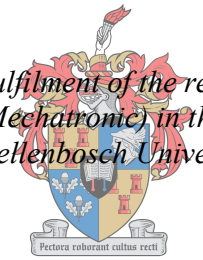
The integration of human workers as resource holons in a holonic manufacturing cell

by
Johannes Cornelius Leuvennink

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



UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

Supervisor: Dr Karel Kruger
Co-supervisor: Prof Anton Basson

March 2018

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: March 2018

Abstract

This thesis documents research conducted into the integration of human workers as resource holons in a holonic manufacturing cell. The research is motivated by the need to develop manufacturing systems that allow smaller enterprises in developing countries, such as South Africa, to be competitive in a global market, without contributing to unemployment. These systems must be selectively automated, so that the critical processes are automated while the other processes retain the use of manual labour.

The objectives of this research are to develop architectures for human integration in holonic manufacturing systems and to evaluate and compare these architectures. To this end, holonic control strategies and the proposed architectures for human integration were implemented with a testbed manufacturing cell at Stellenbosch University.

The structure of the holonic control strategies is based on the PROSA reference architecture and the mapping of the testbed cell components to holons is explained. The holonic control system for the testbed cell was implemented as a multi agent system using the JADE platform. The higher level control and lower level control of the subsystems of the cell are described in detail.

Two architectures for human integration in holonic systems were developed, namely the interface holon architecture (IHA) and the worker holon architecture (WHA). The IHA makes use of a fixed interface to communicate with a worker that is assigned to a specific workstation by a human supervisor. The WHA makes use of a mobile interface, dedicated to a specific worker, to communicate with the worker. The WHA also makes use of an automated supervisor software agent that manages the workers on the factory floor instead of a human supervisor. These architectures, as well as their implementation and integration with the holonic control system of the testbed cell, is described in detail.

A series of experiments were devised to evaluate the two architectures for human integration. The experiments were performed and the results are analysed and discussed. The results show that the WHA is superior to the IHA since it results in higher productivity as well as more flexibility and reconfigurability.

Uittreksel

Hierdie tesis dokumenteer navorsing wat gedoen is oor die integrasie van menslike werkers as hulpbron holons in 'n holoniese vervaardiging sel. Die navorsing word gemotiveer deur die behoefte vir die ontwikkeling van vervaardigingsstelsels wat dit vir kleiner ondernemings, in ontwikkelende lande soos Suid-Afrika, moontlik maak om mededingend te wees in 'n globale mark, sonder om tot werkloosheid by te dra. Hierdie stelsels moet selektief geoutomatiseer wees, sodat die kritieke dele van prosesse geoutomatiseer word terwyl die ander dele die gebruik van handearbeid behou.

Die doelwitte van hierdie navorsing is om argitekture vir menslike integrasie in holoniese vervaardigingstelsels te ontwikkel en hierdie argitekture te evalueer en te vergelyk. Tot hierdie doeleinde, is holoniese beheerstrategieë en die voorgestelde argitekture vir menslike integrasie geïmplementeer in 'n toetsbed vervaardiging sel by die Universiteit Stellenbosch.

Die struktuur van die holoniese beheerstrategieë is gebaseer op die PROSA verwysingsargitektuur en die kartering van die toetsbed-selkomponente tot holons word verduidelik. Die holoniese beheerstelsel vir die toetsbed sel is geïmplementeer as 'n multi-agentstelsel deur van die JADE-platform gebruik te maak. Die hoër vlak beheer en laer vlak beheer van die substelsels van die sel word in detail beskryf.

Twee argitekture vir menslike integrasie in holoniese stelsels is ontwikkel, naamlik die koppelvlak holon argitektuur (IHA) en die werker holon argitektuur (WHA). Die IHA maak gebruik van 'n vaste koppelvlak om te kommunikeer met 'n werker wat deur 'n menslike toesighouer aan 'n spesifieke werkstasie toegewys is. Die WHA maak gebruik van 'n mobiele koppelvlak, toegewy aan 'n spesifieke werker, om met die werker te kommunikeer. Die WHA maak ook gebruik van 'n outomatiese toesighouer sagteware agent wat die werkers op die fabrieksvloer bestuur in plaas van 'n menslike toesighouer. Hierdie argitekture, sowel as die implementering daarvan en integrasie met die holoniese beheerstelsel van die toetsbed sel, word in detail beskryf.

'n Reeks eksperimente is ontwerp om die twee argitektuure vir menslike integrasie te evalueer. Die eksperimente is uitgevoer en die resultate word ontleed en bespreek. Die resultate toon dat die WHA beter is as die IHA, aangesien dit hoër produktiwiteit sowel as meer buigsaamheid en herkonfigureerbaarheid tot gevolg het.

Acknowledgements

I would like to thank everyone who contributed, in any way, to this thesis. Special mention must be made of the contributions of the following people:

- Karel Kruger and Prof Anton Basson, for your advice and all the time you invested in giving me the proper guidance to complete this thesis. I have learnt much from you both.
- Reynaldo Rodriquez, for all your help with the equipment in the lab which allowed me so set up my testbed cell.
- My fellow members of the Mechatronic Automation and Design research group, who assisted me with my experimentation.
- Carlien van Eeden, for all your loving support and motivation. You inspire the best in me.

Table of contents

List of figures

Page

List of tables

Page

List of symbols

n_o Number of operations.

n_w Number of workers.

$\bar{t}_o$ Average operation time.

t_o Operation time.

$t_{o,wn}$ Total operation time of worker n.

t_p Total production time.

$\bar{u}_w$ Average worker utilization.

u_{wn} Worker utilisation of worker n.

List of abbreviations

ACL	Agent communication language
AID	Agent identifier
AMS	Agent management system
AP	Agent platform
CFP	Call for proposal
CIM	Computer integrated manufacturing
DF	Directory facilitator
DOF	Degree of freedom
FIPA	Foundation for intelligent physical agents
FSM	Finite state machine
GUI	Graphical user interface
HMS	Holonic manufacturing systems
ICS	Intelligent control systems
IHA	Interface holon architecture
IMS	Intelligent manufacturing systems
JADE	Java agent development framework
MAS	Multi-agent system(s)
WHA	Worker holon architecture

1 Introduction

1.1 Background

Ever since the start of the industrial revolution there has been a movement toward the use of machines rather than human workers to improve the productivity of manufacturing processes. This movement first inspired mechanisation and later gave rise to automation. Mechanisation provided human operators with machinery to assist them with the muscular requirements of work. Automation is a step beyond mechanization, greatly decreasing the need for human sensory and mental requirements as well.

Automation in manufacturing industries include the use of advanced control systems, information technology, mechanical machinery and robotics to reduce the need for human work in the production of goods. The concept of automation has various advantages and disadvantages when compared to manual labour. Some of the advantages of automation are: higher throughput, increased accuracy and repeatability, less human error, reduced labour costs and increased safety. Some of the disadvantages of automation are: decreased versatility, large initial cost and increased unemployment. (Blue, 2013)

In the modern industry there are many manufacturing processes that have been fully automated as well as many that still rely heavily on manual labour. The decision to automate a process depends on many factors. Full automation can be advantageous or disadvantageous for the manufacturing company, depending on the situation.

The modern manufacturing environment demands shorter lead times and higher product variety without compromising quality or price. The answer for this demand is complex adaptive systems that can provide adequate performance, as well as adapt to changes and disturbances.

One answer to this problem was found in Koestler's theories on complex adaptive systems (Koestler, 1967). Koestler made the observation that complex systems can only arise if they consist of stable, autonomous subsystems. These subsystems must be able to survive disturbances and also be able to cooperate with other subsystems. These theories gave rise to the idea of holonic manufacturing. Holonic manufacturing implies a highly distributed organization of the manufacturing system, where intelligence is distributed over the individual entities. These entities are cooperative, intelligent and autonomous modules called holons (Van Brussel, et al., 1999).

In a holonic manufacturing system, individual entities (holons) work together in temporary hierarchies, called holarchies, to achieve a global goal. A holonic manufacturing system combines performance with robustness against changes and disturbances. Since holons are independent entities, they can easily be rearranged into different holarchies without making major changes. Holonic manufacturing systems are thus highly reconfigurable.

HMS reference architectures are a set of design principles with the purpose of providing a structure for the design of a specific system. Various reference architectures for HMS have been proposed by researchers. Most notable of these are PROSA (Van Brussel, et al., 1998) and ADACOR (Leitao & Restivo, 2006).

1.2 Objectives

The objective of this thesis is to develop and evaluate architectures for the integration of human workers in holonic manufacturing systems. The thesis focusses on the integration of human workers as shop floor resources, being able to perform specified production tasks. Supervisory and management tasks performed by humans are therefore not included in the integration. The research considers a holonic manufacturing system that is based on the PROSA reference architecture, wherein human workers are integrated as resource holons.

The developed resource holon architectures should encompass the integration of human workers in the system, at workstation and interface control levels. The detailed study of ergonomics for the human interfaces is not included in the scope of this research. At the system control level, the resource holon must exchange production execution information with the other holons within the PROSA architecture.

The architectures for human integration are to be implemented as part of testbed manufacturing cell based on a relevant case study. Through experimentation with the testbed cell, the developed architectures can then be evaluated and compared.

1.3 Motivation

In a developing country like South Africa, the decision of whether or not to automate is difficult. On the one hand, automation in the South African industry can be very advantageous. Automation can increase production throughput and quality, resulting in more exported products, which will benefit the economy. Labour difficulties are a big problem in South Africa and has been known to be the reason for many investors to hesitate when investing in South Africa. Implementing automation and removing the need for manual labour can be very attractive to international investors. Removing human workers from dangerous

occupations such as mining would also result in less work-related injuries and fatalities.

On the other hand, automation is better suited to large international manufacturers rather than the newer and smaller companies of a developing country. The initial cost of automated equipment too high for many small companies to afford. The smaller factories of South Africa also generally produce smaller volumes of a larger variety of products, for which the classical approach to manufacturing automation is not suitable. Unemployment is a very big problem in South Africa and therefor, using automated systems to replace human workers becomes an ethical issue.

In many cases, these constraints only allow for the automation of certain processes in the manufacturing system. This approach is referred to as selective automation. The selection of the processes that should be automated is based on several factors. These factors include the ease of which a process can be automated, in terms of the technical knowledge and equipment required, and the value that automation adds to the production process. The impact on production value can be measured in production cost, throughput and the elimination of safety risks.

Considering the needs of the South African manufacturing industry, a possible solution is to use Reconfigurable, selectively automated manufacturing systems. The holonic manufacturing system paradigm is well suited to achieve this. Since the objective is only selective automation, holonic manufacturing systems that allow the integration of human workers as resource holons can be a viable solution for the South African manufacturing industry. Such a system could have all the benefits of selective automation and a reconfigurable holonic system, without replacing all human workers.

In previous research regarding HMS, most general holon architectures contains a human interface component. In most cases this component is intended for supervisory control purposes. There is very little mention of human integration as resource holons in the literature and when it is mentioned, it is only to state that it is possible. No detailed work could be found on exactly how such integration could be performed.

1.4 Methodology & overview

This section briefly outlines the methodology used to achieve the objectives of this research. First, literature concerning holonic manufacturing systems and related research was studied in order to gain the knowledge required to continue this research. The full literature review is given in section 2.

A relevant case study was then selected and used to develop a manufacturing testbed cell that requires both manual and automated resources. The case study, the testbed cell and the product it produces is described in detail in section 3.

A holonic control system for the testbed cell was developed according to the PROSA reference architecture and implemented as a multi-agent system. Agents were developed to represent the higher level control (HLC) of the automated resources in the cell. These agents interface with the lower level control (LLC) that was developed for the automated resource hardware. This HLC and LLC is described in detail in section 4. The HLC and LLC of the human workers in the cell are not described in section 4 and is covered later in the thesis.

Two architectures for human integration in holonic manufacturing systems were then developed. These architectures are based on two different approaches. With one approach, a fixed human interface at a workstation is represented by a holon in the HMS. The workstations are requested to perform operations, and any worker that is assigned to that workstation by a human supervisor then performs that operation. The other approach was to directly represent each individual worker as a holon in the HMS. By using a mobile interface, specific workers can then be requested to perform operations by an automated supervisor holon in the HMS. Each approach presents certain advantages and disadvantages and one may be better suited in some situations than the other. The architectures for human integration are developed in section 5 and implemented in section 6.

A series of experiments with the testbed cell were performed to evaluate the two architectures for human integration. The goal of the experiments was to compare the two architectures in terms of pre-defined set of evaluation criteria. Different scenarios were simulated in the experiments in order to determine which architecture is most suited for certain situations. The evaluation criteria, experimental setup, data acquisition, results and discussion of the results is discussed in section 7.

The conclusions from the results of the experimentation were summarised and recommendations for future research were made. The various conclusions and recommendations are given in section 8.

2 Literature review

In this section a review of literature relevant to the thesis is given. The review focusses on holonic systems, multi agent systems and related work.

2.1 Holonic systems

2.1.1 Background

Modern consumers demand short lead times and higher product variety from a manufacturer without compromising quality or price. These modern demands mean that many traditional manufacturing processes now lack competitiveness in the global manufacturing environment. In response to this growing perception, the field of Holonic Manufacturing Systems (HMS) was initiated in Japan by Suda (1989). Suda hypothesised that the cause of this inability to compete was rigid manufacturing processes that lacked agility and responsiveness to changes and disturbances. Suda further stated that the characteristics of robustness, flexibility and adaptability of holonic systems could be the solution to this problem.

The concept of holonic systems originated from philosopher A. Koestler's theories on complex adaptive systems (Koestler, 1967). Koestler made the observation that complex systems can only arise if they consist of stable, autonomous subsystems that have the ability to survive disturbances. These subsystems must also have the ability to cooperate with other subsystems. These theories gave rise to the idea of holonic manufacturing. Holonic manufacturing implies a highly distributed organization of the manufacturing system, where intelligence is distributed over the individual entities. These entities are cooperative, intelligent and autonomous modules called holons (Van Brussel, et al., 1999).

From 1992-1994, teams of experts from around the world worked together to build a test framework for international collaboration in intelligent manufacturing systems (IMS). The holonic manufacturing systems project along with the HMS consortium was formed as one of the six IMS feasibility studies (Farid, 2004).

2.1.2 Holonic manufacturing system rational

Most modern day industrially implemented manufacturing systems can be broadly categorized as computer integrated manufacturing (CIM). HMS are meant to be an alternative to CIM that can overcome some of the limitations and drawbacks associated with CIM (Farid, 2004).

CIM systems have poor agility because of their fixed control hierarchy that does not support change. Furthermore, reconfiguration and extension of existing CIM systems is difficult, performance is not maintained outside of normal conditions,

data for diagnosis is difficult to access and the automated control excludes human intervention (Bussmann, 1998).

As an alternative to CIM, researchers suggested to replace the rigid hierarchical system with the flat structure of a heterarchical system, where each of the components exhibit full local autonomy. Each component in a heterarchical system cooperates via a negotiation procedure to form temporary relationships. Some of the advantages of heterarchical systems includes: high fault tolerance, local disturbance rejection and reduced complexity. These advantages are, however, ultimately insufficient and heterarchical systems were never industrially adopted due to their inability to achieve a predictable result. (Farid, 2004)

Holonic systems have advantages of both hierarchical and heterarchical systems. Holons can belong to multiple hierarchies and do not rely on the proper function of other holons. Holonic systems also have autonomous and cooperative characteristics, like heterarchical systems do, since they negotiate with each other and make local decisions. (Farid, 2004)

Holonic manufacturing systems can mitigate most unwanted circumstances and are a robust flexible and adaptable alternative to CIM.

2.1.3 Basic theory

A holon is defined as an autonomous and cooperative building block of a manufacturing system for transforming, transporting, storing and/or validating information and physical objects. A holon consists of an information processing part and often, a physical processing part (Christensen, et al., 1994).

Each holon has an autonomous characteristic and thus, its development is independent and its functionality is capable of existing alone. Each holon also has a cooperative characteristic that allows it to depend upon a social framework of holons. Individual holons can thus work together in temporary hierarchies, called holarchies, to achieve a global goal. A holarchy is a system of holons that cooperate to achieve a global goal (Christensen, et al., 1994). Cooperation within holarchies, in the form of coordination and negotiation, develops wherever and whenever necessary.

One of the strengths of a holarchy is that it enables the construction of very complex systems that use resources efficiently. Holarchies are recursive in the sense that a holon itself may be an entire holarchy consisting of many holons. (Giret & Botti, 2004). Since holons are independent entities, they can easily be rearranged into different holarchies without making major changes to the system. This causes holonic manufacturing systems to be resilient to disturbances and adaptable to changes in their environment.

2.1.4 Holon architecture

The HMS consortium defined a set of characteristics that an entity should possess to make it a holon. These holonic characteristics are defined below:

- Autonomy – The capability of an entity to create and control the execution of its own plans and/or strategies (Christensen, 1994).
- Cooperation – A process whereby a set of entities develops mutually acceptable plans and executes these plans (Christensen, 1994).
- Recursivity – A similarity in the informational architecture and communications model between holons (Mathews, 1995).
- Self-Organization – The ability of manufacturing units to collect and arrange themselves in order to achieve a production goal (Christensen, et al., 1994)
- Reconfigurability – The ability manufacturing unit to simply alter its function in a timely and cost effective manner (Christensen, et al., 1994).

For a holon to possess these characteristics, its composition requires certain elements. A holon always contains an information processing component and an optional physical processing component. These components, along with an appropriate communication interface, represents a holon. A holon must also be able to reason and communicate with other holons. The various components of a holon and the way they are interconnected defines the holon architecture.

In 1994, Christensen proposed the first general holon architecture (Christensen, 1994). Figure 1 below shows the main components of this architecture.

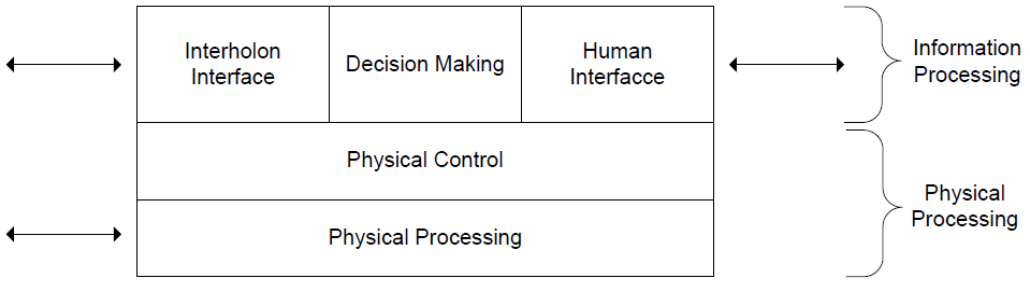


Figure 1: General holon architecture (Christensen, 1994).

The information processing component consists of three main parts: decision making, inter-holon interface and human interface. The decision making part (the kernel of the holon) has reasoning capabilities and makes decisions that control the behaviour of the holon. The inter-holon interface is used to communicate with other holons in the system in order to facilitate cooperation. The human interface is a control interface used to issue commands and monitor the state of the holon.

The physical processing part of the holon consists of two parts: The first part, is the physical possessing part itself, which is traditionally thought of as a hardware resource like a CNC machine or a robot. The second part, is the physical control part, which is the lower level controller of the hardware resource.

Fletcher et al. (2000) developed a more detailed holon architecture, based on Christensen's original work. According to Fletcher et al., a holon may be considered to consist of an intelligent control system (head) and a processing system (base).

The head consists of the process/machine control (PMC), the process/machine interface (PMI), the human interface (HI) and the inter-holon interface (IHI). The PMC is responsible for execution of the control plan for the process that is being controlled. The PMI provides the logical and physical interface to the processing system via a suitable communication network. The HI comprises the interfaces to humans such as supervisors, maintenance personnel and process engineers. The IHI handles the inter-holon communication. (Fletcher, et al., 2000)

The base consists of all processing components necessary to perform a manufacturing activity. The base is thus responsible for the manufacturing functionality. (Fletcher, et al., 2000)

2.1.5 Reference architectures

A holonic manufacturing system (HMS) is defined as: "A holarchy that integrates the entire range of manufacturing activities from order booking through design, production and marketing to realise the agile manufacturing system enterprise" (Farid, 2004).

There are various reference architectures for HMS that have been proposed as a result of the IMS feasibility program. It is important to make the distinction between the holon architecture described in section 2.1.4 and the HMS reference architectures that are described in this section. A holon architecture describes the inner composition of the holon itself. HMS reference architectures are a set of design principles with the purpose of providing a structure for the design of a specific system. This is accomplished by defining a unified terminology, the structure of the system as well as the responsibilities of the system components (Van Brussel, et al., 1998). Thus HMS reference architectures are inter-holonic architectures which identify the types of holons necessary for any manufacturing system, its responsibilities, and the interaction structure in which they cooperate.

Examples of HMS reference architectures include PROSA (Van Brussel, et al., 1998), ADACOR (Leitao & Restivo, 2006), HCBA (Chirn & McFarlane, 1999) and HoMuCS (Langer & Bilberg, 1997). PROSA and ADACOR are the two most commonly accepted holonic reference architectures.

2.1.5.1 PROSA – Product, Resource, Order, Staff Architecture

The PROSA architecture consists of three types of basic holons: order holons, product holons and resource holons. Staff holons can be added to assist the basic holons with expert knowledge (Van Brussel, et al., 1998).

The resource holon, in keeping with the holon architecture described in section 2.1.4, has an information processing component as well as a physical processing component. The physical processing component of the resource holon usually consists of a machine with a certain functionality, such as a conveyor or a robot. The production capacity or functionality of the resource holon is available to be used by the other holons in the system. The information processing component of the resource holon allocates the production resources and holds knowledge and procedures to organise, use and control these production resources (Van Brussel, et al., 1998).

A product holon, unlike a resource holon, does not have a physical processing part. A product holon serves as an information server to the other holons. It contains information concerning the design, process plans, bill of materials, quality assurance procedures, etc. of a certain product (Van Brussel, et al., 1998). It is important to note that there is not a product holon for every physical instance of a product that is being produced. There is in fact only one product holon for every type of product and it only serves a product model.

An order holon represents a task in the manufacturing system and is responsible for managing the physical product that is produced. The order holon contains a model that describes the state of the product and ensures that all the work required to produce the product is performed on time (Van Brussel, et al., 1998).

As seen in Figure 2, the three types of holons exchange knowledge concerning the manufacturing system. Product holons and resource holons communicate process knowledge, for example, information and methods on how to perform a certain process. Product holons and order holons exchange production knowledge, for example, the information and methods on how to produce a certain product. Resource holons and order holons share process execution knowledge, for example, information and methods regarding the progress of executing processes on resources. (Van Brussel, et al., 1998)

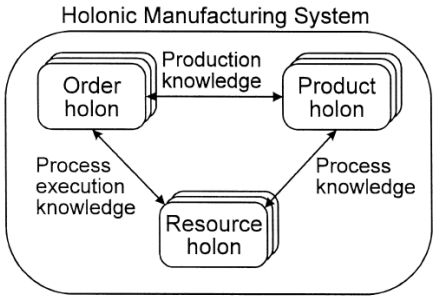


Figure 2: Basic building blocks of a PROSA HMS and their relations

2.1.5.2 ADACOR – Adaptive Holonic Control Architecture

The ADACOR architecture defines four manufacturing holon classes: product holon, task holon, operational holon and supervisor holon.

The product, task and operational holons are very similar to the product, order and resource holons of the PROSA reference architecture described in section 2.1.5.1. The supervisor holon is, however, different from the PROSA staff holon.

Since different levels of hierarchies exist within an HMS, a coordinating holon is required to aggregate the skills of the members of a group of holons. As seen in Figure 3, the supervisor holon introduces coordination and global optimisation in decentralised control and is responsible for the formation and coordination of groups of holons and offer combined services to other holons. Supervisor holons fulfil this role by creating optimised production plans for the operational holons (Leitao & Restivo, 2006).

