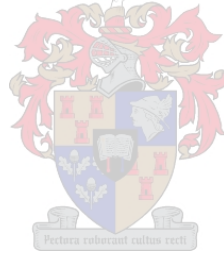


# **Evaluation of C# for a Station Controller in a Reconfigurable Manufacturing System**

by  
Rainer Graefe

*Thesis presented in partial fulfilment of the requirements for the degree  
of Master of Engineering (Mechatronic) in the Faculty of Engineering at  
Stellenbosch University*



Supervisor: Prof AH Basson

December 2016

## DECLARATION

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: .....

Copyright © 2016 Stellenbosch University  
All rights reserved

# Abstract

Reconfigurable manufacturing systems (RMSs) are aimed at dynamic situations, such as varying products, variations in production volume requirements and changes in available resources. RMSs distinguish themselves from other types of manufacturing systems in that they can quickly adapt to a new product being introduced without the need for long reconfiguration times, and can therefore cost effectively produce smaller batch sizes.

RMSs in research environments in most cases used Agent Based Control (ABC), but the main automation vendors in the industry do not support ABC. This inhibits the acceptance of RMSs by the industry. For this research, C# was investigated as an alternative to ABC, since C# can provide for many of the functionalities of agents, yet is a more widely known language than ABC. Furthermore, C# is an object-oriented programming (OOP) language and thus possesses characteristics aligned with the core characteristics of reconfigurable manufacturing systems.

The focus of this thesis is to determine the suitability of C# for the development of the control software for RMSs. This thesis describes the design, implementation, testing and evaluation of a reconfigurable stacking and buffering station. The controller was implemented in C# and made use of the ADACOR architecture.

The physical test-setup was built to evaluate the reconfigurability of the controller in a series of reconfiguration experiments.

The thesis showed that the controller could handle all the hardware interfaces without problems, since C# generally simplifies the task of hardware interfacing. OOP characteristics helped making developing and maintaining the code an intuitive task. The stacking station handled all communication with the cell controller correctly, which proved that it could easily be integrated into a distributed control architecture.

## Uittreksel

"Reconfigurable manufacturing systems" (RMSs) is gemik op dinamiese situasies, soos veranderende produkte, veranderings in produksievolumes en veranderinge in beskikbare hulpbronne. RMSs onderskei hulself van ander tipes vervaardigingstelsels deurdat hulle vinnig kan aanpas by nuwe produkte wat bekendgestel word sonder dat dit nodig is om die stelsel eers lank te herkonfigureer, en kan sodoende kleiner lotgroottes koste-effektief produseer.

RMSs maak in navorsingmilieus meestal gebruik van "Agent Based Control" (ABC), maar die hoof outomatisasie-verkopers in die industrie ondersteun nie ABC nie. Dit belemmer die aanvaarding van RMSs in die industrie. Vir hierdie navorsing is C# as 'n alternatief vir ABC ondersoek omdat C# baie van die funksionaliteite kan voorsien wat aangetref word in ABC, maar terselfdertyd 'n meer bekende taal is as ABC. Verder is C# 'n objekgeoriënteerde programmerings- (OOP) taal en beskik dus oor karakteristieke wat in lyn is met die kernkarakteristieke van RMSs.

Die fokus van hierdie tesis is die geskiktheid van C# vir die ontwikkeling van beheersagteware vir 'n RMS. Hierdie tesis beskryf die ontwerp, implementering, toetsing en evaluering van 'n herkonfigureerbare stapel- en bufferstasie. Die beheerder was in C# geïmplementeer en het van die ADACOR-argitektuur gebruikgemaak.

Die fisiese toets-opstelling was gebou om die herkonfigureerbaarheid van die beheerder te kan evalueer aan hand van 'n reeks herkonfigurerings eksperimente.

Die tesis het gewys dat die beheerder sonder probleme alle hardware-intervlakke kon hanteer, omdat C# dit oor die algemeen vergemaklik om met hardware te kommunikeer. OOP karakteristieke was nuttig om die ontwikkeling en instandhouding van die program intuïtief te maak. Die stapelstasie het alle kommunikasie met die selbeheerder korrek hanteer, wat bewys het dat dit probleemloos in 'n verspreide beheerargitektuur opgeneem kon word.

# Table of Contents

List of Figures.....	viii
List of Tables.....	x
List of Abbreviations .....	xi
<b>1 Introduction .....</b>	<b>1</b>
1.1 Background .....	1
1.2 Objectives .....	3
1.3 Motivation .....	3
<b>2 Literature Review .....</b>	<b>4</b>
2.1 Reconfigurable manufacturing systems.....	4
2.2 Desirable RMS characteristics.....	5
2.3 Autonomous reconfiguration ability.....	7
2.4 Control architectures for manufacturing systems.....	7
2.4.1 Overview of classical control architectures.....	7
2.4.2 Holonic control architectures for RMSs .....	10
2.4.3 PROSA.....	10
2.4.4 ADACOR.....	12
2.5 IEC 61499 Function Blocks.....	12
2.6 Objects vs agents.....	15
2.6.1 Key properties of agents .....	15
2.6.2 Key properties of OOP .....	17
2.6.3 Objects and agents compared .....	18
2.7 Conclusion.....	19
<b>3 Case Study.....</b>	<b>20</b>
3.1 CBI's need for an RMS with traceability .....	20
3.2 Background of stations to be automated.....	25
3.3 Design specifications.....	26
3.3.1 CBI needs analysis .....	27
3.3.2 Functional requirements.....	28
3.4 Design of the stacking station .....	29
3.4.1 Station physical architecture .....	29
3.4.2 Mechanical design.....	30
3.4.2.1 Transportation system / pallets .....	30
3.4.2.2 Fixtures .....	32

3.4.2.3	Robot.....	33
3.4.2.4	Grippers.....	34
3.4.2.5	Buffer.....	38
3.5	Conclusion.....	40
<b>4</b>	<b>Control Software Selection.....</b>	<b>41</b>
4.1	Software requirements .....	41
4.2	Software comparison .....	42
4.2.1	Java .....	42
4.2.2	C.....	43
4.2.3	C#.....	43
4.2.4	C++/CLI .....	44
4.3	Chosen controller and chosen software .....	44
4.4	Evaluation of OOP as an alternative to agents .....	45
4.4.1	Encapsulation of behaviour .....	45
4.4.2	Dynamics, complexity, autonomy and hierarchy.....	46
4.4.3	Modularity and integrability.....	47
4.4.4	Hard and soft real time requirements .....	47
4.5	Conclusions drawn from literature .....	48
<b>5</b>	<b>Control Software Implementation.....</b>	<b>49</b>
5.1	Controller functional requirements.....	49
5.2	Control architecture.....	49
5.3	Separation of tasks among multiple threads.....	52
5.4	Inter-holon communication.....	53
5.5	Responsibilities and functionalities of the various holons .....	54
5.5.1	Generic holon.....	54
5.5.2	Product holon .....	56
5.5.3	Operational holon .....	56
5.5.3.1	Generic operational holon.....	57
5.5.3.2	Pole .....	57
5.5.3.3	Gripper holon .....	57
5.5.3.4	Pole stacking robot holon.....	58
5.5.3.5	Pole storage holon.....	59
5.5.4	Task holon.....	60
5.5.4.1	Stacking task holon .....	61

5.5.4.2	Typical sequence of events for a stacking order .....	62
5.5.5	Supervisor holon .....	64
5.5.5.1	Task coordinator.....	64
5.5.6	Staff holons.....	65
5.5.6.1	Product manager.....	65
5.5.6.2	Pallet manager .....	65
5.5.6.3	Buffer manager.....	66
5.5.7	Communicator holon .....	66
5.6	Conclusion.....	67
<b>6</b>	<b>Evaluation .....</b>	<b>68</b>
6.1	Experimental setup.....	68
6.2	Aspects to measure for reconfiguration .....	69
6.3	Calibrations .....	70
6.3.1	Tool calibration .....	70
6.3.2	Sensor calibration.....	71
6.3.3	Workspace calibration .....	71
6.4	Reconfiguration tests and measurements .....	71
6.4.1	Testing robustness of network communication .....	71
6.4.2	Adding a similar buffer.....	72
6.4.3	Customization test.....	73
6.4.4	Throughput rate tests.....	74
6.4.5	Scalability test – adding another transverse conveyer.....	76
6.4.6	Alterations to operational holon.....	77
6.4.7	Disturbance handling tests.....	77
6.4.8	Ramp-up tests.....	78
6.5	Results.....	78
6.6	Recommendation for shorter reconfiguration times .....	79
6.7	C# evaluation.....	79
<b>7</b>	<b>Conclusions and Recommendations .....</b>	<b>81</b>
<b>8</b>	<b>References .....</b>	<b>83</b>
	<b>Appendix A – Human-Machine Interface screenshots.....</b>	<b>86</b>
	<b>Appendix B – Flow diagrams.....</b>	<b>88</b>
	<b>Appendix C – Code for Gripper Holon .....</b>	<b>91</b>
	<b>Appendix D – Inter-station messages.....</b>	<b>95</b>

<b>Appendix E – Calibration procedures .....</b>	<b>97</b>
E.1 Calibrating the proximity sensor.....	97
E.2 Calibrating tools, such as the gripper .....	98
E.3 Calibrating workspaces such as the buffer, and pallets.....	99



# List of Figures

Figure 1 Mechatronics Automation and Design Research Group (MADRG) laboratory.....	2
Figure 2 Evolution of control architectures.....	8
Figure 3 Basic building blocks of an HMS and their relations.....	11
Figure 4 Examples of function blocks with event signals and data signals .....	13
Figure 5 Function blocks with event and data connections (left) combined into a network (right) .....	13
Figure 6 Function blocks which contain function blocks on their inside, are called composite function blocks.....	14
Figure 7 Evolution of programming approaches.....	18
Figure 8 A fraction of CBI's product variety.....	20
Figure 9 Four variants of the QA-range: 1 pole, 2 pole, 3 pole and 3 pole + neutral .....	21
Figure 10 Tripping mechanisms of poles which form part of a stack are interconnected	22
Figure 11 Stacking station inbetween two conveyers.....	25
Figure 12 Functional analysis.....	28
Figure 13 Stacking station physical architecture.....	29
Figure 14 Stacking and buffering station layout .....	30
Figure 15 Bosch Rexroth 320x320 pallets can carry six fixtures each .....	32
Figure 16 To make stacking possible, poles have to be gripped at their short sides .....	34
Figure 17 Gripper fingers used for stacking poles.....	35
Figure 18 Exploded view of buffer.....	39
Figure 19 Key properties of agents and OOP.....	45
Figure 20 Stacking station control architecture .....	51
Figure 21 Memory shared among various threads.....	52
Figure 22 Inheritance hierarchy .....	54
Figure 23 Station event log simplifies diagnostics .....	55
Figure 24 Robot contains coordinates of surrounding workspaces' reference points.....	60
Figure 25 Sequence of events from order placement to completion of order .....	63
Figure 26 Stacking and buffering station experimental setup.....	68
Figure 27 Buffer added to demonstrate scalability.....	72
Figure 28 Pallet turned 90° to represent new type of product .....	73
Figure 29 Effects of path contour on robot speed.....	77
Figure 30 Stacking station human-machine interface .....	86

Figure 31 Cell controller human-machine interface.....	87
Figure 32 Kuka controller flow diagram.....	88
Figure 33 Flow diagram for multi-threading on stacking station and cell controller.....	89
Figure 34 Communicator flow diagrams.....	90
Figure 35 Adjustable proximity sensor .....	97
Figure 36 HMI for adding a new tool.....	98
Figure 37 HMI for adding a new workspace.....	99
Figure 38 HMI for adding new workspace configuration.....	100

## List of Tables

Table 1 Circuit breaker components.....	21
Table 2 Components used for experiment.....	69
Table 3 Conveyer configurations yielding different throughput rates .....	75
Table 4 Code for Gripper Holon .....	91
Table 5 XML message structure for inter-station communication .....	95
Table 6 Type of messages that the station controller can receive.....	96
Table 7 Type of messages sent out by the station controller.....	96

# List of Abbreviations

ABC	Agent-Based Control
ACL	Agent Communication Language
ADACOR	ADaptive holonic COntrol aRchitecture
AGC	Automatic Garbage Collection
CAD	Computer Aided Design
CC	Cell Controller
CLI	Common Language Interface
DAQ	Data Acquisition Device
DMS	Dedicated Manufacturing Systems
DOF	Degrees of freedom
FB	Function Block
FBD	Function Block Diagram
FBDK	Function Block Development Kit
FIFO	First-in-first-out
FIPA	Foundation for Intelligent Physical Agents
FMS	Flexible Manufacturing Systems
GUI	Graphical User Interface
HMI	Human-Machine Interface
HMS	Holonic Manufacturing System
IDE	Integrated development environment
IEC 61131-3	International Electrotechnical Commission 61131-3
JNI	Java Native Interface
KRL	KUKA Robot Language
Linq	Language integrated query
MADRG	Mechatronics, Automation and Design Research Group
NC	Normally closed
NO	Normally open
OH	Operational holon
OOP	Object oriented programming
PH	Product holon
PLC	Programmable logic controller
PROSA	Product-Resource-Order-Staff Architecture
RMS	Reconfigurable Manufacturing System

RS-232	Recommended Standard 232
SCARA	Selective Compliance Assembly Robot Arm
SH	Supervisor holon
TCP	Tool centre point
TCP/IP	Transmission Control Protocol/Internet Protocol
TH	Task holon
XML	EXtensible Markup Language

# 1 Introduction

## 1.1 Background

Manufacturing systems today face challenges such as short product life cycles, rapidly changing manufacturing technologies, unpredictable demand due to fluctuations in the market and increasing demand from the consumer to be able to manufacture customized products (Van Leeuwen & Norrie, 1997) in high varieties, yet inexpensively and without compromising quality (Van Brussel, et al., 1999). The time that a particular product is on the market is oftentimes far shorter than the time it takes to design a new production line, set it up and get production started. For reasons like these, it is preferable to have manufacturing systems which are easily reused for various new products. Manufacturers who are able to start producing faster can roll out their products sooner and therefore have a significant economic advantage over their competitors. Reconfigurable Manufacturing Systems (RMSs) are designed to shorten these ramp-up times and address the aforementioned challenges.

An RMS is defined as “being designed for rapid adjustment of production capacity and functionality, in response to new circumstances, by rearrangement or change of its components.” (Mehrabi, et al., 2000). Making alterations to a (conventional) dedicated manufacturing line which was designed without future reconfigurations in mind, is costly and can take a considerable amount of time. RMSs on the other hand are designed from the beginning to simplify the process of making changes to both the hardware and the control software of the manufacturing system, thereby drastically shortening ramp-up time while also saving costs and keeping the impact on the ongoing production to a minimum.

Conventional manufacturing systems, like dedicated manufacturing systems (DMSs) still have their place in countries with large, established economies, where production volumes are high and demand is more stable. Reconfigurable Manufacturing Systems offer a more workable solution for fast developing countries with smaller, emerging economies and rapidly changing markets. South Africa falls in this category, where production volumes are generally not as high but where product variety is wide, such that frequent changeovers are required. In South Africa many production processes are still performed by hand, but changing labour laws and quality requirements might necessitate the implementation of new automation systems. It is thus advisable to design those new manufacturing systems to be reconfigurable, even though it is not necessarily advisable to change already existing manufacturing systems to be reconfigurable.

This research project is one of several projects carried out by the Mechatronics, Automation and Design Research Group (MADRG) at the Mechanical and Mechatronic Engineering Department at Stellenbosch University. The MADRG is conducting research into various aspects of reconfigurable manufacturing systems: transportation systems for RMSs; singulation of parts; machine vision for part identification, orientation, quality inspection and autonomous calibration; machine learning for route planning; and various assembly processes and machining processes. Where other researchers focus

mainly on a factory-level, for the MADRG the focus lies mainly on a cell- and station-level. Research is conducted in collaboration with a research group at the Central University of Technology. South African industry partners include CBI Electric: low voltage, hereafter referred to as “CBI” and AAT Composites (manufacturers of aerospace and automotive parts).

Some of the other research projects on a reconfigurable manufacturing cell done by MADRG on the CBI case study are shown in Figure 1. They include the controller for a conveyor system with a pallet magazine, the design of singulation units using machine vision, a revolving helical drum (not shown), and a vibratory bowl; an eye-in-hand camera attached to the end-effector of a robot to eliminate the need for manual calibration, a modular welding robot, and a quality assurance cell. Controllers used include PC's, Siemens PLC's and a Beckhoff embedded PC. Software used for the research projects includes Java, C#, Function Blocks, Agent Based Control, LabView, Erlang and a combination of these.

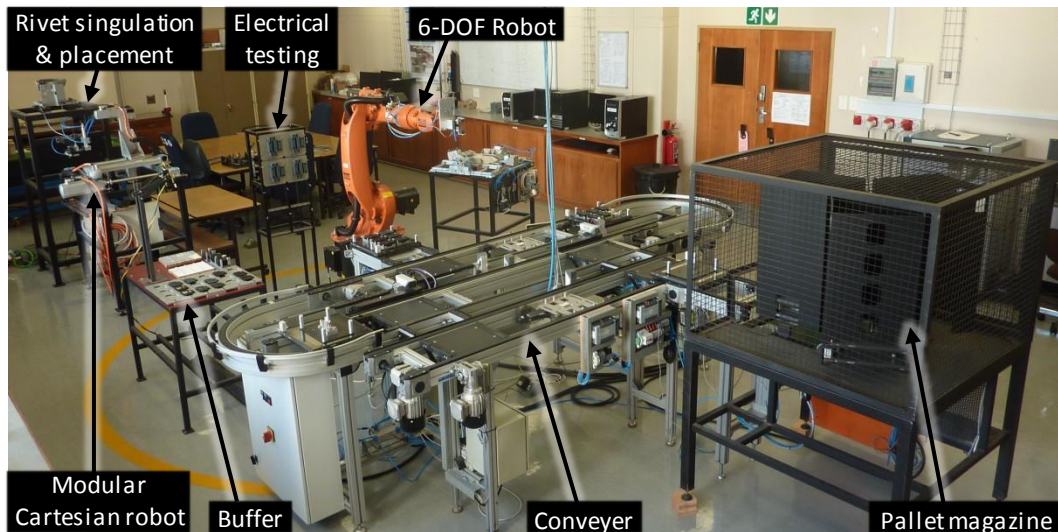


Figure 1 Mechatronics Automation and Design Research Group (MADRG) laboratory

For a case study, it was considered automating one of the production processes for CBI. CBI is a large South African company manufacturing a wide variety of circuit breakers (a few hundred variants) for the local and international market. Batch sizes that they deal with vary from as little as 20 to about 60000 per day. Orders are sometimes unpredictable and they therefore frequently need to changeover between different types of products. At first they wish to automate some of the production processes of those products which are produced in high volumes. Only if and when the need arises, should the existing system be reconfigured so that those products with lower volumes can also be catered for at a later stage but with minimal impact on the already ongoing production and with minimal additional capital investment. CBI would therefore be a potential user of a reconfigurable manufacturing system which is why they agreed to provide material for a case study where some of their production processes are to be automated.

## 1.2 Objectives

The aim of the research presented in this thesis is to evaluate the suitability of C# for a controller in a reconfigurable manufacturing system. This entails:

- Implement a station controller for an RMS cell using C#.
- Evaluate the OOP approach in a case study: Design a reconfigurable stacking and buffering station of which the throughput rates are at least as high as the current manual system.

## 1.3 Motivation

RMSs in research environments in most cases used Agent Based Control (ABC) (Vrba, et al., 2011; Cândido & Barata, 2007; Hall, et al., 2005), but the main automation vendors in the industry do not support ABC. This inhibits the acceptance of RMSs by the industry. The focus of this research is to investigate whether Object-Oriented Programming (OOP) would be suited for implementing RMSs, since OOP is used more widely and therefore is likely to be more acceptable to industry than ABC. C# is being used for this research, since it is a widely used OOP language, which would be acceptable by industry.

The controller implemented for this research was on a lower level (i.e. station-level) than what has usually been done thus far (i.e. cell control level) therefore achieving an optimal throughput rate was more important than autonomous adaptability.

## 1.4 Thesis overview

To place this thesis into context, relevant findings from the literature are discussed in the next chapter. The case study and the design of the experimental setup are covered in chapter 3. Chapter 4 contains a comparison between viable control software and evaluates OOP as an alternative to Agent Based Control. Detail on the implementation of the control software is discussed in chapter 5. Experiments were performed to evaluate the controller's reconfigurability. These experiments are described in chapter 6, followed by a conclusion in the final chapter.



## 2 Literature Review

Research relating to reconfigurable manufacturing systems, fundamentals of established control architectures, and previously used control software is discussed in this chapter. Key concepts of object oriented programming (OOP) are compared to properties of agent based control. These findings aided the design and implementation of a reconfigurable stacking station and its controller, as discussed in later chapters.

### 2.1 Reconfigurable manufacturing systems

To put RMSs in contrast with conventional manufacturing systems, Dedicated Manufacturing Systems (DMSs) and Flexible Manufacturing Systems (FMSs) are briefly discussed here.

DMSs are designed to manufacture a relatively small variety of fairly similar products at high volumes. The manufacturing line is set up to handle only those few selected products, and the processes are optimized to cost-effectively achieve high throughput rates which should match the demand. The equipment installed, the software written and the factory layout chosen only keep the selected products and target production volumes in mind, but make no specific provision for future reconfigurations. When, at some later stage, a new and different product has to be manufactured that has not been catered for in the initial design, then the investments required in terms of time, skilled labour costs and new equipment to make the necessary alterations are often not worthwhile. For this reason, entirely new production lines are built rather than re-using the existing lines. Alternatively, when the decision is made to re-use the existing production line and make the necessary changes to it, then those changes typically require a lot of capital to replace equipment and extensive changes need to be made to the software to accommodate the new product. Finding and removing all the possible hardware and software errors after attempting such an unforeseen reconfiguration, until production can resume at full scale, can take a tremendous amount of time, during which the production of the initial products would also be hindered.

FMSs are manufacturing systems designed to handle a variety of products. To cater for several possible machining requirements, the machinery is usually multifunctional and equipped with a wide variety of tools, usually more than is actually needed for the products at hand. This redundancy results in FMSs to oftentimes be unnecessarily expensive, and according to Mehrabi, et al (2000, p. 403) also have drawbacks such as utilizing inadequate system software, being not highly reliable, and they can easily become obsolete because their software/hardware is fixed and therefore do not foresee advances in manufacturing technologies.

The need for RMS arises in scenarios where production volumes are lower, product variety is wider, changes in the market are frequent and unpredictable, or shorter ramp-up times are required. Due to frequent changes in the market and changes to production methods, the product life cycles are oftentimes shorter than the time it takes to get conventional production lines ready to start production. The need for shorter ramp-up times therefore arises.

RMSs are designed for the exact required production capacity in mind and additional equipment is only added if and when needed at a later stage. Hence, unlike FMSs, RMSs are not unnecessarily expensive and do not easily become obsolete since they can quickly adapt to advances in technology. The additional development time required to cater for possible reconfigurations will quickly be regained by the time and costs saved by being able to reconfigure more easily at some later stage.

Although of course any manufacturing system can be reconfigured, not all systems are designed to facilitate future reconfigurations. RMSs are specifically designed with the possibility of future reconfigurations kept in mind. This applies to both the hardware and the control software. RMSs are designed such that they can be reconfigured quickly and easily in order to keep reconfiguration costs low, impact on ongoing production to a minimum and prevent tedious hours of debugging afterwards. To accomplish these goals, designers should consider the requirements discussed in the next section.

## 2.2 Desirable RMS characteristics

To aid in the design of RMSs, the characteristics of RMS are discussed in this section.

According to Koren & Shpitalni (2010), the six core characteristics of RMSs are:

- Customization: flexibility limited to a product family.
- Convertibility: design for functionality changes.
- Scalability: design for capacity changes.
- Modularity: system components are modular.
- Integrability: interfaces between system components promote rapid integration.
- Diagnosability: design for easy diagnostics which allows for quick ramp-up after reconfiguration.

The first three are characteristics of the whole RMS and are critical for a system to be considered reconfigurable. The last three characteristics, on the other hand, allow reconfiguration to be done efficiently. Therefore a system must reflect modularity, integrability and diagnosability to be considered reconfigurable.

RMSs are generally considered to be holonic manufacturing systems (HMSs, discussed in section 2.4.2) and Christensen (1994) reports that the HMS Consortium identified the following as critical factors for systems:

- Disturbance handling: Provide better and faster recognition of and response to machine malfunctions, rush orders, unpredictable process yields, human errors, etc.
- Human integration: Support better and more extensive use of human intelligence.
- Availability: Provide higher reliability and maintainability despite system size and complexity.
- Flexibility: Support continuously changing product designs, product mixes and small lot sizes.

- Robustness: Maintain system operability in the face of large and small malfunctions.

Except for the human integration factor, the other factors parallel RMS's properties.

When considering IEC 61499 function blocks as a means to control holonic systems, Christensen (1994) identified capabilities required to provide autonomy. These capabilities should also be considered for manufacturing systems:

- encapsulated local data bases;
- local process/machine control;
- local optimization;
- local product tracking;
- self-scheduling;
- self-diagnosis;
- self-repair;
- self-configuration.

Further, for distributed and cooperative holonic architectures, controllers should provide communication and negotiation capabilities, as required by Christensen (1994) of function blocks.

One of the particular capabilities of RMSs is the ability to be dynamically reconfigured. Christensen (1994), when considering the use of IEC 61499 function blocks for RMSs, implied that dynamic reconfiguration is when humans, other holons, or the holonic application itself can:

- dynamically create, modify, destroy and relocate both instances and type definitions of functional units (e.g. function blocks or agents);
- dynamically create and destroy connections among functional units;
- dynamically activate and de-activate functional units;
- perform version management of functional units and applications.

The above is closely related to a system's flexibility and disturbance handling capabilities, i.e. its ability to manage change dynamically. The lack of support in IEC 61131-3 for these capabilities (Brennan, 2007) limits its use in the control of RMSs.

In many practical situations, such as the CBI application mentioned in section 1.1, the situation is dynamic (including requiring occasional reconfigurations), but the context does not require the RMS to be able to autonomously reconfigure. On the contrary, it is the author's impression that many companies would be uncomfortable with such a level of automation. Autonomous reconfiguration is therefore for this study not seen as a requirement, but manual reconfiguration is, as are flexibility and disturbance handling capabilities. The CBI application therefore demonstrates that the relative importance of the requirements given in this section depends on the case being considered. One cannot assume that all practical systems must fully support all of the above requirements.

## 2.3 Autonomous reconfiguration ability

Since RMSs are designed to exactly match the production demand, the capital required for the equipment is generally about the same as for DMSs. However, since designers of RMSs need to keep possible future reconfigurations in mind, software development could potentially take longer and consequently be more expensive, since this task involves highly skilled labour. An optimal level of autonomy for the system should be found: The more intelligent the system is to be, the more adaptations it can make by itself, but the design time and accompanying costs will increase accordingly.

On a hardware level, certain machinery will need to be flexible to handle a variety of products. If the hardware is to be fully autonomous, then during a product change over the machines would need to adjust or exchange tools themselves. For the transportation system to be fully autonomous, it would need to be designed such that it can transport any products or materials to and from any of the stations and provide functionality for accurate, repeatable alignment of transported goods.

On a software level, the system needs to be designed to cater for several different configurations, between which the operator is able to convert back and forth, or able to add/introduce (teach) new configurations. If reconfigurations on a software level are supposed to happen without any human intervention, then commands being passed from the cell controller to the station controller should be responsible for reconfiguring the station. The latter would only be possible if a product changeover occurs for which the system is already physically equipped. Otherwise, manual involvement would be required.

Since costs associated with making systems autonomously reconfigurable increase exponentially, it is generally not worthwhile to develop systems to be fully autonomous. CBI prefers the system to rather not be too autonomously reconfigurable. For this case study all hardware changes required human intervention and the software was designed to dynamically adapt to only minor changes (such as product changeovers), whereas major changes would require human intervention.

## 2.4 Control architectures for manufacturing systems

Holonic control architectures have characteristics which make them well suited for controllers of RMSs. To better understand holonic control approaches, what they are and why they are suitable for controllers of RMSs, other control architectures shown in Figure 2 are discussed first. According to Dilts, et al (1991) "it is the function of the control architecture [of a manufacturing system] to allocate decision making responsibilities to specific control components ... [and to] determine the interrelationships between control components, thereby establishing the mechanism for coordinating the execution of those various decisions".

### 2.4.1 Overview of classical control architectures

Dilts, et al (1991) compared the four control architectures described below along with their advantages and disadvantages in terms of the reliability/fault tolerance, modifiability/extensibility, and reconfigurability/adaptability of the control system:

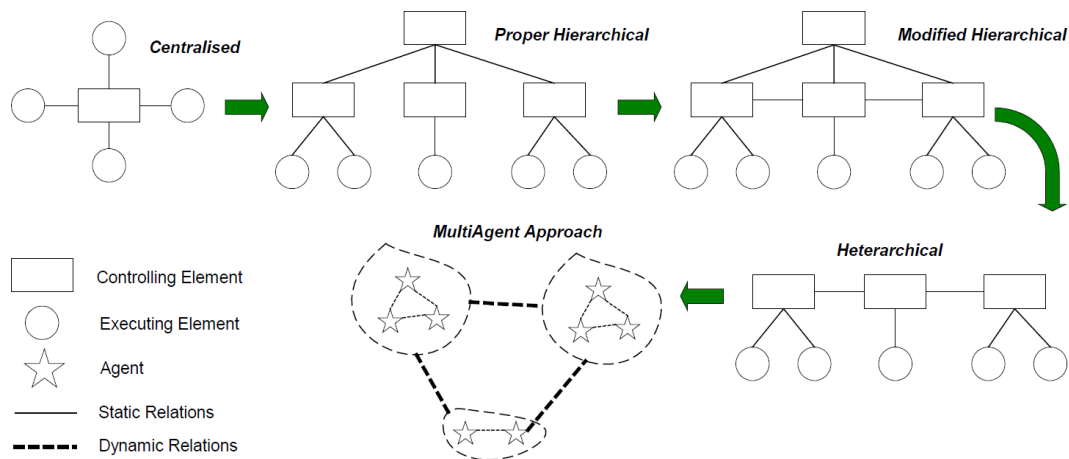


Figure 2 Evolution of control architectures (adapted from Barata (2003) and Dilts, et al. (1991))

**Centralized control architectures** have one central controller on which all information resides and on which all decision-making takes place. These decisions are based on information from sensors and machine controllers. The central controller has a fairly complex logic since it is solely responsible for all of the interactions between the various components of the system.

Advantages are that the centralized controller has access to complete global information, which makes global optimization possible. Its disadvantages are that it is slow and has inconsistent responsiveness due to the variety of tasks carried out by the control unit. Because there is only one control unit, the system is less reliable. If the central control unit malfunctions, the entire system is immediately affected. Modifications, extensions and reconfigurations to the software are difficult, because the logic in centralized architectures is hidden in the program and global data structures (Duffie & Piper, 1987).

