
- Task descriptions
- KRL source code
- C++ source code
- Pictures from the project
- Documents

*Today the need for highly automated yet flexible production equipment is increasing in order to satisfy a new production segment demanding mass production with high flexibility. An autonomous mobile robot is an example of such production equipment as it offers a flexibility superior to that of a stationary robot. The vision of this project is not only to construct a mobile robot, but also to ease the programming of the mobile robot and thus make the programming of a new task available to the production operator. This is accomplished through the introduction of a skill-based approach, which brings the programming of industrial tasks to a level where robotics expertise is no longer needed. This report documents the construction of a mobile robot, the software development and the concluding industrial verification of the entire system.*



**AALBORG UNIVERSITY**

**Master's Thesis in Manufacturing Technology**

**Christian Carøe, Mikkel Hvilshøj & Casper Schou**

**Department of Mechanical and Manufacturing Engineering**