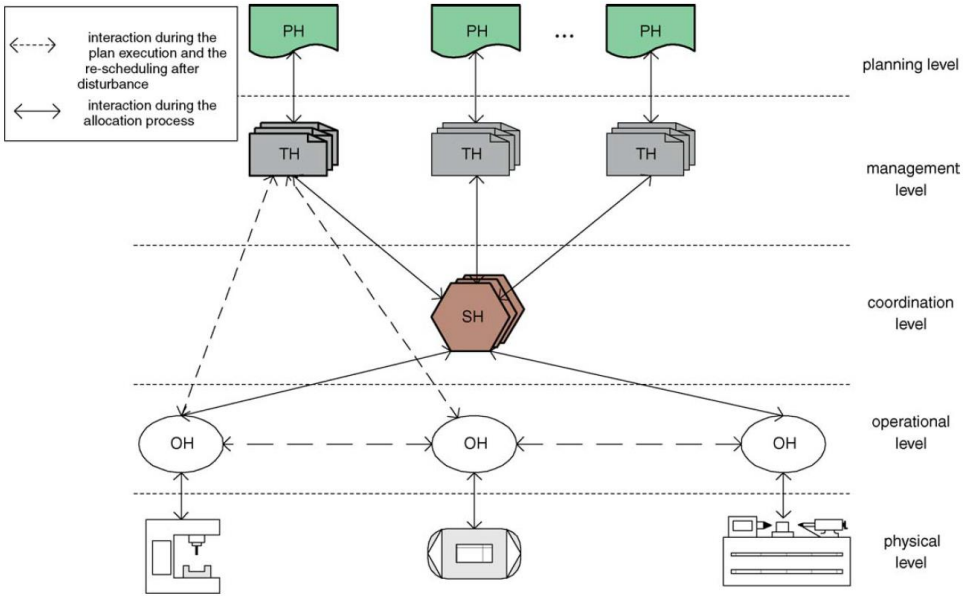


Figure 3: ADACOR holon classes and interactions (Leitao & Restivo, 2006)

The internal architecture of an ADACOR holon is basically the same as the Christensen's general holon architecture described in section 2.1.4 with some differences. Figure 4 below shows a conceptual model for an ADACOR operational holon.

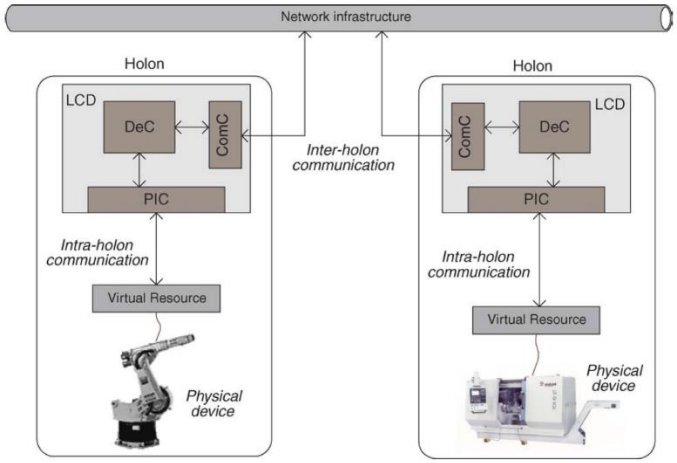


Figure 4: Conceptual model for an ADACOR holon (Leitao & Restivo, 2006)

As seen in Figure 4, the holon consists of a decision making component (DeC), a communication component (ComC) and a physical interface component (PIC). The decision making component controls the behaviour of the holon by performing process planning, scheduling and plan execution. The communication component facilitates inter-holon communication. (Leitao & Restivo, 2006).

Since resource controllers usually have closed control architectures, the physical interface component provides a mechanism to support resource integration based on the virtual resource concept and the client-server model. The server part of this mechanism is much like a virtual machine device that represents the functionalities of the real manufacturing device and supplies primitives to be invoked by the client part of the mechanism (Leitao & Restivo, 2006).

The PIC component acts as the client part of the mechanism. It accesses the real manufacturing resource by invoking the primitives supplied by the virtual resource that represent the services in the physical resource (Leitao & Restivo, 2006).

2.1.6 Open issues for industrial adoption

One of the reasons that HMS has not been widely adopted in the industry is because of a lack of rigorous comparisons with the current best alternatives. In order for companies to accept the risk of implementing HMS the specific advantages of robustness and adaptability to disturbances and failures need to be demonstrated in a real life application (Farid, 2004). To be fully effective, holonic manufacturing requires a complete reorganisation of production operations,

which can become very expensive. Because of this, it is very important to show and quantify the benefits (McFarlane & Bussmann, 2003).

There are very few complete methodologies available for HMS design and implementation. HMS will only become a viable choice for industrial implementation once a complete methodology with clear guidelines has been developed and favourably evaluated against the existing CIM design methodologies that are the main alternative (Farid, 2004). More industrial implementations of these methodologies will also be needed before they become a viable option. Even though some complete methodologies like ANEMONA (Giret & Botti, 2008) do exist, researchers have not yet adopted a single methodology as a common base for their own new developments, making research in the field of HMS inefficient (McFarlane & Bussmann, 2003).

There are also various issues with implementing and maintaining an effective holonic control environment. Before any industrial confidence in holonic manufacturing systems can be established, a comprehensive set of standards is required for the open specification of communications, data formats, systems architectures, algorithms and interfacing of holonic systems. There has, to date, been no comprehensive study of the requirements for standards in this area (McFarlane & Bussmann, 2003).

There has also been little work done on determining the compatibility of the holonic control with the current or the next generation of industrial control and computing systems. Determining how to construct and implement system architectures capable of fully supporting holonic operations while still operating with existing legacy systems will also be a major issue (McFarlane & Bussmann, 2003).

2.2 Multi agent systems

2.2.1 Definition of agents and multi agent systems

There are many definitions of an agent in the literature. Botti and Giret (2008) provide the following definition of an agent: “An autonomous and flexible computational system that is able to act in an environment”. Paulucci and Sacile (2016) provide another definition: “an agent is defined as a computational system which is long lived, has goals, sensors and effectors and decides autonomously which actions to take in the current situation to maximise progress toward its goals”.

Although agent definitions do vary, there is a more accepted consensus regarding the characteristics of an agent. The general characteristics of an agent are listed below (Botti & Giret, 2008), (Paulucci & Sacile, 2016).

- **Autonomy**: Agents should be able to operate without the intervention of humans or other agents.
- **Proactivity**: Agents should be capable of trying to fulfil their own goals.
- **Reactivity**: Agents should be able to perceive their environment and respond to changes in the environment.
- **Social ability**: Agents should be able to communicate with humans or other agents.
- **Rationality**: Agents should be able to reason about perceived data in order to compute an optimal solution.
- **Mobility**: Agents should be able to change their physical location to improve their problem solving capacity.
- **Veracity**: Agents will not knowingly communicate false information.

According to laws et al. (2001) there are 3 types of agent architectures: reactive, deliberative and hybrid. Reactive agents respond to every possible input in a pre-defined manner. Deliberative agents represent goals and, based on the sensory input, they formulate plans to achieve these goals. Hybrid agents use elements of both reactive and deliberative agents.

Multi agent systems can be summarised as “flexible networks of problem solvers that can solve a problem that is beyond an individual solver” (Paulucci & Sacile, 2016). In a MAS agents that have different roles and functions can work together to achieve local as well as global goals. Multi agent systems can be applied to a wide range of domains, like for instance concurrent engineering, electronic commerce, telecommunication, traffic, and in particular manufacturing control (Bussmann, 1998).

2.2.2 Standards and platforms for MAS

The Foundation for Intelligent Physical Agents (FIPA) is a set of standard specifications for the development, communication and coordination of agent-based systems (FIPA, 2002). FIPA was formed in 1996 and its mission was to create software standards for heterogeneous and interacting agents and agent-based systems. The FIPA specifications were built to be used to achieve interoperability between agent-based systems developed by different companies and organisations. The FIPA standards can be divided into the following categories: agent communication, agent management, agent transport, abstract architecture and applications. Of these categories, agent communication is the most important category for the FIPA multi-agent system model. The FIPA agent management

framework and the FIPA-ACL communication language is explained in detail later in section 4.2.2.

Many agent building development environments are available that can be used to create a multi agent system. These include JADE (Bellifemine, et al., 2007), JACK (Winikoff, 2005) and Zeus (Glanzer, et al., 2001).

The Java Agent Development Framework (JADE) is perhaps one of the more widely used platforms for multi-agent system development. JADE was initially developed by the Research & Development department of Telecom Italia s.p.a., but is now a community project and distributed as open source under the LGPL licence. JADE is a completely distributed middleware framework with a flexible infrastructure that allows for easy extension. The JADE framework can be used to develop complete agent-based applications by means of a run-time environment. The run time environment allows the implementation of the life-cycle support features required by agents, the core logic of agents themselves, and a rich suite of graphical tools. JADE is written in Java and thus, it benefits from the large set of language features and third-party libraries that Java provides. Java offers a rich set of programming abstractions which allows developers to construct multi-agent systems with relatively minimal expertise in agent theory. The development of multi agent systems with JADE is discussed in detail in section 4.2.3.

2.2.3 HMS implementation with MAS

Holons and agents are very similar and they poses many of the same characteristics. These characteristics include autonomy, reactivity, pro-activity, social ability, cooperation, rationality, benevolence and mobility.

There are only two differences between a holon and an agent. Firstly, unlike a holon, which can contain other holons, an agent cannot contain other agents. Agents can, however, still be used to form hierarchical structures similar to holarchies. The second difference is that agents are pure software entities, while holons can include both hardware and software components (Babiceanu & Chen, 2006). Agents are still widely considered to be ideal for the implementation of the software component of a holon.

It is almost universally accepted by the HMS consortium that the software part of a holon and holarchies are enabled by agents and multi agent systems. The distributed architecture of multi-agent systems and the agent's characteristics of autonomy and cooperation make MAS a suitable tool for the implementation of the holonic manufacturing concept. (Babiceanu & Chen, 2006)

Brennan and Norrie (2001) noted the similarities between agents and holons and concluded that multi agent systems is a necessary part of HMS implementation. Ulieru et al. (2001) also stated that the multi agent systems paradigm is well suited

to implementing a holonic abstraction of a problem which is fundamentally distributed in nature

Bussmann (1998) argued that agent-oriented techniques can be used to design and implement the information processing part of a holon. Bussmann further stated that when implementing a HMS, the overall manufacturing process should be designed according to the holonic manufacturing paradigm and requirements for the information processing should then be derived from the intended interactions. Bussmann continued by stating that multi agent systems should provide the basic reasoning and cooperation techniques necessary to meet the control requirements and tailor them to the specific needs of holonic manufacturing.

2.3 Human integration in HMS

When it comes to the integration of human workers in HMS, very little detailed work on the subject could be found. There is however many cases where researchers mention that such integration is possible. Researchers often refer to Christensen's work (Christensen, 1994) to show that a human interface is included in his holon architecture. In this case though, Christensen states that the human interface is a control interface used to issue commands and monitor the state of the holon. The literature lacks detailed work on how exactly human integration as a resource is implemented and the most prominent HMS design methodologies do not include methods for human integration.

Bussmann (1998) stated that the process of holon cooperation, in contrast to CIM, also involves humans and that humans are viewed as ordinary resources that show autonomous and cooperative, i.e. holonic behaviour. Bussmann goes on to state that humans can be viewed as resources and that the integration of humans requires a human machine interface at an artificial holon. This suggests that a holon should be created to represent the human interface itself in the HMS.

When comparing the agent approach to the holon approach Leitao (2004) states that In terms of human integration, the human interface is automatically embedded into each holon, while in the agent approach, the human interface is represented by a separated agent. They do not make any further mention of how the interface is implemented.

Babiceanu & Chen (2006) also refer to Christensen's work and state that Christensen developed a broader model of a holon which includes also a human unit functioning as a resource in the same way as the physical processing component, but at the same time, it exchanges information with the environment and can act on the physical processing component just like the software control component.

Giret & Botti (2008) stated that in manufacturing systems, people and computers need to be integrated, with access to required knowledge and information, in order to work together. They go on to state that these requirements are the reason that Christensen added an integrated human interface block to his holon architecture. Each holon must always be able to cooperate with humans whereas in a MAS, human interface is implemented by one or several specialised agents that provide communication services as a whole. Nevertheless, nothing in the agent definition prevents having agents with an integrated human interface block.

Alford et al. (1997) wrote a paper in which they discuss flexible human integration for holonic manufacturing systems through a concept called Human Directed Local Autonomy. Their motivation for integrating humans into a holonic manufacturing system is to take advantage of human intelligence and skill that can be used to interpret robot sensor data, eliminate computationally expensive and error-prone automated analyses and perform trajectory and path planning. They thus focus on integrating humans into robot sensing and motion guidance and coordination. In this role the human is essentially a supervisor that, when requested, can examine sensor data from the robot and directs its movements accordingly using various media including gestures, voice and touch. To implement their test system, Alford et al. used the Intelligent Machine Architecture (IMA) approach that results in a system of concurrently executing software agents. Alford et al. believe that a holonic system can be implemented with IMA, since IMA agents exhibit autonomy and cooperation which are two important characteristics of holons. Alford et al. do not, however, go into the details of the architecture of the human holon, nor do they describe how the information flow takes place.

Kotak et al. wrote a paper that describes a practical system framework for holonic design and operations in a distributed manufacturing environment using multi-agent systems (Kotak, et al., 2003). One of the issues they address is human/system integration. Although in their case, human system integration is mainly used to provide the human user a means to design the system, disturb the system and dynamically communicate with the holonic control system to change system environment. Their human-system integration thus only facilitates human experts' interaction with the system to choose or override the system's holonic solution.

3 Case study and testbed cell description

This thesis uses, as a case study, the control system of a testbed cell that simulates a small part of the electrical circuit breaker manufacturing process of CBI Electric Ltd. The part of the manufacturing process that the testbed cell simulates is the final stage of assembly and quality assurance of the electrical circuit breakers. This case study was chosen because the Mechatronic Automation and Design research group at Stellenbosch University has previously conducted research projects related to CBI Electric's circuit breaker manufacturing process. Various pieces of equipment, as well as product components and knowledge of the process, was therefor available for the development of the testbed cell.

3.1 Assembly and quality assurance of electrical circuit breakers.

3.1.1 Product description

The QA-13 series is a range of miniature circuit breakers produced by CBI Electric. The range consists of a single-pole breaker as well as 2-pole, 3-pole and 4-pole breaker arrays as seen in Figure 5 below.

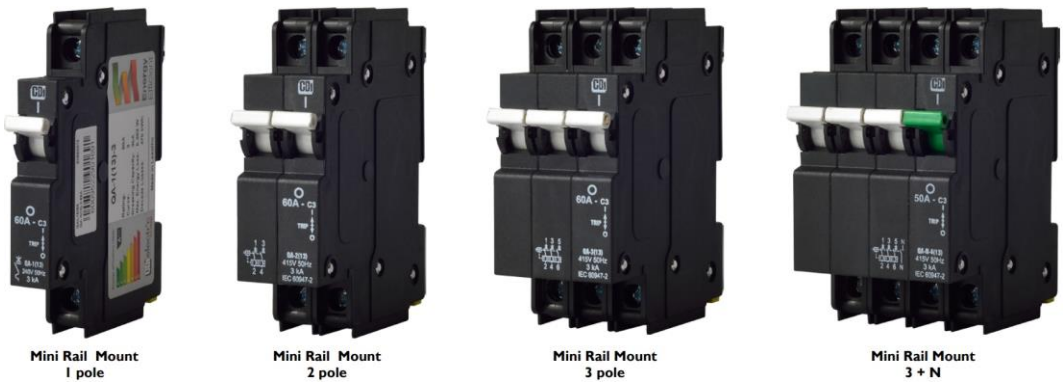


Figure 5: CBI Electric QA-13 Series miniature circuit breakers.

The testbed cell for this case study was designed to be capable of producing all 4 products in the QA-13 range, even though the manufacturing process differs for each product. This was done in order to demonstrate the flexibility of the holonic control system.

3.1.2 Assembly and quality assurance process

The final stages of the process for assembly and quality assurance of the QA-13 range of circuit breakers starts with circuit breakers that are in the state shown in Figure 6. As seen in Figure 6 the complete internal assembly of the circuit breaker has been assembled on the base of the circuit breaker casing.

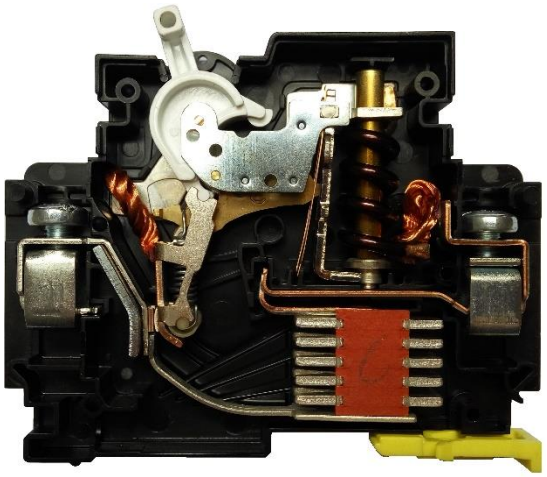


Figure 6: Initial circuit breaker assembly state

This part of the production process ends with the completed and tested product as described in section 3.1.1. The flow diagram in Figure 7 shows the process for the final stages of assembly and quality assurance of the circuit breakers.

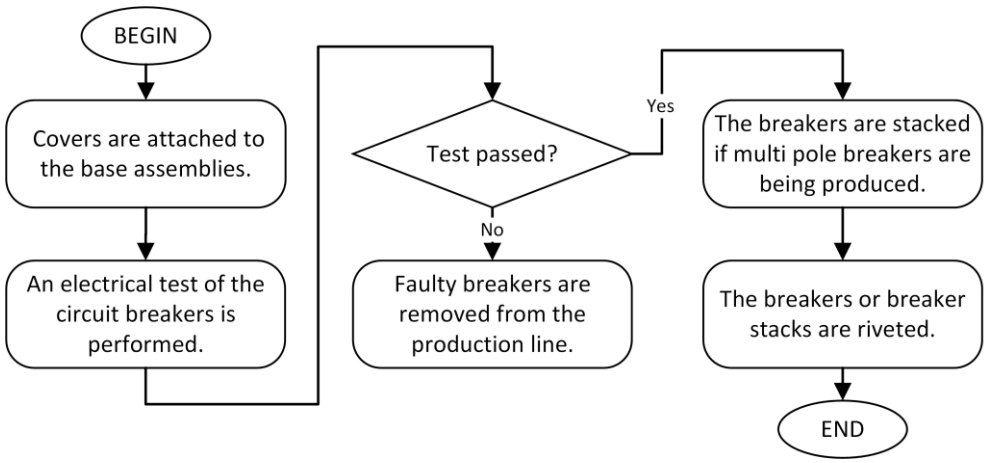


Figure 7: The assembly and quality assurance process of CBI circuit breakers.

3.2 Testbed cell

In this section, the testbed cell, based on the process described in section 3.1, is described.

3.2.1 Testbed manufacturing process

The manufacturing process of the testbed cell is shown as a process flow diagram in Figure 8. During this process the assembly of the circuit breakers are completed and inspected. Each individual breaker is also tested before being stacked and riveted in different configurations to produce any of the products in the range described in section 3.1.1.

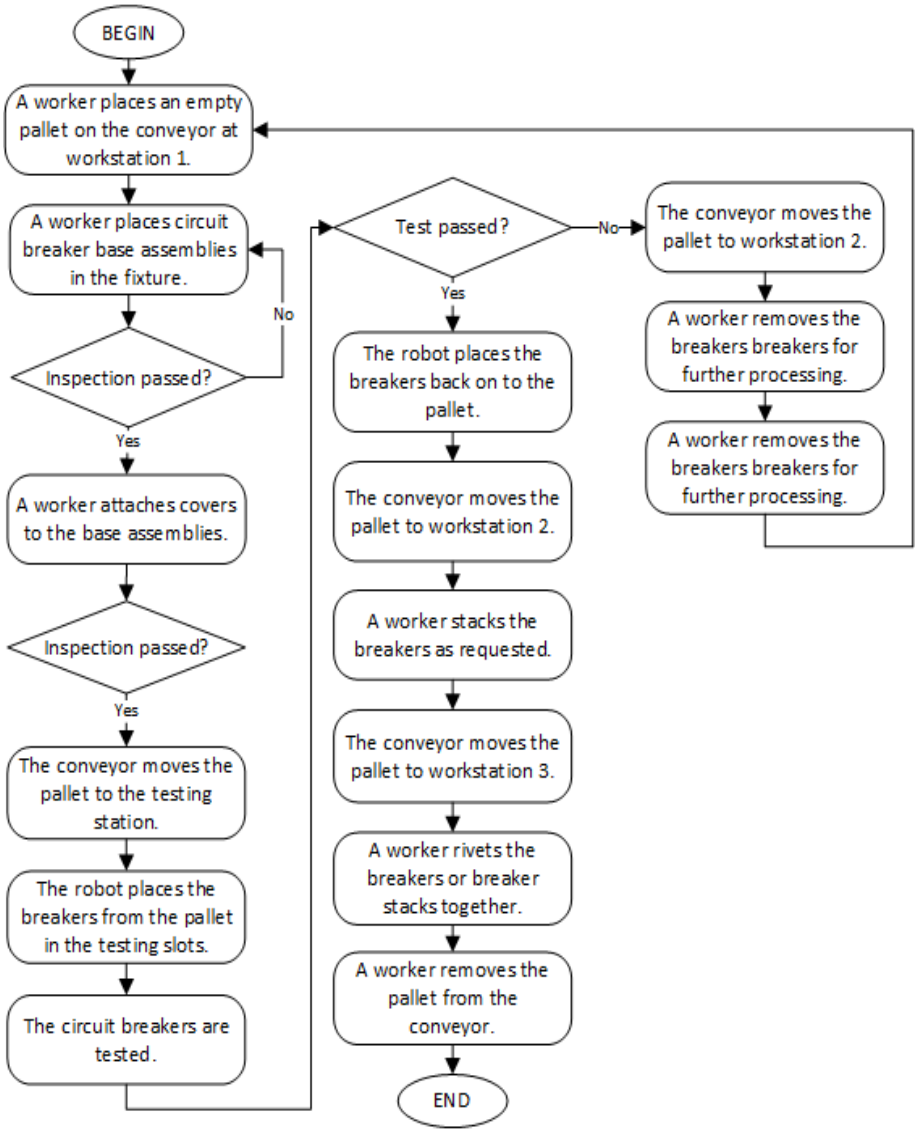


Figure 8 Testbed cell process flow diagram.

3.2.2 Testbed cell architecture

The testbed cell consists of three manual workstations as well as three automated subsystems: the transport subsystem, the machine vision subsystem and the testing subsystem. Figure 9 shows the layout of the cell. A few pictures of the testbed cell can be found in Appendix E.

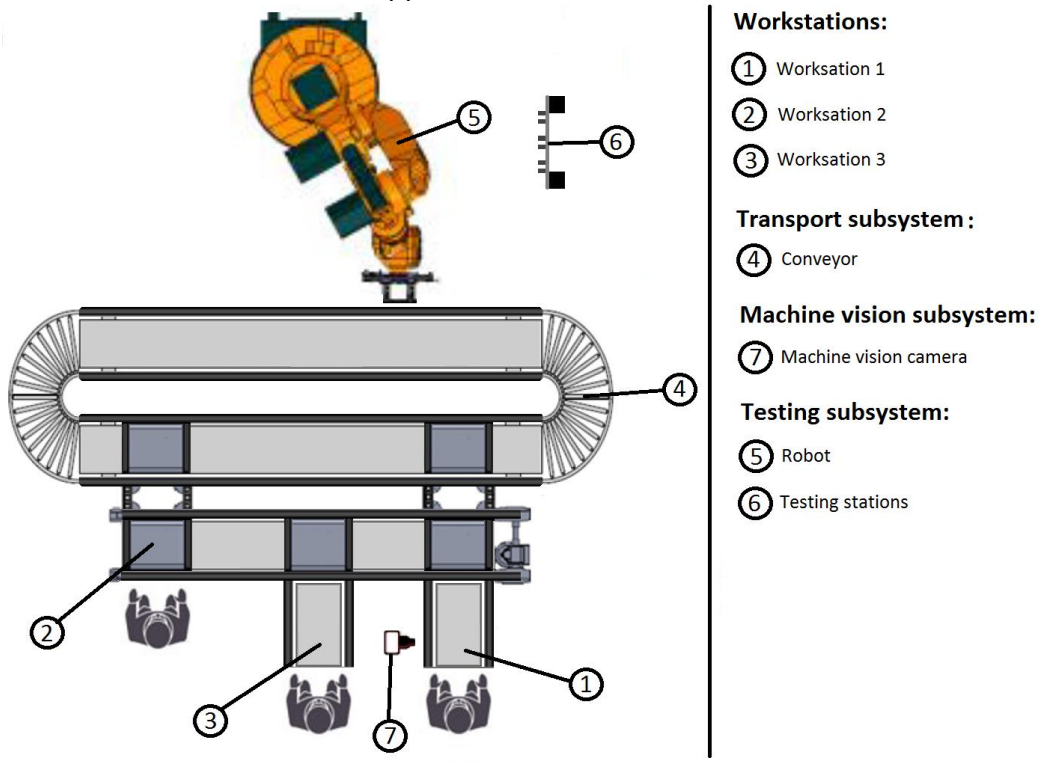


Figure 9: Case study cell layout.