For **proper hierarchical control architectures**, the control tasks are subdivided into branches with distinct levels where the subordinate levels of each branch have no autonomy, but are obliged to always execute the instructions given from the supervisor levels. At the highest level of the hierarchy, “most global goals are decided upon and a long-range strategy is formulated. ... Decisions made at this highest level commit the entire hierarchical structure to a unified and coordinated course of action which would result in the selected goal or goals being achieved” (Simpson, et al., 1982). Interactions between sub-branches have to be handled by the more superior levels of the hierarchy.

Advantages are reduced software development problems, gradual implementation, redundancy, allowance for differing time scales (among various branches), the possibility of incremental addition of vertical slices of the control architecture and fast response times.

Difficulties with dealing with dynamic adaptive control and difficulties with making future unforeseen modifications are some of the disadvantages which make it unsuitable as a control architecture for an RMS. Further, whenever one of the branches has a malfunctioning link, that entire branch will be paralysed and the decision making

process of the hierarchical controller is compromised, because it is missing information that it is relying on.

**Modified hierarchical control architectures** were developed to improve on some of the drawbacks of hierarchical control. They are similar to proper hierarchical control architectures in that there are still distinct levels, but controllers on subordinate levels are able to interact and cooperate directly with one another, thereby gaining some local autonomy and taking over some basic responsibilities of the main controller. This allows the main controller to respond more quickly to requests from the subordinate controllers.

Modified hierarchical architectures have all the advantages and most of the disadvantages of proper hierarchical architectures.

Additional advantages are the ability of local systems to have local autonomy and the ability to off-load some linkage tasks to local controllers. These advantages make this type of control more robust with respect to disturbances (Cassandras, 1986). Since there are less interactions with the supervisory controller, faults are more easily diagnosed.

Additional disadvantages include connectivity problems and increased difficulty of control system design.

Within **heterarchical control architectures**, there are no master/slave relationships between control components of the system. Executing elements, however, are still subordinate to control components. Each control component has full local autonomy and “supervisory decision making [is] located at the point of information gathering rather than in a central location” (Duffie & Piper, 1987).

Advantages of heterarchical control are full local autonomy, reduced software complexity because of enhanced modularity, implicit fault-tolerance since the control strategy is not impacted when one of the nodes fails (Barata, 2003), ease of reconfigurability and adaptability and faster diffusion of information.

Disadvantages are technical limits of controllers, high likelihood of only local optimization (without global information), lack of availability of software, and that a high network capacity is required.

A conclusion that can be drawn in review on the abovementioned control architectures is that the more autonomy each sub-controller has, the more fault-tolerant and robust the system becomes and the easier it is to diagnose errors. The more distributed the intelligence of the system is, the more modular it becomes, and thus easier to extend or modify. On the other hand, it becomes less likely that global optimization is achieved when information gathering and decision making happens only locally. When optimal throughput rates are to be achieved (globally), one should opt for control architectures with supervisors which have a more global view and are thus more likely to find a global optimum.

## 2.4.2 Holonic control architectures for RMSs

Arthur Koestler (1967) defined the term 'holon' which is a combination of the Greek words 'holos' and suffix '-on', meaning 'whole' and 'part' respectively. This word he used to describe units in biological or social systems which on the one hand form a part of a larger system, yet at the same time are a complete self-contained system themselves. This same concept can also be transferred to control architectures of manufacturing systems which is why the building blocks of holonic control architectures are called 'holons'.

Giret and Botti (2005) report that the Holonic Manufacturing Systems Consortium gives the following definitions:

Holon: *"An autonomous and co-operative building block of a manufacturing system for transforming, transporting, storing and/or validating information and physical objects. The holon consists of an information processing part and often a physical processing part. A holon can be part of another holon."*

Holarchy: *"A system of holons that can co-operate to achieve a goal or objective. The holarchy defines the basic rules for co-operation of the holons and thereby limits their autonomy."*

According to Van Brussels, et al. (1998, p. 256) holonic organizations are used to provide stability in the face of disturbances, flexibility in the face of change, and efficient use of resources. The HMS concept combines the best features of hierarchical and heterarchical organisation (Dilts, et al., 1991) by preserving the stability of a hierarchy while providing the dynamic flexibility of a heterarchy (Van Brussel, et al., 1998).

With holonic control, it is possible to choose between a heterarchical and a hierarchical approach (Barata, 2003). A disadvantage of the more heterarchical approach is the unpredictable behaviour that can occur when holons take a lot of initiative themselves and do not coordinate their work with a global scheduler, which can potentially lead to low performance (Bongaerts, et al., 2000).

Two common holon-based architectures are PROSA and ADACOR which are described in the next two sections.

## 2.4.3 PROSA

PROSA is a reference architecture for holonic manufacturing systems, developed at KU-Leuven by the Production engineering, Machine design and Automation division. The architecture consists of the three basic holon types shown in Figure 3, namely the product holon (PH), resource holon (RH) and order holon (OH) and can additionally incorporate staff holons (SH), hence the name PROSA (Product-Resource-Order-Staff-Architecture).

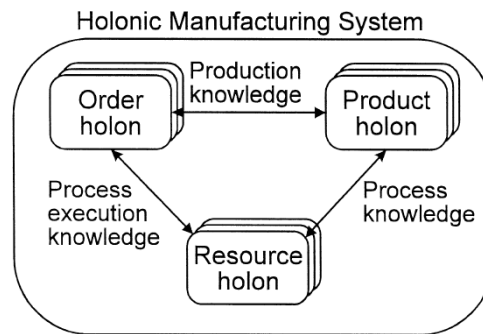


Figure 3 Basic building blocks of an HMS and their relations (Van Brussel, et al., 1998)

Van Brussels, et al. (1998) provide the following descriptions:

A **resource holon** contains a physical part, namely a production resource of the manufacturing system, and an information processing part that controls the resource. It offers production capacity and functionality to the surrounding holons (Wyns, et al., 1996). It holds the methods to allocate the production resources, and the knowledge and procedures to organise, use and control these production resources to drive production. A resource holon is an abstraction of the production means such as a factory, a shop, machines, furnaces, conveyors, pipelines, pallets, components, raw materials, tools, tool holders, material storage, personnel, energy, floor space, etc.

A **product holon** holds the process and product knowledge to assure the correct making of the product with sufficient quality. A product holon contains consistent and up-to-date information on the product life cycle, user requirements, design, process plans, bill of materials, quality assurance procedures, etc. As such, it contains the 'product model' of the product type, not the 'product state model' of one physical product instance being produced. The product holon acts as an information server to the other holons in the HMS.

An **order holon** represents a task in the manufacturing system. It is responsible for performing the assigned work correctly and on time. It manages the physical product being produced, the product state model, and all logistical information processing related to the job. An order holon may represent customer orders, make-to-stock orders, prototype-making orders, orders to maintain and repair resources, etc. Often, the order holon can be regarded as the workpiece with a certain control behaviour to manage it to go through the factory, e.g., to negotiate with other parts and resources to get produced.

Also shown in Figure 3 are the types of information being interchanged between the basic types of holons. The OH is the one that drives production. As soon as an order has been placed, the OH will request the production knowledge from the PH, i.e. the sequence of events that the product must undergo. The OH will then attempt to book and schedule available RHs which are capable of executing the required production processes. The RH which gets assigned the task will request process knowledge from the PH such as coordinates, dimensions and machining parameters (e.g. spindle speed).



After completion of its sub-task the RH will notify the OH, so that the unfinished product can proceed to the next step, where another RH can execute the next process.

Diagnosing logistical or process errors is the responsibility of the OH, whereas machine errors have to be diagnosed by the RH. In the authors view, the PH is not involved in the major decision making and is also not responsible for diagnosing errors, but mainly serves as an information server, i.e. under normal circumstances it simply provides the information that was requested. If that information was not available, the PH will request the missing information from a higher level controller, or an operator (via an HMI). Those holons which requested the information will either have to wait or focus their attention on another task.

The PROSA architecture can also include staff holons which are used to give guidance to the basic holons and provide them with expert knowledge. The final decisions are still made by the basic holons, but they will try to follow the advice of the staff holons wherever possible and will only decide not to follow that advice when the staff holons are performing badly due to disturbances (Van Brussel, et al., 1998).

#### **2.4.4 ADACOR**

ADACOR (ADaptive holonic COntrol aRchitecture) emerged after PROSA, and is based on a set of autonomous and cooperative holons with learning capabilities, self-organization and supervisor entities. It incorporates adaptive control which dynamically balances between a centralized structure when the objective is global optimization, and a more heterarchical structure in the presence of unexpected events and modifications. (Leitão & Restivo, 2006)

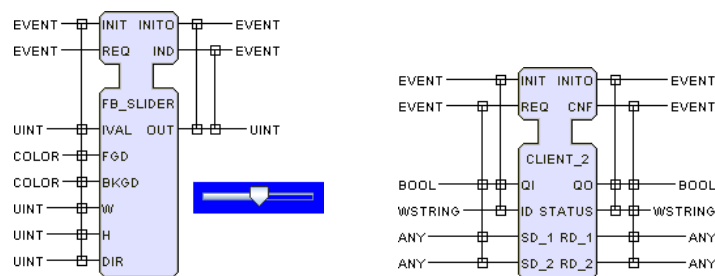
ADACOR incorporates the product (PH), task (TH) and operational (OH) holon classes which closely correspond to the product, order and resource holon of the PROSA architecture. To improve the adaptability of the architecture, ADACOR further incorporates the supervisor holon (SH) which introduces coordination and global optimization in decentralized control and is responsible for the formation and coordination of groups of holons. (Leitão & Restivo, 2006)

In a similar way that ants in nature can learn to locate the path to new food sources based on the smell of pheromones deposited by other ants, so can ADACOR holons learn from disturbances which helps them re-organize themselves after disturbances occurred. This capability to learn allows them to evolve and adapt to the new environment (Leitão & Restivo, 2006). This feature, which is not found in PROSA makes ADACOR more complex and is approaching artificial intelligence.

### **2.5 IEC 61499 Function Blocks**

Others have successfully used function blocks (FBs) to implement holonic controllers for RMSs in research environments. However, function blocks are not widely accepted by industry because there is limited development software available and skills in that field are scarce. Nevertheless, it is investigated to find its strengths and weaknesses so that an informed decision can be made.

Function Block Diagrams (FBD) is a graphical language designed for Programmable Logic Controllers (PLC's), where the programmer produces a network of functional entities with inputs and outputs for events and data. Whenever a function block's input event signal is triggered, its internal logic will change the values of the output variables. Immediately thereafter, or after a specified amount of time, an output event signal will be triggered. The output event signals are typically connected to the input event signals of other function blocks so that a sequence of events can be executed. The output signals can also be used to control hardware, to communicate over networks or to manipulate GUI elements. Similarly, the input signals can be used to pick up signals from sensors, to listen to a network port, or to get user input from the GUI. Figure 4 shows examples of function blocks used for GUI elements and network communication. A complex network of function blocks can be formed by interconnecting the event signals and data signals of several function blocks, as shown in Figure 5. Such a network will then have more functionality than the individual function blocks it is made of.



a) FB for graphical user interface

b) FB for communication services

Figure 4 Examples of function blocks with event signals and data signals (Christensen, 2011)

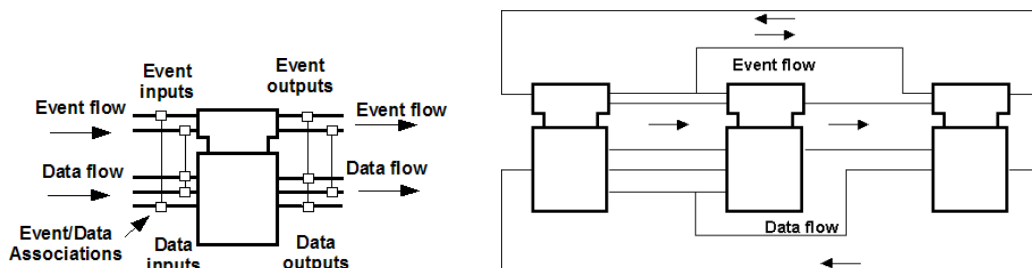
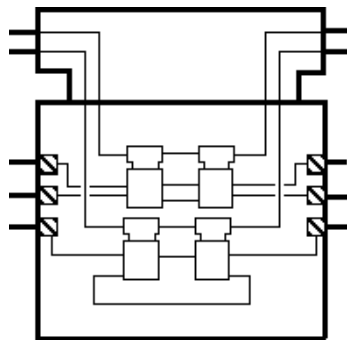


Figure 5 Function blocks with event and data connections (left) combined into a network (right) (Christensen, 2011)

The internal working of a function block can be programmed using 61131-3 standard languages such as Structured Text, Ladder Diagram, Instruction List or Function Block Diagram (FBD). The latter option means that a composite function block can in turn be made up of basic or composite function blocks, as shown in Figure 6. Compared to structured programming, this is very similar to calling several other methods from within an encapsulating method. The difference is that several function blocks can be triggered at the same time, thereby running in parallel, whereas method calls in a program would

need to happen sequentially. The ability to trigger several function blocks simultaneously can cater for processes which require precise timing.

Within the same application, several function blocks can be programmed using different of the abovementioned languages and one can convert between the languages at any time, and the development software will do the translating. This allows several developers with different skills to all work at different parts of the program, using their preferred language, yet all parts can work together harmoniously (Hristu-Varsakelis & Levine, 2005, p. 276). Function blocks from existing libraries can effortlessly be imported, thereby eliminating the need for reinventing the wheel and drastically reducing development time.



*Figure 6 Function blocks which contain function blocks on their inside, are called composite function blocks*

Function blocks encapsulate their internal working and provide standard interfaces which allows them to be easily integrated into a network of other function blocks or to interface with hardware or communication services.

A big drawback with function blocks is that there is limited development software available and skills in that field are scarce. The most well-known is Function Block Development Kit (FBDK) which lacks advanced debugging tools. It is hard to trace back the root of errors by following the many lines on Function Block Diagrams. Not all connections are always indicated by continuous lines but are often indicated by labels or end up inside blocks with no visible connection to another block even when in fact some of the blocks are somehow linked to one another. This makes debugging a very demanding task with the available software, as was also confirmed by Mulubika & Basson (2013).

Also, the concept of dragging and dropping blocks, which are then interconnected by lines is very dissimilar to conventional ways of programming. However, the Structured Text that can be used within the function blocks is syntactically very similar to Pascal and has many of the conventional control structures such as iteration loops, conditional execution, functions and arrays. Additionally, Structured Text provides pre-defined variable types such as timers and counters. One can also write function blocks with shared data, and function blocks which use their own instance of a data block, which is almost equivalent to objects with static variables and creating instances of a class (thus

objects). However, such instances have to be created at compile time since dynamic memory allocation is not supported in the IEC 61131-3 standard.

What speaks for function blocks in terms of reconfigurability, is their reusability, their standard interfaces, their encapsulation and integrability. Furthermore, the development of human-machine interface elements is possible using function blocks (Viatkin, 2007).

In conclusion, function blocks are useful in terms of integrability, modularity and real-time execution, but not in terms of diagnosability (with the currently available development tools). The lack of support of dynamic memory allocation along with the diagnosability issues was seen as too big a constraint so that function blocks were not considered for this research.

## **2.6 Objects vs agents**

Agent based control (ABC) has been successfully implemented in RMS research environments because agents have several characteristics which make them useful for RMS controllers. Since in this thesis' research OOP-control is considered as an alternative to ABC, the differences and similarities between the two concepts are discussed in this section.

### **2.6.1 Key properties of agents**

ABC has been applied in a large proportion of RMS research since it suits the requirements of RMSs so well. Not all aspects can be considered here, but some key capabilities are pointed out:

According to Bellifemine, et al (2007), an agent is essentially a special software component with an interoperable interface to an arbitrary system, and is characterized, among other things, by:

- **Autonomy:** It can independently carry out complex and often long-term tasks, i.e. it operates without the direct intervention of humans or others and has control over its actions and internal state.
- **Pro-activity:** An agent can take initiative to perform a given task even without an explicit stimulus from a user.
- **Ability to communicate:** Agents can interact with other entities to assist with achieving their own and other's goals.

Furthermore (Bellifemine, et al., 2007), an agent is social, because it cooperates with humans or other agents in order to achieve its tasks. An agent is reactive, because it perceives its environment and responds in a timely fashion to changes that occur in the environment. If necessary, it can be mobile, with the ability to travel between different nodes in a computer network, and it can learn, adapting itself to its environment and to the desires of its users.

The abovementioned characteristics make agents a very attractive option for implementing an RMS's control system:

Agents are easy to integrate with one another because they make use of the same communication protocol (e.g. FIPA ACL (Bellifemine, et al., 2007)) and adhere to the same rules. A newly created agent will be registered at the agent management system (described below) and when it wishes to publicize its services, it will announce them at the directory facilitator (described below) so that other agents can request to make use of its services. In this way agents can always be aware of one another's presence and the system is self-structured.

Agents are also modular and integrable because they have an interoperable interface to an arbitrary system, and they keep their internal working to themselves, always showing the same interfaces to the outside world. This means that nothing needs to be changed in the rest of the system when a change is made to an individual agent, e.g. one can let an agent run on a new platform or make it control a different physical resource and as long as the interfaces behave the same, the rest of the system will not be affected. Therefore, humans can seamlessly be integrated into an automated manufacturing system by implementing an HMI on the inside of an agent which will then appear like a regular agent as long as it provides a standard interface to the rest of the system.

Since agents have control over their internal state, they can easily report on their health status while they are still responsive, and since agents are proactive, they are capable of detecting when other agents stopped responding and can try to resolve the problem. This makes them suitable for diagnosability and self-repair.

According to Van Brussel, et al. (1998), disturbances and changes to the system can easily be handled when using the Contract Net Protocol as a negotiation mechanism between agents. Because each agent is autonomous, it will try to find the optimal solution to its local problem, and because it is also proactive, it will, after disturbances occurred, re-evaluate its solution to the current problem. When using the Contract Net Protocol, the system is self-scheduling: idling agents or agents whose current job is nearing completion, will bid more to get a new offer than agents who have already been assigned more than one task. This ensures that when tasks can be performed in parallel, the work load will be distributed amongst agents rather than piling up at an individual agent, and will reduce agent idling time.

In ABC, dynamic reconfiguration is made possible by the following two components of an agent platform: the agent management system and the directory facilitator. Their main roles are as follows (Bellifemine, et al., 2007):

The agent management system is responsible for managing the operation of an agent platform, including the creation and deletion of agents, the migration of agents to and from the agent platform, and maintaining a directory of identifiers of all agents present within the agent platform and their current state (e.g. active, suspended or waiting).

The directory facilitator (DF) provides "yellow pages" services to other agents and maintains a list of services that agents can offer. Every agent that wishes to publicize its services to other agents would request the registration of its agent description in a DF. At any time an agent may request the DF to modify its agent description. The registration with a DF does not imply a future commitment or obligation on the part of

the registering agent and an agent can subsequently request deregistration of its description at any time.

The agent management system and DF allow agents to be developed off-line, and then be dynamically added to the rest of the system without bringing the system to a halt. They also allow dynamically creating, modifying, and destroying instances of and connections among functional units, and the dynamic activation and de-activation thereof.

### **2.6.2 Key properties of OOP**

The four key concepts of OOP are abstraction, encapsulation, inheritance and polymorphism (Van der Linden, 2002). There are some important synergies between these OOP concepts and the modularity, integrability and diagnosability characteristics of RMSs:

Modularity and encapsulation are closely related, in that a software object can be instantiated as many times as required, because each instance's data is independent of other instances, unless explicitly specified differently. RMS modules are often considered to be holons (Van Brussel, et al., 1998) and a set of OOP objects is well suited to form the software part of a holon. New instances of holons without hardware (e.g. the order holon in PROSA) can easily be created in an OOP implementation by instantiating another object. Similarly, when the order holon's tasks have been completed, its object-instance can simply be deleted. Inheritance in OOP provides another dimension of modularity for which there is no parallel in agents. When creating a new holon during reconfiguration, an appropriately designed OOP implementation will allow the object representing the new holon to inherit "modules" of functionality from previously defined super-classes. This allows for code re-use, thus reducing controller reconfiguration times and risks.

Integrability can be promoted through objects' properties of inheritance, polymorphism and abstraction: A key part of abstraction is that an object hides the complexity of its internal working and presents a simple interface to the outside world. Further, inheritance (e.g. of an abstract class) and polymorphism can be used to define a generic interface for all holons that provides, for example, generic diagnostic and communication aspects. Therefore, abstraction, inheritance and polymorphism can be used in an OOP implementation of a holon's information processing part to create generic interfaces, which enhances an object's integrability. However, OOP implementations will still require the development of, for example, communication interfaces using general approaches (e.g. XML strings exchanged over TCP/IP links), while agent platforms have much of that functionality built in.

Mature OOP software development platforms are available, thus providing excellent software tools to diagnose control software problems. Further, the multi-threading abilities of OOP languages allow diagnostic processes to run concurrently with normal operations, thereby enabling near-real time monitoring of the state of the RMS and providing good diagnosability.

### 2.6.3 Objects and agents compared

A number of researchers have compared objects and agents. A brief summary of the comparisons is first given here, followed in the next section by an assessment with this thesis' case study in mind.

Odell (2002), adapting the work of Parunak (1997), placed OOP and agent programming in a broader perspective (Figure 7) and pointed out that a fundamental difference between object-oriented and agent-oriented programming is that agents can invoke their own units, whereas objects do not.

	Monolithic Programming	Modular Programming	Object-Oriented Programming	Agent Programming
Unit Behavior	Nonmodular	Modular	Modular	Modular
Unit State	External	External	Internal	Internal
Unit Invocation	External	External (CALLed)	External (message)	Internal (rules, goals)

Figure 7 Evolution of programming approaches (Odell, 2002)

Two key areas that can differentiate the agent-based approach from the traditional OOP approach are autonomy and interaction (Odell, 2002). Proactive agents poll the environment for events and other messages to determine what action they should take, while objects are conventionally passive.

For Booch (2000, cited by Odell (2002)), employing agents with object systems is useful, because the agent-based approach:

- provides a way to reason about the flow of control in a highly distributed system;
- offers a mechanism that yields emergent behaviour across an otherwise static behaviour; and
- codifies the best practices in how to organize concurrent collaborating objects.

Although there are certain similarities between object- and agent-oriented approaches (e.g. both adhere to the principle of information hiding and recognize the importance of interactions), there are also several important differences (Jennings & Bussmann, 2003):

- Objects are generally passive in nature: they need to be sent a message before they become active.
- Although objects encapsulate state and behaviour realization, they do not, in principle, encapsulate behaviour activation. Thus, any object can invoke any publicly accessible method on any other object.

- Object orientation fails to provide an adequate set of concepts and mechanisms for modelling complex systems.
- Object-oriented approaches provide only minimal support for specifying and managing organizational relationships (basically, relationships are defined by static inheritance hierarchies).

In OOP, there is no "built-in" provision in current languages for an object to "advertise" its interfaces. The result is that the programmer needs to have some idea what interface to ask for (Odell, 2002). In contrast, an agent can employ mechanisms such as the directory facilitator, mentioned above, or specialized broker agents to which other agents can make themselves known for various purposes but are otherwise unlisted to the rest of the agent population.

## **2.7 Conclusion**

In this chapter, research into RMSs was reviewed and the desirable characteristics for reconfigurable manufacturing systems were set out so that the controller designed for the case study could be evaluated against those characteristics. Furthermore, a summary on various types of control architectures and their characteristics was provided, to help deciding on the most suitable control architecture for the station controller at hand. Since agent based control has been used widely in RMS research environments, but C# was to be used for this research, a comparison between objects and agents was drawn after discussing the key properties of agents and OOP. The next section describes the case study which was chosen so that this thesis' research is closely related to an industrial application.



### 3 Case Study

Circuit breakers, also known as trip switches, are electrical safety devices and are installed into circuits which require protection against too high current levels. As soon as a circuit draws too much current for too long (e.g. in excess of 60A for longer than 4ms) the circuit breaker trips to interrupt the current flow through the circuit. Its purpose is therefore to prevent damage to components and to protect humans from injury or death. They play a critical role where safety is a concern, and should therefore be highly reliable and conform to high quality standards.



Figure 8 A fraction of CBI's product variety (CBI Circuit Breakers - Product Listing [S.a.]

#### 3.1 CBI's need for an RMS with traceability

For a case study, it was considered automating one of the production processes for CBI. CBI is a large supplier of circuit breakers which they deliver to the local and international market. Figure 8 shows only a small selection of the wide variety of products that CBI manufacture. The circuit breakers are designed for various applications and circuit configurations, such as single phase or three-phase circuits with or without a neutral line. As a result, they come in different shapes and sizes, make use of different technologies, are made of different materials and have different ampere ratings. The QA-range which forms part of the larger Q-range is shown in Figure 9 and comes in four variants: 1 pole, 2 pole, 3 pole and 3 pole + neutral. The variants are either just a single riveted pole, or a stack of 2 to 4 similar poles, riveted together. Stacked poles work as a unit: if any one pole trips, it will cause the other poles of that unit to immediately trip together along with it. The Q-range has by far the highest production volumes and hence, when deciding to automate, it is most sensible to design the system for that product family first, and only later on reconfigure the system to accommodate the other types of circuit breakers when the need arises.



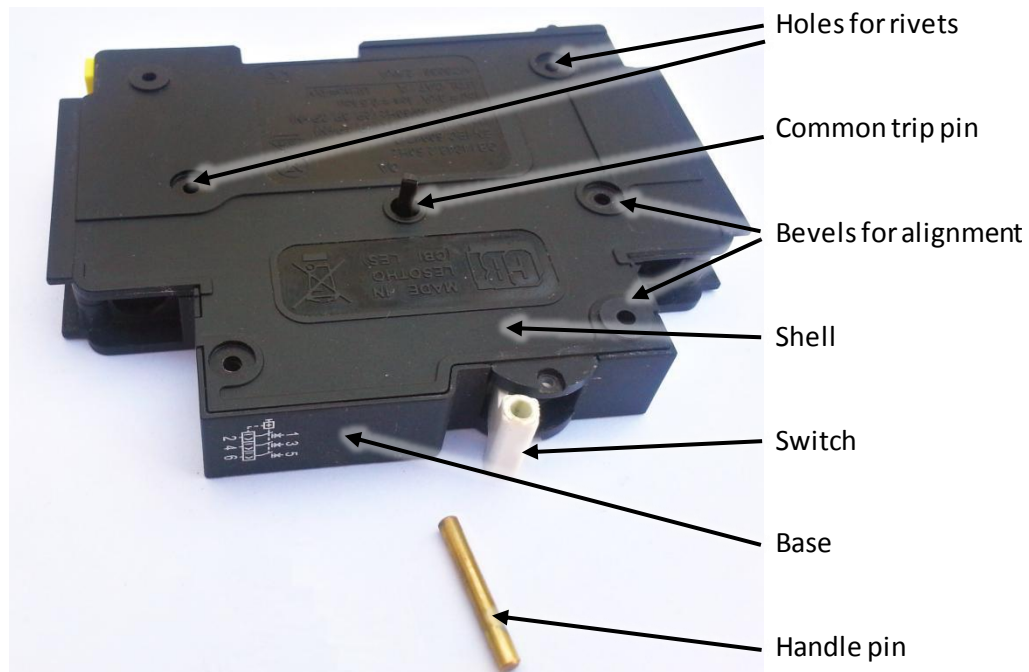
Figure 9 Four variants of the QA-range: 1 pole, 2 pole, 3 pole and 3 pole + neutral  
(Technical Downloads [S.a.]

Figure 10 shows a pole forming part of a stack, before it has been stacked and riveted. The components which are of significance for this case study are listed in Table 1 below.

Table 1 Circuit breaker components

Component	Description
<b>Base</b>	The bottom part of the casing is called the base. It is a plastic mould with features to hold all the internal components in place.
<b>Shell</b>	The shell is the top part of the casing. For stacks, shells of the top-most poles have no bevels for alignment or holes in the middle, but shells of the bottom and middle poles do have bevels which align the poles when stacked and a hole in the middle to let the common trip pin go through.
<b>Common trip pin</b>	The common trip pin is only present in stacks and must be inserted before the stacking process. It interconnects the trip mechanisms of the various poles of a stack and is responsible for letting all other poles trip as soon as any one pole of the stack trips.
<b>Switch</b>	The switch allows a person to reset a tripped circuit breaker or to intentionally trip a circuit breaker. Before the handle pin can be inserted, the holes for all the switches of a stack must first be lined up.
<b>Handle pin</b>	The handle pin connects the handles of all the poles of a stack. When resetting a trip switch after it has tripped, this pin, which penetrates all the handles, ensures that all poles are reset together. The handle pin can be inserted by hand at any time after the stacking process.
<b>Clip-in</b>	A yellow part which is present in each pole. It provides extra grip for when the circuit breaker is installed. The clip-in can be inserted or removed at any time by hand for the Q-range. Other ranges require the clip-in to be inserted before the shell has been placed.
<b>Rivets</b>	There are typically about 6 rivets per breaker. Their primary purpose is to keep the shell and the base together (for single poles), and for stacks, their secondary purpose is to also keep the individual poles of a stack together.

Even though the internal components of the individual poles are almost identical, the poles can nevertheless not be used interchangeably for different layers of a stack. For example, the pole that was intended for the top-most layer of a stack cannot be used for a middle or bottom layer due to some external features which differ.



*Figure 10 Tripping mechanisms of poles which form part of a stack are interconnected by the common trip pin. Handles are connected by the handle pin.*

The individual parts which make up the circuit breakers are produced at the CBI factory in Johannesburg, South Africa, by predominantly automatic or semi-automatic processes such as casting, folding, punching, winding, pressing and bending. The loose parts are then transported to their plant in Lesotho where the circuit breakers are assembled, tested, printed, and boxed. At the Lesotho plant there is not a single fully-automated process. Most processes involve manual labour only, and for those processes which involve machinery, the machinery is operated by hand and sometimes activated by foot.

A wide variety of circuit breakers are being assembled at the Lesotho plant and quantities for batch orders vary from as little as 20 to a few thousand, meaning that a worker could be dealing with several different products during the course of a day. When dealing with such dynamic production scenarios, manual labour is generally the best choice for getting the job done in spite of manual labour being more expensive than the operating costs of automation equipment. When compared to conventional or even sophisticated automation equipment, people are extremely adaptable to changes in production methods. This allows them to quickly change over between known products or, after a brief training period, start assembling an entirely new type of product. For CBI it is thus not economically viable to automate all of their processes, since the overall costs of employees is still less than the total costs associated with an automated assembly line. Nevertheless, it cannot be ruled out that people who do

repetitive work or who are still in the training process occasionally make mistakes without noticing, thereby compromising on quality. Since about 66% of CBI's products are being exported, quality is of utmost importance. Having to recall faulty breakers after they have already been exported, incurs tremendous costs and can affect the company's reputation, especially because superior quality is what sets them apart from their competitors. Automating a few selected processes which can diminish or eliminate the impact of human errors could therefore still be for the better of the company and its employees. The high initial costs related to automation can be justified by the benefits obtained from automating certain parts of the factory, as discussed next.