3.2.2.1 Manual workstations

The manual workstations are where human workers complete the manual operations in the manufacturing process. The workstations are positioned along the conveyor so that pallets can stop at the workstations to allow workers to perform operations on the transported breakers.

At workstation 1, a worker places a pallet on the conveyor, places circuit breaker assemblies in the fixture on the pallet and attaches the cover of the casing to the circuit breaker assemblies in the fixture. At workstation 2, a worker stacks the circuit breakers into stacks that form a circuit breaker with the number of poles requested by the cell controller. The worker then inserts a temporary pin to keep the stacks in place as the pallet moves. This operation is not performed if the desired product is a single-pole circuit breaker. At workstation 3, a worker rivets the single circuit breakers or circuit breaker stacks, resulting in the completed products. The products and the pallet are then removed from the conveyor at workstation 3.

3.2.2.2 Transport subsystem

The transport subsystem consists of a modular conveyor that moves pallets between stations as requested by the cell controller. The pallets house a fixture that can hold up to six circuit breaker assemblies. The conveyor is capable of moving multiple pallets simultaneously while, at the same time, holding other pallets at their current workstations as the various operations are performed. All pallets start at workstation 1 and then moves to all the stations along the conveyor until it reaches workstation 3, where the pallet is removed. The conveyor does not allow pallets to overtake one another.

3.2.2.3 Machine vision subsystem

The machine vision subsystem consists of a machine vision camera that performs various inspections of the circuit breakers following the manual operations at workstation 1. The first inspection performed by the camera is to confirm that the correct number of circuit breaker assemblies have been placed in the correct positions on the fixture and that the breaker assembly contains internal parts and is not just an empty casing. The second inspection checks whether or not the top halves of the casings have been correctly placed on all of the circuit breaker assemblies.

3.2.2.4 Testing subsystem

The testing subsystem consists of a 6-DOF robot as well as a simulated testing station with six circuit breaker slots. The Robot picks up the breakers from the pallet on the conveyor and places them in the testing slots where an electrical test is simulated since no actual testing hardware was available to be used.

4 Holonic control implementation

In this section, the holonic control system of the testbed cell, described in section 3, is discussed. First, the mapping of the cell's components to holons is described. Then, the implementation of the higher level control of the HMS is discussed. Finally, the lower level control of the hardware subsystems of the cell is discussed. The integration of human workers in the holonic control system does not form part of this section and is discussed later in sections 5, 6.2 and 6.3.

4.1 Holonic control architecture

The holonic control approach involves the mapping of the hardware and software components of the testbed cell to holons. A holon may consist of only an information processing (software) component or both an information processing and a physical processing (hardware) component. The mapping of holons was done according to the PROSA reference architecture described in section 2.1.5.1. The PROSA reference architecture was chosen because it is the most established reference architecture to date. The various components of the cell were thus mapped to resource, order and staff holons as further described in this section. No product holons were included in this HMS. The reason for this is that functionality of the product holon can be more easily integrated with the functionality of the order holon, in the case of a simple system such as this. The products are also very similar and the process to produce all the products is virtually the same, which means that the implementation of different product holons is unnecessary and would only bring unnecessary complications.

The automated physical resources of the cell were mapped to resource holons. These physical resources include the conveyor, machine vision camera, robot and testing station. All of these resource holons consist of a software and a hardware component. The software component is responsible for inter-holon communication, higher level control of the resource, as well as interfacing with the lower level control of the resource hardware. The HLC and LLC parts of these resource holons are described in detail in sections 4.2 and 4.3 respectively.

Human workers are also mapped to resource holons. This mapping does however depend on the architecture used for human integration. There are also various staff holons, that perform auxiliary functions related to human integration, accompanying the human resource holons as part of the HMS. These holons and their implementation are later discussed in sections 5.3 and 6.1.

Order holons manage the products that are being produced and contains the product state model and all logistical information processing related to the job. In the HMS for the testbed cell, a single order holon is created to manage the products that are to be produced from the circuit breakers on a single pallet.

An order holon keeps track of the state of the products it is responsible for. It also coordinates the actions of the resource holons in the HMS to complete the production process of the products that it is responsible for. The order holon implementation is described in detail in section 4.2.4.2.

4.2 Higher level control

As discussed in section 2.2.3, agents have been proven to be exceptionally well suited for the implementation of the software component of a holon. The holonic control architecture described in section 4.1 was thus implemented as a Multi-agent System (MAS). This MAS serves as the higher level control (HLC) of the testbed cell described in section 3.

The MAS was developed using the JADE platform. This platform was chosen for two reasons:

- JADE has been established as a suitable platform for the implementation of holonic control systems as multi-agent systems (Kotak, et al., 2003); (Giret & Botti, 2008); (Paulucci & Sacile, 2016).
- JADE has been used before to implement similar control systems by members of the Mechatronic Automation Design Research Group at Stellenbosch University. Extensive knowledge concerning JADE implementations was thus available to the author.

In this section, an overview of the MAS is given and agent communication, coordination, and implementation is described. The functionality, communication and implementation of the various agents of the testbed cell's MAS are also described in detail.

4.2.1 System overview

The MAS is based on the holonic control architecture as described in section 4.1. All holons have an information processing component that is responsible for decision making, communication with other holons and interfacing with the LLC of the physical processing part of the holon if required. JADE agents are perfectly suited to act as the information processing part of a holon since they have built in FIPA-ACL communication protocols and all the functionality of Java, that can be used implement decision making and interfaces with LLC software.

The information processing part of each of the holons described in section 4.1, as well as the human integration holons described later in section 5, were mapped to an agent of the same type. All order, resource and staff holons are thus represented by order, resource and staff agents in the MAS.

In addition to the agents that represent the holons of the HMS, other staff agents are required for the practical implementation of the MAS. The coordinator agent launches a GUI on the PC that the MAS is running on. This GUI is used to input production orders for the cell. The coordinator agent launches the order agents that then coordinate with the other holons in the system to produce the required products.

The final structure of the MAS is dependent on the architecture used for human integration. The two final structures of the MAS, that include all the agents for human integration for the two architectures, are described in section 6.2 and 6.3. The MAS structure without the human integration agents is shown in Figure 10.

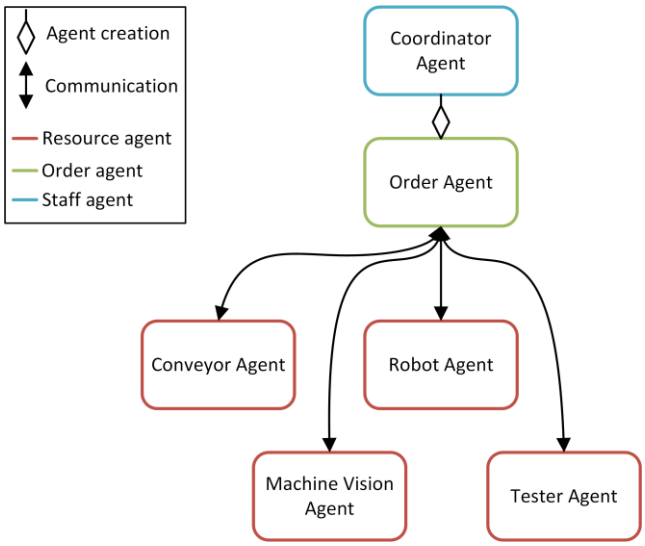


Figure 10: The MAS structure, without the human integration agents.

4.2.2 Agent communication and coordination

JADE uses the FIPA agent communication language (FIPA-ACL) and its protocols. In this section the agent communication and coordination infrastructure of JADE is described, as well as the FIPA-ACL language and its implementation in JADE.

4.2.2.1 FIPA agent management

In addition to communication, the second fundamental aspect of agent systems addressed by the FIPA specifications is agent management. The FIPA agent management framework is a framework within which FIPA agents can exist, operate and be managed. It establishes the logical reference model for the creation, registration, location, communication, migration and operation of agents. This agent management framework consists of the agent platform, directory facilitator and agent management system. (Bellifemine, et al., 2007)

Agent platform

The agent platform (AP) provides the infrastructure in which agents are deployed. It contains the directory facilitator, the agent management system, the agents themselves and any additional support software. A single AP may be spread across multiple computers. The resident agents thus do not have to be co-located on the same host.

Directory Facilitator

The Directory Facilitator (DF) is an agent that maintains a list of all agents that register with it. Any agent in the system can register, with the DF, any service that can be provided by the agent. Other agents in the system can then request the DF for the information regarding agents that can provide a specific service. This then allows the requesting agent to initiate communication with agents that can provide the required service. Agents can register and subsequently de-register from a DF at any time. An AP may support any number of DFs which may register with one another to form federations. (Bellifemine, et al., 2007)

Agent Management System

The Agent Management System (AMS) is a component of an AP that is required by the FIPA specifications. The AMS is responsible for managing the creation and termination of agents and overseeing the migration of agents between AP's and between containers within an AP. Each agent must register with an AMS in order to obtain a FIPA agent identifier (AID) which is then retained by the AMS as a directory of all agents present within the AP. (Bellifemine, et al., 2007)

4.2.2.2 The FIPA-ACL language

FIPA-ACL is considered the most used and studied agent communication language. FIPA-ACL is an agent communication language that is accompanied by a selection of content languages (e.g. FIPA-SL) and a set of key predefined interaction protocols ranging from single message exchange to complex transactions. FIPA-ACL is grounded in speech act theory which states that messages represent actions, or communicative acts, also known as performatives. There are 22 performatives in the FIPA specifications. Some of the most commonly used performatives are *inform*, *request*, *agree*, *not understood*, and *refuse*. (Bellifemine, et al., 2007)

The FIPA-ACL communication protocols make use of ALC messages with certain performatives to facilitate specific types of conversations between agents. Two of these protocols, the request protocol and the contract net protocol, are extensively used in the implementation of the MAS of the testbed cell.

Request protocol

The request protocol allows one agent, the initiator, to request another agent, the responder, to perform an action. The initiator initially sends a *request* message to the responder. The responder then processes the request and makes a decision whether to accept or refuse the request. If the responder accepts the request, an optional *agree* message can be sent to let the initiator know that the requested action will be performed. After the responder has performed the action, it sends either an *inform* or a *failure* message to let the initiator know that the action has been completed, either successfully or unsuccessfully. (Bellifemine, et al., 2007)

Contract net protocol

The contract net protocol describes the case where one agent, the initiator, wishes to have a task performed by one or more other agents, the responders. In most cases there are many agents that are able to perform the task and the initiator must choose one based on a comparison of their respective proposals. Figure 11 shows a process diagram for the contract net protocol.

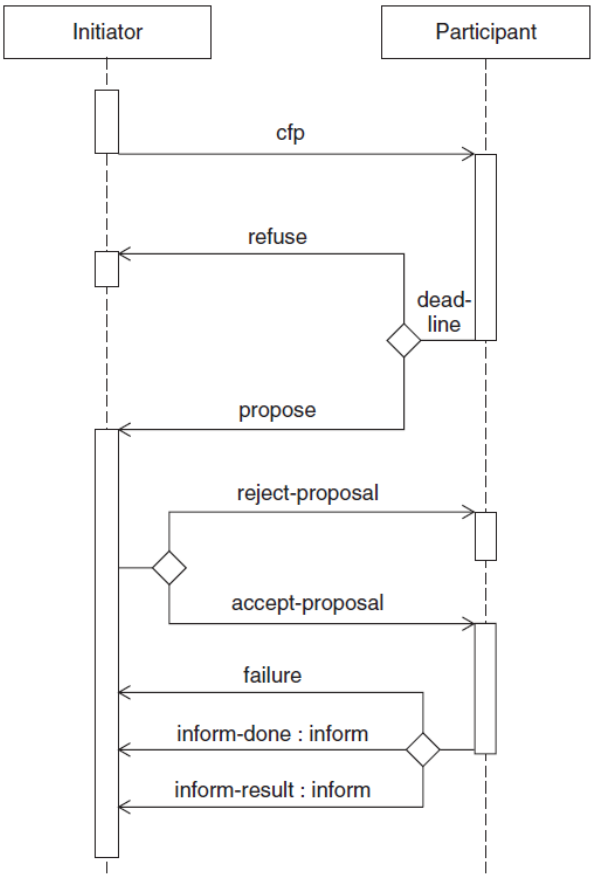


Figure 11: The contract net protocol (Bellifemine, et al., 2007).

The Initiator initially sends a *call for proposal (CFP)* message to the responders - in this case, all the agents that can perform the task. The responders then respond

with either a *propose* or a *refuse* message. A *propose* message is usually accompanied by a proposal value that represents how suited that responder is to perform the task. For example, this proposal value can be a cost or a time required to perform the task. From *x* number of proposals received by the initiator, *y* required responders are selected to perform the action based on their proposal values. The Initiator then communicates an *accept-proposal* to the selected responders and a *reject-proposal* to the rest. After the selected responders have completed the task, they send an *inform* message to let the initiator know that the task has been completed.

4.2.3 Agent development with JADE

The JADE platform can be used to create agents that conform to the FIPA specifications. The functionality of JADE agents is constructed with special JADE classes called behaviours. An agent's behaviours define all the actions and reactions of that agent.

Behaviours can be added to an agent within its *Setup()* method. Each behaviour has two abstract methods. The *action()* method contains the code for the operation performed during the behaviour and the *done()* method returns a boolean that indicates whether or not the behaviour has been completed and is to be removed from the list of scheduled behaviours.

An agent is capable of concurrently executing multiple behaviours. The scheduling of behaviour are, however, not pre-emptive but cooperative. This means that a scheduled behaviour will run until it's *action()* method returns and it is up to the programmer to define when an agent switches from the execution of one behaviour to another. The result of this is that an agent runs in only one thread.

All communication between agents is implemented with special communication behaviours that follow the FIPA-ACL protocols. This section describes the methods and behaviours which are implemented in the agents of the testbed cell's MAS.

The *Setup()* method

The *Setup()* method is the first method that runs when an agent is created. In the *Setup()* method the following operations are typically performed:

- Any arguments that are passed to the agent upon creation are extracted and interpreted.
- The services of the agent is registered with the Directory Facilitator.
- A communication interface is established with any low level control software that is associated with the agent.
- The initial behaviours of the agent are added. This includes all communication responder behaviours.

The *ContractNetInitiator* behaviour

The *ContractNetInitiator* behaviour that allows an agent to initiate a conversation that follows the FIPA-ACL contract net protocol as described in section 4.2.2.2. Some of the methods of this behaviour can be overridden to customize it to perform the intended function.

The behaviour starts by calling the *prepareCfps()* method. This method can be customized and is used to create an ACL message with the *call for proposal* performative. The DF is searched and all agents that can perform the required service are added as receivers of the ALC message. The ACL message is then sent to all the responders to initiate the protocol. The behaviour then waits for all the responders to respond either with a *propose* or a *refuse* message. The behaviour has methods that are called when any messages with performatives appropriate to the protocol are received. These methods can be customized to perform any task if such a message is received. The *handleAllResponses()* method is the most important and is called when all replies have been received. This method contains the code that determines which one of the proposals is to be accepted and generates the *accept proposal* and *reject proposal* messages that are then sent back to the responders. The behaviour then waits to receive *inform* or *failure* messages from the responders. These indicate that the selected responders have completed the requested tasks or failed to do so. The behaviour is then terminated.

The *ContractNetResponder* and *SSContractNetResponder* behaviour

The *ContractNetResponder* behaviour allows an agent to respond to an initiator that has started a conversation that follows the FIPA-ACL contract net protocol as described in section 4.2.2.2.

This behaviour is generally started in the *setup()* method of an agent and constantly listens for incoming *call for proposal* messages. If a *call for proposal* message is received, the *handleCfp()* method is called. This method can be customized and contains the code to generate a *propose* or *refuse* message to send back to the initiator. The behaviour then waits to receive either an *accept proposal* and *reject proposal* message back from the initiator. The behaviour has customisable methods that are called when any messages with performatives appropriate to the protocol are received. The *handelAcceptProposal()* method contains the code to perform the requested action and generate the *inform* or *failure* message that is replied to the initiator. After a single protocol is completed the *ContractNetResponder* behaviour once again listens for the next incoming *call for proposal* message.

The *SSContractNetResponder* is another version of the *ContractNetResponder* with a nearly identical functionality. The *SSContractNetResponder* is a single session version and thus only handles one *call for proposal* message before terminating.

The *AchieveREInitiator* behaviour

The *AchieveREInitiator* behaviour allows an agent to initiate a conversation that follows the FIPA-ACL request protocol as described in section 4.2.2.2. The behaviour starts by calling the *prepareRequest()* method. This method can be customized and is used to create an ACL message with the *request* performative. The directory facilitator is searched for a specific agent that can perform the required action which is then added as a receiver of the ALC message. The ACL message is then sent to a responder to initiate the protocol. The behaviour then waits for the responder to respond either with an *accept* message, indicating that the requested action will be performed, or a *refuse* message indicating that the action will not be performed. If an *accept* was received, the behaviour waits to receive an *inform* or *failure* message from the responder. This indicates that the responder has completed the requested action or failed to perform the action. The behaviour is then terminated.

The *AchieveREResponder* and *SSAchieveREResponder* behaviour

The *AchieveREResponder* behaviour allows an agent to respond to and initiator that has started a conversation that follows the FIPA-ACL request protocol as described in section 4.2.2.2. This behaviour is generally started in the *setup()* method of an agent and constantly listens for incoming *request* messages. If a *request* message is received, the *handleRequest()* method is called. This method can be customized and contains the code to generate an *accept* or *refuse* message to send back to the initiator. The *prepareResultNotification()* method is then called. This method can be customized and contains the code to perform the requested action and generate the *inform* or *failure* message that is then sent back to the initiator. After a single protocol is completed the *AchieveREResponder* behaviour once again listens for the next incoming *request* message.

The *SSAchieveREResponder* is another version of the *AchieveREResponder* with a nearly identical functionality. The *SSAchieveREResponder* is a single session version and thus only handles one *request* message before terminating.

The *SSResponderDispatcher* behaviour

The *SSResponderDispatcher* behaviour is typically added in the *setup()* method of an agent. It constantly listens for incoming ACL messages. If a message fits a pre-defined template, this behaviour then creates a single session responder behaviour to handle the message. The *SSResponderDispatcher* can thus create pre-defined *SSContractNetResponder* or *SSAchieveREResponder* behaviours on demand to handle incoming *call for proposal* or *request* messages respectively.

These behaviours can then execute concurrently. The agent is thus able to facilitate many conversations at the same time.

The *FSMBehaviour* behaviour

The *FSMBehaviour* is used to implement finite state machine (FSM). This FSM can have any number of states. Each state corresponds to a behaviour that is registered to that state. The *FSMBehaviour* provides methods to register these behaviours as FSM states. Each state behaviour returns a value when it is completed. This value is used to determine the next state transition. Methods are also provided to register these state transitions. The FSM behaviour continues to transition between states until it reaches a pre-defined end state, after which the behaviour is terminated.

4.2.4 Agent descriptions

In this section, the agents that are present in the MAS thus far are described in terms of functionality and implementation. Refer to Figure 10 in section 4.2.1 for the MAS structure without the human integration agents. Note that the agents for human integration are later described in section 6.

4.2.4.1 Coordinator agent

The coordinator agent is launched when the MAS is started. In its *setup()* method, the coordinator agents starts a GUI that is used to input production orders. The GUI was implemented using Java Swing components. The GUI has two input fields that needs to be filled to initiate a production order. These fields are the product type (one of 4 possible products described in section 3.1.1) and the quantity. The coordinator agent then launches the required number of order agents needed to fill the production order. When the first order agent of a production order is launched, the coordinator agent logs the start of the production order.

Any number of production orders can be added with the GUI. The coordinator agent has a production order queue and will ensure that only a limited number of order agents are active at any time, this is done to limit the amount of computing resources used by the MAS.

The coordinator agent keeps track of all the active order agents. If an order agent has completed its production process, it sends a message to the coordinator agent using the request protocol. The coordinator agent thus has a *AchieveREResponder* behaviour, that was added in the *setup()* method, to deal with these messages. If such a message is received, the AchieveREResponder behaviour removes the order agent that sent the message from the list of active order agents. This then allows the next order agent in the queue to be launched. If a completion message is received from the last order agent created for a certain production order, the coordinator agent logs the completion of that production order.

The coordinator agent logs the start and completion of production orders by sending a message to the performance tracker agent (later described in section 6.1.1). This is done using an *AchieveREInitiator* behaviour.

4.2.4.2 Order agent

Order agents are launched by the coordinator agent on demand to fill production orders given to the coordinator agent. Each order agent is responsible for one pallet in the testbed cell. Each pallet has six slots for circuit breaker assemblies. This means that one order agent can be responsible for the production of six 1-pole circuit breakers or three 2-pole circuit breakers, etc. It is the responsibility of the coordinator agent to divide a production order up between pallets, i.e. order agents. It is the responsibility of the order agent to coordinate with the other agents in the system and ensure that all the required operations are performed to produce the required products. The code of the order agent can be found in Appendix A.1.

The coordinator agent passes several arguments to an order agent during start up. These arguments contains information regarding the quantity and type of products that the order agent must produce. In the *setup()* method of an order agent, these arguments are saved as variables that modify the process that the order agent follows to produce the required products.

The activities of an order agent is governed by an *FSMBehaviour*, which is added in the *setup()* method. The FSM is set up to follow the production process for the products as shown in Figure 8 in section 3.2.1.

In each state of the FSM, the order agent must communicate with one of the resource agents in the MAS in order to have an operation performed on the circuit breakers on its pallet. A pre-defined *AchieveREInitiator* or *ContractNetInitiator* behaviour is thus registered to every state of the FSM. The FSM waits for the full communication protocol of the behaviour to complete before moving on. The state to which the FSM transitions is dependant type of message received from the responder or the result of the operation contained within the message. Figure 12 shows a state diagram for the FSM of the order holon with all transitions.

For every state of the FSM the same pre-defined *AchieveREInitiator* or *ContractNetInitiator* behaviour is used. At every state, however, different arguments are passed to the new instance of one of these behaviours. The first argument is the service description of the required resource agent that the behaviour must search for in the DF. The second argument is the content of the *request* or *CFP* message that is to be sent. This content contains information regarding operation that is to be performed by the resource agent. This operation information differs for different stages of the process and different product types and quantities. The order agent has methods that generate the message content

accordingly when the product type and quantity is of consequence. These methods would normally form part of a product holon's agent, but in this case it would only have undesirably complicated the system.

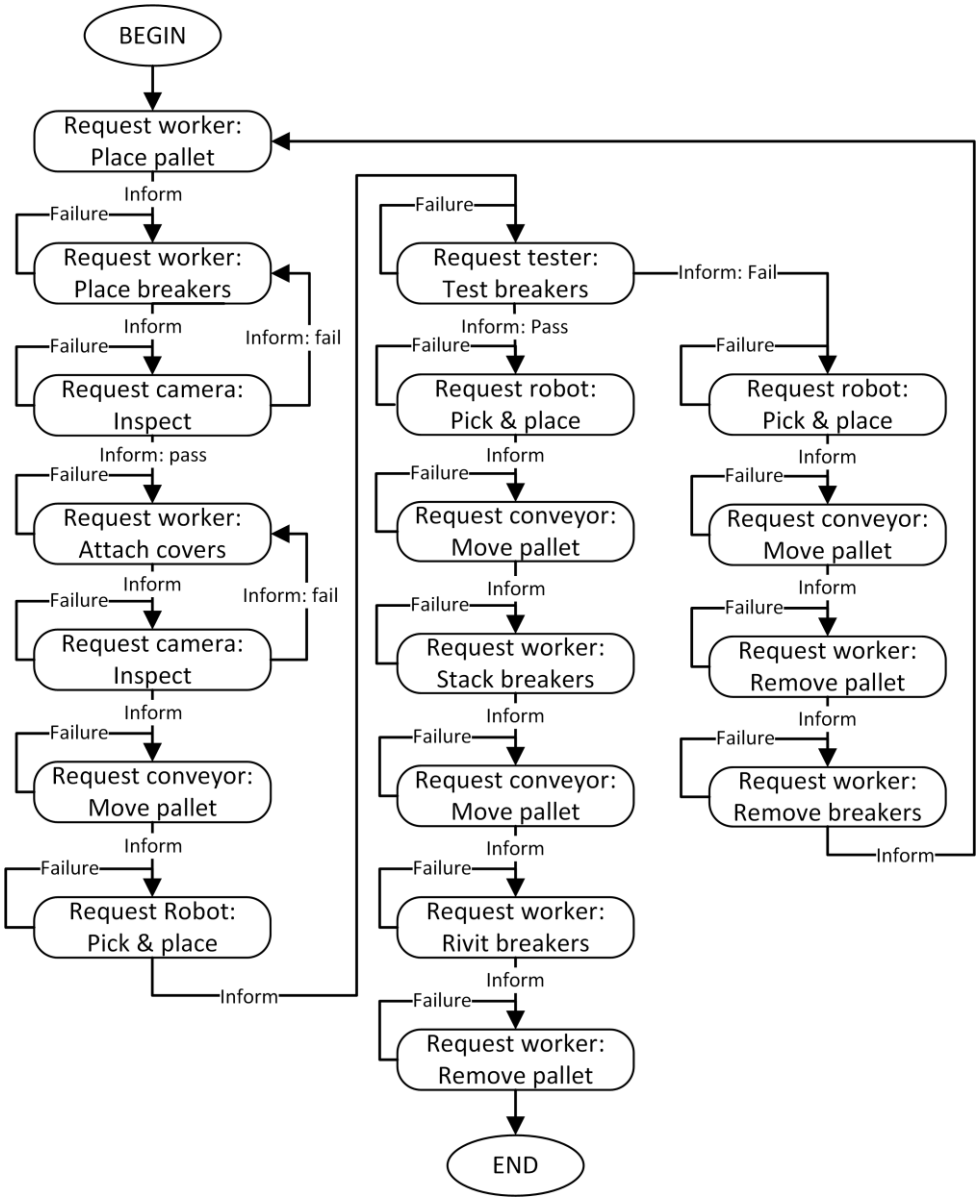


Figure 12: A state diagram for the order agent's FSM.

When the FSM proceeds to its last state, the products that the order agent is responsible for have been successfully completed. The order agent then sends a message to the coordinator agent to confirm the successful completion. This is also done using the pre-defined *AchieveREInitiator* behaviour used for some of the other states.

4.2.4.3 Machine vision agent

The machine vision agent controls a camera that performs inspections on circuit breakers at the request of an order agent. In the *setup()* method, the services of the agent is registered with the DF and an *AchieveREResponder* behaviour is added. This behaviour constantly listens for incoming *request* messages from order agents that require an inspection action to be performed. When a *request* message is received and the *agree* message has been sent back to the initiator, the inspection action is performed. The initial *request* message contains the desired results of the inspection which differs depending on the stage of production and the number of breakers that are supposed to be on the pallet. To continue with the inspection, a TCP-IP socket is opened with the camera and an inspection command is sent. The lower level control of the camera is discussed in section 4.3.2. The camera then replies with the inspection results. The results are then compared with the desired results to determine if the inspection has passed or failed. Finally, an *inform* message containing the inspection result is sent back to the initiator. The result of this inspection, contained in the *inform* message, will determine the next state transition of the initiating order agent.

4.2.4.4 Conveyor agent

The conveyor agent is responsible for controlling the conveyor that moves the pallets between stations along the conveyor. The conveyor agent is capable of moving multiple pallets at the same time as well as maintaining buffers between stations.

In the *setup()* method, the services of the agent is registered with the DF. Four TCP-IP sockets are opened for the four PLC's of the conveyor. Four Java programs that handle the communication between the agent and the PLC's are started in separate threads. An *SSResponderDispatcher* is also added that starts an *SSAchieveREResponder* in a new thread for every *request* message that is received. This enables the conveyor agent to conduct several request protocol conversations concurrently. This is necessary to allow multiple pallets to be moved concurrently. The conveyor agent also maintains action queues for every possible movement to ensure that, if several pallets are waiting in line in a buffer, the correct order agent is informed if the next one in the queue is moved.

When an *SSAchieveREResponder* behaviour is created to handle a *request* message, an *accept* message is immediately sent back to the initiator, which indicates that the action will be performed. The initial request message contains information regarding the starting point and destination of the movement that is required. The first step of performing the movement action is assigning a queue position in the correct queue for this specific movement. The behaviour then waits until it is at the front of the queue before continuing with the movement action. If it is the turn of this movement to be performed, the correct commands to initiate

the movement are sent to the relevant PLC's via the communication programs started in the s*etup()* method. The lower level control of the conveyor is discussed in section 4.3.1. A reply is received from the PLC's when the movement is completed. When these replies are received, an *Inform* message is finally sent to the initiator to indicate that the movement action has been completed.

4.2.4.5 Robot agent

The Robot agent is intended to control a 6-DOF robot that performs a pick and place operation to load circuit breakers form a pallet into testing slots. In the *setup()* method, the services of the agent is registered with the DF and a *AchieveREResponder* behaviour is added. This behaviour constantly listens for incoming *request* messages from order agents that require a pick and place operation to be performed. When a *request* message is received and the *agree* message has been sent back to the initiator, the pick and place operation is performed. The initial *request* message contains the number of circuit breakers on the pallet as well as their positions. To continue with the operation, a TCP-IP socket is opened with the lower level control program of the robot which is discussed later in section 4.3.3. The number of breakers and their positions are then sent over the socket to the robot LLC program. The robot LLC program then replies with a confirmation message when the operation is complete. Finally, an *inform* message sent back to the initiator to confirm the completion of the operation.

The decision was made to simulate the actions of the robot for reasons explained in section 4.3.3. The LLC program for the robot, described in section 4.3.3, just simulates the actions of the robot hardware. If the actions of the robot was not simulated, the LLC program would have controlled the robot hardware and the robot agent would be unchanged.

4.2.4.6 Tester agent

The tester agent is intended to control a testing rig that performs an electrical test on the circuit breakers that were placed into the testing slots by the robot. In the *setup()* method, the services of the agent is registered with the DF and a *AchieveREResponder* behaviour is added. This behaviour constantly listens for incoming *request* messages from order agents that require a testing operation to be performed. When a *request* message is received and the *agree* message has been sent back to the initiator, the testing operation is performed. The initial *request* message contains the number of circuit breakers that are in the slots to be tested. To continue with the test, a TCP-IP socket is opened with the lower level control program of the testing rig which is discussed later in section 4.3.4. The number of breakers in the testing slots are then sent over the socket to the tester LLC program. The tester LLC program then replies with the results of the test when the testing is complete. The results are then compared with the desired results to

determine if the test has passed or failed. Finally, an *inform* message containing the inspection result is sent back to the initiator.

The actions of the tester was also simulated since the hardware to perform the testing was not available for this research. As with the robot, The LLC program for the tester, described in section 4.3.4, just simulates the actions of the tester hardware. If the actions of the tester was not simulated, the LLC program would have controlled the tester hardware and the tester agent would be unchanged.

4.3 Lower level control

Most of the resource holons of the testbed cell described in section 4.1 have a physical processing component that is controlled by the information processing component. The information processing component of these holons takes the form of an agent as described in section 4.2. The physical processing components of these resource holons are the automated hardware. Each of the resource agents interface with some form of lower level control (LLC) software that controls the hardware of their resource. This section describes the LLC of all the automated resources in the testbed cell. The lower level control of human workers is later described in sections 6.2.2 and 6.3.2.

4.3.1 Conveyor LLC

The layout of the conveyor is shown in Figure 13. The conveyor is controlled by four independent PLC's. Each PLC controls a part of the conveyor system as shown in Figure 13.

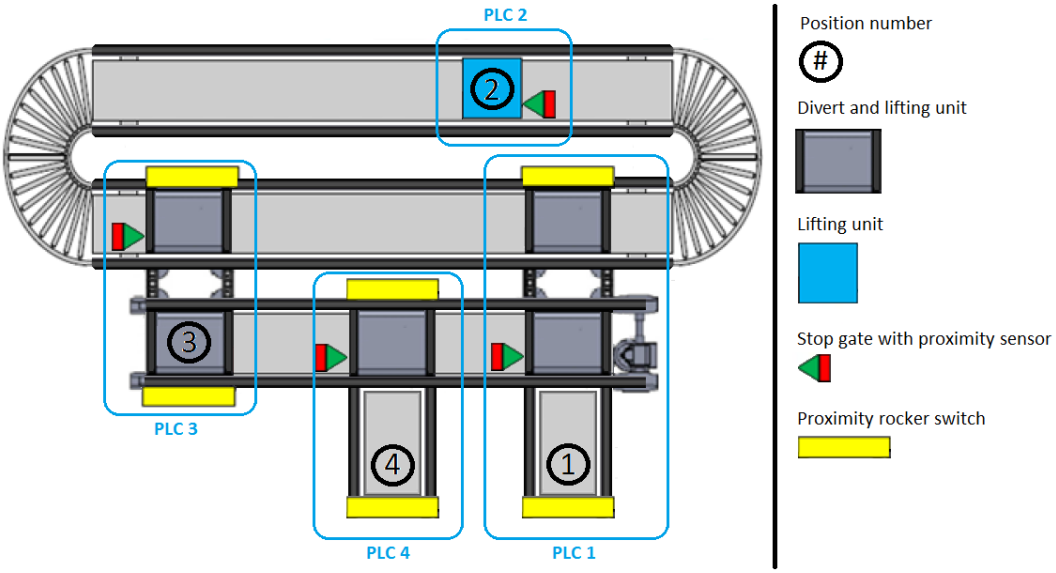


Figure 13: The hardware layout of the conveyor.

Each of the PLC's control access to their respective positions on the conveyor as seen in Figure 13. Each PLC is thus programmed to perform two operations: Accepting a pallet at a position and releasing a pallet from a position. For example, if a pallet starts out at position 1 and needs to be moved to position 2, PLC 1 first releases the pallet from position 1 by controlling the pallet's movement on to the main track. The pallet then moves along the constantly moving main track to the stop gate just before position 2. The pallet stays there until PLC 2 performs the operation to accept a pallet onto the lifting unit at position 2. Using the acceptance and release operations of the four PLC's, the pallet can be moved around the conveyor.

As explained in section 4.2.4.4, the conveyor agent communicates with the four PLC's by means of four separate TCP-IP communication programs. When the conveyor agent requires one of the PLC's to perform an acceptance or release operation, the communication program of that PLC is used to send a single byte as a command to the PLC. The PLC is programmed to save the received byte into memory. Based on the value of this byte, the PLC performs either an acceptance or release operation.

The acceptance and release operations are hard coded on the PLC's. Timers and input from sensors are used to control the actuation sequence of the mechanical components in order to perform the desired movement of the pallet. The PLC code for PLC 3 is given in Appendix B as an example. The code contains a switch-case statement that is based on the value of the command sent to the PLC by the conveyor agent. If the command is 1, the PLC performs an acceptance operation. In the case of PLC 3, the operation starts by lowering the stop gate if the proximity sensor at the stop gate is triggered (meaning there is a pallet there). A series of timers are also started at this point. The continuously moving main track moves the pallet on to the divert and lifting unit. During this time the stop gate is activated once again to stop the next pallet. When the pallet is fully positioned on the divert ant lifting unit it is stopped since the unit is in its middle position and thus does not allow palates to pass. After a certain amount of time has passed to allow the pallet on to the divert and lifting unit, the unit lifts the palate to its high position and the transverse conveyor track is activated. When the pallet reaches position 3, the rocker proximity switch is triggered which then stops the transverse conveyor and lifts the divert and lifting unit under position 3 to its high position. The PLC then returns to its idle state until the next command is received.

The PLC's sends a status byte every 200ms over a TCP-IP socket to a server that is run as part of the communication program of the conveyor agent. After the PLC has completed an operation, the value of this status byte is changed from 0 to 1 for one second. After the communication program of the conveyor agent sends the command to perform the operation to the PLC, it starts to check this status byte every time it is sent. When the value of the byte changes to 1 the conveyor

agent knows that the operation has been completed and continues with its operation.

4.3.2 Camera LLC

The DVT camera used to perform inspections in the testbed cell has on-board image processing capability by means of DVT Intellect inspection control that was set up using DVT Intellect software. A background script program was created to handle the communication with the machine vision agent. This background script also coordinates the camera inspection which is referred to as an inspection product. This product implements several built-in image processing software sensors and a custom foreground script program that determines the inspection result.

The background script runs continuously without interruption from any triggered inspections. The background script creates a TCP-IP server that listens for connection attempts from the machine vision agent. When the machine vision agent requires an inspection to be performed it creates a TCP-IP client that connects to the server of the background script. A byte command is then sent over the TCP-IP socket to the camera. When the background script receives the byte command, the foreground script of the inspection product is triggered to perform the inspection. When the inspection is completed, the foreground script saves the result in memory. It also sets a bit in memory to let the background script know that the inspection is completed. The background script then reads the result from memory and sends it back over the TCP-IP socket to the machine vision agent.

The foreground script was included in the inspection product to generate an inspection result from the softsensor data. Two softsensors (software based image processing sensors) were implemented for every circuit breaker position on the pallet's fixture. The softsensors are simple colour identification sensors that compare the colour in the defined area with a pre-trained reference. With these softsensors, the foreground script can determine whether or not there are breakers in the slots. If there is a breaker in a slot so it can also then determine if they are open and contain the internal parts or if they are closed with the breaker cover. The code for the foreground script can be found in Appendix C.

All the softsensors are shown on an actual inspection image in Figure 14. The larger circular sensors, hereafter referred to as type-1 sensors, are used to determine whether or not there is a circuit breaker in each of the slots. If there is a circuit breaker in a specific slot, the colour obtained by the type-1 sensor of that slot matches the pre-trained blackish colour of that part of the breaker as seen in Figure 14 with the two breakers on the left. If there is no breaker in the slot, the type-1 sensor sees the white circle on the fixture in the same location, which does not match the pre-trained colour as seen in Figure 14 with the two open slots on the right.

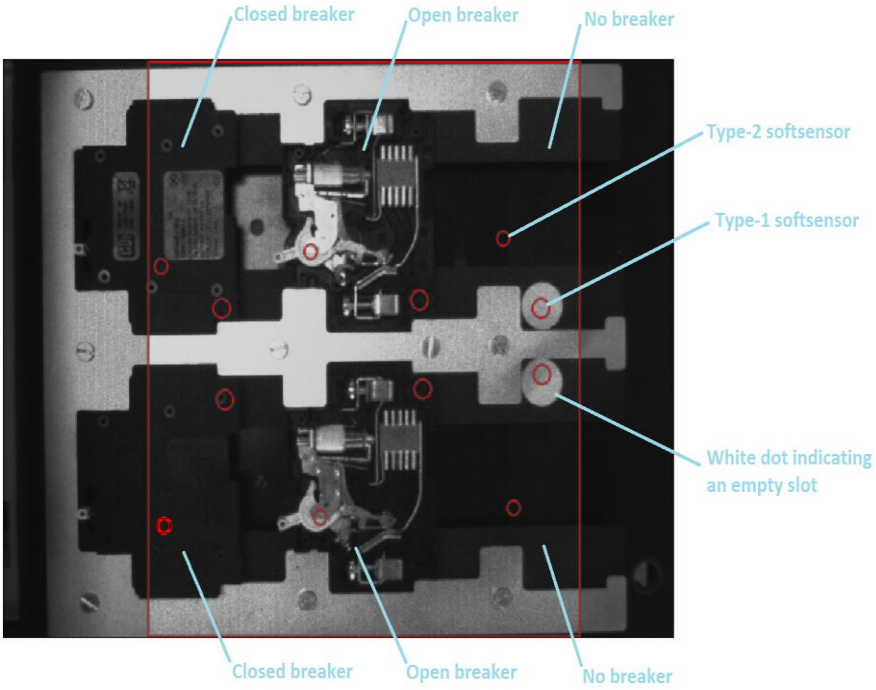


Figure 14: An example of an inspection image with the softsensors shown.

The smaller circular sensors, hereafter referred to as type-2 sensors, are used to determine whether the breakers are open and contain their internal parts, or closed with the breaker shell. The type-2 sensors achieve this by analysing the area where the switch is located in the internal assembly. If it matches the pre-trained colour, the breaker is open and the internal parts are present as seen in Figure 14 with the two open breakers in the centre. If it does not match the pre-trained colour, the breaker is closed with the breaker shell as seen in Figure 14 with the two closed breakers on the left.

Using these softsensors the foreground script determines the result for each breaker slot on the pallet. The results have different meanings based on which of the two inspections was performed. The first inspection in the process takes place after the open breakers have been placed. For this inspection, the result for a slot is *0* if the breaker slot is empty, *1* if the slot contains a breaker which is open and contains internal parts or *2* if the slot contains a breaker with no internal parts. The second inspection in the process takes place after the open breakers have been closed with their covers. For this inspection, the result for a slot is *0* if the breaker slot is empty, *1* if the slot contains a breaker which is open and contains internal parts or *2* if the slot contains a breaker and it is closed with a breaker casing. During the second inspection a closed breaker cannot be confused with an open breaker with no internal parts because the breakers would then not have passed the first inspection, making such a scenario impossible.

After the inspection, the results are sent back to the machine vision agent over the TCP-IP socket. The results can then be compared with the desired results. The results that are desired by the machine vision agent depends on the stage of production and the number of breakers that are supposed to be on the pallet.

4.3.3 Robot LLC

The decision was made to only simulate the LLC and operation of the robot rather than using the real robot for the following two reasons: Firstly, there was only two pallets available with circuit breaker fixtures that have positions that are accurate enough to allow accurate, repeatable operation with the robot. For this research eight new pallet fixtures were made, but these fixtures needed to be cheap and are therefore not accurate enough for use with the robot. Secondly, the LLC implementation of the robot is complicated and time consuming and would not add value to this thesis.

The LLC of the robot was thus implemented as a Java program that the robot agent interfaces with through a TCP-IP socket as it normally would with the real LLC software of the robot. The LLC program simulates the operation of the robot by using a timer that creates a delay which is based on the number of breakers that are being picked and placed. The timer is started when the LLC program receives the command sent from the robot agent. When the time delay is over, a reply is sent back to the robot agent over the socket to confirm the completion of the operation.

4.3.4 Tester LLC

As with the robot LLC, the tester LLC was also simulated, because the hardware to test the circuit breakers was not available. The LLC of the tester was thus implemented as a java program that the tester agent interfaces with through a TCP-IP socket. The LLC program simulates the operation of the tester by using a timer that creates a delay which simulates the time it takes to conduct the test. The timer is started when the LLC program receives the command sent from the tester agent. When the time delay is over, a reply, containing the simulated results, is sent back to the robot agent over the socket. The LLC program can simulate the random failure of circuit breakers during testing. In some cases, as with the experiments described in section 7.3, the failure probability is set to zero in order to minimise variability between experiments.

5 Human integration

In this section, two approaches to integrate human workers as resource holons in holonic manufacturing systems are discussed. Two architectures for human integration are then developed based on these approaches. These architectures include the holon architecture itself, as well as a concept for the human interface.

5.1 Human workers as resource holons

In order to implement holonic control with a manufacturing system that contains human workers, there is a need to somehow integrate humans in the holonic system. One approach is to view human workers as a resources in the system and treat them like any other mechanical/electronic resource. In accordance with the PROSA reference architecture (Van Brussel, et al., 1998), a resource holon can be created to manage a human worker in the system.

Normally, a resource holon interfaces with the low level control system of an automated resource in order to exchange information regarding the tasks that are to be performed by the resource hardware. The difference between a normal resource holon and a human worker resource holon is that the holon cannot control the human worker directly. A human worker is also much more unpredictable than a machine and thus a more robust control mechanism is required to make sure the worker does what is required by the holon. Human workers can also be mobile resources, unlike most machines which are static.

In order to facilitate communication between a holon and a human worker, a bidirectional communication interface is required. This approach allows the holonic system to make use of a human worker in the same way as with any other automated resource.

One of the issues with this approach is how to define the architecture of human resource holons in the HMS. One possibility is to create a holon for every worker, since the PROSA reference architecture suggests that every resource should be represented by a dedicated holon. This will certainly be ideal since every human has a certain skillset and could possibly be utilized at various workstations. The ability of the system to move workers between workstations and perform different tasks at different times can greatly improve the overall productivity and flexibility of the manufacturing system. The challenge with this approach is that for every worker to have their own holon, every human will also need their own mobile communication interface in order to exchange information with their respective holons at all times.

An alternative to the above mentioned approach is to rather create a holon for every workstation where a human could work. Every workstation can then have its own fixed human interface where users can log in to work at that station. In

this way, the HMS can keep track of which workstations are manned and who is manning them. These two approaches are further developed and formulated into complete architectures in the subsequent sections.

5.2 Architectures for human integration

In 1994, Christensen proposed the first general holon architecture (Christensen, 1994). This architecture is still among the most commonly accepted general holon architectures. Christensen's architecture is discussed in section 2.1.4 and Figure 1 in section 2.1.4 shows the main components of this architecture.

Christensen included a human interface component in his general architecture, but this interface was intended for control and supervision purposes. Normally, the physical processing component of a resource holon is a machine or automated subsystem of some kind that is controlled by the physical control component of the holon. In the case of a human worker resource holon, the human worker becomes the physical possessing part of the holon since the worker is the physical resource that is controlled by the holon. A human interface becomes the physical control part of the holon since it is used to control the physical processing part, i.e. the human worker.

Using this modified general holon architecture, as well as the two different approaches of human integration discussed in section 5.1, two human resource holon architectures are proposed in section 5.2.1 and 5.2.2.

5.2.1 Interface holon architecture

Figure 15 shows the first architecture, namely the interface holon architecture (IHA). With this architecture the holon is defined in terms of a fixed human interface located at a workstation. Each workstation is thus represented by a holon. Human workers can log in at the fixed interface to work at the workstation where it is located. Different human workers can work at the workstation but only one at a time. The human workers exchange information with the holon through the fixed human interface.

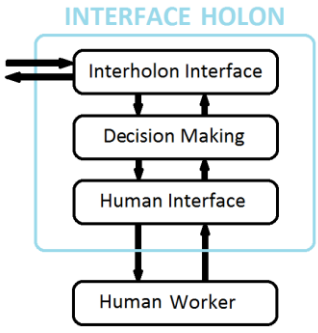


Figure 15: interface holon architecture.

As seen in Figure 15, with this architecture, the human worker is not considered as part of the holon. The human worker in this case is just an interchangeable tool used by the holon to perform tasks by communicating instructions through the holon's human interface. The human workers in the system are thus not directly requested to perform tasks. Rather, a workstation is requested to perform a task and any human that is assigned to work at that workstation then performs the task.

With this architecture, human workers are not directly represented by their own holon and the system can only communicate with human workers when they are logged in at a workstation interface. Human workers on the factory floor thus cannot be directly managed by the HMS. A human supervisor or another external management system is thus still needed to assign workers to workstations and move them between workstations as required.

5.2.2 Worker holon architecture

Figure 16 shows the second architecture, namely the worker holon architecture (WHA). With this architecture, the holon is defined in terms of the individual human worker. Every human worker is represented and managed by a dedicated holon. As seen in Figure 16, unlike with the IHA, the human worker is considered to be a part of the holon, since the worker acts as the physical processing part of the holon as defined by Christensen's general architecture.

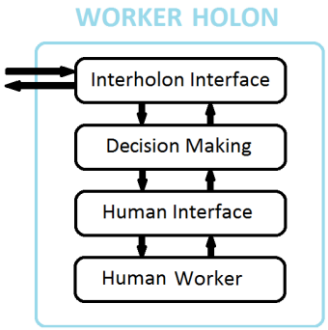


Figure 16: Worker holon architecture.

With this architecture, every human worker requires their own interface in order to exchange information with their respective holons. This approach requires the interface to be mobile so that it can move around the manufacturing environment along with the human worker. The interface will therefore have to be a wearable or mobile device of some kind that communicates wirelessly with the holonic control system.

This architecture is the most compatible with the PROSA reference architecture since the human resource is directly represented by a dedicated holon. With this architecture, the holonic system can directly communicate with the human

workers through their personal interfaces. This direct communication is not possible with the IHA, where the holonic system can only communicate with a worker if that worker is logged in at a fixed interface. The holonic system can thus directly request a human worker to perform any task suitable for their skillset, even if the worker is not currently at a workstation.

Since every worker has their own interface, certain customizations can be made to the interface for a specific worker. These customizations can include different communication languages and communication methods (audio, visual, text, etc.). Sensors on the mobile interface device can also potentially track the working conditions and physical location of the worker. This information can be used by staff holons as is described in section 5.3.2.

The mobility of the human workers and their dedicated interfaces means that human workers can be utilized at several workstations and perform multiple tasks, depending on their individual skillset. This allows the system to automatically move workers between workstations as they are needed. For this architecture, no human supervision/management is thus needed on the factory floor since the human resource management is automatically done by the HMS. This could result in a more productive and flexible system that allows for the formulation and execution of optimized production plans. The eliminated need for human supervision will also reduce personnel expenses. The fact that the holonic system can move workers around as they are needed can also decrease repetitiveness in the workers' day to day activities.