Roughly, the process plan for assembling a circuit breaker of the QA-range is as follows:

1. Electrical components and the common trip pin are placed in the bottom part of the casing (the base).
2. With the casing still open, it is ensured that everything was placed correctly.
3. The top cover of the casing (the shell) is placed and the switch is set to the ON-position.
4. While the pole is still unriveted, it is placed into a ramp-wave tester, an electrical-test device, capable of testing several poles simultaneously. There it is subjected to currents of different intensities which should cause the pole to trip at the right time. After having passed the test, the switch will be in the OFF-position.
5. The electrical-tester indicates by means of lights if the tested poles conformed to specifications or if they have failed the test. Poles which failed the test are removed and sent to be reworked.
6. Poles are stacked in the correct order on top of one another (when the product to be made is a 2-pole, 3-pole or 3-pole+neutral). Since poles are only stacked after having passed the test, their switches will always be in the OFF-position and all the common trip pins will all have the same orientation, which simplifies alignment.
7. Rivets are inserted into the holes of the casing.
8. The product is riveted to seal the casing and to join poles which belong together.
9. For stacks, the handle pin is pushed through the switches of all poles.
10. Clip-in(s) are inserted.
11. Final quality inspection is done by hand.
12. Finished products are packaged.

Quality checks are performed at several stations. The most important test is accomplished by the electrical test station, where each pole is subjected to a pre-programmed electrical current to determine whether it functions correctly in accordance to its ratings. Circuit breakers which consist of multiple poles are not tested as a complete unit, but instead the poles are tested individually before they are stacked and riveted. Testing therefore happens when the casing could potentially open up and allow the internal parts to fall out or disarrange. The ramp-wave tester is currently activated by hand and circuit breakers are placed into it by hand. Only those circuit breakers which pass the test proceed to the next stations, whereas those that did not pass the test are sent back to be reworked. Since each and every breaker is subjected to this test before being packaged and shipped, theoretically no breakers which do not conform to the quality standards would ever be leaving the factory. However, it

occasionally happens that a circuit breaker leaves the factory when in fact it should not have passed the quality tests. In other words, some of the breakers were either never tested or they failed the test but have accidentally proceeded to the next step when they should have ended up in the rework line. This can be attributed to human error and hence CBI is considering the automation of certain processes, to ensure that each circuit breaker leaving the factory has indeed been tested.

To prevent any human errors being introduced during the quality assurance process, not only should the electrical test station itself be automated but also some of the preceding and succeeding processes:

- Visual inspection while internal components are still visible, to ensure no components are missing and everything is in the right place.
- Shell placement.
- Placement of poles into the tester, and activation of the tester.
- Electrical testing.
- Removal of poles from the tester, and the decision what should happen to the pole next.
- Stacking and matching of poles which together form a unit.
- Placing and riveting rivets.
- Printing product information on casing which corresponds to what's inside.

All other non-critical processes should remain manually operated where possible. Having the abovementioned processes automated will eliminate the risk of faulty breakers leaving the factory and allows complete traceability of each and every product, i.e. where each product has been and which processes it was subjected to.

Being able to trace the route that a product has taken through the factory and recording how each product was handled by each machine not only allows one to discard any faulty objects but also gives an insight into which machine or action could have been the cause of a possible fault. Such a tracing tool can save a lot of effort in resolving any issues, thereby reducing ramp-up time. Furthermore, small errors can be detected sooner before they propagate.

To ensure that the integrity of the poles is not affected after the visual inspection, all processes up to and including riveting must be automated. This however, will only prevent non-conforming poles from passing the test but because the product information is only printed at a much later stage, and because different products may have a similar outer appearance it could happen that wrong information is printed on a fully-functional product which can be worse than shipping poor quality products. To allow full traceability, the printing process should therefore also be automated. Once the breakers have been riveted and the information printed, there's not much that can go wrong and it is safe to let humans handle the processes which follow after printing.

Although the capital requirements for new automation equipment is generally much greater than to employ people, the benefits of traceability and having no more defect products can outweigh the costs of automating the quality assurance cell. Due to CBI's large product variety, the automated parts of the system should be reconfigurable to

simplify frequently occurring product changeovers, and to simplify the process of scaling up or extending the system at a later stage to cater for new products. CBI thus needs a reconfigurable quality assurance cell which provides traceability.

Other researchers of the research group are investigating the visual inspection station, the electrical test station, the riveting station and control strategies for logistical aspects and the conveyor system. For this research project, a stacking station in a lab context was automated. Next, the design specifications are determined and the mechanical design of the station is discussed.

### 3.2 Background of stations to be automated

As shown on the diagram below, the stacking station is situated in-between two conveyers, each transporting different types of pallets. Only single poles are transported on the first conveyer. The fixtures on those pallets are therefore only required to accommodate single poles. Pallets on the second conveyer contain riveting fixtures capable of holding stacked 1-pole, 2-pole, 3-pole or 3-pole+neutral circuit breakers. To place the stacking station into context with the preceding and succeeding stations, they are also briefly discussed.

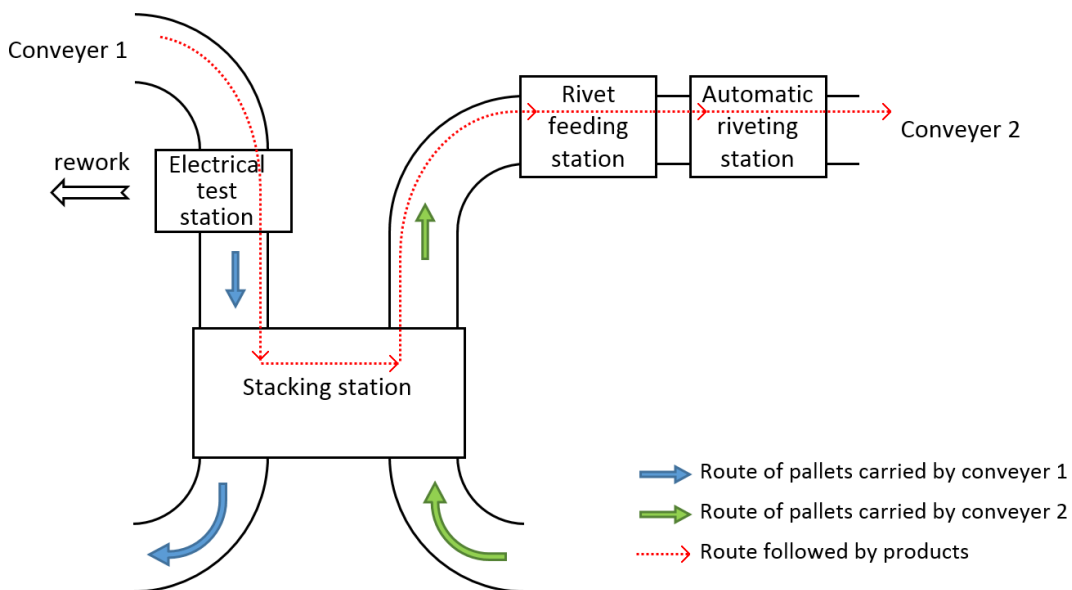


Figure 11 Stacking station inbetween two conveyers

The **electrical test station** receives a pallet filled with fully assembled but unriveted single poles. The poles are then tested individually to see whether they comply with their electrical requirements. Poles which do not pass the test are being discarded for rework, and poles which do pass the test are placed on another pallet and sent to the stacking station. To achieve the desired throughput rate at the electrical test station it was required that the pallets can carry 6 poles at a time.

The **stacking station** receives pallets containing tested single poles, all of which have passed the electrical test. Those poles must then be transferred to pallets containing fixtures in which they can be riveted. In addition, when multiple-pole breakers are to be manufactured, the stacking station needs to stack the individual poles on top of one another to form, for example, triple-pole breakers. Since poles which fail the electrical test do not reach the stacking station, a complete set of single poles is not always available to form a multiple-pole breaker. Therefore, the stacking station also needs the ability to temporarily store poles in a buffer until a complete multiple-pole breaker can be stacked. Once a complete set has been stacked, it is sent to the rivet feeding station.

The **rivet feeding station** receives a stacked set of poles within a riveting fixture on a pallet and must feed rivets of the correct length into the holes of the circuit breaker casing. Rivets are manually fed into a vibratory bowl where they are singulated and thereafter oriented. The placement of rivets into the holes is an automated process. After all holes of the breaker have been filled, the riveting fixture is sent to the **automatic riveting station** where rivets are riveted to prevent the finished assembly from falling apart.

Whenever pallets arrive at a particular station, an accompanying message must be sent by the cell controller to that particular station, informing the station of the pallet contents. Whenever a station requests that a pallet be transported away, the cell controller must be informed of the pallet contents, that it can plan for the pallets' next destinations. The content of the pallets will change at two stations: The electrical test station will only pack poles onto its outgoing pallet if they passed the electrical test, and decides itself in which order the poles will be packed. The stacking station also decides which products are loaded into which fixtures on the pallets and therefore it is crucial for the cell controller to be informed of the pallet contents, so that the next stations will also know how to interact with the pallet.

### 3.3 Design specifications

The formulation of a strategy to find and/or rank the requirements for a specific application is not the focus of this research, but is considered by Hoffman and Basson (2013). However, in this research the following is assumed, taken from the CBI application:

- The system is customizable to the extent that it is able to handle changes within a predetermined part family without being shut down.
- The system is convertible and scalable, but the associated system changes may be implemented during a shut-down period.
- The control system is modular, holonic and distributed, in particular that the operational holon (discussed in section 2.4.4) associated with each hardware resource runs on its own software platform, and that product information is retained in product holons (and not in the operational holons) that reside on a cell controller platform or another controller higher in the holarchy.
- The system can handle disturbances when, for example, a subsystem fails and parts have to be rerouted. The system must also provide HMIs at various points and allow manual override capabilities.

- The system is diagnosable to the extent that each holon can at least assess and report on its own health to a cell controller.

### 3.3.1 CBI needs analysis

Since the Q-range currently has by far the highest volumes, the automated process should specifically cater for the Q-range initially but the system should be designed with the other types of products in mind, so that at a later stage the system can easily be reconfigured without major expenditure. Also during a possible future reconfiguration, the impact on the already ongoing production processes should be kept to a minimum.

The parts of the system to be automated in general, must

- Provide traceability of all products from right before shell placement to where the corresponding product info is printed on the shell. This should be accomplished by recording where each product has been and which processes it was subjected to throughout all processes.
- Enhance consistent quality by eliminating all human errors.
- Must initially be able to at least handle the Q-range, which has the highest production volumes. It would be advantageous if several other product families can also be handled at a later stage.
- The automated stations should have at least the same throughput rate as the manual stations currently in use, preferably faster than 1 pole per seconds.

The stacking station must be able to:

- Transfer poles from the source pallets on the first conveyer into the riveting fixtures on destination pallets on the second conveyer.
- Pick up poles in any order.
- Match correct poles, according to the orders that were placed.
- Record which poles have been matched and send this information back to the cell controller.
- Buffer poles when they cannot yet be stacked, to prevent congestion. This would become necessary whenever some of the poles, which form part of a stack, have been discarded by the electrical test station and replacements for the failed poles must first be made. Once replacements have arrived at the station, they can be used in conjunction with the buffered poles and a complete stack can be built.
- Report on system health periodically and/or upon inquiry.
- If the system should go offline, production should be able to continue manually.

Given:

- When system starts or restarts, the station would be completely empty, i.e. the station does not need to be able to record what is in the buffer when the system goes offline. Furthermore, the controller does not need to make provision for the operator to specify what is loaded into which position of the buffer, because at start-up, all positions would always be empty.



### 3.3.2 Functional requirements

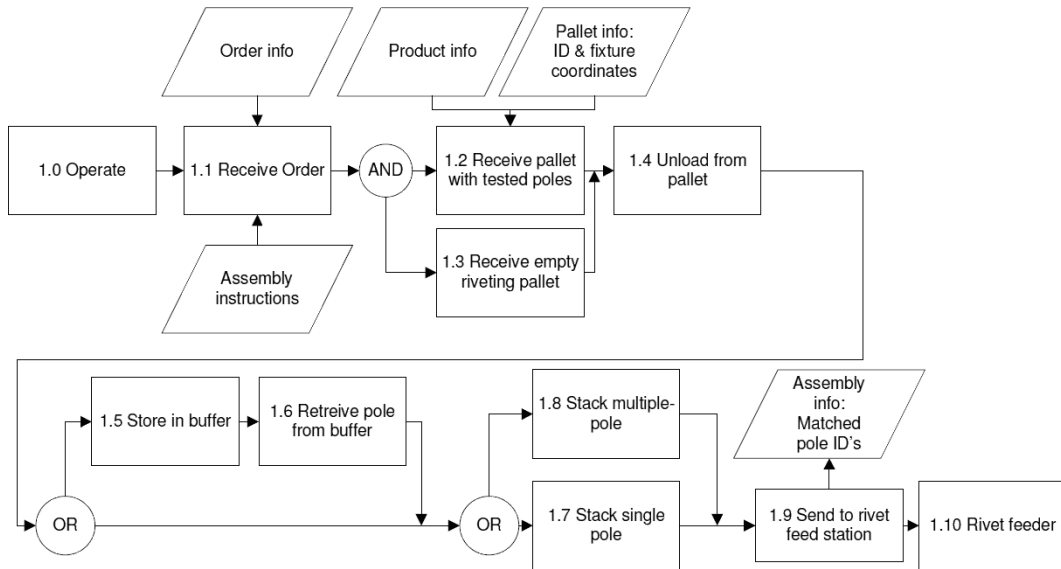


Figure 12 Functional analysis

Figure 12 shows the functional analysis of the stacking station. Below are the functions that the station must fulfil:

- **Receive** a pallet with an empty riveting fixture
- **Receive** a pallet with tested breakers
- **Decide** in which order breakers are to be unloaded from the pallet
- Either temporarily **store** the breakers in the buffer
- Or **stack** directly on the riveting fixture
- **Match** poles which belong together and send information about which poles have been matched back to the cell controller
- **Send** pallet away to the rivet feeding station

Receiving:

- Either: detect presence of pallet
- Or: receive command from cell controller that pallet has arrived
- Request information about the pallet from the cell controller
- Receive information about the pallet from the cell controller

Decide:

- Determine which poles are being waited for
- Determine which breakers need to be assembled with the highest priority
- Determine whether a complete breaker can be built with the available poles
- Decide whether breakers should be unloaded from the pallet or rather from the buffer

Store breakers in the buffer:

- Pick up poles from the pallet
- Relocate to an available position in the buffer
- Release the pole into the location on the buffer

Stack poles

- Pick up poles from either the pallet or the buffer
- Relocate to stacking fixture
- Release pole on top of stack into fixture

Match poles:

- Identify a complete set of poles which all have the same electrical parameters
- Register at the cell controller that those poles are being combined to form a stack

Send pallet away:

- Notify cell controller that riveting fixture is fully loaded and ready to be sent to the rivet feeding station

### 3.4 Design of the stacking station

#### 3.4.1 Station physical architecture

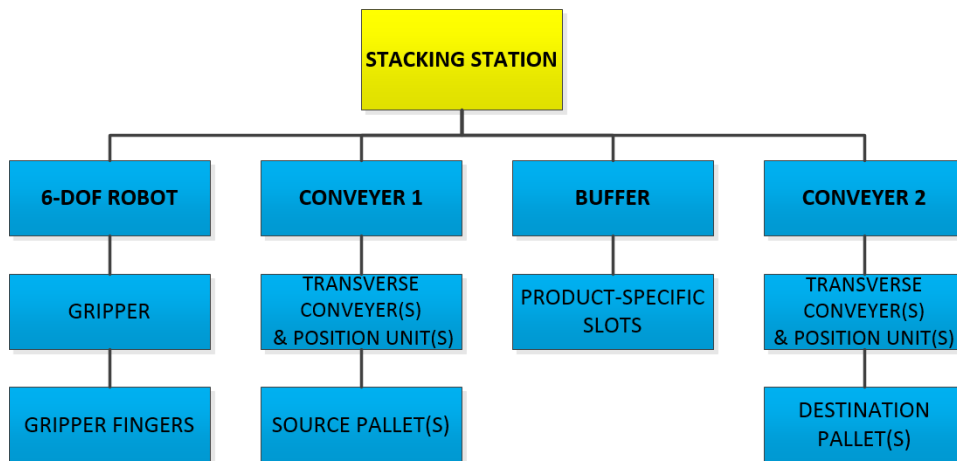


Figure 13 Stacking station physical architecture

As indicated in Figure 13 the stacking station consists of a 6-DOF Robot, a buffer, a conveyor on which poles are being transported to the stacking station and another conveyor on which the poles are being transported away from the station. The station layout is shown in Figure 14.

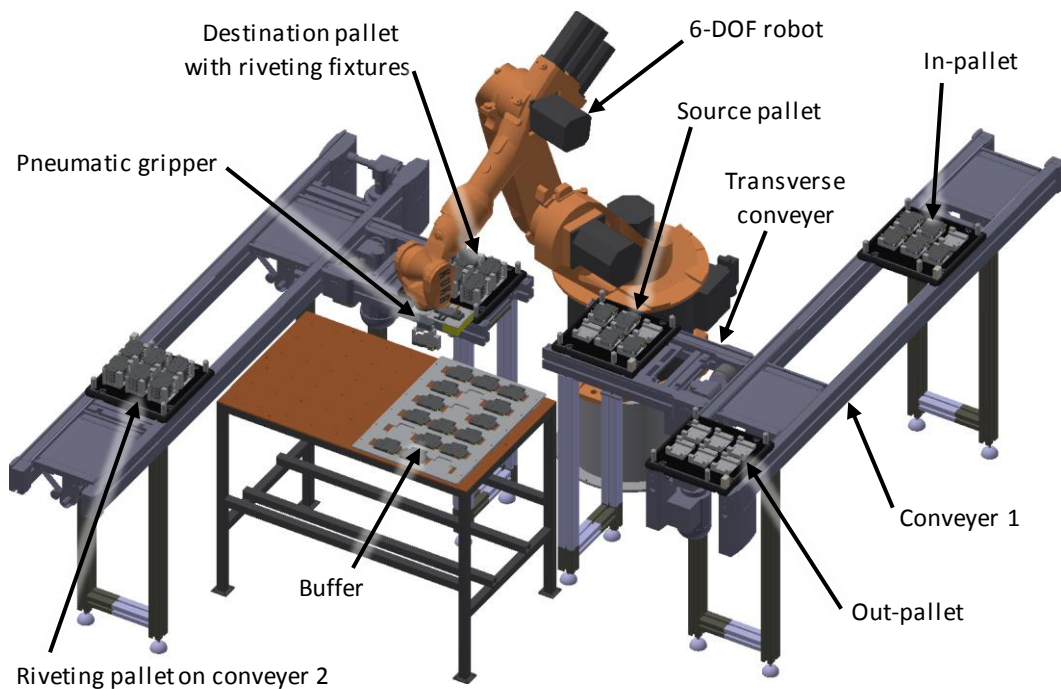


Figure 14 Stacking and buffering station layout

### 3.4.2 Mechanical design

In the sub-sections to follow, the detail design of the buffer, fixtures, and grippers is discussed, and the selection for pallets and the robot is motivated.

#### 3.4.2.1 Transportation system / pallets

Up to date CBI uses plain conveyor belts which run along most of the workstations and onto which poles are placed by hand at any orientation and picked up by hand at the next station. When only humans deal with poles, the orientation of the poles while being transported is trivial since their eye-hand coordination allows humans to quickly and accurately pick up any poles regardless of their position, orientation or product type. However, for automated systems where the poles need to be picked up by machines which are essentially blind, it is beneficial to have all arriving poles in a known and repeatable position and orientation. For this reason, the MADRG decided to choose a **Bosch Rexroth TS 2plus** pallet-based conveyor system with Bosch Rexroth 320mm x 320 mm off-the-shelf pallets and equip each station with a lifting station, which would align and lift all pallets and put them all in a repeatable position for pick-up and placement. Product-specific fixtures were designed to go onto those pallets, as described in the next section. This arrangement allows a wide variety of products to all make use of the same transportation system. Two options would be conceivable for handling product-changeovers: Either, the fixtures on the pallets could be exchanged, or easier would be to have enough pallets stored in a pallet magazine, each containing product-specific fixtures, and upon product change-over, the required pallets would be retrieved from the magazine, while the unused pallets would then be stored in the magazine until required for later usage. The latter option would require no manual intervention.

Other reasons for opting for pallets are that throughput rates for stations such as the printing and visual inspection station can be drastically improved when poles arrive in larger quantities. This is particularly important for the printing process, since industrial laser printers capable of printing on plastic are very expensive, would therefore be bought sparingly and as a result would form the bottleneck. Furthermore, poles are located at distinct positions on the pallet, and pallets can be tracked using identifiers such as RFID tags. This allows precise tracking of each individual pole, even if the poles themselves are not equipped with identifiers and even when pallets overtake one another.

The Bosch Rexroth TS 2plus is highly modular and supports track widths from 160mm to 1200 mm, an overall workpiece mass of up to 240 kg, conveying speeds of 6 m/min up to 18 m/min in increments of 3 m/min.

Bosch Rexroth supplies workpiece pallets in various sizes varying from 160 x 160 mm to 1040 x 800 mm, both square and rectangular-shaped.

Each station is then equipped with at least one lifting station, so that when a pallet arrives, it is stopped and lifted to a pre-defined height, and at the same time aligned so that all pallets being handled by the lifting station would always be in a repeatable position. To achieve the desired throughput rate at the electrical-test station, the robot there had to be able to pick up two poles side-by-side at the same time. From this requirement, the optimal number of fixtures per pallet was derived and found to be 6 fixtures per pallet.

To allow the pallets to be stored on top of one another inside a pallet magazine, as shown in Figure 1, they were kitted with pillars at all four corners which serve to support the pallets which are stacked on top of the current pallet while preventing the fixtures from being compressed.

The ability of the Bosch-Rexroth system to allow for various conveyer components to be added as modules, makes it well suitable for reconfigurations. The following components can be added at most locations along the conveyer system:

- Transverse conveyers.
- Parallel conveyer sections.
- Stopping stations, to prevent traffic jams at intersections, to regulate the exact cycle time for the pallets, and to maintain a safe following distance in-between two consecutive pallets.
- Lifting stations, which align the pallets to a repeatable position and orientation, and lift the pallet by a small amount, to a fixed height.

Figure 15 shows one of the pallets that were used, equipped with six fixtures, four pillars and an RFID module in the far corner of the pallet. The fixtures allow breakers to be picked up from underneath as well as from above, to suit the needs of the electrical test station and the stacking station, respectively.

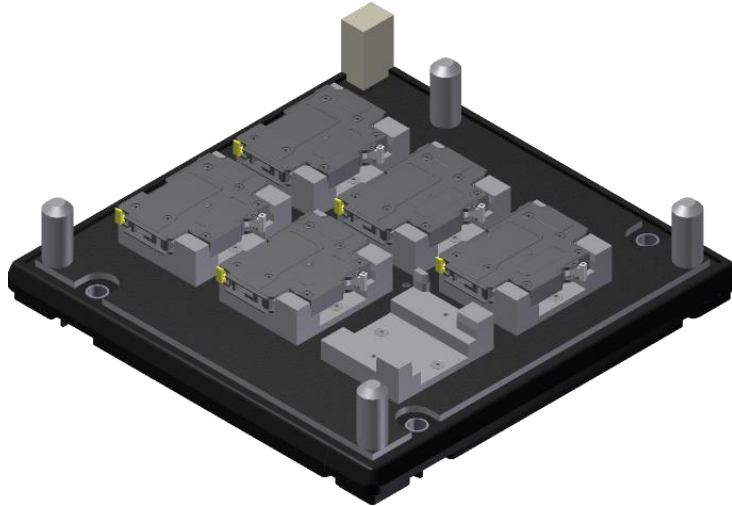


Figure 15 Bosch Rexroth 320x320 pallets can carry six fixtures each

Using Pallets in conjunction with off-the-shelf lifting stations eliminates the need at each station to align the breakers or the need for installing vision-aided pickup systems. The drawback is that the workers would need to place the breakers into fixtures at the correct orientation, whereas with the current conveyer used by CBI, they can place the products in any orientation on the conveyer. Another drawback is that the fixtures on the pallets are product specific. Nevertheless, when reconfiguring between products of different shapes, the same conveyer and pallets can be used, only the fixtures need to be interchanged. Alternatively, to avoid the need for humans to replace the fixtures, several pallets containing all kinds of fixtures can be stocked in the pallet magazine and only be retrieved once they are needed.

Although the pallet fixtures themselves are product specific, whereas a conveyer belt would not have been product specific, each station would require a sophisticated alignment apparatus, whereas the Bosch Rexroth lifting units each have one actuator only and are modular.

#### 3.4.2.1.1 Unloading station

The transverse conveyers are equipped with proximity sensors which detect the arrival of a pallet. At the end of each transverse conveyer is a lifting station which lifts and aligns the pallet to a repeatable position to ensure that the robot can accurately pick up the poles.

#### 3.4.2.2 Fixtures

Fixtures securely hold products in a repeatable position. They contain features which are in contact with the perimeter of the product to prevent it from moving around. Where those features must be, depends on the product that must be kept in place and therefore fixtures are product-specific. Whenever a new type of circuit breaker has to be produced of which the outer dimensions are different to previous models or of which the rivet positions are different, then a new set of fixtures will have to be manufactured.

The fixtures on the first conveyer's pallets are capable of holding one single pole each and are used for the following processes in the order given below:

- Manual placement of the base containing all electrical components.
- Visual inspections.
- Manual placement of the shell.
- Electrical test.
- Transport to stacking station.

To suit the needs of the various stations, the single-pole-fixtures have been designed that workers can place the base with ease and to allow for clear photos to be taken by the overhead camera at the visual inspection station. Also, they allow breakers to be picked up from the front (as required by the electrical test station) as well as from above (as required by the stacking and buffering station). To allow the poles to be picked up from the front, some material was machined away to allow a gripper finger to get underneath the poles. The fixtures were designed to be narrower than the width of the poles they have to carry to allow picking up from above. They are sufficiently narrow, so that if the robot would attempt to pick up a pole which was in fact not on the fixture, then the grippers would not grip the fixture.

On the second conveyer, fixtures had to take up stacks of poles. They had to allow the automatic rivet feeder to feed rivets, and the automatic riveter to rivet the stack. Whether riveting should be done from underneath or from above was not further investigated as this was considered to be out of the scope of this research project. For either scenarios, the poles would need to be stacked from above into the riveting fixture.

### 3.4.2.3 Robot

The main purpose of the robot is to assemble multi-pole breakers by stacking individual poles on top of one another. This is accomplished by picking up poles from the source pallet and transferring them to fixtures on the destination pallet, where they are stacked on a pile of which the height keeps varying. Occasionally, poles must also be transferred from the source pallet to the buffer and from the buffer to the destination pallet. The robot must therefore interact with pallets on the source conveyer, with pallets on the destination conveyer, and with the buffer.

Since the poles are lying on their side upon arrival, and must also lie on their side after stacking, the pick-and-place robot requires only four degrees of freedom. A SCARA robot would have been ideal for this type of application because of its high speeds. However, the only feasible robot available in the research laboratory was a KUKA KR 16-2 robot, which has six degrees of freedom. It could therefore be used for the proof-of-concept setup on which the experiments were carried out.

4-DOF and 6-DOF robots of comparable size do not differ drastically in price but 4-DOF robots are generally less expensive. If provision is to be made for future products that might require the robot to rotate about more than one axis, it would be advisable to rather opt for a more flexible 6-DOF robot due to its higher versatility.

The controller which comes with the robot, deals with all the inverse kinematics calculations. All that needs to be done is to establish a way of communicating and then sending appropriate commands along with coordinates and parameters. Reconfiguring a 6-DOF robot is therefore no more difficult than reconfiguring a 4-DOF robot.

#### 3.4.2.4 Grippers

The way gripper fingers are to be designed depends on their application as well as the shape and size of the poles being handled. Gripper fingers are therefore fairly product specific but the same gripper fingers could still be used for several different products as long as those products are all similar in size and shape. To allow poles being stacked on top of one another, poles have to be lowered into the riveting fixture from above and hence must first be picked up from above. The gripper fingers therefore have to grip the poles at the sides, as shown in Figure 16 and the line of action of the clamping force should pass close through the pole's centre of mass.

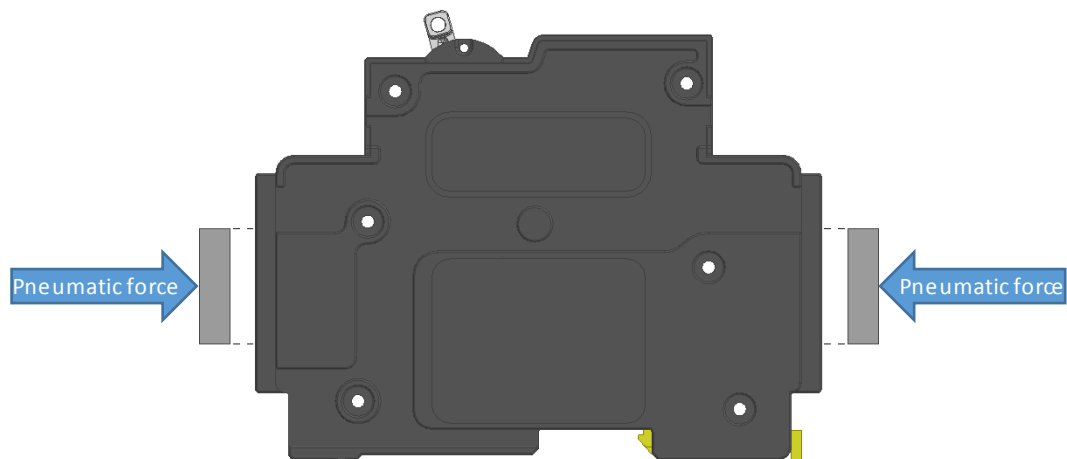


Figure 16 To make stacking possible, poles have to be gripped at their short sides

Requirements for the gripper and gripper fingers are:

- Unriveted single-poles must be picked up, transferred and placed without the casing opening up or the pole being dropped.
- The casing must not be damaged.
- The common trip of a pole already on the stack should be properly aligned with the common trip connector of the pole being stacked on top.
- Fingers should not interfere with the poles already on the stack, when adding another pole to the stack.
- In the event of power loss, poles should remain gripped.
- It must be sensed whether no pole was gripped, or whether it was properly gripped or misgripped.
- Gripper fingers should be designed to cater for quick product changeovers with no or minimal ramp-up time.
- For short cycle times, gripper jaws should open up and close immediately after the corresponding command was given.

- They should be re-usable for various types of products, if possible.

Figure 17 shows an isometric view of how the gripper fingers can be used for stacking. Next, the requirements are addressed in the order mentioned above.