5.3 Staff holons for human integration

The integration of human workers in an HMS creates the opportunity to also automate certain managerial tasks. The holonic system can take over these tasks which are traditionally performed by human managers and supervisors or dedicated to external systems. These managerial tasks are required because, unlike machines, humans do not work around the clock, they can make mistakes and their individual performance varies. These human traits can jeopardise the overall productivity of the system.

In the case of the WHA, the management of human workers on the factory floor can be done by the HMS. There are also other managerial tasks that can be performed by the HMS. These include tracking the performance and monitoring the safety of the human workers.

Staff holons can be implemented to perform these functions and other similar functions are required in a specific scenario. Two staff holons that can improve human integration are proposed in sections 5.3.1 and 5.3.2.

5.3.1 Performance tracker holon

The performance tracking staff holon can collect performance data from the human holons and then record and analyse the data to determine whether workers are meeting certain performance standards. Examples of the performance data that can be recorded include operation execution times, frequency of mistakes and severity of mistakes. A performance tracker holon can also keep track of the amount of hours that a certain worker is working to ensure the worker does not become fatigued and start to make mistakes. This data can also be used to replace time card systems that keep track of working hours and monitor the punctuality of workers. Records of active working time and break times can be used to monitor the working hours of the workers.

In some cases a worker may perform a certain operation better than others. The performance tracker holon can assign a proficiency score to every worker for each operation. This score can be evaluated to determine the best choice when the system is selecting a worker to perform a specific operation.

The performance tracker holon can keep track of which operations the worker performs and calculate a repetitiveness score for every task, which is based on the number of times a certain operation is performed in relation to others. This score can also be evaluated when a human resource is chosen to perform a task which will result in less repetitive work. Performing different tasks rather than one repetitive task may motivate the worker and reduce the chance of mental or physical health problems.

There are many possibilities surrounding the recording and analysis of performance data. The data that is recorded, the methods of analysis and the implications of the results will differ for every manufacturing system. This concept can however be a very effective, automated solution for managing personnel and recording performance data. A human supervisor can only be at one place at a time, but a system such as this can record all data on the factory floor simultaneously. This can make it much more effective than a human supervisor in some cases. Another possibility for utilizing recorded performance data is to supply it to a human supervisor. Based on the data, the human supervisor can then make managerial decisions and affect changes to the system.

The primary purpose of the performance tracker holon in the context of this thesis is, however, just to record performance data for the human workers. This data is used to calculate performance measures which are used to evaluate the two architectures for human integration. The implementation of the performance tracking holon is described in section 6.1.1 and the data that is recorded is described in section 7.2.

5.3.2 Safety monitor holon

Humans, unlike machines, are not expendable and cannot work in harsh environments. There is therefor also a need for the HMS to keep track of the working conditions and the safety of workers.

A safety monitor staff holon can be created to keep track of working conditions through sensors placed at the human interfaces. These sensors can potentially include temperature, air quality and noise sensors. The safety monitor holon can record and analyse the sensor data and send warnings to the relevant human holon interface if working conditions are unsafe. The safety monitor can also react to emergency situations declared by human workers by means of their interfaces. The safety monitor can take the appropriate actions when and emergency is declared. These actions can include issuing a warning to all workers or bringing the entire manufacturing system to a complete halt by issuing stop orders to all the holons in the system. In the case of an emergency, the safety monitor holon could even alert the appropriate emergency services (police, ambulance or fire brigade).

The primary purpose of the safety monitor holon in the context of this thesis is, however, much simpler. Interface devices with sensors capable of monitoring working conditions was not available for this research and the development of such devices are beyond the scope of this research. The sole function of the safety monitor agent in this case, is thus to respond to emergency declarations from the human workers and stop all operations of the system as a precaution.

6 Implementation of human integration

In this section, the implementation of the human integration holons are discussed. The agent implementation for the HLC of the holons are described. The LLC of the workers for both architectures are also described.

6.1 Staff holon implementations

The HMS of the testbed cell requires certain functions to be performed that that do not form part of the functionality of the basic holons of the RPOSA reference architecture. These functions include tracking the performance and monitoring the safety of the human workers in the cell as described in section 5.3. These functions were mapped to their own respective staff holons in the HMS.

6.1.1 Performance tracker agent implementation

The performance tracker holon consists of only an information processing part which is represented by a staff agent called the performance tracker agent.

The performance tracker agent is launched when the MAS is started. The purpose of the performance tracker agent is to record performance data relevant to the human workers in the testbed cell and is identical for both the IHA and the WHA. The functions of the performance tracker is described in section 5.3.1, and the data that is recorded in this case, is later described in section 7.2. The JADE implementation of the performance tracker agent is described below.

In the *setup()* method of the performance tracker agent, an *AchieveREResponder* behaviour is added. This behaviour constantly listens for incoming *request* messages from agents that require the recording of an event. After the behaviour sends an *agree* message back to the initiator, the recording action is performed. The content of the initial *request* message contains the type of event that is to be recorded as well as information regarding the worker represented by the initiating agent. The event is recorded by saving this information, as well as the current time, in a series of array variables. All the events that can be recorded and the data that is sent to the performance tracker agent in each case is explained in detail in section 7.2.

An *inform* message is sent back to the initiator once the recording action has been completed. When the performance tracker agent is shut down after a session, all the recorded data is exported to a series of .csv files for analysis. An example of recorded data can be found in the sample data in Appendix D.

6.1.2 Safety monitor agent implementation

The safety monitor holon consists of only an information processing part which is represented by a staff agent called the safety monitor agent. The safety monitor agent is launched when the MAS is started. The purpose of the safety monitor agent is to monitor the safety of the human workers in the testbed cell. The functions of the safety monitor is described in section 5.3.2. The JADE implementation of the safety monitor agent is described below.

In the *setup()* method of the safety monitor agent, an *AchieveREResponder* behaviour is added. This behaviour constantly listens for incoming *request* messages from agents that wish to declare an emergency. After the behaviour sends an *agree* back to the initiator, the safety monitor agent starts a new AchieveREInitiator behaviour that sends a message to all agents in the system that contains an *all stop* command. This command is interpreted by the agents which then cease all operations.

6.2 Interface holon architecture implementation

In this section the HLC and LLC implementation of the Interface holon (described in section 5.2.1) is described.

6.2.1 Interface holon higher level control

The information processing part of the Interface holon was mapped to a resource agent called the interface agent. The code of the interface agent can be found in Appendix A.2. Interface agents are created to represent fixed workstation interfaces in the testbed cell. These agents form part of the MAS that serves as the HLC of the testbed cell as described in section 4.2.

A staff agent, called the interface registration agent, is required in the MAS when the IHA is used. It is responsible for creating an Interface agent for every workstation interface that attempts to register with the MAS. Using the interface registration agent ensures that workstation interfaces can be dynamically added and removed from the MAS as required. When the system is started, all interface must initially be registered with the MAS.

Figure 17 below shows the complete structure of the MAS when the IHA is used. The performance tracker and safety monitor agents are now included as well as the interface registration agent.

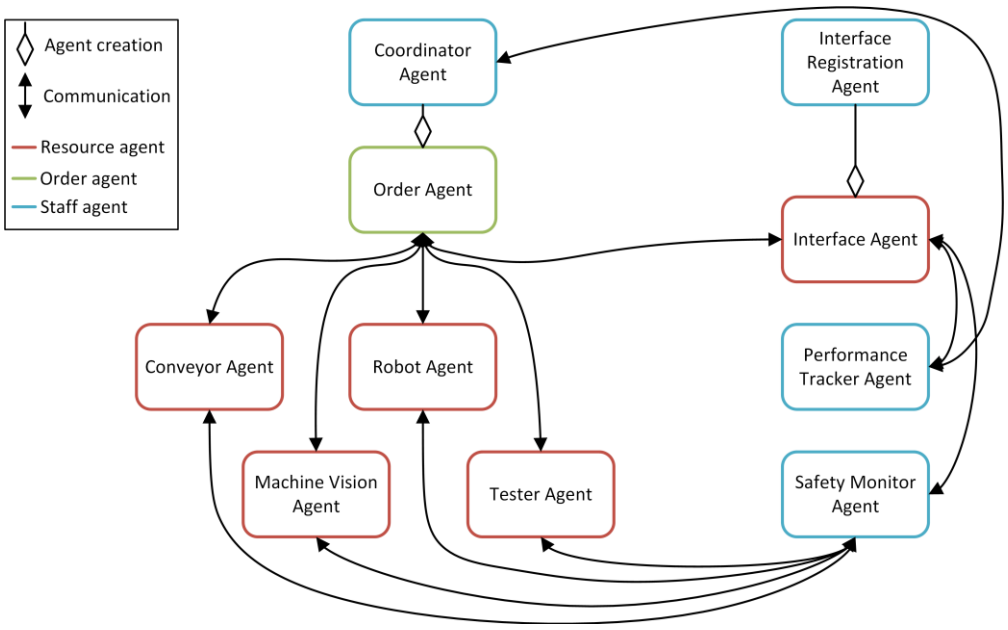


Figure 17: The MAS structure for The Interface holon architecture.

As seen in Figure 17, the order agent directly communicates with the Interface agent that represents a fixed interface at a specific workstation. It is the responsibility of a human supervisor to assign a worker to that workstation in order to complete the requested operation.

6.2.1.1 Interface registration agent description

The interface registration agent is launched when the system is started and if the IHA is being used for human integration. The function of this agent is to launch interface agents that represent fixed interfaces at manual workstations.

When the interface registration agent is started, in the *setup()* method, a TCP-IP server program is started in a separate thread. This server listens for connection attempts from human interface applications that are run on the fixed interface devices at the workstations of the cell. When attempting to register an interface with the MAS, a workstation ID number is entered using the GUI of the human interface application (later described in section 6.2.2). The human interface application then sends the workstation ID along with the IP address of the interface device to the server of the interface registration agent. The interface registration agent then launches an Interface agent (described in section 6.2.1.2) to represent the Interface in the MAS.

When the Interface agent is launched, the workstation ID and IP address of the interface device, as well as the services of that workstation, is passed to the new Interface agent as arguments. The workstation ID and the IP address of the interface devices allows the Interface agent to communicate with its

corresponding Interface device. The services of the workstation are extracted from a text file that serves as an information database for the interface registration agent. This file contains a list of all the workstations of the system as well as the services that can be performed at the workstations.

6.2.1.2 Interface agent description

An interface agent is launched by the interface registration agent when an interface application registers with the MAS. An interface agent represents a fixed human interface at a workstation in the MAS. It is also responsible interfacing with the LLC, which in this case is a device that runs the human interface application at a workstation.

In the *setup()* method of the interface agent, three operations are performed. Firstly, the services that can be performed at the workstation are registered with the DF. These services are passed to the interface agent when it is created by the interface registration agent.

Secondly, In the *setup()* method of the interface agent, a TCP-IP Socket is opened with the interface device using the IP address that was passed to the interface agent when it was created by the interface registration agent. A listener program that listens for incoming communications over the socket is then started in a new thread. If a communication is received over the socket, the *messageRecieved()* method of the interface agent is called to handle the message and take the appropriate actions.

Thirdly, in the *setup()* method of the interface agent, an *SSResponderDispatcher* behaviour is added that starts a new SS*ContractNetResponder* behaviour for every *CFP* message that is received by the agent. This SS*ContractNetResponder* behaviour then handles the *CFP* message received from the order agent that is requesting a manual operation to be performed at the workstation represented by the agent. If a worker is currently logged in at the interface of that workstation, the behaviour sends a *propose* message containing a proposal score to the order agent. If there is no worker logged in at the workstation, a refuse will be sent back to the order agent. The proposal score can be calculated based on any measure, for example a time that a resource expects to take to perform a task. In this case however, the proposal score is meaningless since there is only one workstation in the testbed cell where a certain operation can be performed. Using contract net protocol does however allow for the expansion of the system by adding more workstations that can perform an operation without a need to change the agent code.

If the proposal is accepted, the agent receives an *accept* message from the initiating order agent. The requested operation is then performed by sending the operation instructions to the human interface device via the TCP-IP socket. When all the instructions have been completed, a confirmation message from the interface device is received by the interface agent. Then finally, an *inform* message is sent back to the order agent and the SS*ContractNetResponder* behaviour is terminated.

When a worker that is currently logged in at the workstation logs out, a log out message from the interface device is received by the interface agent. The interface agent then de-registers itself form the DF so that it cannot receive any requests until a worker logs in at the workstation again.

6.2.2 Interface holon lower level control

With the IHA, the interface holon described in section 5.2.1 is used to integrate human workers in the HMS. The interface holon consists of an interface agent as well as a fixed human interface that serves as the LLC of a worker. The human interface is used to facilitate communication between the interface agent and the worker that is assigned to the workstation at any given time.

The human interface consists of a networked device that runs an interface application with a GUI. The device that was chosen to run the interface application is a Raspberry Pi 3 Model B. The Raspberry Pi was chosen because it is cheaper and smaller than a conventional PC and yet has many of the same capabilities. The Raspberry Pi also has Wi-Fi capabilities that allows network communication without the need to lay Ethernet cables to all of the workstations.

As an input device, a QUERTY keyboard with a touch pad was chosen. The QUERTY keyboard is necessary to input data when logging in and the touchpad is used to navigate the GUI. The touch pad is used instead of a conventional mouse since no large flat surface, needed to use a mouse, is available at the workstations. Both the keyboard and the touchpad are also necessary for the general operation of the Raspberry Pi, for example connecting to the Wi-Fi network and launching the interface application. As an output device, an LCD screen is used. The screen and the keyboard is mounted on a bracket that was designed to attach to the conveyor at the workstations. Figure 18 shows the complete interface at one of the workstations.

Figure 18: A fixed human interface at a workstation.

The interface application was developed in Java and is responsible for relaying information between the worker and the interface agent. When the application is started, the interface must first be registered with the MAS. A simple GUI seen in Figure 19 is used to enter the ID of the workstation where the application was started.

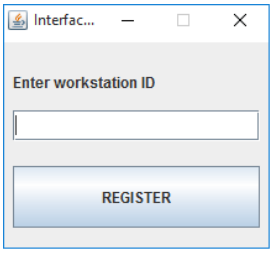


Figure 19: The interface registration GUI.

A TCP-IP socket is then opened with the PC that the MAS is running on and a connection is made with the server of the interface registration agent. The workstation ID that was entered, as well as the IP address of the Raspberry Pi, is sent to the interface registration agent over this socket. An interface agent is then created to represent the interface as described in section 6.2.1.1. When it is launched, the interface agent opens a socket with a server on the Raspberry Pi as described in section 6.2.1.2. This socket is then used for two-way communication between the LLC interface application and the interface agent.

After the interface has been registered with the MAS, the main GUI of the interface application is started. Figure 20 shows the interface GUI with the menu panel on the left and the switch user screen on the right.

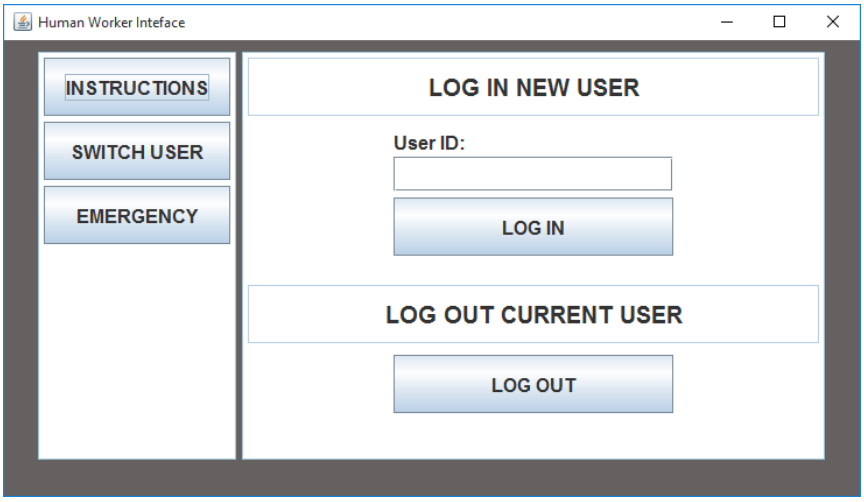


Figure 20: The Interface GUI with the switch user screen.

The panel on the right changes according to the selected menu button on the left. When the main GUI is started the switch user screen is automatically shown since a worker must first log in before the interface can be used. A worker logs in simply by entering their own unique worker ID. A message that contains the worker ID is then sent to the interface agent over the TCP-IP socket. This is to inform the interface agent that a worker has logged in. The interface agent then logs the start of the worker's session with the performance tracker agent. After a worker has logged in, the standby screen (as seen in Figure 21) is shown until instructions are received or the worker manually goes to another screen using the menu buttons.

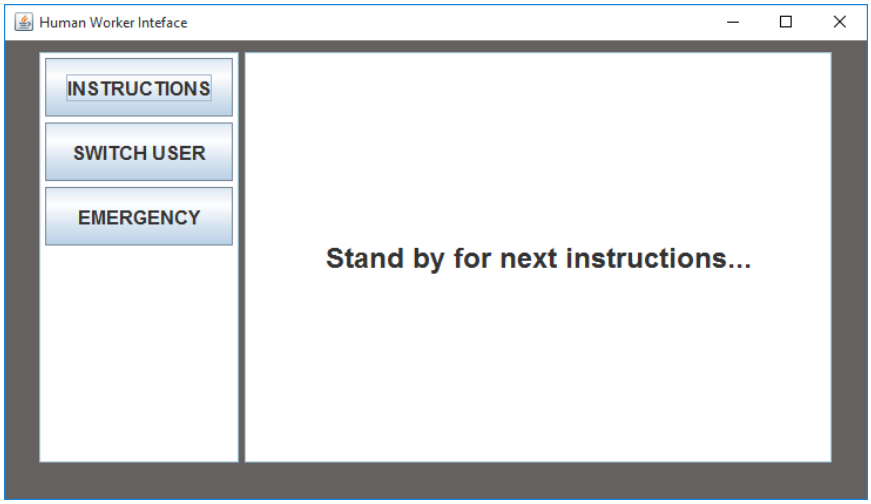


Figure 21: The Interface GUI with the stand by screen.

When the interface agent requires a manual operation to be performed at its workstation and a worker is currently logged in, it sends a series of instructions to the interface application over the TCP-IP socket. When the instructions are received by the interface application, it automatically transitions to the instruction screen and displays the first instruction in the set as seen in Figure 22.

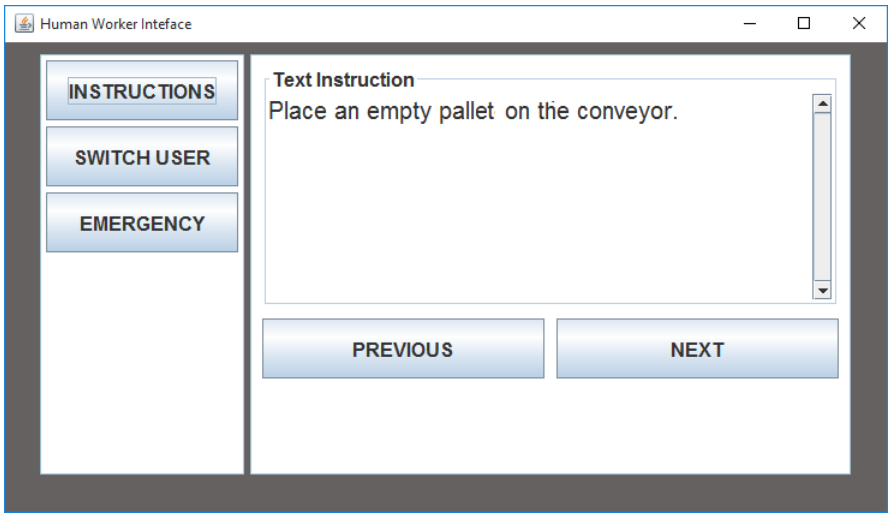


Figure 22: The interface GUI with the instructions screen.

After an instruction has been completed, the worker clicks the *next* button to display the next instruction in the set. The *previous* button can be clicked to go back to the previous instruction if needed. When the instruction set has been completed, the interface application sends a confirmation message over the TCP-IP socket back to the interface agent which then resumes its operations. The standby screen is then be displayed once again until the next instruction set is received from the interface agent.

When the human supervisor orders the worker to move to another workstation, the worker needs to take a break or it is the end of the shift, the worker needs to log out. This is done by going to the switch user screen via the menu button and clicking on the logout button. When a worker logs out, the interface application sends a message over the TCP-IP socket to inform the interface agent which then takes the appropriate actions as described in section 6.2.1.2.

6.3 Worker holon architecture implementation

In this section the HLC and LLC implementation of the worker holon (described in section 5.2.2) is described.

6.3.1 Worker holon higher level control

The information processing part of the worker holon was mapped to a resource agent called the worker agent. The code of the worker agent can be found in Appendix A.3. Worker agents are created to represent individual workers in the testbed cell. These agents form part of the MAS that serves as the HLC of the testbed cell as described in section 4.2.

A staff agent, called the worker registration agent, is required in the MAS when the WHA is used. It is responsible for creating a worker agent for every worker that attempts to register with the MAS through their personal mobile interface. Using the worker registration agent ensures that worker can be dynamically added and removed from the MAS as required.

With the WHA, human workers are directly managed by the MAS rather than by a human supervisor. An additional staff agent, called the supervisor agent, is thus required in the MAS to manage the workers. The code of the supervisor agent can be found in Appendix A.4. The supervisor agent is responsible for assigning requests to workers and moving them between workstations as needed. Figure 17 below shows the complete structure of the MAS when the WHA is used. The performance tracker and safety monitor agents are now included as well as the worker registration agent and the supervisor agent.

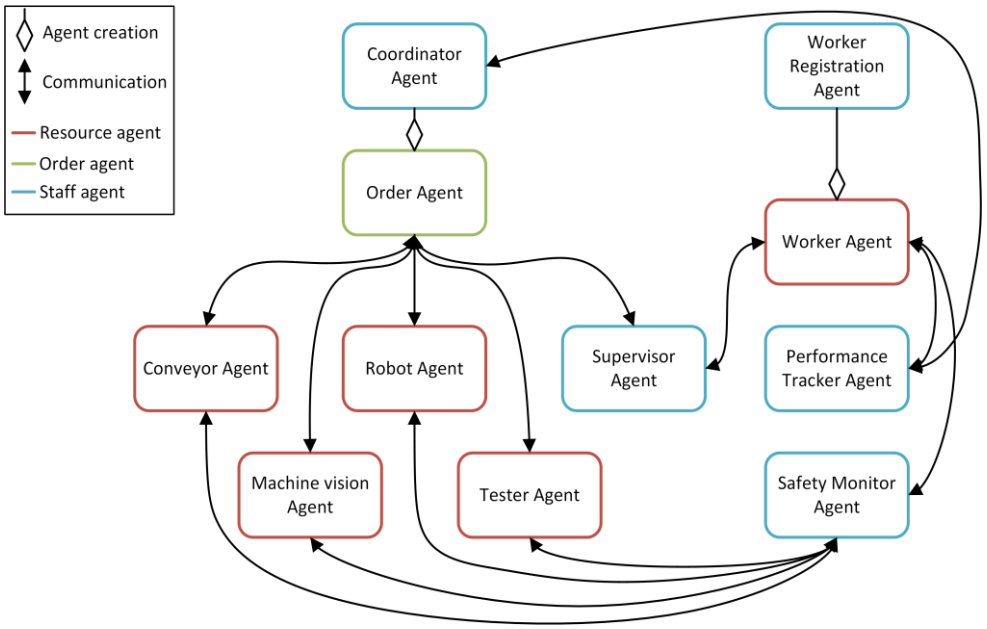


Figure 23: The structure of the MAS for The Worker holon architecture.

As seen in Figure 23, the order agent sends all requests aimed at workers to the supervisor agent. The supervisor agent then delegates these tasks to available workers and moves them between workstations as needed. The worker registration agent is also present as previously discussed in this section.

6.3.1.1 Worker registration agent

The worker registration agent is launched when the system is started and the WHA is being used for human integration. The function of this agent is to launch worker agents to represent active workers in the system.

When the worker registration agent is started, in the *setup()* method, a TCP-IP server is started in a separate thread. This server listens for connection attempts from human interface applications that run on mobile interface devices. In order for a worker to register themself as an active resource in the system, the worker must log in on their personal interface device. The worker enters a worker ID number and a password using the GUI of the interface application. The interface application then sends the worker ID along with the IP address of the mobile interface device to the server of the worker registration agent. The worker registration agent has access to a text file that serves as an information database. This file contains information regarding all the workers, including their worker ID's, passwords and services that they can perform. When the worker ID and password of the worker attempting to log in is received, the file is checked to authenticate the login information of the worker. If the login information is authentic, a worker agent (as described in section 6.3.1.2) is launched to represent the worker in the MAS. When the worker agent is launched, the worker ID and IP address of the interface device as well as the services of that worker is passed to the new worker agent as arguments. The IP address of the mobile interface device allows the worker agent to communicate with its corresponding mobile interface device. The interface application is described in detail in section 6.3.2.

6.3.1.2 Worker agent

A worker agent is launched by the worker registration agent when a mobile interface device attempts to register with the MAS. A worker agent represents a human worker in the MAS. It is also responsible interfacing with the LLC, which in this case is a mobile device that runs the human interface application described in section 6.3.2.

In the *setup()* method of the interface agent, three operations are performed. Firstly, the services that can be performed by the worker are registered with the DF. These services are extracted from a text file and passed to the worker agent when it is created by the worker registration agent. Secondly, In the *setup()* method of the worker agent, A TCP-IP socket is opened with the mobile interface device using the IP address that was passed to the worker agent when it was

created by the worker registration agent. A listener program for incoming communications over this socket is then started in a new thread. If a message is received over the socket, the *messageRecieved()* method of the interface agent is called to handle the message and take the appropriate action.

When using The WHA, a supervisor agent (described in section 6.3.1.3) is also present in the MAS to serve as a middle man between the order agents and the worker agents. An order agent thus requests the supervisor agent to assign the requested task to a worker. The supervisor agent initiates a contract net protocol with the active worker agents to achieve this. In the *setup()* method of the worker agent, an *SSResponderDispatcher* behaviour is added that starts a new SS*ContractNetResponder* behaviour for every *CFP* message that is received by the agent.

If a *CFP* message is received, the dedicated SS*ContractNetResponder* behaviour sends a *propose* message, containing a proposal score, back to the Supervisor agent. The proposal score in this case is based on the current state of the worker. The proposal score consist of various components with different weights:

- The component of the proposal score with the largest weight is based on whether or not the worker is currently at the correct workstation to perform the requested operation. This is to avoid any unnecessary movement between workstations. The worker at the required workstation will thus always have the best proposal.
- The component with the second largest weight is based on whether the worker is currently at no workstation. An idle worker will thus always have a higher proposal score than another worker that is currently at a different workstation. This avoids that a worker at another workstation is not moved to the required workstation if there is an idle worker in the system.
- The component with the third largest weight is based on the current action queue length of the worker agent. The proposal score is reduced for every action in the queue. This ensures that if the scores of two workers are very close, that the one with the shortest queue is assigned the task.
- The component with the smallest weight is a random component that will make the difference if two workers are identical in every other way.

If the proposal is accepted, the worker agent receives an *accept* message from the initiating supervisor agent. The requested operation is then added to the action queue of the worker agent. The SS*ContractNetResponder* that is handling the conversation waits until the current operation matches its queue number. The operation is then performed by sending the operation instructions to the human interface device via the TCP-IP socket. When all the instructions have been completed, a confirmation message from the interface device is received by the

worker agent. Finally an *inform* message is sent back to the supervisor agent and the SS*ContractNetResponder* behaviour is terminated.

When a worker logs out from their mobile interface, a log out message from the interface device is received by the worker agent. The worker agent then de-registers itself from the DF and then terminates itself.

6.3.1.3 Supervisor agent

The supervisor agent is only present in the MAS if the WHA is used. It serves as a middle man between the order agents and the worker agents and fulfils the role that the human supervisor has when using the IHA. The supervisor agent assigns operations requested by order agents to worker agents.

The worker allocation strategy programed into the supervisor agent is simply to assign the first available worker with the highest proposal score to the order agent that has been waiting the longest. This simple strategy was implemented because it can be easily mimicked by a human supervisor for the experimentation with the two architectures. It ensures that the management strategy is controlled and does not influence the results.

In the *setup()* method of the supervisor agent, its services are registered with the DF. An *SSResponderDispatcher* behaviour is also added that starts a new *SSAchieveREResponder* in a new thread for every *request* message that is received from an order Holon.

The supervisor agent needs to make contact with the worker agents in the MAS to assign the requested operation to one of them before it can complete the initial request protocol with the order agent. When a new *SSAchieveREResponder* is started to handle an incoming *request* message, it thus immediately starts a new *ContractNetInitiator* behaviour. This behaviour starts by sending *CFP messages* to the worker agents that can perform the operation according to the DF. The worker agents reply with proposal scores that are evaluated to determine the best proposal. The best proposal is accepted by sending an *accept* message to the relevant worker agent and the operation is added to the action queue of the worker agent as described in section 6.3.1.2. The *ContractNetInitiator* behaviour waits for the *inform* message from the worker agent, indicating that the operation is complete before the behaviour is terminated.

After the contract net protocol with the worker agents has completed, the supervisor agent resumes its request protocol with the order agent by sending an *Inform* message back to indicate that the operation has been completed.

6.3.2 Worker holon lower level control

For the WHA, the worker holon described in section 5.2.2 is used to integrate human workers in the HMS. The worker Holon consists of a worker agent as well as a mobile human interface that serves as the LLC for the worker. The mobile interface is used to facilitate communication between the worker agent and the worker that is represented by the agent.

The mobile interface consists of a networked mobile device that runs an interface application with a GUI. This device is dedicated to one worker and stays with the worker at all times. The initial idea for this mobile interface was to use a wearable device based on a micro controller with a touch screen. Because of cost restrictions however, the decision was made to rather implement the interface as an Android application that can then be run on any Android device. All Android devices have all the required hardware for the interface device, including Wi-Fi connectivity and a touch screen that serves as both an input and output device.

The mobile interface application was developed in Android Studio. All the scripting is done using Java and the layouts of all screens (or *activities* as they are named in the development environment) are done using XML. The mobile interface application is responsible for relaying information between the worker and the worker agent. When the device is connected to the Wi-Fi network and the application is started, the worker must first be registered with the MAS. The application opens with the login screen as shown in Figure 24.

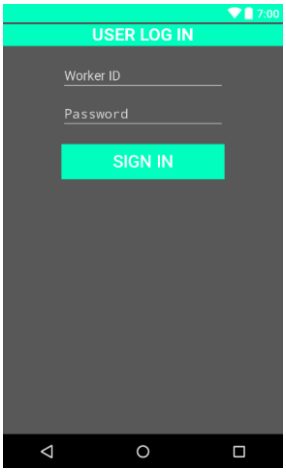
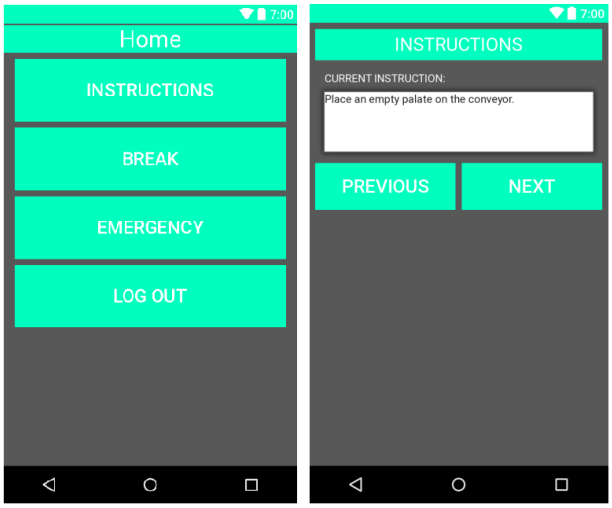


Figure 24: The login screen of the mobile interface application.

The worker enters a worker ID and password to log in. A TCP-IP socket is then opened with the PC that the MAS is running on and a connection is made with the server of the worker registration agent. The worker ID and password that was entered as well as the IP address of the Android device is sent to the worker registration agent over this socket. If the log in information checks out, a worker agent is created to represent the worker as described in section 6.3.1.1.

When it is launched, the worker agent opens a socket with the Android device as described in section 6.3.1.2. This socket is then used for two-way communication between the mobile interface application and the worker agent.

After the worker has logged in the application transitions to the home screen as shown in Figure 25 (a). The home screen contains four menu buttons that, when clicked, results in a transition to their respective screens.



(a) (b)

Figure 25: (a) The home screen of the mobile interface application. (b) The instructions screen of the mobile interface application.

The instruction screen, as seen in Figure 25 (b), can only be accessed if there currently are instructions to perform. Otherwise, a pop up message will be displayed that states that there are currently no instructions. If instructions are received from the worker agent via the TCP-IP socket, the *instruction* button on the home screen starts to flash to indicate that there are pending instructions. The worker can then click the *instruction* button to transition to the instructions screen which will then display the first instruction in the set as shown in Figure 25 (b).

After an instruction has been completed the worker clicks the *next* button to display the next instruction in the set. The *previous* button can be clicked to go back to the previous instruction if needed. While busy with the current instruction set, the worker can return to the home screen with the back button to gain access to the other menu options. When the instruction set is complete, the interface application sends a confirmation message over the TCP-IP socket back to the worker agent which then resumes its operations accordingly. The application then transitions back to the home screen where the *instructions* button will have stopped flashing.

When a worker needs to take a break, the break screen can be accessed with the *break* button from the home screen. The application then transitions to the break screen as seen in Figure 26 (a). The worker starts a break by tapping the *start break* button. A message is then sent to the worker agent informing it that the worker cannot perform any tasks until the break ends. The worker agent then de-registers itself from the DF, meaning that it will no longer receive requests from other agents. It also sends a message to the performance tracker agent to log the start of the break.

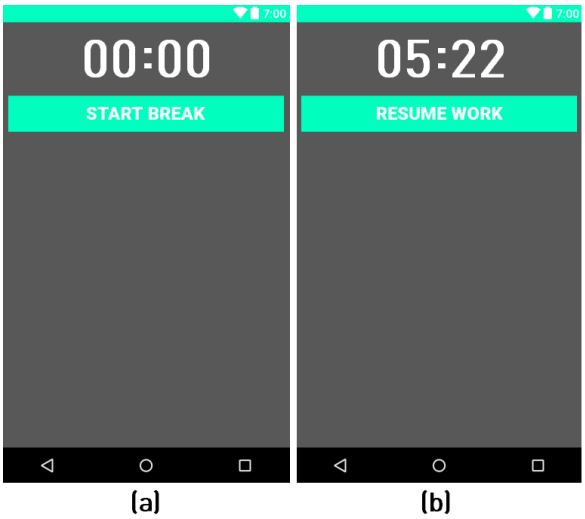


Figure 26: The break screen (a) before and (b) after a break has been started.

When the break is started, the timer on the screen starts and can be used by the worker to keep track of the duration of the break. The *start break* button also changes to the *resume work* button as seen in Figure 26 (b). The user can then tap this button to end the break. When the break is ended another message is sent to the worker agent, which then re-registers itself with the DF and sends another message to the performance tracker agent to log the end of the break.

When it is the end of the worker's shift, the worker must log out. The worker can do this simply by tapping the *log out* button on the home screen. A message is then sent to the worker agent which then records the end of the session and terminates itself.

7 Evaluation

In this section the architectures for human integration, as implemented in section 6, are evaluated. First, the evaluation criteria is defined. Then, the data acquisition and experiments are described. Finally, the results are presented and discussed.

7.1 Evaluation criteria

In this section, evaluation criteria are defined to evaluate the two architectures for human integration. The desired characteristics of the manufacturing system are described, as well as the requirements that influence these characteristics. The performance measures that were used to evaluate these requirements are also described.

7.1.1 Characteristics and requirements

Two important characteristics for a manufacturing system to possess are high productivity as well as flexibility and reconfigurability (Koren & Shpitalni, 2010). Productivity is a desirable characteristic, since higher productivity directly results in a higher production volume, which in turn results in increased profit for the manufacturing company. Flexibility and reconfigurability are also desirable characteristics, since they allow a system to easily respond and adapt to changes and disturbances. It also makes a system more capable of producing a larger variety of products.

Productivity

For the case study described in section 3, the productivity of the testbed cell is mostly dependent on the following:

- **Working speed** is here defined as the speed at which the human workers are able to perform manual operations. The working speed is dependent on the human interface and the speed at which it can be used to communicate instructions to the worker.
- **Worker utilisation** is here defined as the percentage of production time that the workers are actively working. The worker utilisation is mostly influenced by the testbed cell configuration and worker management.
- **Quality of work** is here defined as the rate at which the workers make mistakes when performing manual operations.

Flexibility and reconfigurability

Flexibility in this case is the ability of the system to produce various products without making changes to the functionality of the system. Reconfigurability is the ability to easily reconfigure the system by adding or removing components, with the aim of producing different products, or changing the capacity of the system. In this section, flexibility and reconfigurability only relate to the human integration. flexibility and reconfigurability of the testbed cell is mostly dependent on the following:

- **Resource flexibility** is the ability of a resource to perform various operations as required by the system.
- **Resource mobility** is here defined as a worker's ability to move between workstations in order to perform different operations at different workstations. Resource mobility is dependent on the ease and the speed by which workers can be moved between workstations. Resource mobility directly influences reconfigurability.

7.1.2 Performance measures

The main purpose of the performance tracker (described in section 5.3.1 and implemented in section 6.1.1) is to record data that can be used to calculate a set of performance measures. The data that is recorded is discussed in detail later in section 7.2. The performance measures calculated from this data can be used to compare the two proposed architectures for human integration in terms of the criteria discussed in section 7.1. In this section these performance measures are described. The information that the performance measures can provide regarding the evaluation criteria is also discussed. The calculation of the performance measures are also described. Table 1 shows a matrix that relates performance measures to characteristics

Table 1: matrix relating the performance measures to the characteristics

	CHARACTERISTICS	PRODUCTIVITY			FLEXIBILITY & RECONFIGURABILITY	
	REQUIREMENTS	Working speed	Worker utilisation	Quality of work	Resource flexibility	Resource mobility
PERFORMANCE MEASURES	Average operation time	X				
	Total production time		X		X	X
	Average worker utilisation		X		X	X
	Number of mistakes			X		

7.1.2.1 Average operation time

Operation time is the time that a human worker takes to complete a single operation. It is measured from the moment operation instructions are sent to the human interface, to the moment when the completion message is received from the interface application. The average operation time is calculated using the operation records from the performance tracker as described is section 7.2.2. The sum of all the operation times for the experiment is simply divided by the total number of operations as shown in equation 1 below.

$$\bar{t}_o = \frac{\sum t_o}{n_o} \tag{1}$$

With every experiment that was conducted, the same production order was given to the cell controller as discussed later in section 7.3. This means that for every experiment, the same operations were performed. The time it takes to perform an operation is related to the speed that the human interface can be used. The average operation time can thus give an indication of the working speed, which has an influence on the overall productivity.

7.1.2.2 Total production time

The total production time is simply the time it takes to complete a specified production order. Since the production order is the same for every experiment, the total production times of experiments with different conditions can be directly compared. The total production time is measured from the start time of the first order to the completion time of the last order produced in the experiment.

During an experiment, the time it takes for workers to move between workstations is not included in the operation time, as described earlier in this section. The total production time thus provides an indication of the speed at which workers move between workstations. This speed provides an indication of the mobility and flexibility of the resource which has an influence on the flexibility and reconfigurability of the cell. It also provides an indication of the worker utilisation of the workers which has an influence on productivity.

7.1.2.3 Average worker utilisation

Worker utilisation is the percentage of the total production time that a certain resource is being utilised. In this case, the time that a human worker is seen as being utilized is the time that the worker is actively completing instructions.

To calculate the average worker utilisation, first, the worker utilisation of each of the workers is calculated. As seen in equation 1 the worker utilisation of an individual worker, in this case worker 1, is calculated by dividing the total operation time of that worker with the total production time.

$$u_{w1} = \frac{t_{o,w1}}{t_p} \quad (1)$$

To calculate the average worker utilisation of all the workers, the sum of resource utilisation of all the workers is divided by the total number of workers and then multiplied by 100 to provide a percentage value as seen in equation 2 below.

$$\bar{u}_w = \frac{u_{w1} + u_{w2} + \ldots + u_{wn}}{n_w} \times 100\% \quad (2)$$

The worker utilisation indicates how much the human resources of the system are used during production. A higher percentage indicates that the resource was used more during production. The worker utilisation has an influence on the productivity of the cell.

7.1.2.4 Number of mistakes

The number of mistakes made by human workers is calculated from data in the operation records. It is calculated simply by counting the number of operations that are logged as *completed-incorrect*.

7.2 Data acquisition

For all the experiments, data was captured and stored by the performance tracker agent described in section 6.1.1. The recorded data includes records of work sessions, operations performed by the worker, breaks taken by the worker and product orders produced by the cell.

7.2.1 Work session records

The work session records contain information regarding every work session of all the workers in the system. With the IHA, the session record is created as a user logs in to the fixed interface at a work station. For the WHA the session record is created as the user logs in to the interface application on his/her dedicated device. For both architectures, the worker ID and the log in time are recorded. For the IHA, the workstation ID where the relevant fixed interface is located is also recorded. For both architectures, when the user logs out, the end time of that session is recorded and the total time of the session is calculated. Work session sample data is given in Table 2.

Table 2: Work session records sample data.

Ses. No.	Worker ID	Workstation ID	Start Date	Completion Date	Ses. Time (s)
0	222222	3	Fri Oct 13 08:57:52 CAT 2017	Fri Oct 13 09:24:07 CAT 2017	1564,47
1	111111	2	Fri Oct 13 08:58:02 CAT 2017	Fri Oct 13 09:24:05 CAT 2017	1562,55
2	333333	1	Fri Oct 13 08:58:04 CAT 2017	Fri Oct 13 09:24:02 CAT 2017	1557,45

7.2.2 Operation records

The operation records contain information regarding all the individual operations performed by the human workers in the cell. The operation records for both architectures contain the same information. An operation record is created every time a human worker starts to perform a task. The worker ID, workstation ID, operation ID, and start time is immediately recorded. When the human worker has completed the operation, the completion time is recorded and the total operation time is calculated. Table 3 shows a sample of operation records.

Table 3: Operation records sample data.

Op. No.	W ID	WS ID	Op.	Start Date	Completion Date	Op. Time (s)	Op. Result
0	333333	1	1	Fri Oct 13 08:58:25 CAT 2017	Fri Oct 13 08:58:41 CAT 2017	15,196	Completed
1	333333	1	2	Fri Oct 13 08:58:42 CAT 2017	Fri Oct 13 08:59:11 CAT 2017	29,142	Completed-Correct
2	333333	1	3	Fri Oct 13 08:59:17 CAT 2017	Fri Oct 13 08:59:59 CAT 2017	42,573	Completed-Correct
3	333333	1	1	Fri Oct 13 09:00:12 CAT 2017	Fri Oct 13 09:00:38 CAT 2017	25,985	Completed
4	333333	1	2	Fri Oct 13 09:00:39 CAT 2017	Fri Oct 13 09:01:29 CAT 2017	49,47	Completed-Correct

The operation records also include an operation result field. Possible results include:

- *Completed*: An operation is recorded as *completed* if the worker has completed all the instructions for the operation.
- *Aborted*: An operation is recorded as *aborted* if the worker has ended the operation without completing all the instructions.
- *Completed-correct*: An operation is recorded as *completed-correct* if the result of the operation was inspected by another sub-system and passed the inspection. In this case, a camera inspects selected operations and determines whether the operation is recorded as correct or incorrect.
- *Completed-incorrect*: An operation is recorded as *completed-incorrect* if the result of the operation was inspected by another sub-system and failed the inspection.

7.2.3 Break records

The break records contain information regarding the breaks from active work that the human workers take. The break records are only necessary for the WHA since all workers have dedicated interfaces and the system needs to know whether or not a worker is currently available. With the IHA, if a worker needs to take a break he/she can simply log out of the interface allowing another worker to fill the spot if needed.

A break record is created when a worker starts a break by clicking the *start break* button on the *break* screen of the interface application. The worker ID and break start time is immediately recorded. The worker concludes the break by clicking the *resume work* button on the *break* screen of the interface application. When the break is concluded, the break end time is recorded and the total break time is calculated. Table 4 shows a sample of break records.

Table 4: Break records sample data.

Break Number	Worker ID	Start Date	Completion Date	Break Time (s)
0	222222	Fri Oct 13 10:31:27 CAT 2017	Fri Oct 13 10:37:30 CAT 2017	363,35
1	333333	Fri Oct 13 10:35:35 CAT 2017	Fri Oct 13 10:45:39 CAT 2017	526,21
2	222222	Fri Oct 13 11:10:17 CAT 2017	Fri Oct 13 11:17:39 CAT 2017	422,00

7.2.4 Order records

The order records contain information regarding the production orders given to the system. The order records include the size of the order, the product that is produced, the start time of the order, the completion time of the order and the total production time of the order. For more information on how orders are managed refer to section 4.2.4.1. The order times are needed to calculate some of the performance measures discussed in section 7.1.2. Table 5 below shows a sample of order records.

Table 5: order records sample data.

Order No	Product Type	QTY	Start Date	Completion Date	Order Time (s)
0	1-pole	6	Fri Oct 13 08:58:25 CAT 2017	Fri Oct 13 09:07:30 CAT 2017	545,584
1	2-pole	3	Fri Oct 13 09:00:12 CAT 2017	Fri Oct 13 09:14:51 CAT 2017	879,695
2	3-pole	4	Fri Oct 13 09:05:12 CAT 2017	Fri Oct 13 09:20:37 CAT 2017	925,421
3	4-pole	3	Fri Oct 13 09:09:56 CAT 2017	Fri Oct 13 09:24:02 CAT 2017	846,035

7.3 Experiment description

This section describes the experiments that were performed to test the agent based holonic control system that was developed for the testbed cell. The goal of the experiments is to gather data that can be used to evaluate the holonic control system and to compare the two architectures for human integration in terms of the criteria described in section 7.1. In this section the experimental procedure and experimental scenarios are discussed.

7.3.1 Experimental procedure

Three scenarios were devised to evaluate the proposed architectures for human integration. These scenarios are described in detail in section 7.3.2.

For every experiment, the same production order was given to the cell controller. This was done to ensure that the data acquired while using different architectures

and scenarios can be directly compared. Table 6 below shows the production order that was used for all experiments. There are four orders in total. To produce these orders eight pallets of breakers are required.

Table 6: The production order for all experiments.

ORDER	PRODUCT	QTY
1	1-pole circuit breaker	6
2	2-pole circuit breaker	6
3	3-pole circuit breaker	4
4	4-pole circuit breaker	3

The ramp-up and ramp-down time of production times in the experiments make up a significant part of the total production time. Ideally, a larger production order could increase the period of steady state manufacturing in the experiments. Unfortunately, the size of the production order used for the experiments is limited by the following practical reasons:

- A limited number of circuit breaker assemblies was available to use in the experiments. The disassembly of completed products takes too much time to allow for the reuse of breaker assemblies during an experiment.
- Larger production orders greatly increases the time it takes to complete and experiment. A large number of experiments needed to be conducted and thus this increased production time would result in an impractical time requirement from the test workers.

The large proportions of the ramp-up and ramp-down time does have an influence on the results, but the influence is the same for every experiment. It will thus not have an impact on the accuracy of comparisons made between experiments.

For every experiment the same test workers were used. This was also done to ensure that the data acquired while using different architectures and scenarios can be directly compared. The test workers who took part in the experiments are all independent engineering students from Stellenbosch University that were asked to collaborate on this research project. None of them had any prior knowledge of the system or the performance measures that are used. They were only given a brief instructional briefing on how to use the interfaces and were instructed to try to be as consistent as possible in terms of working speed.

For the experiments conducted with the IHA, a human supervisor was required to move the workers between workstations as needed. One of the test workers was tasked with acting as the supervisor when necessary. This supervisor was instructed to manage the workers with the same strategy used by the supervisor agent, which manages the workers when the WHA is used. This strategy is a *first in first out* strategy. This means that the first available worker is always assigned to the workstation where a pallet has been waiting the longest time for a manual

operation. This strategy is not necessarily the optimal strategy, but it does ensure consistent worker management for all experiments. This strategy also forces a high amount of worker movement in certain scenarios which is desired in order to evaluate worker mobility.

Experiments were conducted with each of the scenarios described in section 7.3.2. For very scenario, an experiment was done using the IHA and the WHA. Every experiment that was conducted was also repeated to ensure that the results are repeatable and conclusive.

As the human workers complete more tasks and become familiar with the process as the experiments progress, they may become increasingly efficient at performing the various operations. When the experiments are repeated, it is thus done in a reversed order. The average results of the two rounds of experiments was used as the final results. This was done in an attempt to minimise the influence that the varying performance of the human workers have on the results.

7.3.2 Description of scenarios

The scenarios that were devised are based on the case study described in section 3. The setup of the cell for each scenario differs slightly with the intent of isolating certain evaluation criteria. This allows the two architectures to be easily compared in terms of these criteria.