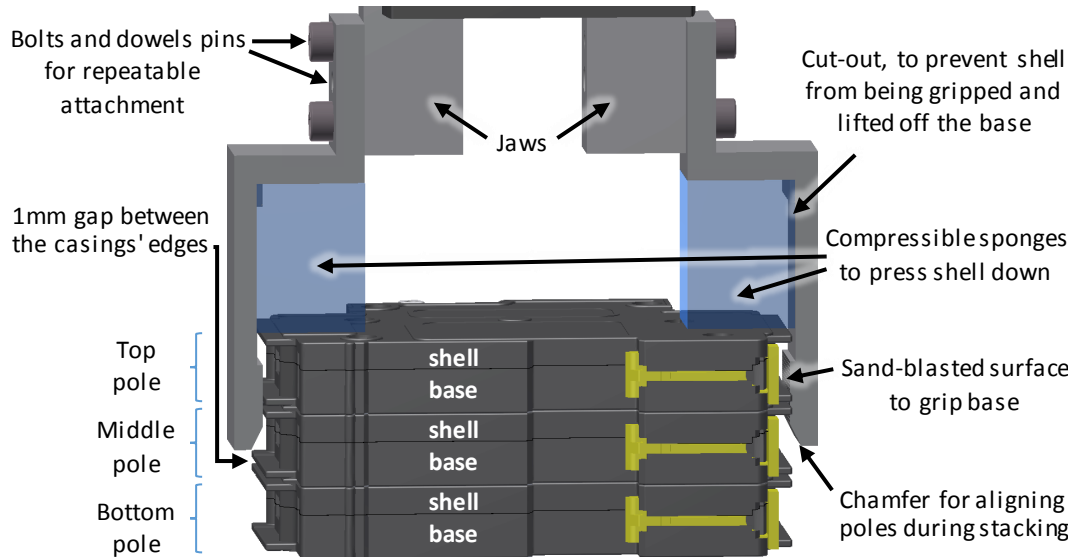


Figure 17 Gripper fingers used for stacking poles

#### 3.4.2.4.1 Picking up & placing without casing opening up

Since the poles are in an unriveted state, the stacking station has to handle them carefully enough to prevent the casing from opening up. Viewed from above, bases and their corresponding shells have identical geometries. If the base and shell would be gripped simultaneously, then it could happen that the shell is being lifted off the base, leaving the base behind in the fixture. To prevent this, only the base should therefore be gripped and contact with the shell should be avoided. This was accomplished by cutting away material from the gripper fingers at the location where they would've made contact with the shells, as can be seen in Figure 17.

When a pole is being lowered into the riveting fixture, which has very tight tolerances, then both the base and shell will experience an upward friction force. Since the gripper fingers only hold onto the base, the shell could be separated from the base when the friction force acting on the shell becomes too large. To prevent the case from ever opening up, two spring-like sponges (blue parts in Figure 17) were added to the grippers to prevent the shell from sliding open while the poles are placed into the riveting fixture. While picking up poles out of the source pallet, the sponge would be slightly compressed to exert a downward force on the pole which is large enough to counteract the upward friction force of the tight riveting fixture, but small enough to not push the pole out of the gripper fingers. Sufficient space was left open between the contact surfaces and top of the fingers that an entire stack of poles could also have been gripped by the same grippers that would usually only transfer one pole at a time. Although this feature was never utilized for any experiments, for it to work the sponge would need to



be made smaller and the stack being transferred would need to be gripped at the base of the bottom-most pole.

As indicated in Figure 17, the edges of the casings of stacked poles are approximately 1mm apart leaving very little room for the grippers to grip underneath the casing. When the poles are picked up at their sides, the available surface of contact is only a very small area and enough friction needs to be developed for a firm grip. Three options were considered:

The first option was to use a thin, small rubber pad to provide the friction. When adding a pole onto a stack, this rubber pad should not interfere with the pole underneath, so the pole being transferred would need to be held close to the bottom edge of the rubber pad. When the pressure is increased too much, the edge of the rubber pad would give in more than its centre, thereby causing the resultant force exerted on the pole to have a downward component, causing the pole to slip out of the fingers. Furthermore, extensive usage would eventually wear out the rubber pads and result in non-repeatability. Because of possible unreliability caused by those two aspects, this idea was discarded.

The second option was to use small  $\text{Ø}1\text{mm}$  pins to fit underneath the poles, similar to fingernails, thin enough to fit in between the casings of two stacked poles. These pins would provide a shoulder for the poles to rest on, preventing them from falling out of the fingers. Because of their small size they could easily be damaged and they would need to be very accurately positioned relative to the pole to be picked up which would often be very difficult to accomplish and there would be very little compliance. Also, this concept does not provide a means for keeping the shell on the base while the pole is pushed onto the stack, which is why this idea was also discarded.

The third and chosen option was to give the aluminium a rough sandblasted surface finish, rough enough to increase the friction coefficient, but smooth enough as to not damage the casings. This concept has proved successful, as poles have been transferred to the riveting fixture numerous times without any poles ever being dropped or casings opening up.

#### 3.4.2.4.2 Damage to casing

To prevent damage to the casing, the gripper fingers were designed sufficiently wide that the contact surface between the grippers and the casing was large enough to keep the resulting pressure sufficiently low as to not damage the casing.

#### 3.4.2.4.3 Aligning poles of a stack

To line up the common trip with the common trip connector during the stacking process, the grippers have to align the edges of the pole being stacked with the edges of the poles which are already on the stack. To accomplish this task, the end of the gripper fingers were chamfered, so that upon lowering the top-most pole of the stack, its sides would line up with the sides of the pole onto which it is being stacked.

#### 3.4.2.4.4 Loss of power

Festo DHPS-20A pneumatic parallel grippers were used, which are double-acting and they come with the option of having backup force retention for either the normally open (NO) or normally closed (NC) state of the gripper. To prevent the pole from being dropped in the event of power loss, it was opted to have force retention in the NC state. Even in the absence of air pressure or loss of input signal, an NC gripper keeps holding onto the part, because it contains an internal spring which exerts a closing force.

#### 3.4.2.4.5 Quick product changeovers

The left and right gripper were designed to be identical, so that any of the two fingers could be attached to any jaw and that there would not be an incorrect way to assemble them. For cases where gripper fingers are not symmetrical, it is advisable to label them appropriately to aid with assembly. To ensure that the fingers will always be aligned in a repeatable way, they were designed with dowel pins to align them w.r.t. the grippers.

The grippers were attached to a mounting plate w.r.t. which it was aligned using dowel pins. Various members of the research group have been using the KUKA robot for various experiments. To allow for quick and easy changeover from one configuration to another, another member of the research group has designed a mounting plate which is to be attached to the flange at the end effector of the robot and onto which various types of tools can be attached in a repeatable position, when used along with dowel pins. The flange was designed so that one would not need to screw and unscrew onto the robot directly with every changeover. The mounting plate for the gripper was designed such that it would be easy to rotate the grippers by 90° if need be.

Dowel pins ensure all parts are correctly aligned to one another, thereby eliminating the need for re-calibration after product changeovers, and drastically reducing ramp-up times. Only one or two test runs would be required to ensure everything was assembled correctly.

#### 3.4.2.4.6 Near-real-time control

The pneumatic air control valve was controlled by a **National Instruments** Data acquisition device containing 8 digital inputs and 8 digital outputs. For powering the valve, an external 24VDC power supply was necessary, since the Festo actuators work with 24V as well as compressed air supply.

The DAQ is connected to the PC via USB. National Instruments drivers had to be imported into the C# program so that the DAQ could be used.

#### 3.4.2.4.7 Feedback

The Festo DHPS-20A gripper features a slot inside which a proximity sensor can be anchored which can be moved up or down inside the slot. An SMT-10G/10G proximity sensor was used, which is a digital sensor and can only produce a "false" or a "true" signal. Using this signal in conjunction with the command sent to the gripper control valve, four possible states can be determined:

1. Whether a pole has been properly gripped – A command would be given to close the gripper, and the sensor returns "true" to indicate that the jaws are in the position for which the sensor has been calibrated, i.e. that the grippers are partially closed because the pole being gripped prevents them from closing completely.
2. Whether a pole has been misgripped or is not present where expected – A command would be given to close the gripper but the sensor returns "false" because the jaws would have moved past the calibrated position and closed completely.
3. Whether air pressure has been lost – A command is given to open the gripper, but due to low air pressure, the jaws remain closed at the calibrated position and the sensor returns "true".
4. Whether the gripper is open – A command would be given to open the gripper, which moves the jaws away from the calibrated position, causing the sensor to return "false".

The position of the jaws for which the sensor returns "true" depends on the pole width. The sensor must therefore be re-adjusted whenever breakers of different pole widths have to be accommodated. Alternatively, two or more sensors could have been inserted into the slots to determine the thicknesses of various products without the need for manual adjustment when switching between product types.

#### 3.4.2.5 Buffer

The buffer is used to temporarily store poles that cannot yet form part of an assembly until those poles can be used at a later stage. Poles being stored in the buffer are in an unriveted state.

The following points have been considered for the design of the buffer:

- Since a SCARA robot would typically be used to interact with the buffer, only four degrees of freedom should be required to retrieve poles or place poles into the buffer.
- It should be possible to pick up any pole at any time in any order, i.e. poles should not block access to other poles, e.g. should not be stacked on top of one another, unless they are identical and can thus be used interchangeably.
- Since poles have common trip connectors protruding upward (as was shown in Figure 10), not even poles of the same kind can be stacked on top of one another.
- Poles should be placed repeatable and should not move in case the buffer gets bumped into.
- The buffer itself should ideally not consist of any actuators or controllers to make retrieval of poles possible.
- People should be able to easily insert or remove poles by hand in case of manual override mode or when the system has to be cleared after a restart.

It would have been possible to utilize the existing hardware present in the lab to form a buffer: Unused poles could have been stored on empty pallets which would then be sent

to the pallet magazine until poles on those pallets would be required at a later stage. However, the logistical, practical and economic aspects of this approach were far from desirable and other buffer concepts were therefore considered.

The concept that was chosen is shown in Figure 18. It was designed such that all poles lie on the same plane, to simplify calibration, which in turn simplifies reconfigurations. When all fixture positions are lying on the same plane, it further simplifies the task of choosing a reference point and expressing the coordinates of the individual fixtures relative to that reference point.

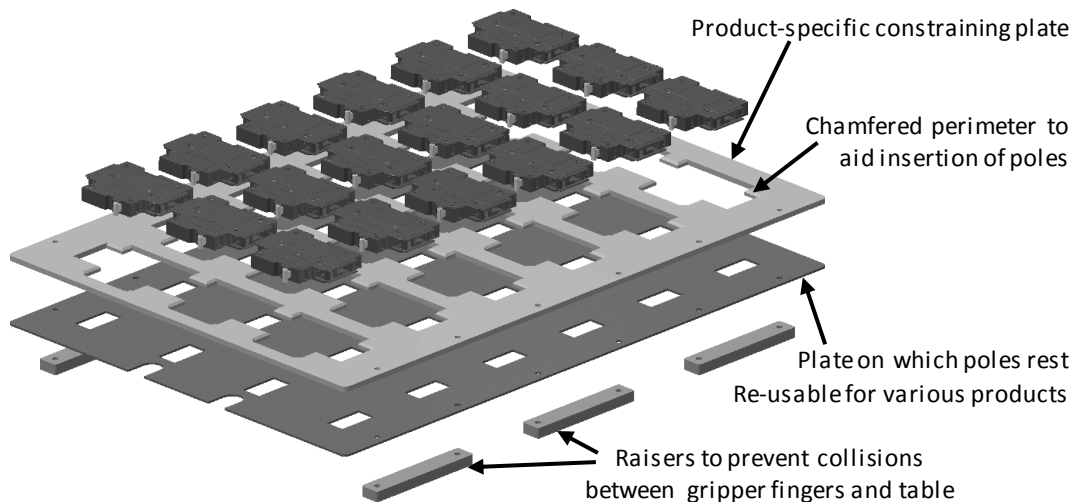


Figure 18 Exploded view of buffer

As shown in the figure above, the buffer provides space to store up to 18 single-poles. For stacking, the robot needs to grip the poles from above which is why the buffer has been designed such that poles can be removed and inserted from above. The top-most layer contains several holes which have the shape of the outline of the circuit breakers, so that breakers are restricted from moving horizontally. These holes have chamfered edges to aid the robot placing the poles. The top-most plate is product-specific as the holes are custom-made to match the outline of the poles to be stored. The rest of the buffer is product-independent. Therefore, when the shape of a circuit breaker model is modified, then only the topmost plate of the buffer needs to change to accommodate poles with a different shape. The rest of the buffer hardware does not need to be changed. When loaded into the buffer, poles will be resting on the middle plate which has holes big enough for the gripper fingers to fit through in a fully opened and fully closed state. The entire buffer rests on raisers so that the tips of the gripper fingers would not collide with the table while picking up a pole.

The advantages of the chosen concept are:

- Poles can be placed in repeatable positions.
- Poles of various thicknesses can be stored.
- Poles will not move in case that the buffer is pushed.

- Storing all poles horizontally, which is the same orientation that they are transported on the pallets, allows them to be handled by 4-DOF robots.
- The buffer was inexpensive and easy to manufacture.
- No complications occurred while installing the buffer and after removal, the buffer could be repositioned at its previous location without requiring re-calibration.
- Choosing a reference point on the buffer for calibration purposes, and calibrating the buffer's workspace coordinates was done without problems.

The disadvantages of the chosen concept are:

- The contours of the top-most layer are dependent on the outer geometry of the poles that have to be stored. This means that only one type of circuit breaker can be stored in the buffer.
- A relatively large footprint was claimed by the buffer when installed horizontally. A vertical design would occupy a fraction of the floor space.

### **3.5 Conclusion**

A background on the case study was given in this chapter, along with the design specifications for the stacking and buffering station. The mechanical design aspects of the transportation system, fixtures, grippers and buffer was then described. Chapter 4 provides the details of how control software for the station controller was chosen.

## 4 Control Software Selection

Control software alternatives for a reconfigurable controller are investigated in this chapter. They are evaluated in terms of their capability to interface with the hardware chosen in the previous chapter, their suitability for holonic control architectures, their prevalence in industry and in terms of the requirements discussed in the section below. Only a high-level comparison is made to identify some of the weaknesses and strengths of C# but a detail comparison is considered out of scope. Furthermore, OOP is evaluated as an alternative for agents at the end of this chapter.

### 4.1 Software requirements

Since the controller in this research is not in control of an entire factory but has to control only one station, it would not be of much value to have all the individual holons run in separate threads. What was seen as a much higher priority on a station-level was to achieve optimal throughput rates for which real time or near-real time control of hardware plays an important role, while also allowing global optimization. Multi-threading capabilities were thus seen as a requirement to allow for near-real time execution of hardware being controlled by the main thread, while also having the following tasks run in parallel:

- Communication over TCP/IP or any other asynchronous communication channels.
- CPU-intensive forecasting and/or global optimization algorithms.
- Control of a human-machine interface (i.e. graphical user interface and mouse/keyboard event listeners).
- Diagnostic tools which might require regular access to the hard drive when log files are to be written to gather fault data for diagnostic statistics.

The station controller must be able to communicate with the cell controller and conveyer controller over Ethernet, to receive orders, product information and logistical information and to allow for diagnostics to happen remotely over the network. Support for TCP/IP communication is thus required.

To improve modularity among the stations and the cell controller, communication should happen by interchanging XML strings due to the following advantages of XML:

- XML is platform independent.
- XML can be designed to be extensible (Obasjano, 2004) which allows for changes to the XML structure to be made in one application, and an older or newer application will still be able to function properly and read the XML message as easily as before the changes have been made. The extensibility of XML allows reconfiguration to happen gradually as needed, so that only critical parts of the system need to be changed, while not having to make changes to the entire system at once.
- XML strings are human-readable, which can simplify diagnostics in some occasions.

For the case study in particular it was further required that it should be easy to interface with the KUKA robot via RS232, and Festo gripper sensor and control valve via a USB-DAQ.

The following aspects were other important factors in the search for suitable control software:

- To allow for a fully reconfigurable controller, a holonic control architecture should be implementable, i.e. the controller should support OOP concepts so that holons or holon-like objects can easily be created.
- To promote industry acceptance, the language should be widely known, proven successful over many years and easy to learn.
- For shorter development time reusable libraries should be available so that relatively basic functionalities do not need to be developed from the ground up.
- Advanced diagnostic tools should exist for fault finding during development and also during operation.
- Development of an HMI/GUI should not be a major effort.
- Near-real time control of hardware for optimal throughput rates.
- Multi-threading capabilities will allow asynchronous tasks such as network communication and CPU-intensive tasks such as optimization algorithms to run while still having near-real time control of time-critical hardware tasks.
- Support for TCP communication and XML handling.
- Dynamic memory allocation to cater for changes in product information or order information at runtime without the need for recompiling the code and restarting the controller.
- Integrable, thus ability to run on various platforms and support for various types of interfaces.

## 4.2 Software comparison

Although numerous programming languages could be used for the implementation of the station controller, only some of the most widely used programming languages were investigated, since wider usage implies higher likelihood of industry acceptance. The languages considered here are Java, C, C#, and C++.

### 4.2.1 Java

Java is the most widely used programming language (Cass, 2015). It is a high-level programming language with support for OOP concepts, multi-threading and dynamic memory allocation. Developing a GUI does not require major effort. However, it is cumbersome to interface with hardware and to get serial communication working. FBD and ABC that were mentioned in sections 2.5 and 2.6 are both Java-based applications which makes hardware interfacing a cumbersome task for those approaches too.

A great advantage of Java is that it is platform independent, i.e. the programmer does not require any knowledge of the operating system or the machine where the code will run on, but can be assured that the code will run the same way on different platforms without having to recompile the code. This enhances the modularity of code written in

Java. The Java Native Interface (JNI) however, which must be used to interface with hardware, is not platform independent.

To ensure the controller can run for months without interruptions or failure, memory leakage must be prevented. Java comes with automatic garbage collection (AGC) which simplifies software development in that developers do not need to keep track of the memory allocated to pointers themselves and removes the chances of human errors in this respect. A disadvantage of AGC is that it is handled by the operating system which makes it impossible for the programmer to predict when it will occur. Whenever AGC kicks in, other processes could experience minor delays in their execution, which is acceptable when no precise timing between two or more processes is required, or when timing-critical processes are executed by a hard-real-time controller such as a PLC.

#### **4.2.2 C**

C supports dynamic memory allocation but has no built-in AGC. Correctly allocating and disposing memory is thus the responsibility of the programmer. Since in C error checking (such as array bounds checks) is not performed at runtime and because the programmer is in control of memory disposal, C can execute slightly faster and with more precise timing than common OOP languages with AGC. Essential parts of the code can easily be written in assembly which makes C suitable for near-real time control.

The lack of AGC and automatic error checking leaves those tasks to the programmer which, if neglected, could cause unreliable behaviour and produce sudden failures after weeks of normal operation, when due to memory leakage suddenly no new memory can be allocated. Especially after reconfigurations, when another programmer had to make changes to the code, chances are that all allocated memory is no longer disposed of correctly. Neglecting to properly dispose of memory does not necessarily produce any immediate runtime errors because controllers today typically have an abundance of available memory. Memory leakage happening at a slow rate is therefore not likely to be detected during ramp-up tests but will only cause failure at a later stage. In the author's view, the lack of AGC makes C less reliable than languages with AGC.

C is a much lower level language than common OOP languages and has very poor support for OOP concepts. Hence, implementing a holonic architecture in C would probably take significantly longer. Although C is not a multi-threading language per se, it does support multi-threading using libraries. There is no direct support for implementing a GUI, but GUIs can be created using wrappers.

#### **4.2.3 C#**

C# is almost equivalent to Java from an application developer's perspective (Radeck, 2004) and syntactically is very close to Java, and thus almost equally easy to learn. Programmers knowing Java should easily be able to learn C# (and vice versa) since they are conceptually very similar. Like Java, C# also has support for AGC, multi-threading, OOP concepts and easy GUI development.

Hardware interfacing, however, is easier in C# than in Java. Drivers for most of the hardware are usually available for C, C++ and C#.



C# is a strongly-typed (type-safe) language, i.e. errors relating to conflicting types are detected at compilation time and code cannot be compiled until these errors are resolved. Having such errors already detected during the development phase effectively means that some built-in diagnosability is automatically provided for during the development phase resulting in more reliable code.

#### 4.2.4 C++/CLI

C++/CLI (C++ over Common Language Interface), like Java and C#, supports all OOP concepts, multi-threading, AGC, and GUI development. It targets the .Net framework and can easily be integrated with other .NET code, such as code written in C#.

C++ is one of the languages that allows multiple inheritance, i.e. that a sub-class can inherit from more than one base class. Java and C# don't allow this, but they do allow inheriting from multiple interfaces, as this would cause no ambiguity for the compiler which one of the inherited methods to implement, since interfaces have no implementation.

Managed languages in the .NET Framework do not support multiple inheritance, i.e. only one base class can be specified for a derived class.

Garbage collection works for managed objects for which destructors are not required but unmanaged resources are not cleared up by garbage collection and destructors must therefore be implemented for them.

### 4.3 Chosen controller and chosen software

Compared to ABC and FBD, the languages discussed in the previous section are easier to learn and much more widely used. ABC is a very high level language and provides many useful functionalities for holonic control which, when built from scratch, would take a long time to implement. However, for the CBI case study not all those functionalities are required and hence a more general language should be more suitable.

TCP communication and XML parsing is supported in all of the languages considered above and could therefore not be seen as a determining factor.

High-level languages are easier to use to program holons, because they allow programming to happen on a level which is close to human thinking. Since C is on a much lower level than other OOP languages and because of the lower reliability due to the lack of AGC, C was deemed less suitable. Java was decided against because for a station controller, lots of hardware interfacing would be required which would be a cumbersome task to do in Java.

Although no significant advantages could be pointed out that C# has over C++ it was nevertheless decided to use C# for this case study since it fulfilled to all the requirements set out in section 4.1.

The integrated development environment (IDE) used was Microsoft Visual Studio 2012 which has advanced debugging tools. A nifty feature is that, when an exception was

thrown, hovering the mouse over an object variable will display its value along with the values of all the member variables. This can very quickly shine light onto the possible cause of an error and allows for quick diagnosing.

#### 4.4 Evaluation of OOP as an alternative to agents

The comparisons in sections 2.6 between agents and objects are not specific to manufacturing, nor to control. One should therefore question to what extent the above conclusions are applicable, particularly when considering a relatively simple application such as the control of a manufacturing cell or a subsystem contained by it, as in the intended application of RMS for CBI. This section therefore reconsiders the comparisons and also a number of other issues, with the specific application in mind. Figure 19 summarizes the key differences.

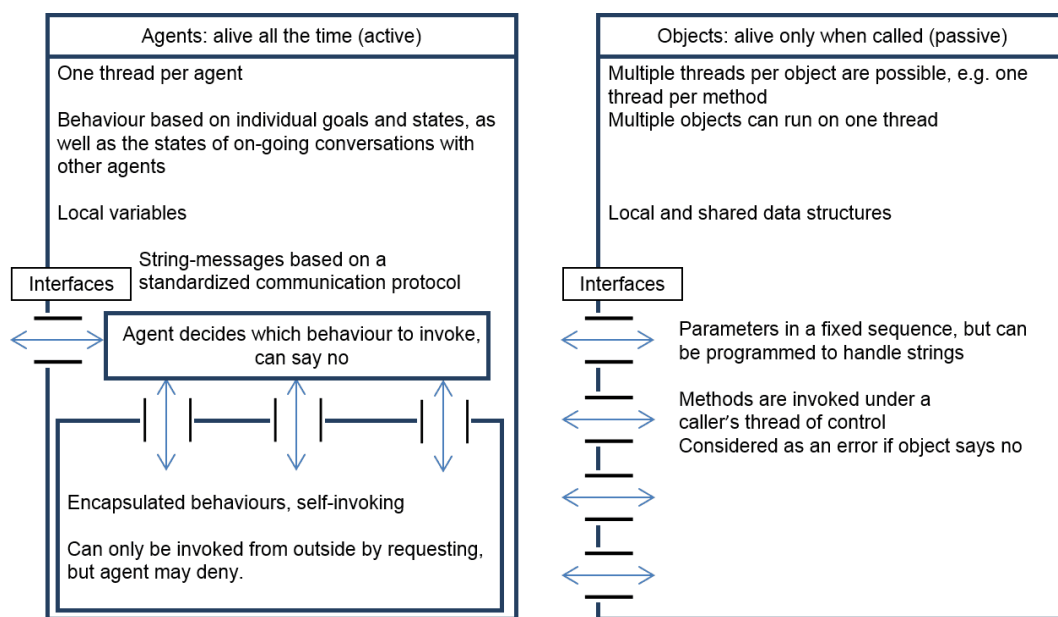


Figure 19 Key properties of agents and OOP

##### 4.4.1 Encapsulation of behaviour

Consider the external vs internal "unit invocation" difference shown in Figure 7 and the related issue of encapsulation of behaviour, in the context of a holonic control architecture: for a holon there is little difference in the logic, and its implementation in coding, between an agent-based and an OOP approach. An object can just as easily be programmed to initiate its own methods as an agent can initiate its own behaviours. Also, the programmer can decide which of an object's methods are publicly accessible and can therefore expose no more than a communication interface, thereby mimicking this property of an agent. In agent platforms such as JADE, the ideal is that each agent runs in its own thread, which can be limiting for manufacturing control scenarios. However, objects running different methods in different threads (e.g. to handle communication or diagnostics in parallel with other activities) is commonplace.

Regarding the, in theory, different active vs reactive natures of agents and objects, again in a holonic control architecture for a manufacturing cell, there is little difference between the logic and programming that will be required for agents and objects. This is even more so since some OOP implementations now have event-listener frameworks giving objects some of the dynamic capabilities of agents.

#### **4.4.2 Dynamics, complexity, autonomy and hierarchy**

As mentioned in the Introduction, RMSs are aimed at dynamic situations. As can be seen from Section 2.6, agents are well suited to adapt themselves to dynamic situations. Objects do not have all those capabilities built into them. For an OOP based system to be as autonomously reconfigurable as agents, designers would need to implement many of the standard ABC features in OOP, and then agents might just as well be used in the first place.

The advantages that agents offer in terms of emergent behaviour and reasoning are significant for complex systems and highly dynamic situations. Unstable and unpredictable environments benefit from decentralization and self-organization (Odell, 2002). In manufacturing systems, complexity arises from the large number of interacting subsystems, for instance the transportation system responsible for the material flow in a factory containing many cells. Optimizing such a system would be challenging and time consuming, but systems consisting of autonomous, proactive agents have emergent behaviours which allow for optimization to be done automatically if the correct rules are set for the agents.

On the other hand, for scenarios where the systems are relatively simple and/or reconfigurations are infrequent, the autonomous reconfiguration capabilities are of less value, and it might be more cost effective for humans to provide inputs (Hoffmann & Basson, 2013). The systems that are being considered for CBI are cells, containing subsystems such as 6-DOF-robots and automated riveters, for which material flow and resource management is much simpler than for an entire factory. The control system of a manufacturing cell with limited (if any) redundancy, is a relatively simple system. Handling of, for example, disturbances (such as subsystem break-downs or unanticipated changes to production schedules) and throughput optimization are therefore simple enough to be handled effectively by a hierarchical control approach.

The Contract Net Protocol is well suited for a holonic system, but is not needed for a hierarchical system, since decision-making is done on a higher level, and decisions are enforced on the holons situated on a lower level in the hierarchy. The directory facilitator of agent platforms is very valuable when handling autonomous reconfiguration of systems. However, there is little need for such autonomous reconfiguration in many industry applications. Particularly if reconfigurations occur relatively infrequently or have a fairly predictable nature, it is feasible to provide HMLs that can be used to provide not only the functions that a directory facilitator would, but also diagnostic and manual override capabilities.

### 4.4.3 Modularity and integrability

In terms of modularity and integrability, which are key aspects of RMSs as described in Section 2.2, agents offer little advantages over objects. Both agents and objects that represent holons will have to provide a communication interface to other holons. An agent platform will provide some infrastructure to handle these messages, while an OOP-based approach will have to build it up from a lower level. In the case of agents, at least an application specific ontology, and possibly also an inter-agent communication language, must be designed. The equivalent effort will be required for objects that are running in different platforms, for example using TCP/IP sockets and XML encoding. Further, for objects in the same executable code, a library of object types based on one or more abstract classes, as used in e.g. C++ and C#, can provide a means of specifying a standardized interface, thereby simplifying integration and customization.

Another practical consideration is that many popular agent platforms are programmed in Java, which is much more cumbersome to interface with hardware than, for example, common OOP platforms like C#. In the CBI application, which involves the control of the subsystems of reconfigurable manufacturing cells, there are a large number of interfaces between controllers and hardware. Using Java-based tools in such a context is therefore less integrable than C#.

### 4.4.4 Hard and soft real time requirements

One of the significant limitations of agents, in a machine control situation, can be their autonomy: each agent has autonomous control over its own behaviour. In many machine control situations, specific sequencing or timing of actions is required for safe and efficient operations. Close timing or rapid sequencing that are guaranteed and safe, is therefore difficult to achieve when multiple agents are involved. Such timing is easier to achieve with objects running in the same thread, but unless a real time operating system is used, there is still a measure of uncertainty in the timing. One can therefore conclude that agents are not suited to any form of real time control, while objects are suited to so-called "soft real time", where latencies of approximately 50ms or more are acceptable.

Another limitation of the agent platforms considered by the author is that no provision is made for allocating agents different levels of priority in their allocation of CPU time, while it is easier to allocate priorities to threads using standard OOP implementations. A manufacturing scheduling optimization algorithm can be therefore run in one thread at a lower priority on the same CPU than another thread running timing-sensitive machine control routines. This approach may not fully obviate the need to separate the "intelligent" part of the system from the real-time part of the system (Brennan, 2007) and to use a layered architecture consisting of low level controllers (LLCs) and high level controllers (HLCs). Traditionally LLCs would be written in software which allows for exactly predictable timing, whereas HLCs would be written in software more suited to implementing complex algorithms and that are more amenable to reconfiguration. LLCs are usually more difficult to reconfigure, particularly when more complex algorithms have been implemented. Therefore using OOP instead of ABC for the HLC will allow more functionality to be moved from the LLC to the HLC, thereby enhancing reconfigurability.

## 4.5 Conclusions drawn from literature

ABC and FBD both have characteristics which make them suitable for controllers of RMSs and both have successfully been implemented in research environments. However, neither ABC nor FBD are widely accepted by the industry. ABC performs well in terms of modularity, integrability and diagnosability, but performs poorly in terms of real-time execution, while FBD performs well in terms of modularity, integrability and real time execution, but not in terms of diagnosability and does not support dynamic memory allocation.

The Agent Based Control (ABC) approach is the de facto standard for controllers for Reconfigurable Manufacturing Systems. However, due to industry's reluctance to adopt ABC, an object-oriented programming (OOP) approach is considered in this research as an alternative. OOP is more widely used and has many capabilities that are valuable when implementing an RMS.

Sections 2.6 and 4.4 have shown that ABC's advantages can be decisive in complex, highly dynamic systems requiring autonomous reconfiguration. However, in simpler systems and systems where timing and sequencing is important, OOP will have significant advantages. For CBI, the industry partner of this research, the advantages of OOP exceed that of ABC, primarily since autonomous reconfiguration and emergent behaviour are not high priorities in their situation, while OOP provides better integrability with hardware.