7.3.2.1 Scenario 1: Three workers for three stations (3W3S)

In this scenario, 3 workers man the 3 workstations of the cell. Because there is a worker at every workstation, workers do not move between workstations. This scenario thus focusses on comparing the interfaces of the respective architectures in terms of their speed of use.

7.3.2.2 Scenario 2: Two workers for three stations (2W3S)

In this scenario, two workers are present to man the three workstations in the cell. This scenario is able to clearly illustrate one major difference between the two architectures, which is the way workers are moved between workstations. With the IHA, the workers are moved around by a human supervisor. With the WHA, the supervisor agent (described in section 6.3.1.3) manages the movement of the workers on the floor. The mobility of the human workers, in the case of each of the architectures, can be compared using the performance data from experiments conducted with this scenario.

7.3.2.3 Scenario 3: One worker for three stations (1W3S)

In this scenario, there is only one worker to man the three workstations in the cell. This scenario, like 2W3S, focusses on the mobility of the workers when using each of the each architectures. This is an extreme case since the cell is severely undermanned with only one worker, but it does force a large amount of movement and will thus illustrate the effects of mobility even more clearly than 2W3S. The results of experiments with 3W3S 2W3S and 1W3S can be used to see the effect on worker mobility as the number of workers in the system increases and less movement is required.

7.4 Results

In this section the results for all the experiments are given. The results are grouped together according to the different experiment scenarios as described in section 7.3.2. As mentioned earlier in section 7.3.1, each experiment was conducted twice to ensure repeatability. The result tables in this section show a summary of the results of both experiments for every scenario architecture combination, as well as the average between the two. The performance measures shown in the tables are based on the records from the performance tracker as described in section 7.2. An example of the raw data of an experiment can be found in Appendix D. These performance measures were calculated as described in section 7.1.2.

Scenario 1: Three workers for three stations (3W3S)

Table 7 shows a summary of the results of the experiments conducted with 3W3S.

Table 7: 3W3S results summary.

	SCENARIO: 3W3S					
	INTERFACE HOLON ARCHITECTURE			WORKER HOLON ARCHITECTURE		
	Exp. 1	Exp. 2	**Average**	Exp. 1	Exp. 2	**Average**
Total operation time (s)	2257	2187	**2222**	2143	2043	**2093**
Total production time (s)	1487	1410	**1449**	1434	1321	**1378**
Average operation time (s)	48,0	46,5	**47,3**	45,6	43,5	**44,5**
Average resource utilisation	50,6%	51,7%	**51,2%**	49,8%	51,5%	**50,7%**
Number of worker transfers	0	0	**0,0**	0	0	**0,0**
Number of mistakes	0	0	**0,0**	0	0	**0,0**

As seen in Table 7, for 3W3S, all of the performance measures are very similar for the two architectures. The total operation time, and subsequently the average operation time, was slightly less when the WHA was used. This translates to a difference of 2.8 seconds per operation (5.8 %). The total production time was 71 seconds (or 5 %) faster with the WHA. The worker utilisation for the two architectures differs by less than 1 %.

Scenario 2: Two workers for three stations (2W3S)

Table 8 shows a summary of the results of the experiments conducted with 2W3S.

Table 8: 2W3S results summary.

	SCENARIO: 2W3S					
	INTERFACE HOLON ARCHITECTURE			WORKER HOLON ARCHITECTURE		
	Exp. 1	Exp. 2	**Average**	Exp. 1	Exp. 2	**Average**
Total operation time (s)	2232	2183	**2207**	2142	2080	**2111**
Total production time (s)	1668	1562	**1615**	1478	1420	**1449**
Average operation time (s)	47,5	46,4	**47,0**	45,6	44,3	**44,9**
Average resource utilisation	66,9%	69,9%	**68,4%**	72,5%	73,2%	**72,8%**
Number of worker transfers	10	10	**10,0**	11	10	**10,5**
Number of mistakes	0	0	**0,0**	0	0	**0,0**

As seen in Table 8, the total operation time, and subsequently the average operation time, was lower when the WHA was used. This translates to a difference of 2.1 seconds per operation (4.4 %). There is also a fair difference in the total production time. With the WHA, the total production time was 166 seconds (or 10 %) faster than with the IHA. The average worker utilisation with the WHA was 4.4 % higher than with the IHA.

Scenario 3: One Worker for three stations (1W3S)

Table 9 shows a summary of the results of the experiments conducted with 1W3S.

Table 9: 1W3S results summary.

	SCENARIO: 1W3S					
	INTERFACE HOLON ARCHITECTURE			WORKER HOLON ARCHITECTURE		
	Exp. 1	Exp. 2	**Average**	Exp. 1	Exp. 2	**Average**
Total operation time (s)	2149	2040	**2095**	2014	2083	**2049**
Total production time (s)	2951	2834	**2893**	2223	2285	**2254**
Average operation time (s)	45,7	43,4	**44,6**	42,9	44,3	**43,6**
Average resource utilisation	72,8%	72,0%	**72,4%**	90,6%	91,2%	**90,9%**
Number of worker transfers	36	34	**35,0**	35	35	**35,0**
Number of mistakes	0	0	**0,0**	0	0	**0,0**

As seen in Table 9, the results for the experiments conducted with 1W3S is very consistent, with minimal difference between the two experiments conducted for each architecture. The total operation time, and subsequently the average operation time, was slightly less when the WHA was used. This translates to a difference of 1 second per operation (2.2 %). There is, however, a large difference in the total production time. With the WHA, the total production time was 639 seconds (or 22 %) faster than with the IHA. There is also a large difference in the average worker utilisation. The average worker utilisation with the WHA was 18.5 % higher than with the IHA.

7.5 Discussion of results

In this section the results given in section 7.4 are discussed. Possible trends in the results are identified and explained and conclusions regarding the evaluation criteria are drawn for each architecture.

Average operation times

Figure 27 compares the average operation times of all three scenarios for the two architectures for human integration. The graph is based on data from the tables given in section 7.4. The horizontal axis on the graph represents the number of workers in the cell and thus corresponds to the 1W3S, 2W3S and 3W3S scenarios. The error bars show the deviation of the results of the two experiments that were conducted for each scenario-architecture combination, in relation to the average that is used to generate the graph.

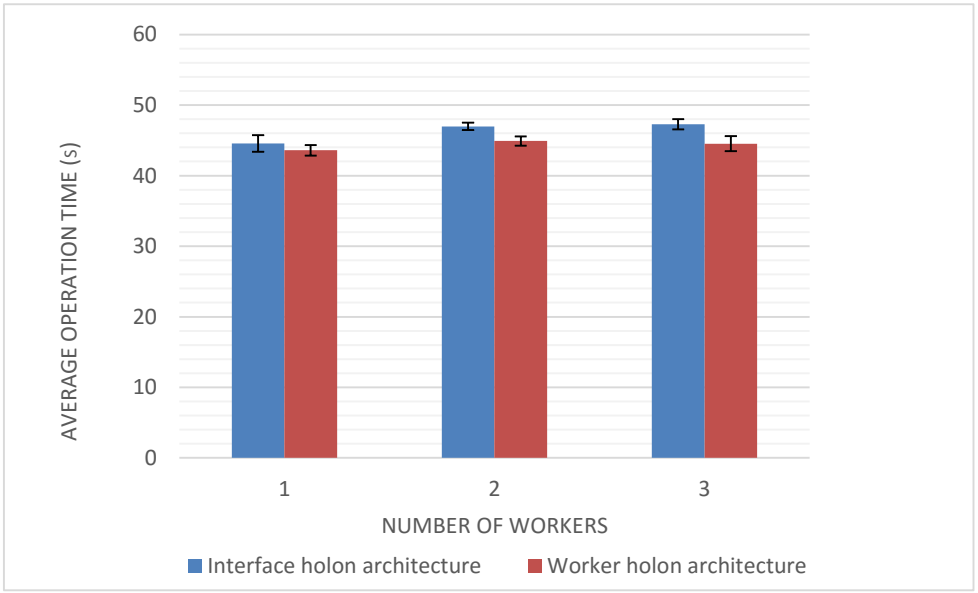


Figure 27: Average operation times of scenarios 1-3 for the two architectures.

As seen in Figure 27, the average operation times are slightly faster for every scenario when the WHA is used. The differences are quite small with a minimum difference of 2.2 % with one worker in the cell, and a maximum difference of 6 % with three workers in the cell. The average operation times for both architectures are very slightly lower with one worker in the cell and the difference between the two architectures is smaller when compared to the other scenarios. One possible explanation for this is that the worker selected to perform the single worker experiments was quicker to perform the operations than the average of the multiple workers used in the other experiments.

Since the same workers were used and the same operations were performed for all experiments, the only variable that the difference in average operation time can be attributed to is the different interfaces used with each of the architectures. It thus seems that the interface of the WHA is faster to use than the interface of the IHA. The interface of the WHA consists of an interface application that runs on a mobile device with a touch screen. The interface of the IHA consists of an interface application that runs on a Raspberry Pi and uses a keyboard with a touchpad as an input device. Both of the GUI's of the respective interface applications are very similar in functionality and navigation. One possible reason that the interface of the WHA is faster to use, is the input method. It is possible that the interface is more quickly navigated using a touch screen than using a keyboard and touchpad since only a tap is required with the touch screen as opposed to a curser movement with the touchpad followed by a click.

Another conclusion that can be drawn from Figure 27 is that the average operation time for each architecture stays relatively constant for all the experiments. This makes sense when considering that the same operations are performed for every experiment and that that the number of workers in the system have no influence on operation times, since the time it takes to move between workstations is not included (see section7.2.2).

From the error bars in Figure 28 it is noted that in all cases the difference between the two experiments that were conducted for every scenario-architecture combination is relatively small. This provides some indication that the results are repeatable with minimal error. The small error that is present can be attributed to the human factor in the experiments.

Figure 28 below shows the overall average operation times of all the respective operation types when using the two architectures. The total average time of all operations is also shown for the two architectures. To generate Figure 28, the operation records from the all the experiments conducted with all scenarios were used. This results in a total number of 405 data points for each architecture. For each of the experiments, the same operations were performed and the same number of experiments were conducted for each architecture and scenario combination. The only variable is thus the architecture that was used.

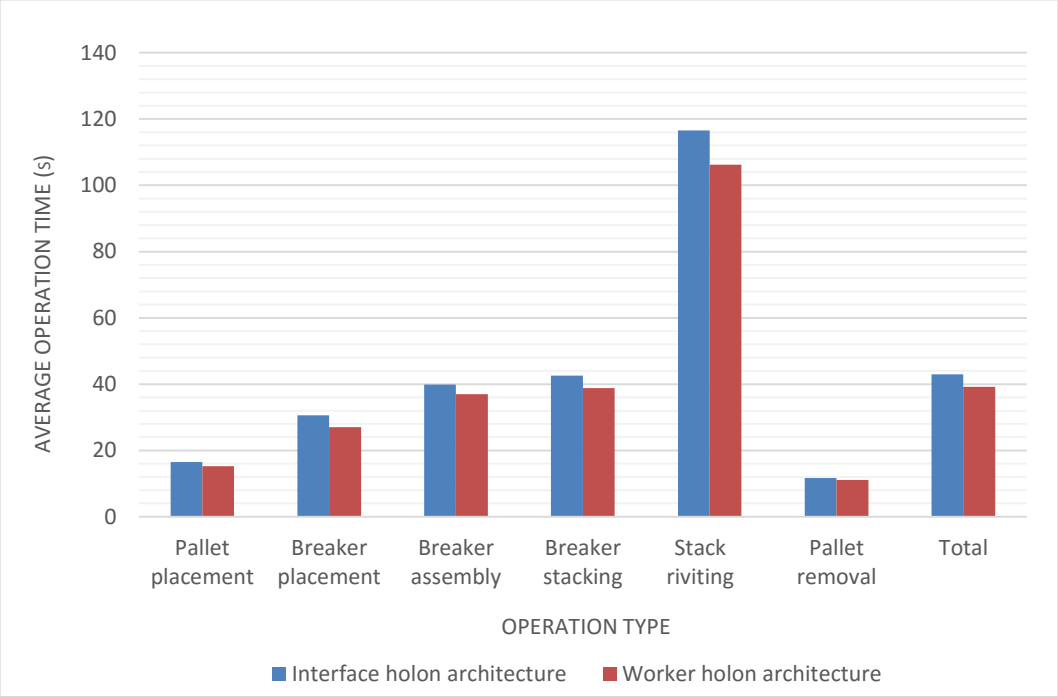


Figure 28: Overall average operation times for each operation.

As seen in Figure 28, for each of the respective operation types, the overall average operation time across all scenarios was faster when the WHA was used. Consequently, the total average operation time of all the experiments is 8.6 % faster when the WHA was used. This further supports the conclusion reached earlier in this section that the interface of the WHA is faster to use.

From the results discussed in this section, it can be concluded that the slightly faster speed of use of the WHA's interface results in a higher working speed and consequently, higher productivity when compared to the IHA.

Total production time

Figure 29 compares the total production times of all scenarios for the two architectures for human integration. The graph is based on data from the tables given in section 7.4. The horizontal axis on the graph represents the number of workers in the cell and thus corresponds to the 1W3S, 2W3S and 3W3S scenarios. The error bars show the deviation of the results of the two experiments that were conducted for each scenario-architecture combination, in relation to the average that is used to generate the graph.

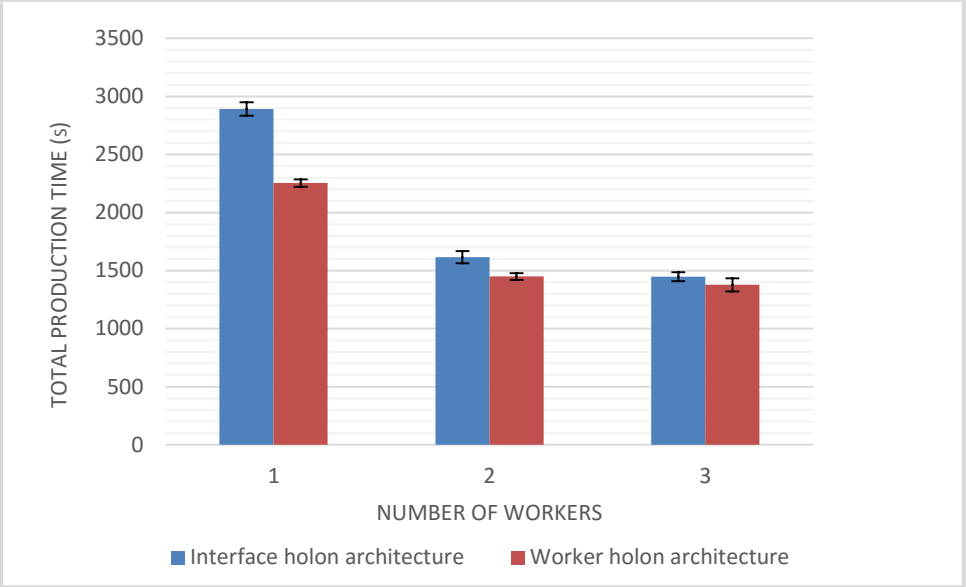


Figure 29: Total production times of all scenarios for both architectures.

As seen in Figure 29 and the tables in section 7.4, in the case of both architectures, the total production time decreases as the number of workers in the cell increases. This makes sense, because the more workers there are in the cell, the more manual operations can be performed simultaneously. Also, the more workers there are in the cell, the less movement of the workers between workstations is required.

From Figure 29 it is interesting that, in the case of both architectures, the total production time decreases exponentially as the number of workers increases. There is a large difference in total production time between the scenarios with one and two workers respectively. This can be attributed to the fact that double the amount of operations can be performed concurrently when two workers are used as opposed to one.

The difference in total production time between the scenarios with two and three workers respectively, is much smaller. One contributing factor is the fact that with three workers, only 50 % more operations can be performed concurrently as opposed to two workers in the cell. Another possible contributing factor comes from observations made during the course of the experiments. It was observed that when a third worker is added in the cell, most of the time, one of the workers is standing idle while waiting for others to complete their operations. This occurs because some operations take longer to perform than others as seen in Figure 28. This is further supported by the worker utilisation data from the results tables in section 7.4. The worker utilisation when there are three workers in the cell is much lower than when there are fewer. This means that workers are standing idle for much longer periods when three workers are in the cell.

Another interesting observation that can be made from the graph in Figure 29 is that the difference in total production time for the two architectures seems to decrease as the number of workers in the cell increases. With one worker in the cell, the difference in production time between the architectures is a significant 22 %. In comparison, with three workers in the cell, the difference in production time between the architectures is only 5 %. The operation times for all experiments are fairly constant and the influence of the different number of workers on the total production time should be the same in the case of both architectures. The decreasing difference in total production time is thus most likely related to differences in resource mobility with the two architectures.

For workers to move between workstations takes time, thus the cause of these differences in production time could occur because the act of moving between workstations takes less time when the WHA is used. This makes sense, since there is no need for a worker to log out at one interface and log in at another when using the WHA. Since the interface is mobile with the WHA, the worker can just take it along to the next workstation. Consequently, with the IHA, the extra time it takes to repeatedly log in and out can have a dramatic effect on the total production time. The effect of this extra time increases as the number of worker transfers between workstations increases.

The difference in total production time between the two architectures is so large when there is only one worker in the cell, because the worker has to transfer between workstations very frequently (an average of 35 times per experiment). With two workers in the cell the difference is much less since there is then only an average of 10 worker transfers between workstations per experiment. With three workers in the system, the difference in production time between the two architectures is almost negligibly small, because there are no worker transfers in that case.

From the error bars in Figure 29 it is once again noted that in all cases the difference between the two experiments that were conducted for every scenario-architecture combination is relatively small. This provides some indication that the results are repeatable with minimal error. The small error that is present can be attributed to the human factor in the experiments.

From the results discussed in this section, it can be concluded that the total production time, when the WHA is used, is substantially less than when the IHA is used and resource mobility is a factor. Resource movement is much faster with the WHA and there is no need for a human supervisor, or at least, it will result in a reduced workload of said supervisor. With a more sophisticated version of the supervisor agent, the workers can possibly be managed even more effectively, resulting in higher productivity.

The faster production time with the WHA in scenarios where there are less workers than workstations in the cell strongly indicates the superior resource mobility of the WHA. This results in increased flexibility and reconfigurability of the cell. It also results in a higher worker utilisation, since movement takes less time - this further contributes to a higher productivity, when the WHA is used.

Worker utilisation

Figure 30 compares the worker utilisation of all scenarios for both of the architectures for human integration. The graph is based on data from the tables given in section 7.4. The horizontal axis of the graph represents the number of workers in the cell and thus corresponds to the 1W3S, 2W3S and 3W3S scenarios. The error bars show the deviation of the results of the two experiments that were conducted for each scenario-architecture combination, in relation to the average that is used to generate the graph.

Figure 30: Worker utilisation of all scenarios for both architectures.

As seen in Figure 30, the worker utilisation of both architectures follows a downward trend as the number of workers in the cell is increased. As explained earlier, when the total production time was analysed, this downward trend in worker utilisation can be attributed to the fact that the more workers are in the cell, the more workers have to wait for each other to complete operations. This time spent waiting is time that the resource is not being utilized, resulting in a lower worker utilisation percentage.

When there is only one worker in the cell, that worker should always be doing something. The reason that the worker utilisation is not 100 %, is because of the following reasons: Firstly, the processing time of the MAS that controls the cell is not instant. It takes a few seconds for all the necessary agent communication

protocols to be completed before instructions are sent to the waiting worker. Secondly, the time spent moving between workstations is not included when calculating the worker utilisation.

As seen in Figure 30, the difference in worker utilisation between the two architectures decreases as the number of workers in the system increases. With one worker in the cell, the worker utilisation with the IHA is 19 % less than with the WHA. With two workers, the difference is substantially less at 4.4 %. With three workers in the cell the worker utilisation is more or less the same with a difference of only 0.5 %.

As with the total production time, the difference is minimal with three workers in the cell, i.e. when there is no movement between workstations. Thus, this decreasing difference can be attributed to the difference in resource mobility between the two architectures. As explained earlier, the time it takes to log out and in when a worker transfers to another workstation is time that the workers are not completing instructions. This explains why the worker utilisation is less with the IHA when there are less than three workers in the cell. It also explains why the difference increases when there are less workers in the cell and more movement between workstations is required.

From the results discussed in this section, it can be concluded that the worker utilisation, when using the WHA, is substantially lower compared to that of the IHA when resource mobility is a factor. This is a result of the faster movement between workstations when using the WHA. This faster movement, as explained earlier is due to the fact that it is not necessary to log in and out for every worker transfer as with the IHA. The higher worker utilisation with the WHA increases the productivity of the cell. The superior resource mobility of the WHA, which is responsible for the higher worker utilisation, results in increased flexibility and reconfigurability of the cell.

Number of mistakes

No mistakes were made by the workers in any of the experiments, thus no difference in the number of mistakes can be observed between the two architectures. There is thus no indication of the quality of work and it can be stated that for these experiments the quality of work had no influence on the productivity of the cell.

8 Conclusions and recommendations

This thesis documents the research conducted into the integration of human workers as resource holons in holonic manufacturing systems. The main objectives of the research is the development, implementation and evaluation of architectures for human integration.

As a case study, two architectures for human integration were implemented as part of a holonic control system for a testbed manufacturing cell. This cell simulates a part of the circuit breaker manufacturing process of CBI electric. The cell contains various automated resources including a conveyor, an inspection camera and a robot. The various manual operations required in this process were performed by human workers at three manual workstations placed along the conveyor.

The holonic control strategy was implemented according to the PROSA holonic reference architecture. The physical and information processing components of the cell were mapped to resource, order and staff holons in accordance with the PROSA reference architecture. The holonic system was implemented as a multi agent system which was developed using the JADE platform. The information processing component of all the holons of the holonic system are represented by resource, order and staff agents in the multi agent system. The resource agents represent the resources of the cell and control the physical resource by interfacing with the LLC of the resource. The order agents are responsible for coordinating the resources in order to complete the manufacturing process of the products that the order agent are responsible for. The staff agents perform several additional functions required for the operation of the cell.

Two different approaches were used to develop two architectures for human integration. These architectures governed the architecture of the resource holon, the structure of the MAS and the human interface used to facilitate communication between the control system and the worker. The two architectures were the interface holon architecture (IHA) and the worker holon architecture (WHA). With the IHA, a fixed interface at a workstation is represented by an agent in the MAS and workers can log in at this interface to perform operations requested by the control system. With the IHA the movement of the workers are managed by a human supervisor. With the WHA, every worker is directly represented in the MAS and a mobile interface is used to facilitate communication when the control system directly requests a specific worker to perform an operation. With the WHA, the movement of workers is managed by an additional staff agent called the supervisor agent.

Several experiments were devised and conducted in order to acquire data that could be used to evaluate the architectures for human integration. The architectures were evaluated in terms of their resulting productivity, flexibility and reconfigurability. Various performance measures that could provide insight into the requirements that influence these characteristics, were calculated. The results were analysed and the following conclusions were drawn regarding the two architectures for human integration:

- The mobile interface of the WHA is slightly faster to use than the fixed interface of the IHA. This is most likely because of the different input methods used by the two interfaces. The faster speed at which the interface can be used results in higher productivity when using the WHA.
- The time it takes for workers to transfer between workstations is shorter with the WHA than for the IHA. This is because of the fact that a worker is required to log out and in when transferring to another workstation with the IHA. With the WHA the interface moves along with the worker and thus, no time is wasted by logging out and in for every transfer. This results in increased mobility when using the WHA, which increases productivity in a case where worker movement is required. The faster and easier movement also results in increased flexibility and reconfigurability of the cell when the WHA is used.

After comparing and discussing the results, it was concluded that the WHA is superior to the IHA in terms of productivity, flexibility and reconfigurability. This is especially true in situations where highly mobile workers are required.

The IHA is still a viable option, especially in situations where human supervision is preferred or required. In complex situations the automation of worker management and supervision may not be practical or possible. In such situations using the IHA and retaining human supervision may be the better option.

In light of the research in this thesis, the following recommendations are made for future research:

- Research can be done into the development of a more advanced mobile interface device. In addition to its communication functions, this device can house sensors that can keep track of working conditions and warn the worker if they are unsafe. Working conditions that can be monitored include noise levels, temperature and air quality. If mass produced, such a device could also become a more economically viable option.
- The performance tracker can be improved to allow for queries regarding the performance of a worker when performing a certain task. This information can be a factor when generating a proposal score. The performance tracker can also alert the plant manager if a worker is not meeting required performance standards.

- The interface application in the case of both architectures can be improved. Text instructions can be accompanied by images that assist the worker in completing the operation. Interface customization options can like language selection can also be added.
- Research can be done into the improvement of the supervisor agent that manages the human workers when the WHA is used. Much more complex algorithms can be developed to manage workers in complex situations in such a way that productivity can be maximised.
- The architectures for human integration can be investigated more thoroughly for larger, more complex systems over longer periods by using simulation techniques.