## 5 Control Software Implementation

This chapter describes the implementation of the station control software and how OOP principles were utilized for defining holon classes, with references to C# features that are useful when it comes to designing a controller for an RMS.

### 5.1 Controller functional requirements

The hardware that is controlled by the station controller includes the robot, the gripper that is attached to the end effector of the robot, the buffer, and the poles that are moving through the station. The pallets which are transported to and from the station are not controlled by the station controller directly, but rather by a separate conveyer controller. The station controller nevertheless needs to be aware of the pallets' contents and locations. The following are the main functions of the station controller:

- The station controller must be able to receive product information, pallet information and orders from the cell controller.
- When orders are placed the station controller must ensure that the corresponding products are assembled as soon as possible and attempt to complete all orders before their desired completion date.
- Source pallets filled with tested, unriveted poles must be received on the one conveyer, whereas stacked assemblies must be placed into destination pallets containing riveting fixtures on another conveyer.
- Poles must be obtained either from the source pallets or from the buffer and get stacked into a riveting fixture on one of the destination pallets.
- Unused poles must be transferred to the buffer until they can be used at a later stage.
- Pallets must not be moved by the station controller directly but messages must be sent to the cell controller, to request that pallets are being moved away once no longer in use.
- The cell controller must be notified when orders have been completed or are overdue, when required information is missing, when pallets are to be moved away or in case of errors or warnings.
- The cell controller must be informed of the IDs of poles that have been matched to form an assembly.
- The status of various aspects of the system should be displayed on a human-machine-interface. The station controller must be able to process input received from the operator via the HMI.

### 5.2 Control architecture

To optimally utilize the robot and to achieve the highest possible throughput rates, which should be aimed for at a station-level, one would traditionally opt for a centralized approach. However, due to some poles failing at the electrical test station in an unpredictable pattern and due to the unpredictable arrival time of pallets, the stacking station is constantly subjected to disturbances, which are better handled when holons have a higher level of autonomy, i.e. when a heterarchical approach is taken. To

achieve optimal throughput rates while also having the flexibility to react to disturbances, ADACOR was chosen as the architecture which can dynamically adapt between a hierarchical and heterarchical structure.

ADACOR consists of the product, task, operational and supervisor holons. In addition, some staff holons have been adopted from the PROSA architecture.

The operational holons were used to represent anything physical that had to be managed. Product holons were Task holons were driving production

A supervisor holon was required to ensure optimal execution of competing task holons. If several task holons, each with a local view, were to individually try and get their orders completed as quickly as possible, the system would be at risk of becoming congested and urgent orders might be completed too late. Without supervision congestions could occur when all fixtures on the destination pallets were occupied by partially stacked assemblies of which none could be completed due to missing poles. These congestions could eventually lead to the buffer reaching its full capacity and bring the station to a complete halt. Supervisor holons were thus required to prevent those congestions and also to handle any poles on the source pallets for which none of the task holons would take responsibility and transfer those poles to the buffer.

Due to the failure of some poles in the preceding test station, the products will take different routes through the station so that task holons will keep interacting with different operational holons. At times, poles will have to be retrieved from the buffer, and when several source and destination pallets are each holding poles meant for different types of products, then possible interactions between the various holons could be manifold. A heterarchical approach was therefore needed in conjunction with supervision.

Holons of the ADACOR architecture proposed by Leitão and Restivo (2006) have the ability to learn. This functionality however was not implemented in the current research since it could result in unpredictable behaviour occurring suddenly after a long time of normal operation and CBI would not be comfortable with this. This feature would further make the system unnecessarily complex without adding much value. On station-level relatively few devices are to be controlled and hence humans would be able to easily identify the changes required when a new type of disturbance occurs, for which holons with a learning ability would not necessarily find as good a solution. Holons were therefore programmed to continuously operate according to a predefined set of rules until those rules would be redefined by a human.

Figure 20 shows the interrelationships between the different types of holons as well as some of the communication between the station controller and the cell controller.

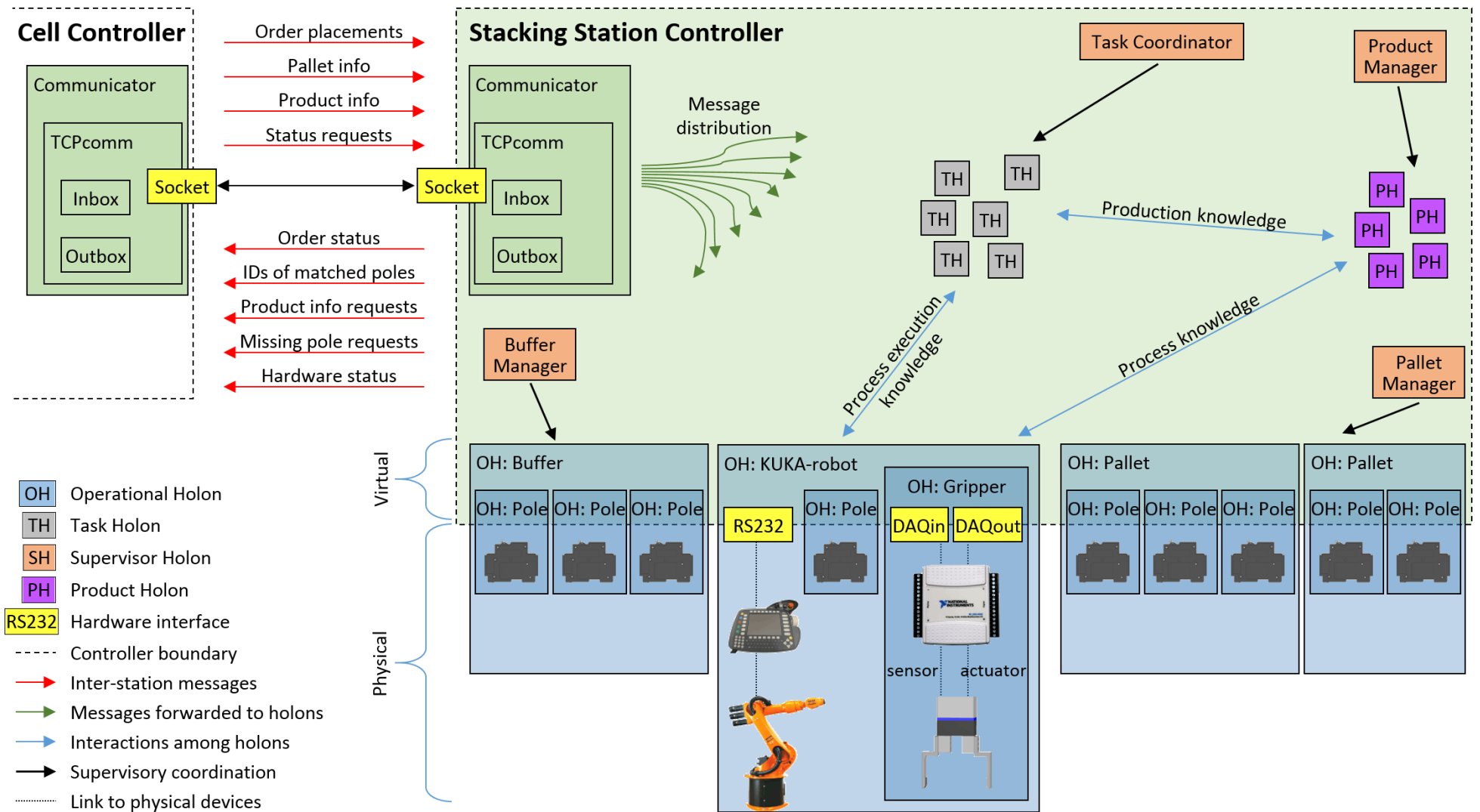


Figure 20 Stacking station control architecture



### 5.3 Separation of tasks among multiple threads

To ensure near-real time execution of the hardware and to minimize robot idle time, time-critical processes and CPU-intensive tasks or asynchronous tasks were run in separate processes.

As shown in Figure 21, three different processes were run, each in its own thread:

- The GUI had to be in its own thread since it could hamper the execution of hardware processes whenever major updating (refreshing) is to be done on the user form.
- Sending and receiving of XML messages over Ethernet had to be dealt with in its own thread since TCP/IP communication happens asynchronously and would otherwise produce delays of unpredictable duration.
- All the holons, including the supervisor holon were run in the stacking station's main thread.

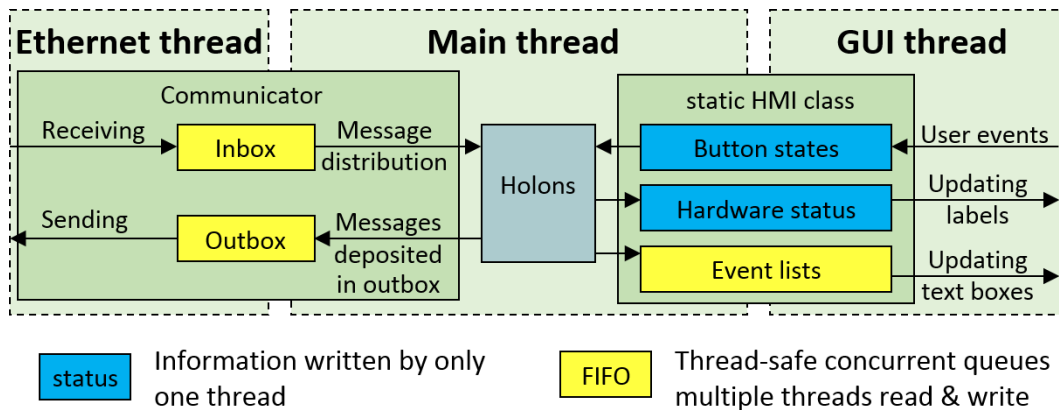


Figure 21 Memory shared among various threads

Although “autonomy of holons” could refer to all of the holons running the entire time on separate threads and communicating with one another asynchronously, this approach was not taken for the stacking station. Instead, all holons were running on the main thread only since all of the stacking station’s hardware would always be controlled sequentially, never in parallel. Although, for example, pallets could be moving while at the same time the robot is in motion, the pallets would not be controlled by the stacking station controller directly. The station controller would only send a corresponding command to the conveyer controller when a pallet is to be moved.

By having the various task holons all run in turns on the same thread, only one of them would try to book resources at a time. Had they been running on separate threads quasi-simultaneously, task holons could potentially undermine one another’s plans by booking certain resources which other task holons would require to complete a task.

Although with this approach holons are only active when given their turn, they are nevertheless autonomous since they used all the information they had available to make

all decisions themselves without having to pass information on to a higher level and ask for advice. The supervisor (discussed in section 5.5.5.1) made no decisions on their behalf, other than deciding on the optimal order in which they are to be activated to achieve highest possible throughput rates.

To ensure safe operation between the threads, inherently thread-safe queues had to be used from the .NET libraries (System.Collections.Concurrent.ConcurrentQueue<T>), which act as a first in first out (FIFO) buffer. The **Enqueue()** method is used to add an element to the end of the queue. Adding elements will generally be a successful operation, even when multiple threads are working on the same ConcurrentQueue. In contrast, removing elements is not guaranteed to be a successful operation when multiple threads are busy working with the same ConcurrentQueue. To remove the first element of a queue the **TryDequeue()** method must be used, which returns the requested element if successful, or null when unsuccessful.

Thread-safe FIFO buffers shared among the threads are indicated in yellow in Figure 21. They include the inbox and outbox which allow holons to communicate asynchronously over the Ethernet. Thread-safe FIFO buffers were also used for lists of events for the event log textboxes (discussed in section 5.5.1).

A static class called 'HMI' was used to allow the station controller thread and the GUI thread to share information such as user input or the status of hardware. To output the status of task holons or operational holons, event logs were stored inside the above mentioned thread-safe FIFO-lists within the main thread, and removed within the GUI thread.

Each class which was to be run in a separate thread had amongst others, a **Run()** and **Terminate()** method. The **Run()** method would be executed inside a thread of its own and would loop as long as that object's run-flag is set. The **Terminate()** method can be invoked publically and, when executed, un-sets that run-flag. By this approach threads can be "asked" by other threads to safely finish the current iteration and thereafter terminate. A flow diagram indicating how the threads are initiated and interact with one another can be found in Figure 33 in Appendix B.

## 5.4 Inter-holon communication

Since it was decided to let all holons run on the same thread, the ability to communicate asynchronously was not required, except when communicating over the network. Simple method calls are therefore sufficient for communication among holons on the same thread. Using method calls means that the calling holon will have to wait for the called holon before being able to continue its execution. A communicator holon (discussed in section 5.5.7) was developed to handle asynchronous communication over Ethernet. Holons could simply deposit messages into the communicator's outbox and immediately continue with their actions without having to wait for a reply, since the communicator would deal with the message from that point onwards and will then forward the reply it received to the corresponding holon.

Methods which needed to be invocable by other holons were declared public where methods which were to be encapsulated and hidden from other holons were declared private or protected. Where appropriate, interfaces were used to enforce standardized signatures for methods on certain types of holons. Using method calls, it is generally very intuitive for the programmer to know how holons are to be called. For example, when information is requested from a product holon, the product holon will simply return a reference to the requested information if available, or a null pointer otherwise.

C# has a keyword **this**, which can be used by holons to pass references to themselves to other holons as an argument forming part of a method call, or as a return value. When for example an operational holon agrees to provide a certain service to a task holon, the operational holon would use **this** to pass a reference to itself to the requesting task holon. At some later stage the task holon can thereby easily get hold of the operational holon that agreed to offer the service.

### 5.5 Responsibilities and functionalities of the various holons

Using inheritance and polymorphism, the various holons running on the station have all directly or indirectly been derived from a generic holon, as shown in Figure 22 below. The various types of holons are discussed in the sections that follow. As part of the discussion, some features on the graphical user interface, shown in Appendix A, are referred to.

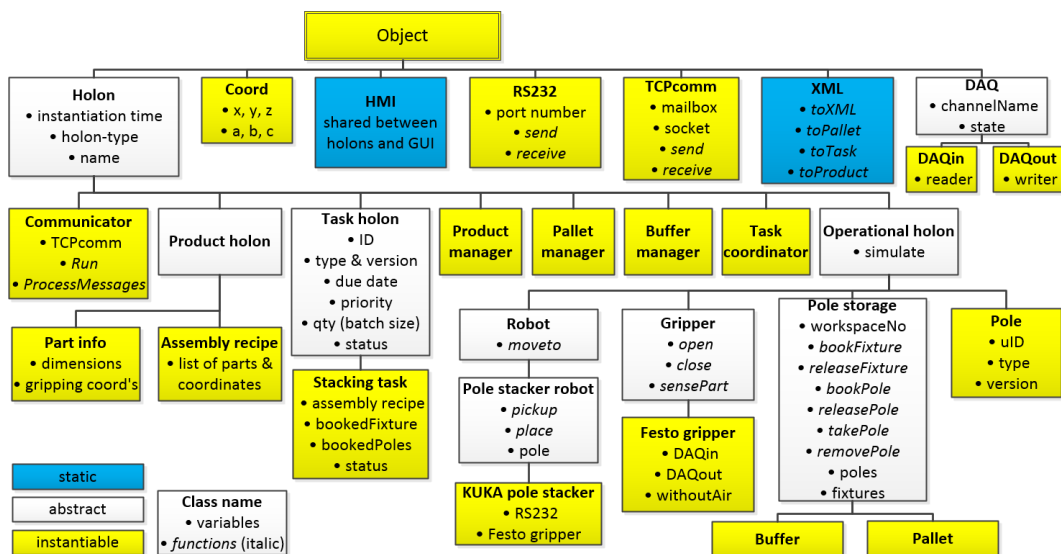


Figure 22 Inheritance hierarchy

#### 5.5.1 Generic holon

The generic holon was declared an abstract class and given the following properties that all the holons would inherit regardless of their type: name, holon-type and timestamp.

The **name** property contains a description of the holon, for example “pallet 7” or “KUKA KR16-2 robot”. As holons perform actions or detect problems their activity is being

recorded and logged in chronological order in an event textbox on the HMI as shown in Figure 23. To allow the operator better insight into their activities, some event log entries are accompanied by their names. The event textbox was implemented since having a good insight into the sequence of events can drastically simplify diagnostics.

The **holon-type** property is used to distinguish the different types of holons from one another. For a clearer overview in the event textbox described above, holons produce their event logs in a unique colour which is determined by their type. On the right in Figure 23, it can further be seen how those events can be filtered by selectively muting the activities of certain types of holons.

Other than for the event textbox, the name property is further used as the sender-field when constructing XML messages, as will be described in section 5.5.7. Another use of the name property comes into play within the file handling class. To allow reverting back to previously taught configurations, certain holons have to store and retrieve settings on the hard drive. Being able to use the holons' names made it an easy task to retrieve the correct settings for each of the holons.

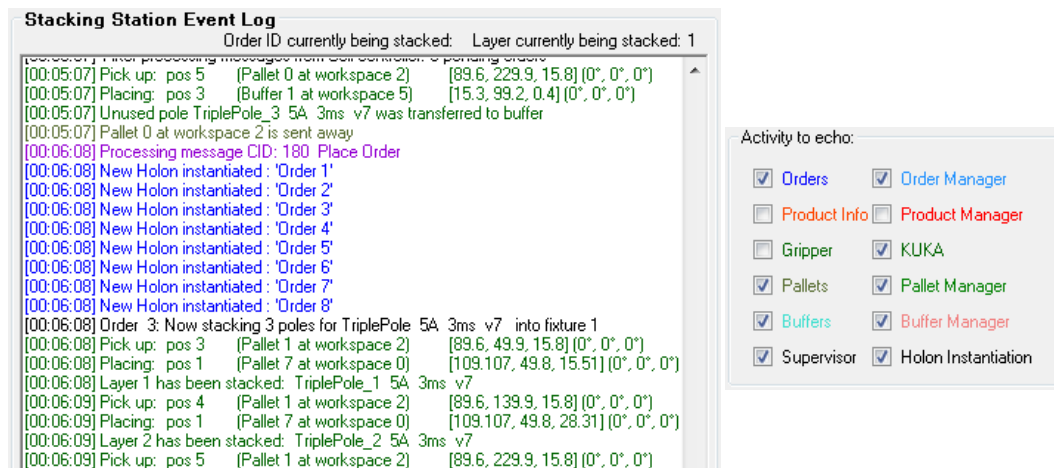


Figure 23 Station event log simplifies diagnostics. Holon activity can selectively be muted.

Within the generic holon's constructor method, the current system time is stored in the **timestamp** property which effectively records the instantiation time of the holon. This allows the **age()** method to determine the total running time of any holon and can be used to detect if, for example, a pallet or pole is stagnant in the system for much longer than the norm. The constructor method further lets the holon announce itself in the event log, unless "Holon instantiation" (Figure 23) is unchecked, or when that particular type of holon is set to be mute.

All derived holon classes inherit those properties and methods of the generic holon. However, polymorphism makes it possible for the derived holon classes to override the inherited methods and to define additional properties and methods. To let the constructor of the derived class extend the constructor of the parent class (i.e. to re-use the functionality of the parent constructor while possibly adding more functionality in

the derived constructor), the **base** keyword is placed directly after the constructor's header. Using this concept, new instances of derived holon classes would automatically announce themselves when their constructor method was called, but had the option to perform additional actions upon instantiation.

Abstract classes cannot be instantiated. Yet classes which are derived from an abstract class can be instantiated as long as they are not abstract themselves. This means an object of type "Holon" cannot be instantiated, whereas for example an object of type "Stackingtask holon" (non-abstract) could be instantiated.

### 5.5.2 Product holon

The purpose of the product holons is twofold. Firstly, to help the task holons with planning the execution of their processes, by specifying the sequence of events that must be scheduled by the task holons in order to get products produced; and secondly, to provide the operational holons with process knowledge, i.e. the machine parameters required by the physical devices in order to make the product according to specifications.

For this purpose, two types of product holons were defined for the stacking station, both of which have been derived from the abstract product holon :

The first type is a **partInfo** product holon which contains product information about individual parts, such as the thickness and dimension of a particular part (e.g. pole), and the coordinates where the part is to be picked up, relative to a reference point. This is the information that the product holons share with the operational holons.

The second type of product holon contains an **AssemblyRecipe** which contains an ordered list, where each entry of the list specifies a part required along with the orientation and coordinates where that part has to be placed. This recipe is shared with the task holons.

The product holons do not perform much decision-making, but serve mainly as information-servers. When the requested information is not available inside the product holons, then the inquiring holon will receive *null* as a reply and the product holon will request the missing information from the cell controller. The latter only happens if such a request was not already sent in the past 15 seconds to prevent the cell controller from being flooded with requests for product information. For that means, the abstract generic product holon was given a property to store the time of the last request.

Another property of the abstract product holon was the version number of the product information it was holding. This allowed for gradual changes to be made to some physical parts and assemblies and keeping their virtual representation up to date, while also accommodating some older versions of parts or assemblies that might still be used.

### 5.5.3 Operational holon

Operational holons of ADACOR are quite similar to the resource holons of PROSA which represent the abstraction of production means and include, among others, machines

such as robots, conveyers, pallets, components and raw materials (Van Brussel, et al., 1998). For this reason, anything physical that needs to be managed was defined as an operational holon, namely the robot, gripper, poles, buffers and pallets.

### 5.5.3.1 *Generic operational holon*

The abstract generic operational holon has the Boolean property *simulate* which, when set to true, lets only the virtual part of the operational holon execute. In this mode, no actual hardware is controlled, and feedback from sensors is disregarded. Simulation mode allows the developer to make changes to the software and run virtual tests offsite, where no physical hardware is available. This reduces ramp-up time and eases the impact of reconfigurations on ongoing production since a great part of testing can happen on another controller without having to pause production. Furthermore, reconfiguration costs are reduced by requiring fewer highly skilled personnel on-site.

Since there are no hardware-related delays in simulation mode, the virtual system can be run at much higher speeds which allows observing the long-term system behaviour under certain conditions (e.g. where bottlenecks are most likely to occur or the effect of disturbances) without actually having to wait that long.

### 5.5.3.2 *Pole*

Poles have the following properties: *uid*, *partNo*, and *versionNo*.

***partNo*** is the part number of the pole and conveys information about the pole's type, such as its ampere rating and for which layer (i.e. bottom, middle or top) of a stack it is suitable, and what type of stack it is made for (e.g. triple pole of the QA-range).

***versionNo*** specifies the version number of the pole and is used to ensure that poles are only used for a stack when they are compatible with the other poles of the same stack. When for instance an external feature of a pole would change, it might no longer interlock correctly with other poles of that stack. Therefore, whenever an adjustment was made to the design of a pole, its version number would be updated to reflect that change.

***uid*** stands for unique identifier and was assigned by the cell controller. To allow traceability of each and every part which eventually ended up in an assembly, various manufacturing details were recorded, in order to easily find the cause of a possible problem. Along with the pole's ***uid***, it was recorded which task holon was driving the production, which operational holon has handled the pole, and into which fixture the pole has been placed. When building stacks it is important to record which poles have been put together. Should the end product be faulty, one will know which routes were taken by the individual parts.

### 5.5.3.3 *Gripper holon*

The code for the gripper holon is shown in Appendix C. Grippers need the ability to open and close their jaws and, for diagnostic purposes, the ability to sense whether the part being picked up is properly gripped. The abstract generic gripper holon was therefore

designed with three methods, namely `open()`, `close()` and `sensePart()` to provide those abilities.

When gripper fingers are being exchanged with gripper fingers of different dimensions, then the gripper needs to be brought closer or further away from the object being picked up to compensate for the change in gripper finger length. Gripper holons thus need to be aware which gripper fingers are attached to the jaws, which is why the generic gripper holon was assigned two properties. The first property contains a list of gripper fingers that can be attached to the gripper jaws, and the second property specifies which set of gripper fingers out of that list is actually attached.

The Festo gripper holon was derived from the generic gripper holon. Since the air control valve and the position sensor are controlled via the National Instrument DAQ, the Festo gripper holon must know how to interface with the DAQ. For this purpose, `DAQin` and `DAQout` classes were created which incorporate National Instrument drivers for reading and writing digital signals from and to the DAQ. These hardware interfaces serve as the link between the operational holon's virtual part and physical part. In Figure 20 they are therefore indicated by the yellow blocks which are lying on top of the station controller boundary.

The ***withoutAir*** property was defined to allow the virtual part of the gripper holon to run smoothly even when no air supply is present. During ramp-up tests this feature can be used to perform dry runs, i.e. to let the robot move to all the pickup and place positions but without the gripper actually picking up and transferring parts.

#### 5.5.3.4 Pole stacking robot holon

Since all robots, regardless of their type, are able to move to a specified location, an abstract robot class with an abstract method `moveto()` was defined. Declaring an abstract method in the base class forces all derived non-abstract classes to provide an implementation for that method, e.g. the `moveto()` method. Not all robots (e.g. painting or welding robots) have the ability to pick up and place objects which is why the generic class does not provide methods to cater for such functionality.

For the case study, a 6-DOF KUKA robot was used to stack poles, but for the sake of reconfigurability it should be possible to use any other capable robot in its place. The internal working of those robots would differ and should be hidden, but to enhance integrability, the same standard interfaces to the outside world should be used by all pole stacking robots. This allows for one robot to easily be swapped with another and only the operational holon responsible for managing the new robot would need to be programmed, while no reprogramming of other cooperating holons would be required.

To achieve integrability, an interface was written, containing two methods: `pickup()` and `place()` which would be required by any pole stacking robot, regardless of its type. An abstract pole stacker robot class was then derived from the generic robot class, and was made to also inherit the `pickup()` and `place()` methods of the abovementioned interface. As an additional field a pole object was included to represent the physical pole being held by the robot.

From the abstract pole stacker robot class several types of robot classes can be derived which represent actual robots. For the case study, a (non-abstract) KUKA pole stacker class was implemented, which further contained a Festo gripper holon and an RS232 class responsible for serial communication with the robot.

An instance of the Festo gripper holon was encapsulated within the KUKA pole stacker holon, rather than having the gripper run as a separate entity of its own. The reasoning behind this is that the gripper and the robot work closely together, i.e. the gripper will never open or close unless the robot is motionless, and the robot will never start moving until the signal from the gripper holon has confirmed that the part is properly gripped. In other words, these two devices will never run concurrently, but always sequentially. This encapsulation eliminates any direct communication between the task holons and the gripper holons. The cooperation between the gripper and the robot are thus hidden inside an operational holon and the task holons can always rely on using the interface described above to get poles transferred.

Part of the KUKA pole stacker holon is running on the KUKA controller which has built-in functionalities for handling various arbitrarily oriented coordinate systems, inverse kinematics and procedures for calibrating workspaces. On-board the KUKA controller, the coordinates for a limited number of calibrated workspaces (e.g. buffers and stopping positions of pallets) and tools (e.g. grippers) can be stored.

To avoid unnecessary recalibration when reverting to a previously calibrated configuration (e.g. reusing a previously used set of grippers), a history of calibration coordinates must be stored for future reuse. The KUKA controller, however, does not support dynamic memory allocation, but can only store a very limited number of coordinates and is therefore unsuitable for keeping a history of calibrated workspaces. After calibrations, the coordinates are therefore retrieved and stored on-board the station controller, which is capable of keeping a complete history of the calibration data.

The flow diagram for the KUKA Robot Language (KRL) code running on-board the KUKA controller is shown in Figure 32, Appendix B. Workspace and tool calibration procedures are described in Appendix E.

#### *5.5.3.5 Pole storage holon*

Although the buffers and pallets used in this research are not equipped with any actuators or sensors and therefore require no machine interfaces, they must still have a virtual representation since the poles stored inside them need to be managed, as well as the slots into which poles can be placed. Somewhere it must be recorded which poles are inside which slots, along with the coordinates of those slots. A pole storage holon class was therefore defined containing a list for poles and fixtures to fulfil those requirements. Buffer and pallet classes were both derived from this class since they both contain fixtures/slots into which poles can be placed and fulfil very similar functions.

The task holon (discussed in section 5.5.4), which is responsible for getting the poles stacked, must interact with the pole storage holons to find available fixtures and to book all poles required for building a stack. The pole storage holons were therefore given the



bookPole() and bookFixture() methods along with the releasePole() and releaseFixture() methods, since bookings being made might sometimes need to get cancelled.

Whenever a robot removes a pole or places a pole into a fixture, the pole storage holon must update its inventory list. For interactions with robot holons, the pole storage holon class was therefore further given the methods takePole() and removePole().

Robots which interact with buffers or pallets need to know the location and orientation (i.e. coordinates) of those buffers and pallets. The coordinates of the fixtures relative to some reference point on the pallet/buffer are stored in the pole storage holon. The coordinate of the reference point itself, however, is not stored in the pole storage holon, but in the robot holon that has to interact with the pole storage holon. The pole storage holon only stores a workspace number (Figure 24) which it communicates to the robot holon. The robot then maps the workspace number to the correct set of coordinates which were stored on-board the KUKA controller when that particular workspace was first calibrated. With this approach the pole storage holons can be left completely untouched in case a robot is being exchanged with another robot. Only the new robot would need to be calibrated for each of the surrounding workspaces, which cannot be avoided.

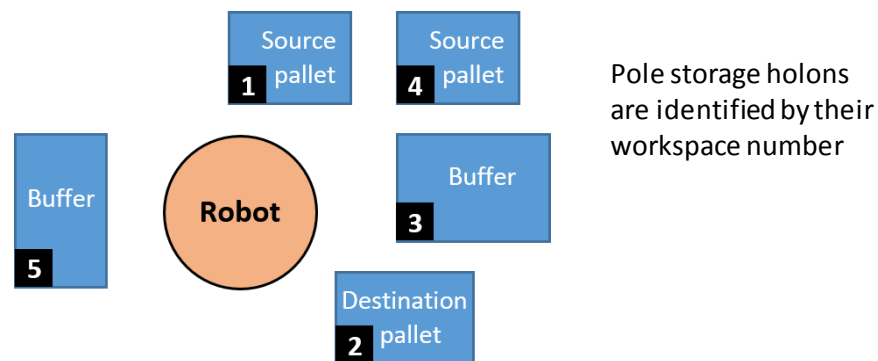


Figure 24 Robot contains coordinates of surrounding workspaces' reference points

When for example a pole is to be picked up out of a pallet, the robot must be given the coordinate of the pole relative to some reference point on the pallet along with the workspace number of that pallet. The absolute coordinate to which the end-effector must then move to is calculated on-board the KUKA controller which maps the workspace number to the calibration data, and adds the relative coordinate of the pole.

#### 5.5.4 Task holon

The generic task holons were designed to process batch orders (i.e. multiple basic orders of exactly the same type) and were given the properties described below.

The **ID** (i.e. unique identifier) is used to distinguish between orders. Also, whenever a part is being handled, the process it underwent is recorded along with the ID for traceability purposes.

**productType** and **version** are properties which together are used to unambiguously refer to the type of product to be produced, and are used when communicating with the product holon.

The **priority** property is zero by default, but can be set to a higher level to indicate that the particular order should be given preference above orders with a lower priority level.

**BatchSize** and **BatchCounter** respectively refer to how many products need to be produced for a batch order, and how many of them have already been completed.

**CompleteBefore** is the time by which an order ought to be completed and can be used to determine the urgency of an order. In case orders become overdue, the cell controller is to be warned. To prevent the cell controller from being flooded with warnings, the time when the last warning was sent is recorded inside the **lastTimeoutError** property and the next warning is only sent if at least 15 seconds have elapsed.

Lastly, the **status** property is used to store the current state of a (partially complete) order.

#### 5.5.4.1 *Stacking task holon*

A stacking task holon class was derived from the generic task holon class and contains these additional properties:

The **assemblyRecipe** property represents a list of parts that need to be assembled in a specified order and also contains the coordinates to specify how and where those parts are to be oriented and placed.

Since the poles are to be riveted after being stacked, they need to be placed into a riveting fixture. Such a fixture must first be booked, for which the **bookedFixture** property was designed which holds the reference to an available fixture on one of the destination pallets.

The individual poles which are required to form a stack must also be booked before stacking can begin. The **bookedPoles** property contains a list of references to pole storage holons which hold the required poles. Rules have been set such that stacking task holons will first negotiate for poles with the source pallet holons before trying to obtain poles out of the buffers. By following these rules, poles would not be transferred to the buffers unnecessarily, thereby utilizing the robot more efficiently.

Partially stacked assemblies which cannot be built to completion could congest the system. Rules have therefore been set so that the task holons would not initiate the stacking process unless the following requirements have all been met:

- Product holons hold all the information about the assembly to be produced as well as all the information about the individual parts that the assembly is made of.
- A fixture capable of holding the entire stack of poles is available on one of the destination pallets.

- All poles required for the assembly are already present in the station, be it on one of the source pallets or inside one of the buffers.

#### 5.5.4.2 *Typical sequence of events for a stacking order*

Since task holons are the ones driving production, it is sensible to describe here the sequence of events (Figure 25) that would typically play out from the placement of an order until its completion. Several disturbances can happen to which the controller will react differently, and the cell controller can send messages in any order which could result in a different sequence of events. However, for the sake of this discussion it is assumed that the cell controller has sent the messages listed below and that the communicator (discussed in section 5.5.7) has already distributed the messages to the various holons so that the stacking station holds all information required to complete an order.

It is assumed that the cell controller has already sent the following messages:

- Placement for an assembly order, which will lead to the instantiation of a stacking task holon.
- Notifications that two pallets have arrived: One containing tested poles and another pallet with empty riveting fixtures. Two pallet instances will be created due to these notifications: a source pallet and a destination pallet.
- Product information for the assembly for which an order has been placed. A newly created assemblyRecipe holon will store this information.
- Part information for all the parts of the abovementioned assembly. One partInfo holon will be created for each part of the assembly.

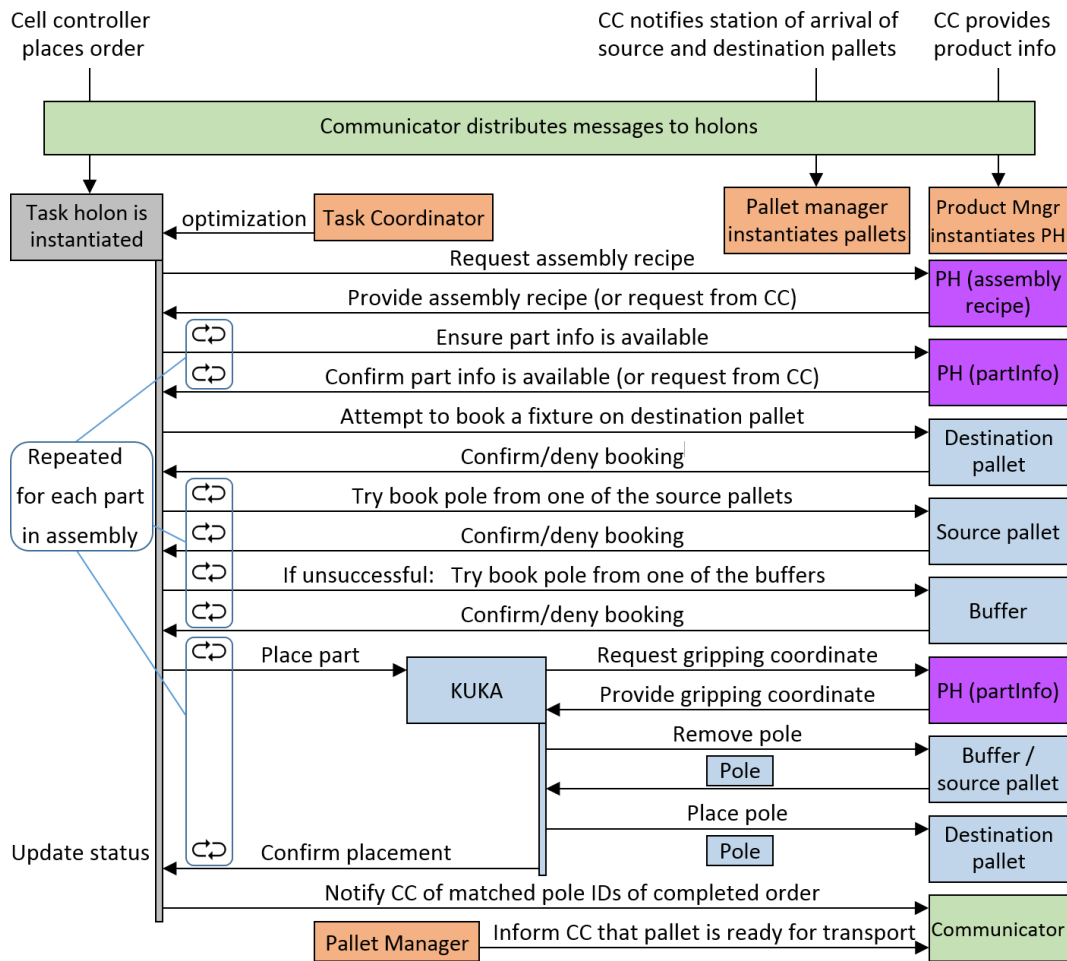


Figure 25 Sequence of events from order placement to completion of order

If several task holons are active on the station controller, the task coordinator will try to optimize their order of execution, as described in section 5.5.5.1.

Since all task holons are running on the same thread, they will execute one by one. Once a task holons gets its turn, it will:

1. Try to obtain the assemblyRecipe from the corresponding product holon. For this, the product type and version number are used as reference. The recipe is not copied, only a reference to the product holon is stored inside the task holon. To get hold of the correct product holon, the task holon will initially direct its query to the product holon manager which contains a list of product holons.
2. For each part listed in the assemblyRecipe, it will ensure that the correct partInfo is available.
3. A fixture will be booked on the destination pallet, which must be capable of holding the entire stack.
4. For each part listed in the assemblyRecipe, the task holon will try to book that pole from one of the source pallets, where possible. To get hold of a pallet

holon, the task holon will initially direct its query to the pallet manager which contains a list of pallet holons.

5. If none of the source pallets are holding the required pole, the task holon will try to book it from the buffer.
6. Once all fixtures and poles have been booked, the robot can start stacking the poles into the fixture one by one.
  - a. In order to grip the pole correctly as indicated in Figure 16, the robot holon requests the gripping coordinates from the partInfo holon.
  - b. After the pole is lifted out of its pickup-position and the gripper's proximity sensor has confirmed that the pole is properly gripped, the pole will be de-registered with the pole storage holon.
  - c. Once the pole has been placed into the riveting fixture, it will be registered with the destination pallet.
7. The robot confirms that a pole has been stacked, and the task holon updates the state of the order.
8. Upon completion of an order, the cell controller will be informed about the ID's of the matched poles. The task holon will then be disposed of.
9. If any source pallet becomes empty or if any destination becomes full, a request is sent to the cell controller to transport the pallet away and the pallet holon will be disposed of.

To prevent partially stacked assemblies from congesting the system, the task holons would only start the stacking process once all required parts are available along with the part information. If, during the booking process any fixture or pole cannot be booked, then all those resources which have already been booked are being released again, to be used by another order.

### **5.5.5 Supervisor holon**

The supervisor holons have been implemented to allow scalability, to prevent congestion, to ensure an orderly manner of execution in an attempt to achieve optimal throughput rates, and to help with the instantiation of some of the basic holons.

Next, the responsibilities of the task coordinator, product manager, pallet manager and buffer manager are discussed.

#### *5.5.5.1 Task coordinator*

The task coordinator fulfilled the role of a supervisor holon. It was implemented to ensure an orderly manner of execution in an attempt to achieve optimal throughput rates

Transferring poles to and from the buffer is generally an unproductive robot manoeuvre which should be avoided where possible. By letting the task coordinator determine the order in which task holons execute, buffer utilization can be reduced resulting in higher throughput rates.

The task coordinator periodically sorts the tasks according to their priority, urgency and complexity. More complex tasks (i.e. assemblies consisting of many parts) have a smaller

chance of having all required parts available at the station at any given stage, and should therefore be addressed first. Since sorting is not a very CPU intensive process, the task coordinator can be run in the main thread without causing noticeable delays of the hardware.

Using the task coordinator, some form of adaptive control is achieved: Within each cycle the control architecture alternates between being centralized and being heterarchical:

At the beginning of each main cycle, when the task coordinator decides on the order in which task holons should be executed, the task holons need to pass their local information on to a higher level, so that decisions can be made at that higher level. This is typical for hierarchical or centralized control architectures.

However, after the task coordinator has finished the sorting algorithm, the system switches back to heterarchical control. All holons then make decisions themselves based on their local view in order to effectively handle disturbances (e.g. poles having failed the electrical test, missing product information, pallets of which some are unsuitable for a given task, etc.).

Besides determining the execution order of the task holons, the task coordinator also counts the total number of orders and conveys this number to the cell controller upon inquiry. When several stacking stations are working in parallel, the cell controller can then decide to place further orders with the stacking station which has the fewest pending orders.

### **5.5.6 Staff holons**

Staff holons assist the basic holons with their tasks. In particular, they provided information that was required during the instantiation of new holons.

#### *5.5.6.1 Product manager*

When the cell controller provides product information, it is the responsibility of the product manager to prevent multiple instances of the same product holon from being instantiated.

Some of the messages sent by the cell controller are directed towards particular product holons directly and will be treated by those holons themselves. Other product-related messages which do not have a particular recipient will be processed by the product manager. Upon request from the cell controller or the operator, the product manager can also provide a list of all available product information in a structured way.

#### *5.5.6.2 Pallet manager*

Whenever pallet-related messages are sent over the network, only the pallet ID is sent along as a reference. To reduce communication overhead, the pallets' fixture coordinates are not sent along with the message but are stored on-board the station controller. Upon arrival of a pallet, the pallet manager uses the pallet ID to load the fixture dimensions for the newly instantiated pallet holon.

Initially, only the cell controller contains the fixture coordinates for all the pallets in the factory. As pallets arrive at the station for which the coordinates are not known yet, the pallet manager will obtain those coordinates from the cell controller and store them locally. This approach enhances the robustness of the station. In case the cell controller would go offline for a while, the stacking station can continue its operations since it can rely on the local copy of the pallet information.

Within the stacking station each transverse conveyer on which a pallet can arrive has a unique workspace number which is used by the pallets to interact with the robot. Since those workspace numbers are for internal use only, they cannot be provided by the cell controller but are provided by the pallet manager.

The pallet manager periodically performs checks to see if a pallet can be sent away. Whenever a pallet arrives with poles, task holons will try and use as many of those poles as possible for their orders. Some poles cannot be used straight away since they are meant to form part of a stack of which other poles are still missing (e.g. due to failing the electrical test). The pallet manager will then arrange for those unclaimed poles to be transferred to the buffer so that the pallet can move along and give way for the next pallet.

#### 5.5.6.3 *Buffer manager*

The buffer manager is a staff holon. During reconfigurations when the system is scaled up by adding new buffers, the buffer manager provides unique workspace numbers for each of the buffers, to be used in interactions with the robot.

Furthermore, whenever the total capacity of all the buffers combined reaches a utilization level of 75%, 90% or 100%, the cell controller will be notified.

### 5.5.7 Communicator holon

The communicator holon was developed to allow holons on the stacking station to communicate with the cell controller and other collaborating stations (for example the conveyer or additional stacking stations that might be working in parallel). It contains a **TCPcomm** class which handles asynchronous communication over network sockets. The TCPcomm class contains a mailbox, which in turn consists of an inbox and an outbox.

Other holons on the stacking station that wish to communicate with the cell controller deposit their messages in the communicator's outbox. Messages which are received over the socket are deposited in the communicator's inbox. The inbox and outbox are thread-safe FIFO buffers. They are being accessed by the communicator's **Run()** and **ProcessMessages()** methods which are running on two separate threads. The flow diagrams for these methods are shown in Figure 34 in Appendix B.

The **Run()** method runs continuously in its own thread and is responsible for asynchronously sending and receiving XML messages over the network socket. When a holon places its message in the outbox, it will be converted to XML and the communicator will immediately attempt to send it. The message will temporarily remain in the outbox and will only be removed once receipt thereof has been confirmed. If

receipt is not confirmed within 15 seconds after sending, the message will be resent repeatedly with increasing time intervals (i.e. after 30, 60, 120 seconds, etc.) until eventually receipt has been confirmed.

Should the connection to the cell controller temporarily be lost, messages will be sent through after the connection has been re-established. In the meantime, the station controller could run out of tasks but will not fail. By letting the communicator ensure all messages eventually get through to the intended recipient enhances the robustness of the controller and shifts this responsibility away from the basic holons to the communicator holon.

The *ProcessMessages()* method runs periodically on the main thread and is responsible for forwarding the messages from the inbox to the local holons. Since the messages are received in XML format, they must be converted to a format appropriate for the holon.

Converting pallet, product and task holons to XML messages, and vice versa, is handled by the communicator only. When for example a pallet is to be sent away and the cell controller must be informed of its contents, then the communicator receives a pallet holon which it converts to an XML message. Likewise, when the cell controller places an order as an XML message, the communicator must convert it to a task holon.

It was decided to keep the conversion responsibility with the communicator only, and to not introduce XML to any of the other holons. If one would later on decide to use an alternative to XML, only the communicator's code would need to change while leaving all other holons untouched.

In C# the System.Xml.Linq library can be used to easily construct and parse XElement s. The XML message structure that was agreed upon by the members of the research group is shown in Table 5 in Appendix D. The type of messages that the station controller can send and expect to receive are listed in Table 6 and Table 7 in Appendix D.

## 5.6 Conclusion

This chapter discussed the detail software implementation of the controller. The chosen control architecture is ADACOR which incorporates the task, operational, product and supervisor holon classes. To this architecture, the communicator holon was added to facilitate communication between the cell controller and holons on the station controller. To evaluate the station controller in terms of reconfigurability, a series of reconfiguration experiments were carried out which are described in the next chapter.



## 6 Evaluation

To be able to evaluate the performance of the stacking station controller, a small scale laboratory setup was built on which a variety of tests have been carried out. First some of the rudimentary procedures that would be needed for many of the reconfigurations are described, followed by the actual reconfiguration experiments, such as product changeovers, scaling up, readjusting the system, etc.

The basic hardware based procedures include: calibrating the gripper's sensors, the gripper fingers and the workspaces surrounding the robot. Software based procedures include: the creation of holons on-the-go, and the programming of new type of holons.

### 6.1 Experimental setup

The setup that was used for the experiments in the research laboratory is shown in Figure 26. Some compromises had to be made due to limited space and resources in the automation laboratory which resulted in the following differences between the ideal factory layout and the experimental setup:

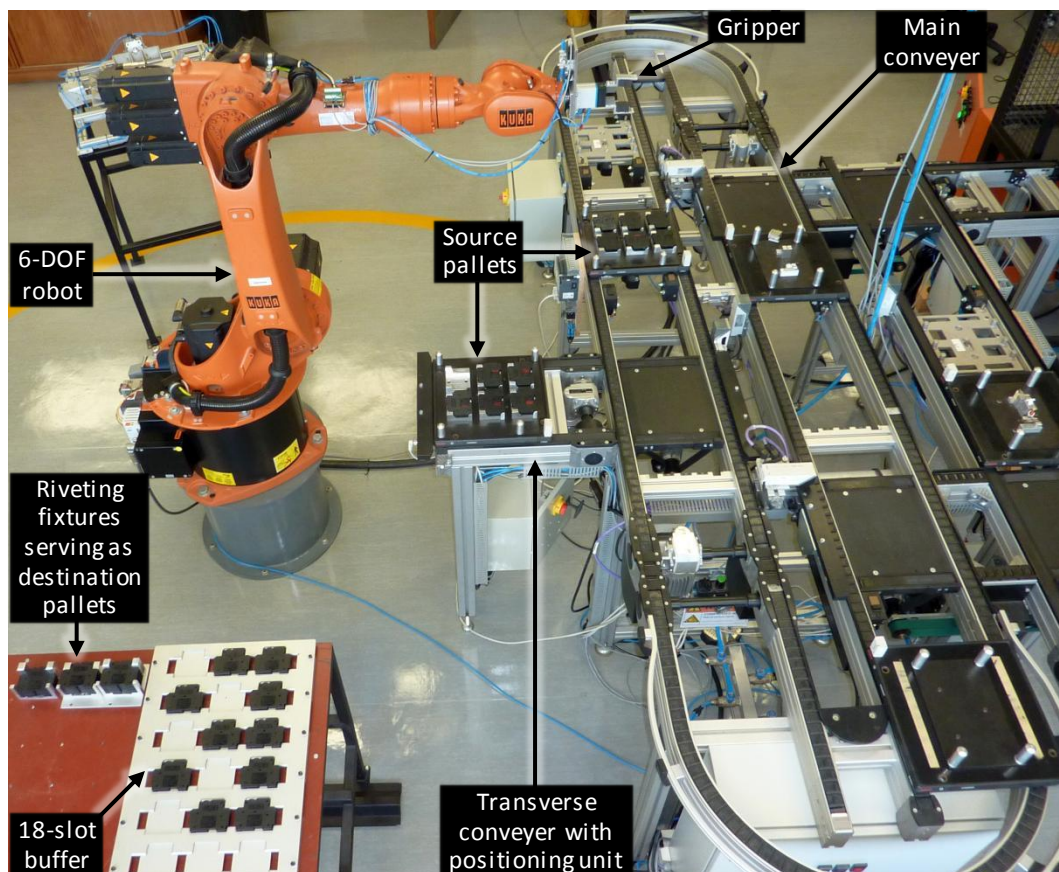


Figure 26 Stacking and buffering station experimental setup

Only one conveyer, with one transverse conveyer was used. A second transverse conveyer was imitated by a positioning unit on the main conveyer which allowed precise and repeatable positioning of the second source pallet from which poles could also be picked up. Instead of a second conveyer carrying the destination pallets with riveting fixtures, those fixtures were directly mounted onto a table to play the role of the destination pallets.

Table 2 lists the components that were used for the experiment along with their model number and the suppliers, where applicable. The circuit breakers used for this case study were supplied by CBI.

*Table 2 Components used for experiment*

<b>Component</b>	<b>Supplier</b>	<b>Model number</b>
<b>Conveyer</b>	Bosch Rexroth	TS 2plus
<b>Transverse conveyer</b>	Bosch Rexroth	EQ 2/TE
<b>Positioning unit</b>	Bosch Rexroth	PE 2-320/320
<b>Pallets</b>	Bosch Rexroth	320x320
<b>Fixtures</b>	Custom-made	n/a
<b>Grippers</b>	Custom-made	n/a
<b>Buffer</b>	Custom-made	n/a
<b>6-DOF robot</b>	KUKA	KR 16-2
<b>Air control valve</b>	Festo	CPE10-M1BH-5L-MS
<b>Parallel gripper</b>	Festo	DHPS-20-A
<b>Gripper proximity sensor</b>	Festo	SMT-8G-PS-24V-E- 2,5Q-OE Prox. sensor
<b>Data acquisition device</b>	National Instruments	NI USB-6525
<b>USB-to-serial interface</b>	MOXA	Uport 1410
<b>Development software</b>	Microsoft .NET	Visual Studio 2012 for .NET with C# add-on

## 6.2 Aspects to measure for reconfiguration

The following aspects were considered/measured:

- Time required for calibrating hardware.
- Time for first time configuration.
- Levels of expertise required.
- Time and effort required for reverting to a previously taught configuration.

- Ramp-up time (tests that are to be run to confirm system works fault-free after reconfiguration).
- Throughput rate, and impact on throughput rate.
- Which software and hardware components need to change for reconfiguration.
- Which parts of the system must be tested during ramp-up before one can gain certainty that the system runs without complications.
- Impact that reconfiguration has on ongoing production.

## 6.3 Calibrations

Certain types of reconfigurations would require (re-)calibration of robot workspace coordinates (or workspaces for brevity), tools or sensors. Sensor calibration could be done without requiring a restart of the controller, as long as the controller is paused. On the other hand, tool and workspace calibrations require that the controller is completely offline and must be restarted afterwards.

For reconfigurations, when a previously defined tool or workspace is to be reused, the station controller would also need to be restarted although no calibrations would be required. When reverting to previously taught configurations, it would take an unskilled operator less than 30 seconds to select the tool or workspace settings from a dropdown list on the GUI (Figure 30 in Appendix A).

All of these calibration procedures can quickly be taught to an operator. No programming skills are required but for tool and workspace calibrations the operator should have basic knowledge of the KUKA system.

First-time calibration of tools, workspaces and the sensor are discussed next.

### 6.3.1 Tool calibration

In the context of the KUKA controller, a tool is anything that can be attached to the robot's end effector, for example grippers or machining tools. The KUKA controller can store tool data for 16 different tools. Each set of tool data contains a description of the tool and a coordinate. The latter is defined as the vector from the centre point of the robot's flange to some reference point on the tool. The reference point on the tool is known as the tool centre point (TCP). After a tool has been defined and the robot is told to move to a certain coordinate, it will be the TCP that is brought to the specified coordinate, and any rotation will happen about this TCP.

Since grippers are more expensive than gripper fingers, one would opt for exchanging only the gripper fingers rather than the gripper when a new type of circuit breaker is to be handled. For this discussion, 'tool' therefore refers to the exchangeable gripper fingers, which are attached to the gripper at the robot's end effector.

As mentioned, calibration data for 16 different tools can be stored on-board the KUKA controller. Therefore, 16 different tools can be interchanged at any time without requiring calibration, as long as they have been calibrated before and can be attached to the robot's end effector in a repeatable way (e.g. using some form of alignment such as dowel pins).

When new tools are introduced, they will first need to be calibrated using the 4-point XYZ approach described in Appendix E. Calibrating a tool and storing a copy of the tool data on-board the station controller takes 11 minutes. This would require a technician who is familiar with the KUKA, but the technician is not required to have programming skills. Physically exchanging gripper fingers takes 5 minutes.

### **6.3.2 Sensor calibration**

After a product changeover, the parts that are to be picked up may have a different width, therefore the gripper fingers may have to be exchanged and the proximity sensor on the gripper may have to be readjusted. Appendix E gives more details. The sensor is digital, meaning that it can only produce two distinct signals: true or false. The sensor should be positioned such that a true signal is produced when the pole is properly gripped and a false signal otherwise.

The sensor calibration procedure is described in Appendix E.1, and takes about 4 minutes to complete. After one pole was picked up and the correct signal sensed, then all other poles of the same range will produce the same signals since they are identical. Furthermore, the gripper fingers are equipped with dowel pins, so that reattaching them will get them into the same location they have been in before. No ramp-up tests are therefore required after the sensor has been recalibrated.

### **6.3.3 Workspace calibration**

A workspace is defined by means of a plane with a certain orientation lying in a 3D space within reach of the robot. It defines where hardware, which the robot needs to interact with, is located and oriented relative to the robot. For the pole stacking robot, workspaces thus refer to the location of the buffer and the positions on the transverse conveyers where pallets come to a stop.

Workspaces are calibrated using the 3-point approach described in Appendix D.3. It takes about 15 minutes to calibrate a workspace and transfer the calibration data to the station controller.

## **6.4 Reconfiguration tests and measurements**

### **6.4.1 Testing robustness of network communication**

To be able to test the stacking station in its entirety, a basic cell controller was developed which could perform tasks such as placing orders and providing the station controller with product and pallet information. Appendix A shows a screenshot of the cell controller's user interface. The cell controller was run on a separate computer so that network communication could be tested. For this purpose, the same communicator holon (discussed in section 5.5.7) that was developed for the station controller was effortlessly reused for the cell controller. Only network settings such as the IP address, port number and a recipient's name had to be specified.

To put the communicator's abilities to the test, the stacking station was given several tasks to complete along with the product information required to complete those tasks. The cell controller was then intentionally disconnected to prevent messages from the

station controller from getting through. In this disconnected state, the station controller continued to execute all tasks for which the required parts were available and tried to inform the cell controller of its progress. After a while the connection was re-established and shortly thereafter all the messages which the station controller had tried to send came through without any message getting lost. This showed that the holons can rely on asynchronous communication over the network.

#### 6.4.2 Adding a similar buffer

Another buffer was added to see how the system performed in terms of scalability. Due to limited resources, a second buffer was not physically built. Instead, the existing 18-slot buffer was used to serve as two separate identical buffers, of which each consisted of 9 slots as shown in Figure 27.

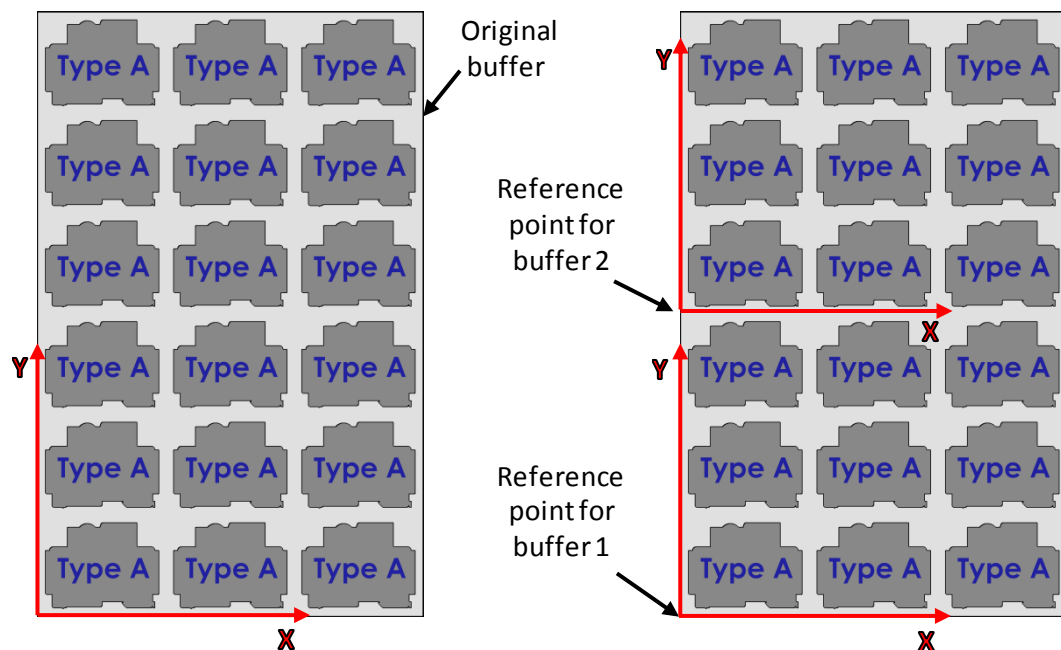


Figure 27 Buffer added to demonstrate scalability

The second buffer was internally represented as a pole storage holon which required a unique workspace number, which the buffer manager assigned to it. The coordinates of the slots relative to the chosen reference point (see Figure 27) had to be obtained from the buffer's CAD file which took about 10 minutes.

The additional buffer formed an extra workspace that the robot had to interact with and therefore had to be calibrated. As was already described in section 6.3.3, this procedure takes about 15 minutes for which the controller must go offline.

As a ramp-up test, a few poles should be placed into and picked up out of various slots of the added buffer. This test would ensure that the correct reference point was used when the coordinates have been obtained from the CAD file and that the workspace has been calibrated accurately. The ramp-up test together with the foregoing calibration procedure will cause production to be interrupted for about 40 minutes.

### 6.4.3 Customization test

Customization refers to a product changeover where the new product is part of the same product family that has been catered for previously and therefore requires no additional functionality. In the context of the case study, this means that only stacking and buffering capabilities were required by the new product. However, different parameters had to be used for some of the hardware.

When a different circuit breaker model has to be manufactured, then the individual parts it consists of would be different to that of other models. Likewise, different knowledge is required for the assembly process. The fixtures capable of holding the different parts would also be different and would therefore be on other pallets.

Figure 28 shows a pallet with products that were already known (type A) and a pallet with a new type of product (type B) that had to be introduced to the station controller. The new poles that have been used were physically the same poles as those used before but were represented as a different type of pole by rotating the pallet on which they arrived by 90°. This rotation made it appear as if the pallet itself was different too. Further, for type B poles, the fixtures had different coordinates relative to the pallet's reference point and type B poles had different pickup coordinates relative to those fixtures.

Since poles of type B have different pickup coordinates (indicated by the turquoise markers in Figure 28), new *partInfo* product holons were needed to store their pickup coordinates. New *assemblyRecipe* product holons were required to hold information for assemblies that make use of type B poles.

Pallets carrying type B poles have their fixtures arranged differently relative to the reference point (bottom left corner) of the pallet. New *pallet* holons for the source and destination pallets were therefore required with different sets of fixture coordinates.

Lastly, new stacking task holons were needed which, when an order for type B assemblies has been placed, would ensure that the assemblies were being produced according to the plan in the *assemblyRecipe* holon.

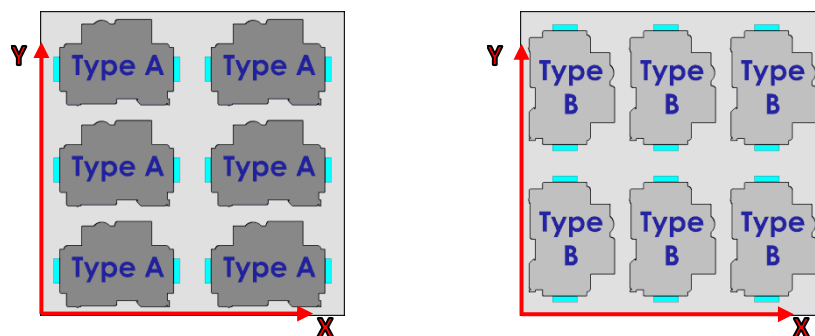


Figure 28 Pallet turned 90° to represent new type of product

The new information regarding part info, assembly instructions and the pallets' fixture coordinates was delivered by the cell controller in XML format, and the stacking station controller dynamically instantiated the corresponding *partInfo* holons, *assemblyRecipe* holons and *pallet* holons. Whenever a pallet arrived with type B holons, then pole holons have been instantiated to represent those poles. As soon as an order had been placed for type B assemblies, a stacking task holon was created to drive the production for type B assemblies.

Since type B assemblies required no additional functionality, no changes to the station controller software were required to accommodate the new product. For this reason, and because all of the abovementioned holons were instantiated dynamically, the station controller did not need to be restarted. The information for type B poles nevertheless had to be fed into the system at the cell controller from where it was distributed to other stations requiring that information (i.e. the stacking station). For this purpose, a GUI for the cell controller (Appendix A) was developed, allowing the operator to specify the *assemblyRecipe*, and place orders for the new product and send this information over Ethernet to the station controller.

Since type B and type A poles were transported on pallets which arrived on the same transverse conveyers, it was not necessary to define and calibrate new workspaces. However, the fixtures on those pallets were arranged differently and therefore the coordinates for those fixtures first had to be obtained from a CAD file so that the cell controller could send the pallet information to stations which needed to interact with those pallets (i.e. the stacking station).

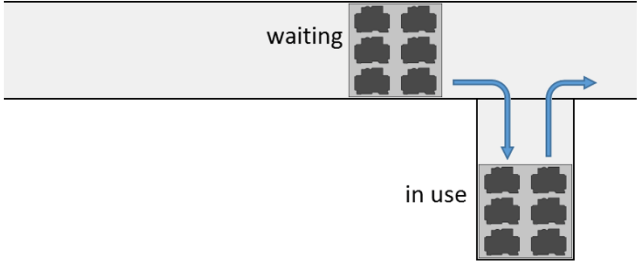

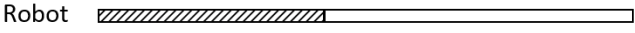


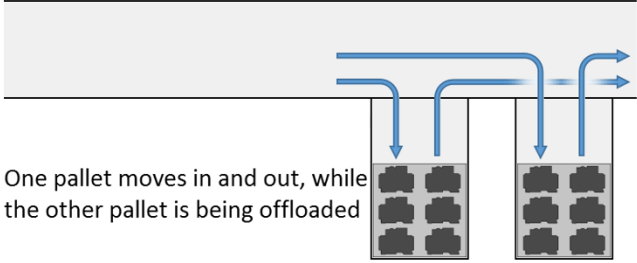


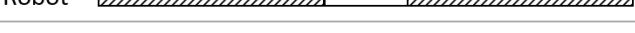



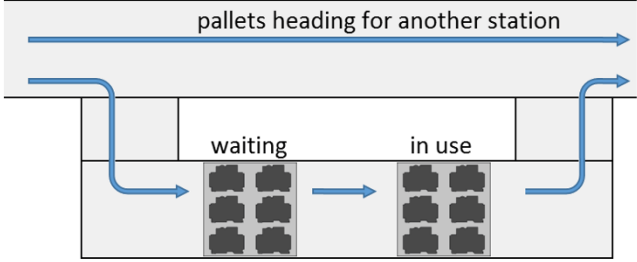

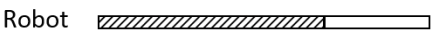


Once the cell controller had all the part info, assembly instructions and pallet info, the system could seamlessly switch between type A and type B products without requiring further human intervention since the station controller was able to handle customization changes dynamically. If type B poles would have had a different pole-width, then the sensor on the gripper would need to be re-calibrated manually, which would take 4 minutes, but would require no special skills.





#### 6.4.4 Throughput rate tests

To determine whether the system achieved the desired throughput rate of 1 pole per second, and to determine which conveyor configuration yielded the highest throughput rates, the throughput rates for three different layouts were measured. For these tests, it was assumed that the destination conveyor would never form the bottleneck, but that the throughput rate could only be impacted by the source conveyor, which is the conveyor from which the robot had to pick up poles. Therefore, only the source conveyor was configured for different layouts to determine the effect on the throughput rate.

The three layouts that were considered for the source conveyor are shown in Table 3: using one transverse conveyor; using two transverse conveyers; and using a parallel conveyor. For each of those three layouts, two different cases were tested, resulting in a total of six different scenarios. For the first case (reported in black), no poles had failed at the electrical test station and the buffer did not need to be used but poles could be transferred directly from the source pallet to the destination pallet. For the second case

Table 3 Conveyer configurations yielding different throughput rates

Configuration	Timing diagram	Robot utilization	Throughput time per pole
a) One transverse conveyer 	6 poles handled every 23.4 s Pallet 1  Robot 	42.2%	3.90 s
	6 poles handled every 26.4 s Pallet 1  Robot 	48.7%	4.40 s
b) Two transverse conveyers  <p>One pallet moves in and out, while the other pallet is being offloaded</p>	12 poles handled every 23.4 s Pallet 1  Pallet 2  Robot 	84.4%	1.95 s
	12 poles handled every 26.4 s Pallet 1  Pallet 2  Robot 	97.4%	2.20 s
c) Parallel conveyer  <p>pallets heading for another station</p>	6 poles handled every 14.5 s Pallet  Robot 	68.3%	2.42 s
	6 poles handled every 17.4 s Pallet  Robot 	73.6%	2.91 s

 Pallet stationary    
  Robot active when at least one pallet is stationary    
 Black – all poles are transferred directly from source pallet to destination pallet  
 Pallet moving    
  Robot idle when all pallets are moving    
 Red – 33% of poles are transferred to buffer and obtained from buffer



(reported in red), 33% of the poles had to be transferred to the buffer and retrieved from the buffer, while 67% were transferred directly to the destination pallet.

The main conveyer in the laboratory only had one transverse conveyer but a positioning unit located on the main conveyer was used to imitate a second transverse conveyer. The parallel conveyer configuration could not be built with the available hardware and the associated throughput rate could therefore not be measured directly. However, the existing hardware configuration was used to measure the duration of various pallet movements and these measured values were used to calculate the throughput rates of the other configurations and to construct the timing diagrams shown in Table 3.

The robot can only pick up poles from pallets which are stationary and aligned on a positioning unit. When all six poles can be transferred directly from the source to the destination pallet, it takes 9.89 seconds to unload a pallet. When only four of the six poles can be transferred directly to the destination pallet, while the remaining two poles have to be transferred to the buffer, and two other poles have to be retrieved from the buffer, then it takes 12.84 seconds for the robot to unload a pallet. In contrast, it takes 13.4 seconds for a pallet to move out of a transverse conveyer and a next pallet to move in and become aligned. Since the overall pallet motion takes longer than the robot motion, the robot is idle more than 50% of the time while waiting for the next pallet to move in (when only one transverse conveyer is used). It is therefore advisable to use a second transverse conveyer so that there can always be at least one source pallet from which poles can be picked up from.

The results show that when two transverse conveyers are used, the robot has to wait the least amount of time, which results in the fastest throughput rates of 1.95 seconds per pole. This is about half the speed that CBI requires. By letting another identical stacking and buffering station work in parallel, the combined throughput rate would be faster than the desired rate of one pole per second. The KUKA KR16 robot that was used for these experiments is a big robot and consequently relatively slow. Smaller variants, or SCARA robots, would be able to transfer poles much quicker. To exploit the benefits of a faster robot, more than two transverse conveyers should be used to ensure that a pallet would always be ready for the robot.

#### **6.4.5 Scalability test – adding another transverse conveyer**

For improved throughput rates, the robot should always be able to do work (i.e. handle poles) and never have to be idle while waiting for a pallet. For this reason, there should be more than one source pallet within reach from the robot, so that the robot can pick up poles from the first pallet while the second pallet simultaneously is moving in or out, and vice versa. The laboratory setup only contained one transverse conveyer with a positioning unit, but the main conveyer system was also equipped with a positioning unit within the robot's reach, which was used to represent the additional transverse conveyer.

Changes that had to be made to the software to accommodate the additional transverse conveyer were minimal, since the controller was programmed to handle this type of scalability from the ground up. It was only necessary to change the value of a variable

that represented the number of stopping positions for source pallets, and to calibrate the workspace for the additional transverse conveyer on the KUKA. The newly calibrated workspace coordinates were then transferred to the station controller by using the GUI shown in Appendix A. The controller had to be restarted.

#### 6.4.6 Alterations to operational holon

For the initial implementation of the robot holon, the path that the end effector had to follow was specified by a series of coordinates, which the robot had to accurately move to. It was observed that the robot did not maintain full speed along the entire path from the pickup to the place position, but slowed down significantly whenever it came close to any of the specified coordinates describing its path (blue line in Figure 29). For optimal throughput rates, the robot should be able to transfer poles as quickly as possible. Therefore, the robot holon was re-programmed such that the robot would accurately visit only the first and last coordinate of the path, while the other coordinates along the path would be visited with some deviations, thereby allowing it to maintain higher speeds (red line in Figure 29).

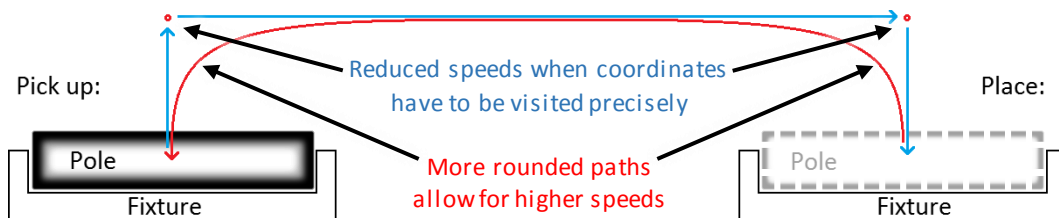


Figure 29 Effects of path contour on robot speed

The KUKA controller has built-in capabilities for movements along rounded paths and movements to exact coordinates. To change the way the robot had to move, both halves of the robot holon had to be changed. The part residing on the C# station controller had to be able to send two different commands: one for sending exact coordinates to specify the starting and end points, and another command for sending approximate coordinates. The part of the robot holon which resides on the KUKA controller, had to be able to receive the additional commands via RS232 and invoke the corresponding method on the KUKA controller. None of the other holons required any changes.

This experiment showed that the rest of the station controller is completely unaffected by the change in the operational holon, and that holons can therefore be easily altered independently from one another, which enhances modularity. For example, the KUKA robot could have been replaced with a SCARA robot without effecting other parts of the system.

#### 6.4.7 Disturbance handling tests

The stacking station controller was programmed to inherently handle disturbances arising from interrupted supplies of poles due to failures at the electrical test station. Whenever poles arrive on pallets which cannot (yet) be used to build a stack, they are transferred to the buffer. The following scenarios could all result in poles being transferred to the buffer:

- When the pole is useful for a placed order, but the other poles which are needed to form a complete stack have not arrived at the station yet. The “early” pole must wait for matching poles but the source pallet must be emptied to give way for the next pallet.
- When none of the placed orders specifies the usage for those poles.
- When all the poles which together form a complete stack have already arrived at the station but the recipe which specifies how they must be stacked is not yet available. The request for that recipe would have been sent to the cell controller but the response to that request might be delayed because of a temporary network fault or the cell controller being over utilized.
- When poles have arrived on the source pallet but no destination pallet is at the station, or the fixtures on the destination pallet do not have the height required for the stack.

Whenever a pole is transferred to the buffer so that the buffer becomes full, an appropriate message is sent to the cell controller.

Since the functioning of the station relied on Ethernet communication, the robustness of the communication protocol had to be tested. The cell controller and stacking station were both run, and the cell controller gave some orders to the stacking station. The cell controller then was terminated. None of the messages that the stacking station had to send were lost but were kept in the outbox until the cell controller came back online. It could therefore be concluded that the station controller could handle disruptions on the network without requiring human intervention or a restart.

#### **6.4.8 Ramp-up tests**

To test whether workspaces were calibrated accurately or not, poles have to be picked up out of a pallet fixture, and brought to a known position on the buffer to see if the poles fit without interference. This test should be done for a couple of positions (say the three or four corner positions, which are farthest apart). Once all four of those positions have been confirmed to be placeable without interference, then it can be deduced that the pick-up positions in the middle of the pallet would also be accurate.

To ensure the robot can reach all pick-up and place positions accurately without collisions occurring, the system should be run in Ramp-up-test mode. In this mode, the critical coordinates of the pole-storage devices (such as buffers and pallets) are tested for whether or not the robot can successfully remove a pole and put it back into that position.

### **6.5 Results**

The series of experiments have tested various aspects of the station controller. It was shown that when hardware changes were required (i.e. for convertibility and scalability tests) the controller must go offline and the technician must have KUKA and C# skills. After such changes, ramp-up tests are necessary to ensure proper working before resuming full-scale production. The system will be offline for at least an hour to perform

the controller changes and hardware calibrations, and ramp-up tests can take another 10 minutes.

Customization tests have shown that the system need not be restarted when a new product is being introduced which is part of the same product family that has been catered for previously and requires the same hardware functionality. The controller could handle such customizations dynamically without human interaction or a restart.

Convertibility experiments, when reverting to a previously taught configuration, can happen by a few mouse clicks, and need not take long if the required hardware is still in place and does not need to be re-calibrated. Nevertheless, the controller needs to be restarted even if the configuration has been used previously.

## **6.6 Recommendation for shorter reconfiguration times**

For the hardware setup that was built for the experiments, the gripper on the robot was equipped with only one digital sensor which could sense with certainty only the presence of products of a certain width. If another product with a different width has to be catered for, then the sensor needs to be readjusted, recalibrated, and thereafter undergo a couple of ramp-up tests to confirm its proper functioning. To avoid this unnecessary recalibration of the proximity sensor after each product changeover, two approaches could be taken. One approach would be to install an array of proximity sensors on the gripper which could sense the width of a larger number of products. Another approach would be to use a single analogue sensor capable of reporting the exact distances by which the jaws are apart. Using one of these approaches can obviate the need for manual sensor calibration and allow the system to seamlessly change over to different products because the control software could use the positional data to determine whether or not the poles have been picked up correctly. Furthermore, the chance of human error can be reduced and consistent gripper behaviour can be ensured.

## **6.7 C# evaluation**

Using C#, being a high-level OOP language, made it an easy and intuitive task to develop the control software, and make alterations to it at a later stage. Automatic garbage collection took a major load off the programmer and ensured that memory leakage would not occur as easily. None of the hardware that had to be controlled was causing interfacing difficulties. DLL files supplied by automation vendors could be wrapped inside a class and then reused easily. .NET libraries with built-in support for the more rudimentary functionalities such as Ethernet communication, thread-safe linked lists, XML handling and user forms shortens development time and allows the programmer to focus more on the application-specific tasks.

Some of the benefits of C# can be attributed to the fact that C# is an OOP language, and therefore also apply to any other OOP languages such as Java, C++, etc. As discussed in earlier sections, OOP characteristics allowed for the intuitive implementation of holons. Also, during reconfigurations, it was intuitive where changes to the software had to be made, and software had to be changed only locally. Diagnostic functionalities built into

the basic generic holon have helped drastically during the development phase already, and also to point out hardware errors in normal operation mode.

Encapsulation allowed hardware-specific drivers to be wrapped inside operational holons, thereby hiding the implementation detail and providing a standard interface to the surrounding holons. This allowed for changes to be made to operational holons, without affecting any of the other holons.

Having parts of the system run synchronously, while other parts communicate asynchronously, enhanced the system's robustness. The station controller could continue operating without requiring human intervention even when the connection to the cell controller has been lost, or when the cell controller would be restarted.

Dynamic memory allocation allowed for the system to continue running while performing reconfigurations for which no hardware calibration or changes to the control software was required. For customization, where for new products only existing functionality is required, not having to reprogram and restart the controller means that ongoing production is not impacted at all, and that no human intervention was required since the station controller was able to handle all the changes dynamically.

## 7 Conclusions and Recommendations

The Agent Based Control (ABC) approach is the de facto standard for controllers for Reconfigurable Manufacturing Systems. However, due to industry's reluctance to adopt ABC, an object-oriented programming (OOP) approach using C# was considered in this thesis as an alternative. OOP is more widely used and has many capabilities that are valuable when implementing an RMS.

The research has shown that ABC's advantages can be decisive in complex, highly dynamic systems requiring autonomous reconfiguration. However, in simpler systems and systems where timing and sequencing is important, OOP will have significant advantages. For CBI, the industry partner of this research, the advantages of OOP exceed that of ABC, primarily since autonomous reconfiguration and emergent behaviour are not high priorities in their situation, while OOP provides better integrability with hardware. The learning curve for C# is not as steep as for ABC.

OOP concepts such as inheritance and polymorphism made it possible to efficiently reuse code and to compartmentalize code, thereby enhancing modularity. When a physical device is to be exchanged with another of similar capabilities, and the same interfaces is enforced on the new holon, then the new holon can seamlessly be integrated into the rest of the code, and no other holons need to be reprogrammed at all. All of the hardware that was used for the case study could easily be integrated since C# has built-in support for hardware interfacing and serial communication, unlike Java. Java could have been used for the core of the control software, but additional interfaces would then need to be written in another language (such as C#) whereas when C# was used from the beginning, then no other languages for interfacing would be required.

Experiments have shown that reconfigurations such as customization can be done with no human effort at all as long as the physical hardware can cater for all the needs of the added products. Using C#, which is an OOP language, the system can easily be expanded, since inheritance provides for code re-use. Further, interfaces allow for integrability.

Object-oriented concepts make OOP languages well suited for implementing holonic architectures. Dynamic memory allocation makes it possible to cater for any amount of orders, resources or products without requiring changes to the code to be made. Since C# is a strongly typed programming language and very strict, the chances of programming errors during development is very scarce. Multi-threading capabilities allow hardware-critical processes to run in (near) real-time, while at the same time allow asynchronous communication over the network via TCP/IP.

It was found that when trying to debug code while running multiple threads, it becomes difficult to see which thread is running in which part of the code.

The pre-defined objects designed for graphical-user interfaces allow the developer to effortlessly produce an HMI in a short amount of time. The reconfiguration tests have shown that all six core characteristics of RMSs can be addressed using an OOP-based controller.

The system could successfully handle orders for single, double, triple- or four-poles and could use the buffer effectively when certain poles have failed the electrical test to pair matching poles and assemble them to complete the desired order. Furthermore, the controller was able to provide some diagnostic tools and had the ability to trace poles, and this information can be used for further diagnostics.

Dynamic creation of holons made it possible to add as many holons as needed at runtime. Having no fixed memory limitations makes the system more flexible for reconfigurations.

For future research, diagnostic holons could be developed, which run the entire time and use the data of traced poles to determine where in the system possible faults could lie. On another thread, an optimizer can be run to prevent collisions between robots in case more than one robot is to be used within the same station. To reduce ramp-up time, an automatic calibration device could be developed, such as an eye-in-hand camera, to let the robot autonomously calibrate the workspace after equipment was moved around on the factory floor. This would make it unnecessary for the operator to require KUKA skills.

## 8 References

- Anon., n.d. *CBI Circuit Breakers - Product Listing*. [Online]  
Available at: <http://www.cbi-lowvoltage.co.za/productlist>  
[Accessed 12 September 2015].
- Anon., n.d. *Technical Downloads*. [Online]  
Available at: [http://www.cbi-lowvoltage.co.za/sites/default/files/downloads/CBI\\_QA\(13\) - Series Circuit Breakers Data Sheet.pdf](http://www.cbi-lowvoltage.co.za/sites/default/files/downloads/CBI_QA(13) - Series Circuit Breakers Data Sheet.pdf)  
[Accessed 15 September 2015].
- Barata, J. A., 2003. *Coalition Based Approach for Shop Floor Agility – A Multiagent Approach*, Lisboa: Universidade Nova de Lisboa.
- Bellifemine, F., Caire, G. & Greenwood, D., 2007. *Developing multi-agent systems with JADE*. s.l.:Wiley.
- Bongaerts, L., Monostori, L., McFarlane, D. & Kádár, B., 2000. Hierarchy in distributed shop floor control. *Computers in Industry*, 43(2), pp. 123-137.
- Booch, B., 2000. *private communication* [Interview] 2000.
- Brennan, R. W., 2007. Toward Real-Time Distributed Intelligent Control: A Survey of Research Themes and Applications. *IEEE Transactions on Systems, Man and Cybernetics*, 37(5), pp. 744-765.
- Cândido, G. & Barata, J., 2007. *A Multiagent Control System for Shop Floor Assembly*. Berlin/Heidelberg, Springer-Verlag, pp. 293-302.
- Cassandras, C. G., 1986. Autonomous Material Handling in Computer-Integrated Manufacturing. *Modelling and Design of Flexible Manufacturing Systems*, pp. 81-98.
- Cass, S., 2015. *The 2015 Top Ten Programming Languages*. [Online]  
Available at: <http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages>  
[Accessed 18 September 2015].
- Christensen, J. H., 1994. *Holonic Manufacturing Systems: Initial Architecture and Standards Directions*. Hannover, s.n.
- Christensen, J. H., 2011. *IEC 61499 - A Standard for Software Reuse in Embedded, Distributed Control Systems*. [Online]  
Available at: <http://www.holobloc.com/papers/iec61499/overview.htm>  
[Accessed 19 September 2015].



- Dilts, D. M., Boyd, N. P. & Whorms, H. H., 1991. The Evolution of Control Architectures for Automated Manufacturing Systems. *Journal of Manufacturing Systems*, 10(1), pp. 79-93.
- Duffie, N. A. & Piper, R. S., 1987. Non-hierarchical control of a flexible manufacturing cell. *Robotics and Computer-Integrated Manufacturing*, 3(2), pp. 175-179.
- Giret, A. & Botti, V., 2004. Holons and agents. *Journal of Intelligent Manufacturing*, 15(5), pp. 645-659.
- Giret, A. & Botti, V., 2005. Analysis and Design of Holonic Manufacturing Systems. *18th International Conference on Production Research*.
- Hall, K. H., Staron, R. J. & Vrba, P., 2005. *Holonic and Agent-Based Control Systems*. Berlin/Heidelberg, Springer-Verlag, pp. 1571-1580.
- Hoffmann, A. J. & Basson, A. H., 2013. *Towards Alternatives for Agent Based Control in Reconfigurable Manufacturing Systems*. Munich, Germany, s.n.
- Hristu-Varakelis, D. & Levine, W. S., 2005. *Handbook of Networked and Embedded Control Systems*. New York: Birkhäuser.
- Jennings, N. R. & Bussmann, S., 2003. Agent-based control systems: Why are they suited to engineering complex systems?. *Control Systems, IEEE*, 23(3), pp. 61-73.
- Koestler, A., 1967. *The Ghost in the Machine*. London: Hutchinson & Co..
- Koren, Y. & Shpitalni, M., 2010. Design of reconfigurable manufacturing systems. *Journal of Manufacturing Systems*, 29(4), pp. 130-141.
- Leitão, P. & Restivo, F., 2006. ADACOR: A holonic architecture for agile and adaptive manufacturing control. *Computers in Industry*, 57(2), pp. 121-130.
- Mehrabi, M. G., Ulsoy, A. G. & Koren, Y., 2000. Reconfigurable manufacturing systems: Key to future manufacturing. *Journal of Intelligent Manufacturing*, Volume 11, pp. 403-419.
- Mulubika, C. & Basson, A. H., 2013. Comparison of IEC 61499 and Agent Based Control for a Reconfigurable Manufacturing Subsystem. *International Conference on Competitive Manufacturing*.
- Obasjano, D., 2004. *Designing Extensible, Versionable XML Formats*. [Online] Available at: <https://msdn.microsoft.com/en-us/library/ms950793.aspx> [Accessed 12 April 2015].
- Odell, J., 2002. Objects and Agents compared. *Journal of Object Technology*, 1(1), pp. 41-53.

- Parunak, H. V. D., 1997. "Go to the Ant": Engineering Principles from Natural Multi-Agent Systems. *Annals of Operations Research*, Volume 75, pp. 69-101.
- Radeck, K., 2004. *C# and Java: Comparing Programming Languages*. [Online] Available at: <https://msdn.microsoft.com/en-us/library/ms836794.aspx> [Accessed 29 September 2015].
- Simpson, J. A., Hocken, R. J. & Albus, J. S., 1982. The Automated Manufacturing Research Facility of the National Bureau of Standards. *Journal of Manufacturing Systems*, 1(1), pp. 17-32.
- Van Brussel, H. et al., 1999. A conceptual framework for holonic manufacturing: Identification of manufacturing holons. *Journal of Manufacturing Systems*, 18(1), pp. 35-52.
- Van Brussel, H. et al., 1998. Reference architecture for holonic manufacturing systems: PROSA.. *Computers in Industry*, 37(3), pp. 255-274.
- Van der Linden, P., 2002. *Just Java 2*. Upper Saddle River(NJ): Prentice Hall.
- Van Leeuwen, E. H. & Norrie, D., 1997. Holons and holarchies. *Intelligent Manufacturing*, 76(2), pp. 86-88.
- Viatkin, V., 2007. *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. North Carolina: ISA.
- Vrba, P. et al., 2011. Rockwell Automation's Holonic and Multiagent Control Systems Compendium. *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, 41(1), pp. 14-30.
- Winkelman, P., 2011. A Theoretical Framework for an Intelligent Design Catalogue. *Engineering with Computers*, 27(2), pp. 183-192.
- Winkelman, P. & Yellowley, I., 2006. A Theoretical Framework for an Intelligent Design Catalogue. *Proceedings of the Canadian Engineering Education Association*, pp. 259-269.
- Wyns, J., Van Brussel, H., Valckenaers, P. & Bongaerts, L., 1996. *Workstation architecture in holonic manufacturing systems*. Johannesburg, s.n., pp. 220-231.
- Zwegers, A., 1998. *A study in shop floor control to determine architecting concepts and principles*, Eindhoven: University Press Facilities.

# Appendix A – Human-Machine Interface screenshots

This section contains screenshots of the graphical user interfaces that were developed for the stacking station and the cell controller.

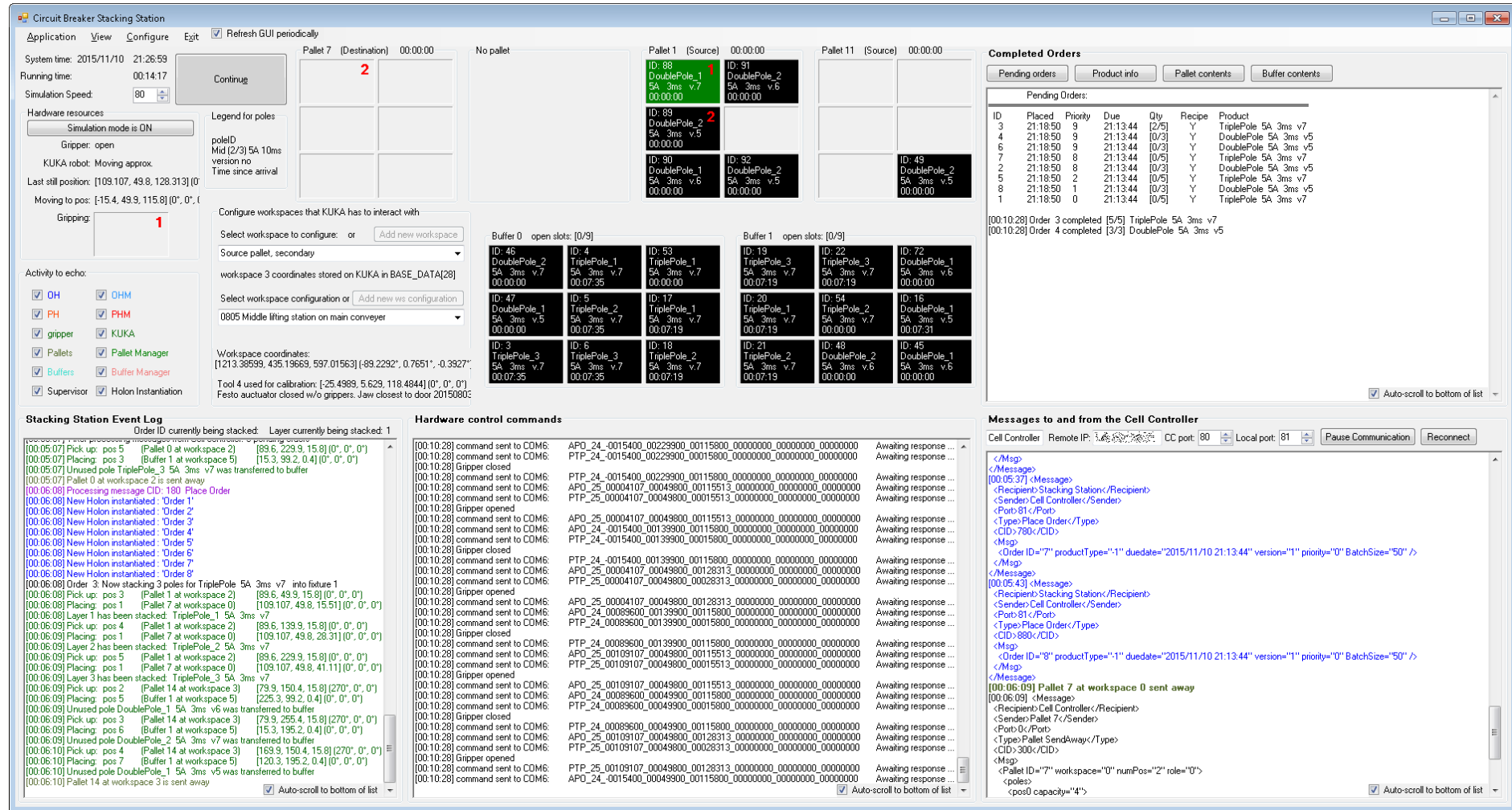


Figure 30 Stacking station human-machine interface

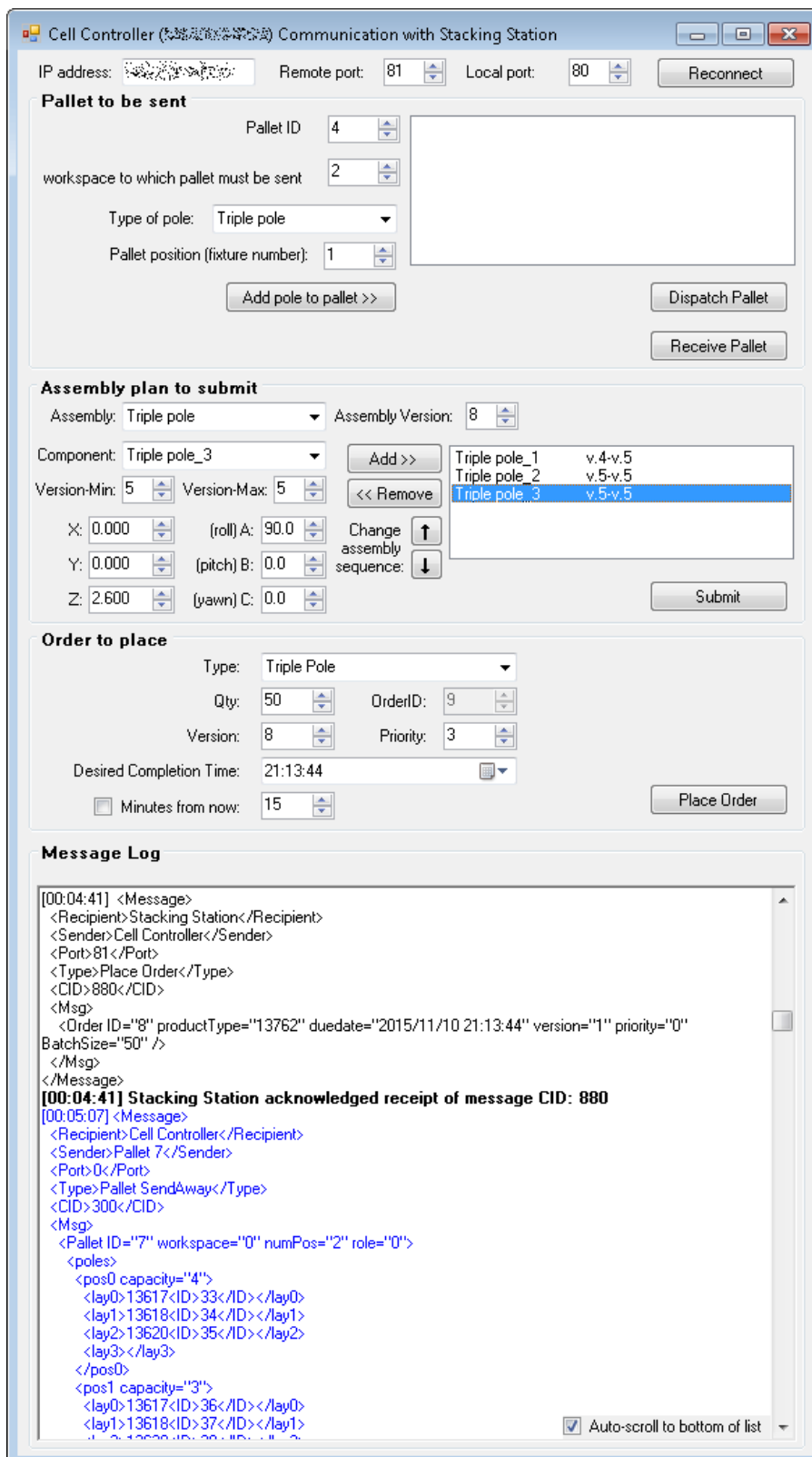


Figure 31 Cell controller human-machine interface

## Appendix B – Flow diagrams

This appendix contains various flow diagrams that have been referred to in chapter 5.

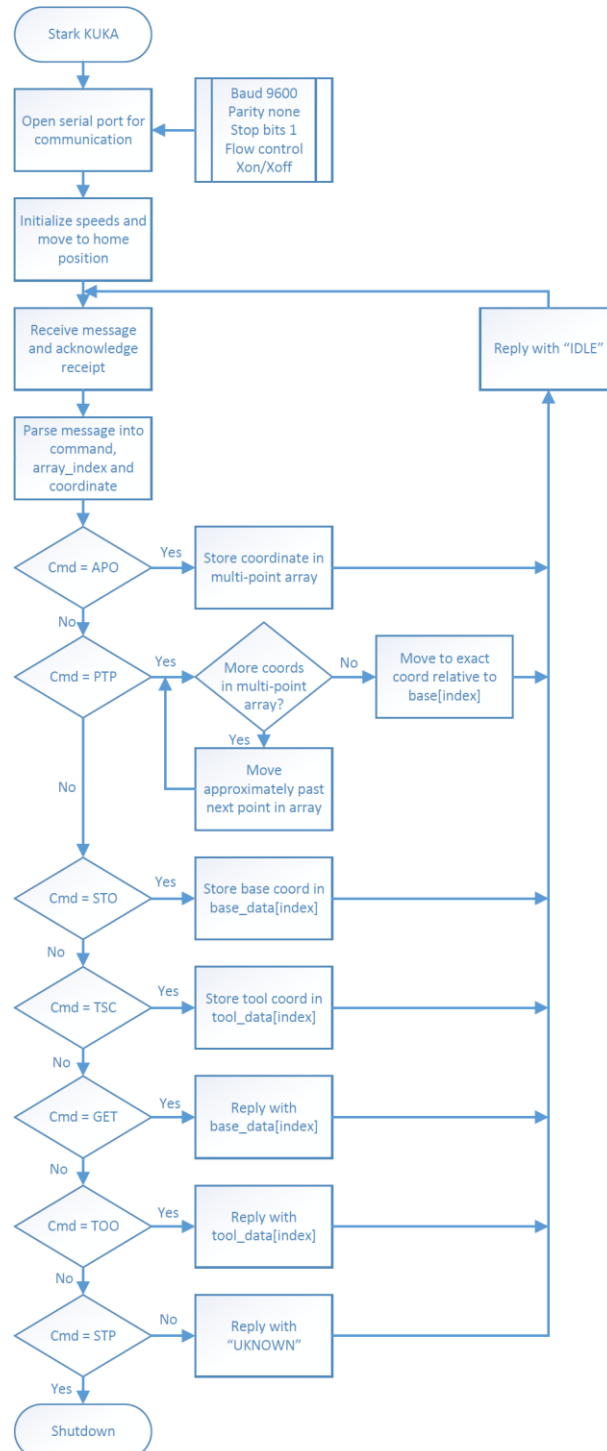


Figure 32 Kuka controller flow diagram

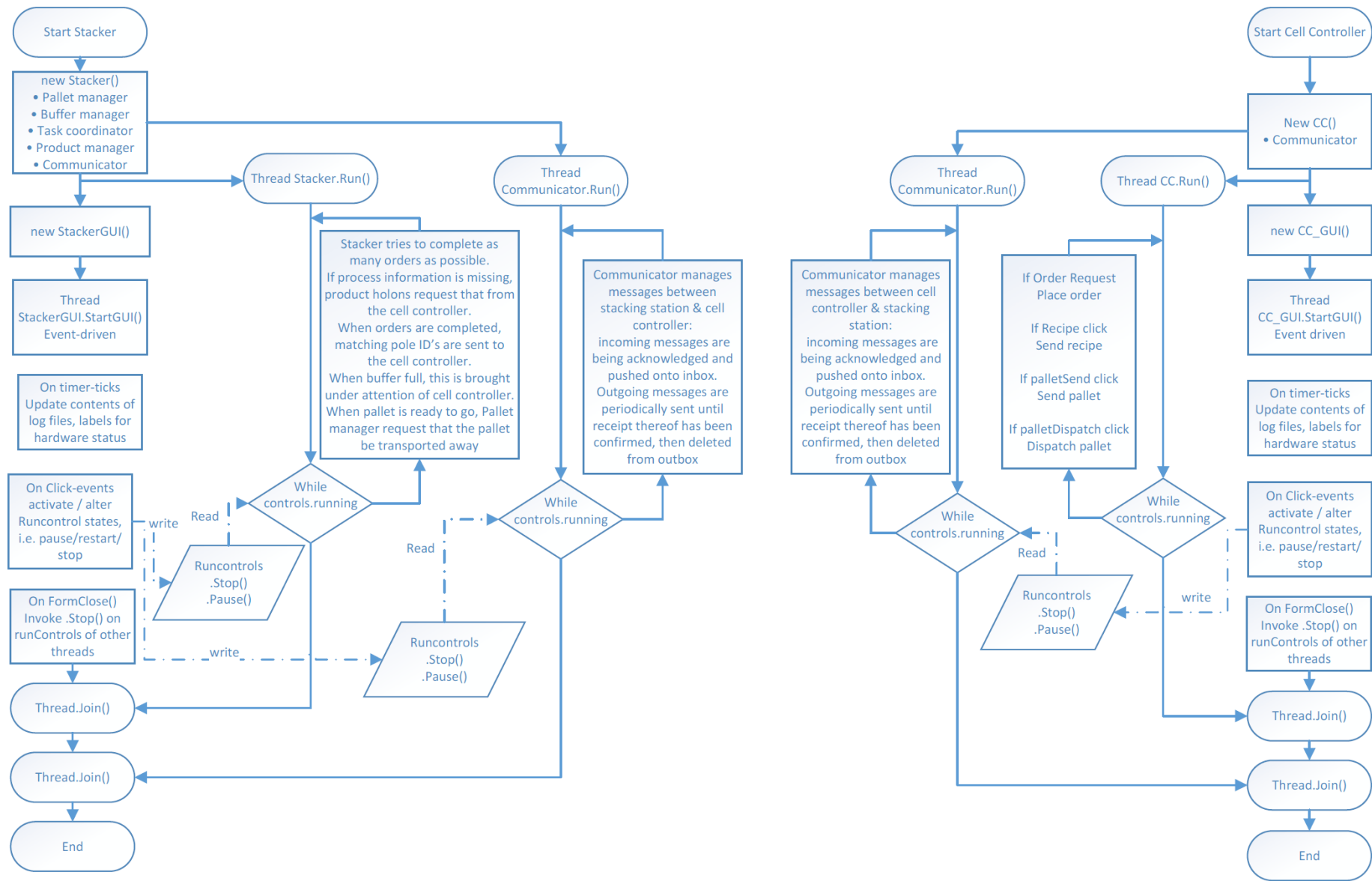


Figure 33 Flow diagram for multi-threading on stacking station and cell controller

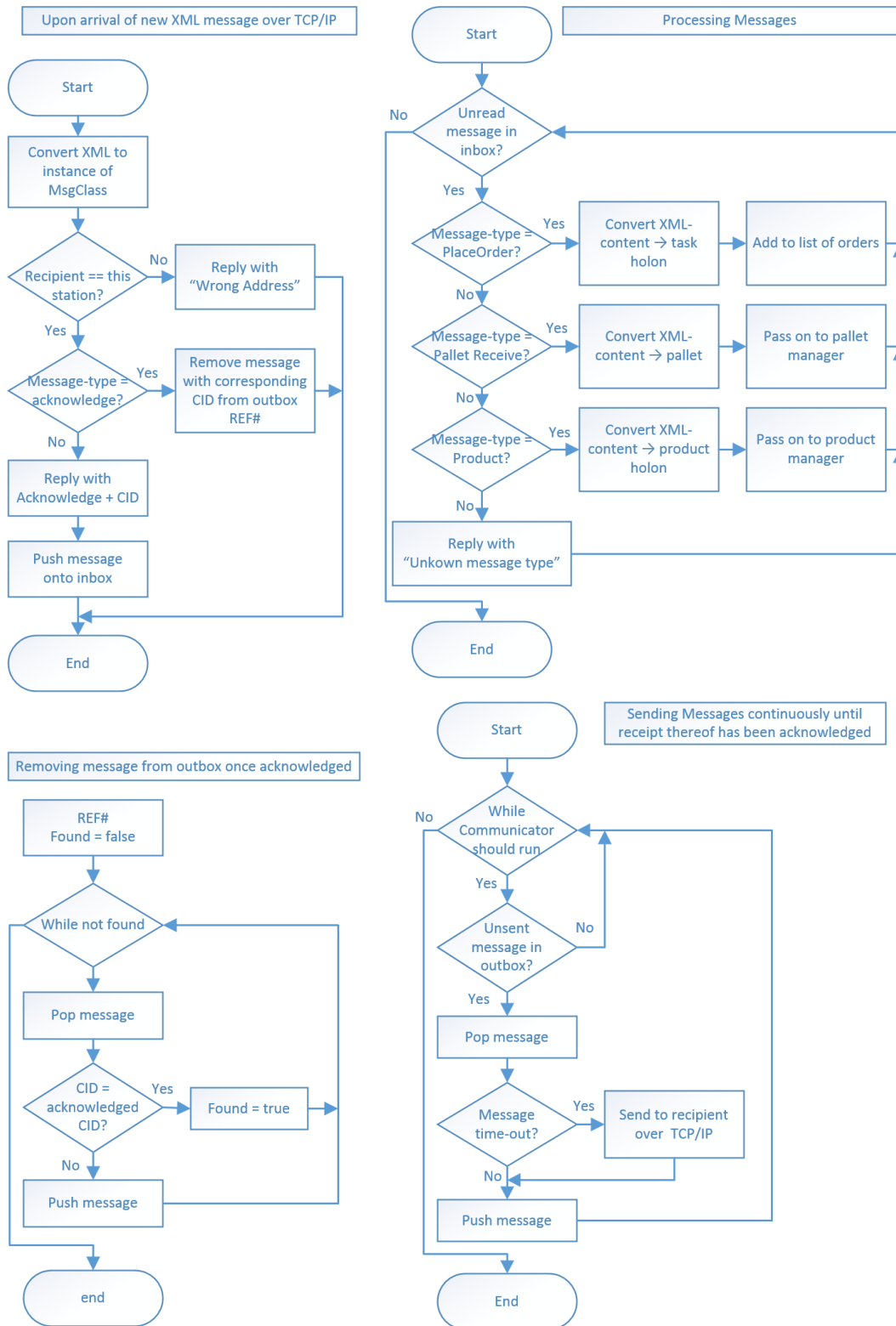


Figure 34 Communicator flow diagrams

## Appendix C – Code for Gripper Holon

To give the reader an idea of how the code was designed, the code for the Festo Gripper Holon is provided below. Since it has been derived from the generic gripper class which in turn implements the IGripper interface, the code for the interface and parent class is also shown.

Table 4 Code for Gripper Holon

```
interface IGripper
{
    void Open();
    void Close();
    bool SensePart();
}

public abstract class GripperGen : OperationalHolon, IGripper //generic
grripper
{
    public GripperGen(string name, int type) : base(name, type) { }
    public abstract void Open();
    public abstract void Close();
    public abstract bool SensePart();

    protected Boolean CurrentlyOpen = false; //Boolean used to store the
state that the gripper is SUPPOSED to be in.
    protected System.Collections.Generic.List<toolConfiguration> tools = new
List<toolConfiguration>(); //Holds coordinate of tool that was used to
calibrate bases
    protected int selectedToolConfig; //Out of the tools list, which entry
is being used
    protected System.Collections.Generic.List<grripperConfiguration> grippers
= new List<grripperConfiguration>(); //Holds coordinate of gripper that is
attached to end effector
    protected int selectedGripper; //Out of the grippers list, which
grripper is currently mounted to the end effector

    public toolConfiguration getSelectedTool()
    {
        return tools[selectedToolConfig];
    }
    public grripperConfiguration getSelectedGripper()
    {
        return grippers[selectedGripper];
    }
}

public class FestoGripper : GripperGen
{
    private DAQin sensor; //senses state of gripper
    private DAQout actuator; //digital out for opening/closing gripper valve
    private Boolean withoutAir; //Gripper holon continues with normal
operation even when valve is not supplied with air

    public FestoGripper() //runs before UserForm is shown
```



```

        : base("Festo DHPS-20-A gripper", HMI.HolonType_RH)
    {
        if (FileHandling.FileExists(this)) //retrieve last used
        configurations from HDD
        {
            string[] Lines = FileHandling.ReadAllLines(this);
            int lineCounter = 0;
            HMI.gripperOutputChannel_SelectedIndex =
            Convert.ToInt32(Lines[lineCounter++]);
            HMI.gripperInputChannel_SelectedIndex =
            Convert.ToInt32(Lines[lineCounter++]);

            int numGrippers = Convert.ToInt32(Lines[lineCounter++]);
            selectedGripper = Convert.ToInt32(Lines[lineCounter++]);
            for (int t = 0; t < numGrippers; t++)
            {
                grippers.Add(new gripperConfiguration(Lines[lineCounter],
            Coord.toCoord(Lines[lineCounter + 1])));
                lineCounter += 2; //2 lines of information per gripper in the
            file
            }

            int numTools = Convert.ToInt32(Lines[lineCounter++]);
            selectedToolConfig = Convert.ToInt32(Lines[lineCounter++]);
            for (int t = 0; t < numTools; t++)
            {
                tools.Add(new toolConfiguration(Lines[lineCounter],
            Convert.ToInt32(Lines[lineCounter + 1]), Coord.toCoord(Lines[lineCounter +
            2])));
                lineCounter += 3; //3 lines of information per tool in the file
            }
            HMI.KUKA_tools = tools; //links tools on HMI form to tools within
            gripper holon
        }
    }

    public void Initialize() //runs as soon as user clicked Start. Writes
    user selected values to configuration file and initializes DAQ I/O's
    {
        string lines = HMI.gripperOutputChannel_SelectedIndex + "\t\t;DAQ:
        Selected Output channel\r\n" +
        HMI.gripperInputChannel_SelectedIndex + "\t\t;DAQ: Selected Input channel";
        lines += "\r\n\r\n" + grippers.Count() + "\t\t; number of grippers" +
        "\r\n" + selectedGripper + "\t\t; currently attached to end effector";
        foreach (gripperConfiguration gripperConfig in grippers)
        {
            lines += "\r\n\r\n" + gripperConfig.description + "\r\n" +
            gripperConfig.offset.ToString();
        }
        lines += "\r\n\r\n\r\n;===== TOOLS =====\r\n" +
        tools.Count() + "\t\t; number of tools" + "\r\n" + selectedToolConfig +
        "\t\t; currently selected";
        foreach (toolConfiguration toolConfig in tools)
        {
            lines += "\r\n\r\n" + toolConfig.description + "\r\n" +
            toolConfig.ToolNumber + "\r\n" + toolConfig.TCP.ToString();
        }
    }

```

```

        FileHandling.Write(lines, this);
        //values for HMI.gripperOutputChannel & HMI.gripperInputChannel are
        obtained from GUI before Start button is pressed
        actuator = new DAQout(HMI.gripperOutputChannel, "gripperJawControl");
        sensor = new DAQin(HMI.gripperInputChannel,
        "gripperPositionSensing");

        if (HMI.simAir) return;
        if (SensePart()) //If the gripper is holding a pole while the station
        is powered on,
        {
            CurrentlyOpen = true;
            Close(); //let the grippers hold on tight, to prevent pole
        from slipping out
            HMI.log("ERROR", "An unknown pole seems to be gripped by the
        gripper. Please remove", this);
            if (!HMI.simMode)
                while (SensePart()) ; //wait for operator to remove pole from
        gripper
        }
        Open(); //Open up the jaws to prevent collisions when picking
        up the first pole
    }

    override public void Open()
    {
        if (CurrentlyOpen) //If gripper is already open, but again sent
        command to open
            HMI.log("WARNING", "Consecutive calls for opening the gripper have
        been made", this);
        HMI.gripperStatus = "open";
        CurrentlyOpen = true;
        if (!HMI.simAir && !HMI.simMode) //Don't activate hardware when in
        Simulation mode or no air supply
            actuator.write(true);
        HMI.HWlog("Gripper opened");
    }

    override public void Close()
    {
        if (!CurrentlyOpen) //If gripper is already closed, but again sent
        command to close
            HMI.log("WARNING", "Consecutive calls for closing the gripper have
        been made", this);
        HMI.gripperStatus = "closed";
        CurrentlyOpen = false;
        if (!HMI.simAir && !HMI.simMode) //Don't activate hardware when in
        Simulation mode or no air supply
            actuator.write(false);
        HMI.HWlog("Gripper closed");
    }

    override public Boolean SensePart()
    {
        Boolean GripperHalfway = false;
        if (HMI.simMode || HMI.simAir)//When in Simulation mode or when
        testing without air supply:

```

```
        return !CurrentlyOpen; //assume gripper always manages to pick
up parts as intended
        GripperHalfway = sensor.Read();
        if (!CurrentlyOpen && GripperHalfway)
            HMI.gripperStatus = "Item MIS-GRIPPED!";
        HMI.HWlog("Gripper closed halfway: " +
Convert.ToString(GripperHalfway));
        return GripperHalfway;
    }
}
```

## Appendix D – Inter-station messages

Appendix C contains information relating to the messages that were interchanged between the stacking station and the cell controller.

Table 5 XML message structure for inter-station communication

```
<Message>
  <Recipient>
    Intended recipient of message
  </Recipient>
  <Sender>
    Sender of the message
  </Sender>
  <Port>
    Number of the port through which the message has been sent
  </Port>
  <Type>
    Type of message
  </Type>
  <CID>
    Unique ID of the conversation of which the message forms part
  </CID>
  <Msg>
    Message content
  </Msg>
</Message>
```

Table 6 Type of messages that the station controller can receive

Message type	Purpose
<b>PlaceOrder</b>	When the cell controller places a (batch) order.
<b>PalletArrive</b>	Indicates that a pallet has arrived, and conveys information as to what items are loaded into which position of the pallet.
<b>PalletAway</b>	To confirm that pallet has been transported away.
<b>PalletInfo</b>	Contains the fixture coordinates of a pallet. To reduce communication overhead, usually only the pallet ID is used as reference for the above three commands. When the fixture coordinates of a particular pallet are not known yet, but are needed, the stacker will requested them first.
<b>ProductInfo</b>	Information regarding assemblies or their parts.
<b>ReqReportStatus</b>	Request to report on status, for diagnostic purposes and for scalability purposes: If several stacking stations were working in parallel the cell controller would place orders with stations that have the smallest workload. Or when a lot of failures occurred at the e-test, then those pallets with the most irregularities can be sent to the station which has the most available space in the buffers.

Table 7 Type of messages sent out by the station controller

Message type	Purpose
<b>OrderComplete</b>	Inform the cell controller of the ID's of poles being matched.
<b>SendPalletAway</b>	When all fixtures on the pallet have been filled, the pallet can be sent away. The cell controller is informed of the exact pallet contents, i. e. which products are placed into which positions of the pallet.
<b>OrderOverdue</b>	A warning being sent when an order could not be completed within its desired completion time.
<b>ReqPole</b>	When a pole for an order is needed but not present at that station, it will be requested by the task holon in need of that pole.
<b>ReqProductInfo</b>	When an order has been placed, but the corresponding product info is not yet available on the stacking station, it will be requested.
<b>BufferCap</b>	Warns the cell controller that the buffer is getting full and that its capacity has reached a percentage of 75%, 90% or 100%.
<b>TaskCount</b>	Notifies the cell controller of the number of pending orders on the station controller.

## Appendix E – Calibration procedures

Some of the reconfigurations require calibration of hardware. Those procedures are described in this section.

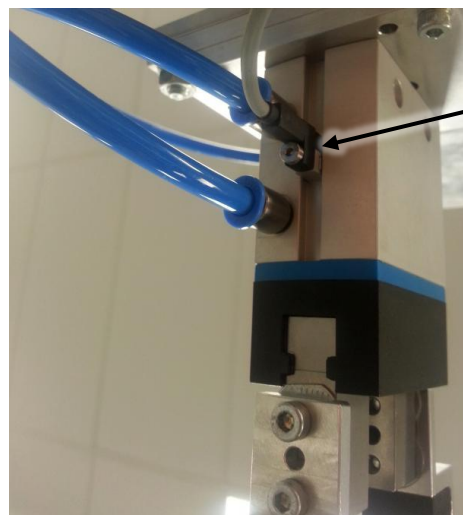
### E.1 Calibrating the proximity sensor

When a new part with a different width is to be picked up, then the sensors should be re-adjusted to output the correct signals. **True** should be signalled whenever the part is properly gripped and **false** otherwise, such as when the grippers are completely open or completely closed (i.e. when the part has not been properly gripped).

The sensor is equipped with an LED which turns on when the proximity sensor gives out a **true** signal.

Steps:

1. Let the grippers grip the product in the proper way.
2. Loosen the screw of the sensor.
3. Adjust the position of the sensor until the LED comes on while the product is still being held.
4. Tighten the screw to fix the sensor position.
5. Open the jaws and remove the product.
6. If the jaws are almost completely open or almost completely closed while the product is held properly, it could happen that the region being sensed as “gripped” overlaps with the region being sensed as “fully open” or “fully closed” due to some tolerance of the sensors. To ensure that the sensor does not give out a high voltage when the jaws are completely open or completely closed, move the sensor a small distance away from the incorrectly sensed position by repeating steps 1-5 above until three distinct positions can be measured.



Sensor which can be moved to anywhere in the slot to sense a specific position

Figure 35 Adjustable proximity sensor

## E.2 Calibrating tools, such as the gripper

On-board the KUKA controller, dimension-data for 16 different tools can be stored inside an array called TOOL\_DATA[ ].

When calibrating the gripper fingers (the “tool”), make sure the jaws are in a fully opened position. Choose a reference point at the edge of the fingertip. When a certain tool is selected, the reference point will become the tool centre point (TCP) and any subsequent rotations will be executed about that point.

1. Place an object with a sharp point at a fixed position within close reach of the robot.
2. On the KUKA controller, terminate any running programs and navigate to Setup > Measure > Tool > XYZ 4-Point.
3. Select the index of the TOOL\_DATA array inside which the tool coordinates are to be stored and specify a name for the tool.
4. Use the  $\pm X$ ,  $\pm Y$ ,  $\pm Z$ ,  $\pm A$ ,  $\pm B$ , and  $\pm C$  buttons on the pendant to move the end-effector and let the reference point on the tool only just touch the tip of the sharp object.
5. On the pendant, select “Measure” and “Continue”.
6. Repeat steps 4 and 5, approaching the tip from four different directions, which should all be different\* to one another.
7. On the pendant, select “Accept” to store the data.
8. The procedure may be cancelled at any stage by selecting “Cancel” on the pendant.
9. On the station controller, select “Add new tool” and follow the instructions on the screen. This is to ensure that the settings file for the gripper holon reflects the correct tool index chosen in step 3. Recording such changes allows to revert back to a previously calibrated tool during any future reconfigurations.

\* The bigger the difference between the lines of approach, the more accurately the controller will be able to calculate the relative position of the reference point.

The tool calibration procedure described above takes about 10 minutes and adding the tool to the range of tools on the controller (Figure 36) takes another 30 seconds.

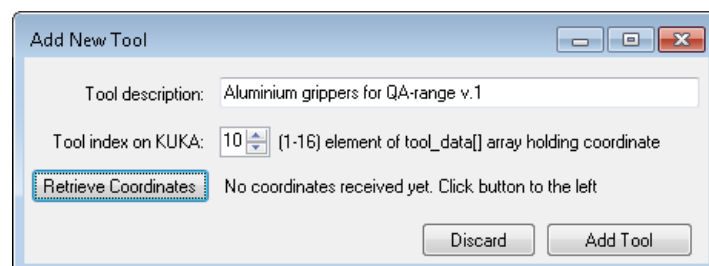


Figure 36 HMI for adding a new tool

### E.3 Calibrating workspaces such as the buffer, and pallets

The XYZ 3-point method is used to define a plane unambiguously in the 3D space:

On-board the KUKA controller, dimension-data for 32 different workspaces (planes) can be stored inside an array called BASE\_DATA[].

Some reference point on the workspace should first be chosen which will serve as the origin of the plane, and with respect to which the CAD dimensions can easily be expressed.

Also, a previously calibrated tool must be attached to the end effector

1. On the KUKA controller, terminate any running programs and navigate to Setup > Measure > Base > 3-point
2. Select the index of the BASE\_DATA array inside which the workspace coordinates are to be stored and specify a name for the workspace.
3. From a list, select a previously calibrated tool which is to be used for the workspace calibration procedure. This tool must also be attached to the end effector.
4. Use the  $\pm X$ ,  $\pm Y$ ,  $\pm Z$ ,  $\pm A$ ,  $\pm B$ , and  $\pm C$  buttons on the pendant to move the reference point on the tool (the TCP) to the reference point of the workspace (the origin of the plane).
5. On the pendant, select "Measure" and "Continue".
6. Move the TCP to any point on the positive X-axis of the plane and select "Measure" and "Continue".
7. Move the TCP to any point on the plane which has a positive Y-value, and Select "Measure" and "Continue".
8. On the pendant, select "Accept" to store the data.
9. On the station controller, select "Add new workspace" and follow the instructions on the screen. This is to ensure that the settings file for the corresponding operational holon reflects the correct workspace index chosen in step 2. Recording such changes allows to revert back to a previously calibrated workspace during any future reconfigurations.

The workspace calibration procedure takes about 12 minutes to complete.

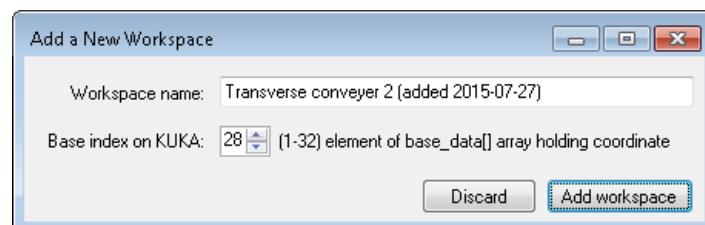
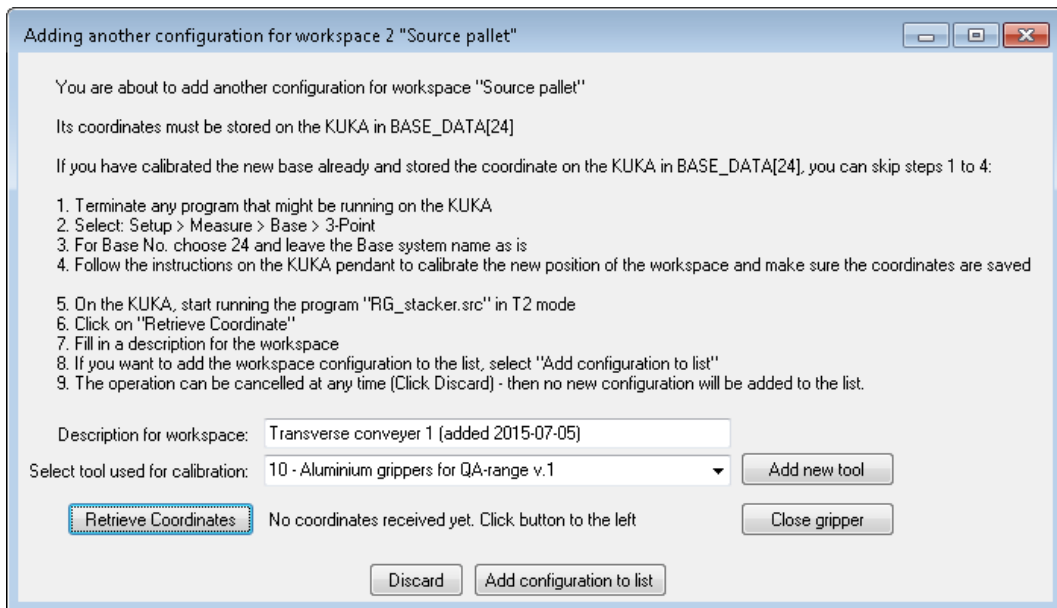


Figure 37 HMI for adding a new workspace



Adding an entirely new workspace on to the station controller ( Figure 37), takes about a minute, and defining a new configuration for an already existing workspace ( Figure 38) takes 2 minutes.



The screenshot shows a dialog box titled "Adding another configuration for workspace 2 'Source pallet'". The dialog contains the following text and controls:

You are about to add another configuration for workspace "Source pallet"  
Its coordinates must be stored on the KUKA in BASE\_DATA[24]  
If you have calibrated the new base already and stored the coordinate on the KUKA in BASE\_DATA[24], you can skip steps 1 to 4:

1. Terminate any program that might be running on the KUKA
2. Select: Setup > Measure > Base > 3-Point
3. For Base No. choose 24 and leave the Base system name as is
4. Follow the instructions on the KUKA pendant to calibrate the new position of the workspace and make sure the coordinates are saved
5. On the KUKA, start running the program "RG\_stackercr" in T2 mode
6. Click on "Retrieve Coordinate"
7. Fill in a description for the workspace
8. If you want to add the workspace configuration to the list, select "Add configuration to list"
9. The operation can be cancelled at any time (Click Discard) - then no new configuration will be added to the list.

Description for workspace:

Select tool used for calibration:

No coordinates received yet. Click button to the left

Figure 38 HMI for adding new workspace configuration