9 References

Alford, A., Wilkes, D. M., Kawamura, K. & Pack, R. T., 1997. *Flexible Human Integration for Holonic Manufacturing Systems,* Nashville: Vanderbilt University.

Babiceanu, R. F. & Chen, F. F., 2006. Development and applications of holonic manufacturing systems: a survey. *Journal of Intelligent Manufacturing,* Volume 17, pp. 111-131.

Bellifemine, F., Caire, G. & Greenwood, D., 2007. *Developing Multi-Agent Systems with JADE.* 1 ed. s.l.:Wiley.

Blue, B., 2013. *Advantages and Disadvantages of Automation in Manufacturing.* [Online] Available at: http://www.vista-industrial.com/blog/advantages-and-disadvantages-of-automation-in-manufacturing/

Botti, V. & Giret, A., 2008. *ANEMONA A Multi-Agent Methodology for Holonic Manufacturing Systems.* s.l.:Springer.

Brennan, R. & Norrie, D. H., 2001. Agents, holons and function blocks: Distributed intelligent control in manufacturing. *Journal of Applied Systems Science: Special Issue,* 2(1), pp. 1-19.

Bussmann, S., 1998. *An Agent-Oriented Architecture for Holonic Manufacturing Control,* Berlin: Daimler-Benz AG.

Chirn, J. L. & McFarlane, D. C., 1999. A holonic component-based architecture for manufacturing. *MAS '99,* pp. 219-223.

Christensen, J. H., 1994. *Hlonic Manufacturing Systems: Initial Architecture and Standards Directions.* Hannover: s.n.

Christensen, J., Prado, J. M., Tamura, S. & Garcia-Herreros, E., 1994. *Ims - holonic manufacturing systems: System components of autonomous modules and their distributed control,* s.l.: HMS Consortium.

Cossentino, M. et al., 2010. ASPECS: an agent-oriented software process for engineering complex systems. *Auton Agent Multi-Agent Syst,* Issue 20, pp. 260-304.

Farid, A., 2004. *An evaluation of the dacs methodology,* Cambridge: University of Cambridge Institute for Manufacturing.

Farid, A. M., 2004. *A Review of Holonic Manufacturing Systems Literature,* Cambridge: University of Cambridge.

FIPA, 2002. *Standard FIPA specifications.* [Online]
Available at: http://www.fipa.org/repository/standardspecs.html
[Accessed 13 11 2016].

Fletcher, M. et al., 2000. *An Open Architecture for Holonic Cooperation and Autonomy.* s.l., s.n., pp. 224-230.

Giret, A. & Botti, V., 2004. Holons and agents. *Journal of Intelegent Manufacturing,* pp. 645-659.

Giret, A. & Botti, V., 2008. *ANEMONA: A Multi-agent Methodology for Holonic Manufacturing Systems.* s.l.:Springer.

Glanzer, K., Hammerle, A. & Geurts, R., 2001. *The appliance of zeus agents in manufactuirng systems.* Prague, s.n.

Koestler, A., 1967. *The Ghost in the Machine.* London: Hutchinson and Co.

Koren, Y. & Shpitalni, M., 2010. Design of reconfigurable manufacturing systems. *Journal of Manufacturing Systems,* Issue 29, pp. 130-141.

Kotak, D., Wu, S., Fleetwood, M. & Tamoto, H., 2003. Agent-based holonic design and operations environment for distributed manufacturing. *Computers in Industry,* Issue 52, pp. 95-108.

Langer, G. & Bilberg, A., 1997. *Architectural considerations for holonic shop floor control.* Aukland, world Manufacturing Congress.

Laws, A. G., Taleb-Bendiab, A. & Wade, S. J., 2001. Towards a viable reference architecture for multi-agent supported holonic manufacturing systems. *Journal of Applied Systems Science: Special Issue,* 2(1), pp. 61-81.

Leitao, P. J., 2004. *An Agile and Adaptive Holonic Architecture for Manufacturing Manufacturing,* Porto: University of Porto.

Leitao, P. & Restivo, F., 2006. ADACOR: A holonic architecture for agile and adaptive manufacturing control. *Computers In Industry,* Issue 57, pp. 121-130.

Mathews, J., 1995. Organizational foundations of intelligent manufacturing systems - the holonic viewpont. *Computer Integrated Manufacturing Systems,* 8(4), pp. 237-243.

McFarlane, D. C. & Bussmann, S., 2003. Holonic Manufacturing Control: Rationales, Developments and Open Issues. In: *Agent Based Manufacturing: Advances in the Holonic Approach.* s.l.:Springer.

Paulucci, M. & Sacile, R., 2016. *Agent-Based Maufacturing and Control Systems.* s.l.:CRC Press.

Suda, H., 1989. Future factory system formulated in japan. *Japanese Journal of Advanced Automation Technology,* 2(10), pp. 15-25.

Ulieru, M., Walker, S. S. & Brennan, R. W., 2001. *The Holonic Enterprise as a Collaborative Information Ecosystem.* Montreal, s.n.

Van Brussel, H. et al., 1999. A Conceptual Framework for Holonic Manufacturing: Identification of Manufacring Holons. *Journal of Manufacturing Systems,* 18(1).

Van Brussel, H. et al., 1998. Reference architecture for holonic manufacturing systems: PROSA. *Computers in Industry,* 1(37), pp. 255-274.

Van Brussel, H., Wyns, J., Valckenaers, P. & Ginderacher, T. V., 1999. A Conceptual Framework for Holonic Manufacturing: Identification of Manufacturing Holons. *Journal of Manufacturing Systems,* 18(1), pp. 35-52.

Winikoff, M., 2005. *Jack Intelligent Agents: An Industrial Strength Platform.* s.l.:Springer.

Wooldridge, M., Jennings, N. R. & Kinny, D., 2000. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems,* Issue 3, pp. 285-312.

Appendix A: Testbed MAS code

A.1: Order agent code

```
public class OrderAgent extends Agent
{
   //Globals
   OrderAgent a = this;
   int arraysPerFixture;
   int breakersPerArray;
   int orderNo;
   private static final String STATE_0 = "0";
   private static final String STATE_1 = "1";
   private static final String STATE_2 = "2";
   private static final String STATE_3 = "3";
   private static final String STATE_4 = "4";
   private static final String STATE_5 = "5";
   private static final String STATE_6 = "6";
   private static final String STATE_7 = "7";
   private static final String STATE_8 = "8";
   private static final String STATE_9 = "9";
   private static final String STATE_10 = "10";
   private static final String STATE_101 = "101";
   private static final String STATE_11 = "11";
   private static final String STATE_111 = "111";
   private static final String STATE_12 = "12";
   private static final String STATE_121 = "121";
   private static final String STATE_13 = "13";
   private static final String STATE_14 = "14";
   private static final String STATE_15 = "15";
   private static final String STATE_16 = "16";
   private static final String STATE_17 = "17";
   private static final String STATE_18 = "18";

   //Setup method
   public void setup()
   {
      //extract product info from agent arguments
      arraysPerFixture = Integer.parseInt((String)a.getArguments()[0]);
      breakersPerArray = Integer.parseInt((String)a.getArguments()[1]);
      orderNo = Integer.parseInt((String)a.getArguments()[2]);

      System.out.println(a.getLocalName()+": "+"Order holon started.
arrays per fixture = "+arraysPerFixture+". breakers per array =
"+breakersPerArray);

      //====Create FSM behaviour====
      FSMBehaviour fsm = new FSMBehaviour(){
         public int onEnd()
         {
            System.out.println(a.getLocalName()+": "+"FSM Completed");
            myAgent.doDelete();
            return super.onEnd();
         }
      };

      //====Register states====
      //Wait for user input
      fsm.registerFirstState(new OneShotBehaviour(){
```

```
            public void action()
            {
               System.out.println(a.getLocalName()+": "+"STATE 0");
            }
            public int onEnd()
            {
               return 1;
            }
         }, STATE_0);

         //1 Conveyor holon - accept palate at station 1
         String s1service = "conveyor";
         String s1Content = "0";
         fsm.registerState(new ARI(a,1,s1service,s1Content), STATE_1);

         //2 Human holon - place palate
         String s2service = "placePalate";
         String s2Content = "11Place an empty palate on the conveyor";
         fsm.registerState(new CNI(a,2,s2service,s2Content), STATE_2);

         //3 Human holon - place breakers
         String s3service = "placeBreaker";
         String s3Content = "21"+generatePlaceBreakerInst();
         fsm.registerState(new CNI(a,3,s3service,s3Content), STATE_3);

         //4 Machine vision holon - check breakers placed and filled
         String s4service = "machine vision";
         String s4Content = generateInspectionInst(1);
         fsm.registerState(new ARI(a,4,s4service,s4Content), STATE_4);

         //5 Human holon - assemble breakers
         String s5service = "assembleBreaker";
         String s5Content = "31"+generateAssembleBreakerInst();
         fsm.registerState(new CNI(a,5,s5service,s5Content), STATE_5);

         //6 Machine vision holon - check assembly
         String s6service = "machine vision";
         String s6Content = generateInspectionInst(2);
         fsm.registerState(new ARI(a,6,s6service,s6Content), STATE_6);

         //7 Conveyor holon - release palate from station 1
         String s7service = "conveyor";
         String s7Content = "1";
         fsm.registerState(new ARI(a,7,s7service,s7Content), STATE_7);

         //8 Conveyor holon - accept palate at station 2
         String s8service = "conveyor";
         String s8Content = "2";
         fsm.registerState(new ARI(a,8,s8service,s8Content), STATE_8);

         //9 Robot holon - test breakers
         String s9service = "robot";
         String s9Content =
Integer.toString(arraysPerFixture*breakersPerArray);
         fsm.registerState(new ARI(a,9,s9service,s9Content), STATE_9);

         //10 Conveyor holon - release palate from station 2
         String s10service = "conveyor";
         String s10Content = "3";
         fsm.registerState(new ARI(a,10,s10service,s10Content), STATE_10);

         //10.1 if testing failed: Conveyor holon - release palate from
station 2
```

```
      String s101service = "conveyor";
      String s101Content = "3";
      fsm.registerState(new ARI(a,101,s101service,s101Content),
STATE_101);

      //11 Conveyor holon - accept palate at station 3
      String s11service = "conveyor";
      String s11Content = "4";
      fsm.registerState(new ARI(a,11,s11service,s11Content), STATE_11);

      //11.1 if testing failed: Conveyor holon - accept palate at station
3
      String s111service = "conveyor";
      String s111Content = "4";
      fsm.registerState(new ARI(a,111,s111service,s111Content),
STATE_111);

      //12 Human holon - stack breakers
      if(breakersPerArray>1)
      {
         String s12service = "stackBreaker";
         String s12Content = "42"+generateStackBreakerInst();
         fsm.registerState(new CNI(a,12,s12service,s12Content), STATE_12);
      }
      else
      {
         fsm.registerState(new skipState(), STATE_12);
      }

      //12.1 if testing failed: Human holon - remove palate
      String s121service = "removePalate";
      String s121Content = "62Testing failed, remove breakers from
plalate//nRemove the palate from the conveyor.";
      fsm.registerState(new ARI(a,105,s121service,s121Content),
STATE_121);

      //13 Conveyor holon - release from station 3
      String s13service = "conveyor";
      String s13Content = "5";
      fsm.registerState(new ARI(a,13,s13service,s13Content), STATE_13);

      //14 Conveyor holon - accept palate at station 4
      String s14service = "conveyor";
      String s14Content = "6";
      fsm.registerState(new ARI(a,14,s14service,s14Content), STATE_14);

      //15 Human holon - rivit breakers
      String s15service = "rivitBreaker";
      String s15Content = "53"+generateRivitBreakerInst();
      fsm.registerState(new CNI(a,15,s15service,s15Content), STATE_15);

      //16 Human holon - remove palate
      String s16service = "removePalate";
      String s16Content = "63Remove the palate from the conveyor.";
      fsm.registerState(new CNI(a,16,s16service,s16Content), STATE_16);

      //17 Conveyor holon - release palate from station 4
      String s17service = "conveyor";
      String s17Content = "7";
      fsm.registerState(new ARI(a,17,s17service,s17Content), STATE_17);

      //18 last state
      String s18service = "coordinator";
```

```
      String s18Content = Integer.toString(orderNo);;
      fsm.registerLastState(new ARI(a,18,s18service,s18Content),
STATE_18);

      //====Register state transitions====
      fsm.registerTransition(STATE_0, STATE_1, 1);
      fsm.registerTransition(STATE_1, STATE_1, 0, new String[]{STATE_1});
      fsm.registerTransition(STATE_1, STATE_2, 1);
      fsm.registerTransition(STATE_2, STATE_2, 0, new String[]{STATE_2});
      fsm.registerTransition(STATE_2, STATE_3, 1);
      fsm.registerTransition(STATE_3, STATE_3, 0, new String[]{STATE_3});
      fsm.registerTransition(STATE_3, STATE_4, 1);
      fsm.registerTransition(STATE_4, STATE_4, 0, new String[]{STATE_3});
      fsm.registerTransition(STATE_4, STATE_5, 1);
      fsm.registerTransition(STATE_4, STATE_3, 2, new String[]{STATE_4,
STATE_3});
      fsm.registerTransition(STATE_5, STATE_5, 0, new String[]{STATE_5});
      fsm.registerTransition(STATE_5, STATE_6, 1);
      fsm.registerTransition(STATE_6, STATE_6, 0, new String[]{STATE_6});
      fsm.registerTransition(STATE_6, STATE_7, 1);
      fsm.registerTransition(STATE_6, STATE_5, 2, new String[]{STATE_6,
STATE_5});
      fsm.registerTransition(STATE_7, STATE_7, 0, new String[]{STATE_7});
      fsm.registerTransition(STATE_7, STATE_8, 1);
      fsm.registerTransition(STATE_8, STATE_8, 0, new String[]{STATE_8});
      fsm.registerTransition(STATE_8, STATE_9, 1);
      fsm.registerTransition(STATE_9, STATE_9, 0, new String[]{STATE_9});
      fsm.registerTransition(STATE_9, STATE_10, 1);
      fsm.registerTransition(STATE_9, STATE_101, 2);
      fsm.registerTransition(STATE_101, STATE_101, 0, new
String[]{STATE_101});
      fsm.registerTransition(STATE_101, STATE_111, 1);
      fsm.registerTransition(STATE_111, STATE_111, 0, new
String[]{STATE_111});
      fsm.registerTransition(STATE_111, STATE_121, 1);
      fsm.registerTransition(STATE_121, STATE_121, 0, new
String[]{STATE_121});
      fsm.registerTransition(STATE_121, STATE_2, 1);
      fsm.registerTransition(STATE_10, STATE_10, 0, new
String[]{STATE_10});
      fsm.registerTransition(STATE_10, STATE_11, 1);
      fsm.registerTransition(STATE_11, STATE_11, 0, new
String[]{STATE_11});
      fsm.registerTransition(STATE_11, STATE_12, 1);
      fsm.registerTransition(STATE_12, STATE_12, 0, new
String[]{STATE_12});
      fsm.registerTransition(STATE_12, STATE_13, 1);
      fsm.registerTransition(STATE_13, STATE_13, 0, new
String[]{STATE_13});
      fsm.registerTransition(STATE_13, STATE_14, 1);
      fsm.registerTransition(STATE_14, STATE_14, 0, new
String[]{STATE_14});
      fsm.registerTransition(STATE_14, STATE_15, 1);
      fsm.registerTransition(STATE_15, STATE_15, 0, new
String[]{STATE_15});
      fsm.registerTransition(STATE_15, STATE_16, 1);
      fsm.registerTransition(STATE_16, STATE_16, 0, new
String[]{STATE_16});
      fsm.registerTransition(STATE_16, STATE_17, 1);
      fsm.registerTransition(STATE_17, STATE_17, 0, new
String[]{STATE_17});
      fsm.registerTransition(STATE_17, STATE_18, 1);
```

```
      //====Add behavior====
      addBehaviour(fsm);
   }

   public String generatePlaceBreakerInst()
   {
      String instStr = "Place open circuit breaker assemblies in the
following slots: ";

      for(int i=0;i<arraysPerFixture*breakersPerArray;i++)
      {
         if(i==0)
            instStr = instStr+(i+1);
         else
            instStr = instStr+", "+(i+1);
      }

      instStr = instStr+"//nMove away from the view of the inspection
camera and press NEXT to start inspection. There will be a 3 second delay
before inspection starts.";

      return instStr;
   }

   public String generateInspectionInst(int type)
   {
      String instStr = "";

      int num = 0;
      if(type==1)
         num = 2;
      else
         num = 1;

      for(int i=0;i<6;i++)
      if(i<arraysPerFixture*breakersPerArray)
         instStr = instStr+num;
      else
         instStr = instStr+"0";

      return instStr;
   }

   public String generateAssembleBreakerInst()
   {
      String instStr = "Place circuit breaker covers on all the assemblies
on the palate";
      instStr = instStr+"//nMove away from the view of the inspection
camera and press NEXT to start inspection. There will be a 3 second delay
before inspection starts.";

      return instStr;
   }

   public String generateStackBreakerInst()
   {
      String instStr = "";

      for(int i=0;i<arraysPerFixture;i++)
      {
         instStr = instStr+"Stack "+breakersPerArray+" circuit breakers on
position "+(i+1)+".//n";
```

```
        instStr = instStr+"Intest a temporary pin in one of the holes of
the stack.//n";
      }

      instStr = instStr+"Keep hands clear of the palate. Press NEXT to
alow the palate to move away.";

      return instStr;
   }

   public String generateRivitBreakerInst()
   {
      String instStr = "";

      String rivitLen = "";
      if(breakersPerArray==1)
         rivitLen = "single braker";
      else if(breakersPerArray==2)
         rivitLen = "double breaker";
      else if(breakersPerArray==3)
         rivitLen = "triple breaker";
      else if(breakersPerArray==4)
         rivitLen = "quadrouple breaker";


      for(int i=0;i<arraysPerFixture;i++)
      {
         instStr = instStr+"Remove the temporary pin from the stack in
position "+(i+1)+".//n";
         instStr = instStr+"Place "+rivitLen+" rivits in the four holes of
the stack in position "+(i+1)+".//n";
         instStr = instStr+"Place the stack in position "+(i+1)+" in the
riviting machine and wait for riviting to complete"+".//n";
         instStr = instStr+"Remove the breaker array from the riviting
machine and place it on the table"+".//n";
      }

      instStr = instStr+"Riviting instructions complete. Press NEXT
finish.";

      return instStr;
   }
}

//============================================================================
//Contract net initiator
//============================================================================
class CNI extends ContractNetInitiator
{
   Agent oh;
   String service, content;
   int successFlag;
   int stateNo;

   public CNI(Agent a, int stateNo1, String service1, String content1)
   {
      //invoke the constructor of the ContractNetInitiator class
      super(a,null);

      //save arguments to global variables
      oh = a;
      service = service1;
      content = content1;
```

```
      stateNo = stateNo1;
   }

   protected Vector prepareCfps(ACLMessage m)
   {
      System.out.println(oh.getLocalName()+": "+"STATE "+stateNo);

      //====Prepare the request message====
      ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
      cfp.setProtocol(FIPANames.InteractionProtocol.FIPA_CONTRACT_NET);
      cfp.setContent(content);

      //====search directory facilitator and add receivers====
      DFAgentDescription[] result = null;
      DFAgentDescription template = new DFAgentDescription();
      ServiceDescription sd = new ServiceDescription();
      sd.setType(service);
      template.addServices(sd);

      while(true)
      {
         System.out.println(oh.getLocalName()+": "+"Searching DF for
"+service);

         try {result = DFService.search(oh,template);}
         catch (FIPAException fe){fe.printStackTrace();}

         if(result.length!=0)
         {
            break;
         }
         else
         {
            System.out.println(oh.getLocalName()+": "+"Retrying search for
"+service);
            try{Thread.sleep(2*1000);}
            catch (InterruptedException e){e.printStackTrace();}
         }
      }

      for(int i=0;i<result.length;i++)
         cfp.addReceiver(result[i].getName());

      //====return ACL message to be sent====
      Vector v = new Vector(1);
      v.addElement(cfp);
      return v;
   }

   protected void handlePropose(ACLMessage propose, Vector acceptances)
   {
      System.out.println(oh.getLocalName()+": "+"Propose recieved from
"+propose.getSender().getLocalName());
   }

   protected void handleRefuse(ACLMessage refuse)
   {
      System.out.println(oh.getLocalName()+": "+"Refuse recieved from
"+refuse.getSender().getLocalName());
   }

   protected void handleInform(ACLMessage inform)
   {
```

```
      System.out.println(oh.getLocalName()+": "+"Inform recieved from
"+inform.getSender().getLocalName());
      successFlag = 1;
   }

   protected void handleFailure(ACLMessage inform)
   {
      System.out.println(oh.getLocalName()+": "+"Failure recieved from
"+inform.getSender().getLocalName());
      System.out.println(oh.getLocalName()+": "+"Retrying CFP");
      successFlag = 0;
   }

   protected void handleAllResponses(Vector responses, Vector
acceptances)
   {
      //====message variables====
      String acceptContent = "accept";
      String rejectContent = "reject";

      //====Determine best proposal====
      ACLMessage bestResponse = null;
      int propVal = 0;
      int bestPropVal = 0;

      Enumeration e = responses.elements();
      while (e.hasMoreElements())
      {
         ACLMessage response = (ACLMessage) e.nextElement();
         if (response.getPerformative() == ACLMessage.PROPOSE)
         {
            propVal = Integer.parseInt(response.getContent());
            if(propVal>bestPropVal)
            {
               bestResponse = response;
               bestPropVal = propVal;
            }

         }
      }

      //====Respond to all proposals====
      e = responses.elements();
      while (e.hasMoreElements())
      {
         ACLMessage response = (ACLMessage) e.nextElement();
         if (response.getPerformative() == ACLMessage.PROPOSE)
         {
            if (bestResponse == response)
            {
               ACLMessage accept = response.createReply();
               accept.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
               accept.setContent(acceptContent);
               acceptances.addElement(accept);
            }
            else
            {
               ACLMessage reject = response.createReply();
               reject.setPerformative(ACLMessage.REJECT_PROPOSAL);
               reject.setContent(rejectContent);
               acceptances.addElement(reject);
            }
         }
```

```
      }

      //====If there were no proposals wait and retry====
      if(bestResponse == null){
         try
         {
             Thread.sleep(2*1000);
             System.out.println(oh.getLocalName()+": "+"Retrying CFP");
            successFlag = 0;
         }
         catch(InterruptedException
ex){Thread.currentThread().interrupt();}
      }
   }

   public int onEnd()
   {
      return successFlag;
   }
}

//============================================================================
//Achieve RE initiator
//============================================================================
class ARI extends AchieveREInitiator
{
   Agent oh;
   String service, content;
   int successFlag;
   int stateNo;

   public ARI(Agent a, int stateNo1, String service1, String content1)
   {
      //invoke the constructor of the ContractNetInitiator class
      super(a,null);

      //save arguments to global variables
      oh = a;
      service = service1;
      content = content1;
      stateNo = stateNo1;
   }

   protected Vector prepareRequests(ACLMessage request)
   {
      System.out.println(oh.getLocalName()+": "+"STATE "+stateNo);
      System.out.println(oh.getLocalName()+": "+"Searching DF for
"+service);

      DFAgentDescription[] result = null;
      DFAgentDescription template = new DFAgentDescription();
      ServiceDescription sd = new ServiceDescription();
      sd.setType(service);
      template.addServices(sd);

      while(true)
      {
         System.out.println(oh.getLocalName()+": "+"Searching DF for
"+service);

         try {result = DFService.search(oh,template);}
         catch (FIPAException fe){fe.printStackTrace();}
```

```
            if(result.length!=0)
            {
               break;
            }
            else
            {
               System.out.println(oh.getLocalName()+": "+"Retrying search for
"+service);
               try{Thread.sleep(2*1000);}
               catch (InterruptedException e){e.printStackTrace();}
            }
         }
         AID reciever = result[0].getName();

         //Prepare the request message
         ACLMessage rmsg = new ACLMessage(ACLMessage.REQUEST);
         rmsg.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
         rmsg.addReceiver(reciever);
         rmsg.setContent(content);
         Vector v = new Vector(1);
         v.addElement(rmsg);
         return v;
      }

      protected void handleAgree(ACLMessage agree)
      {
         System.out.println(oh.getLocalName()+": Agree recieved from
"+agree.getSender().getLocalName());
      }

      protected void handleInform(ACLMessage inform)
      {
         System.out.println(oh.getLocalName()+": Inform recieved from
"+inform.getSender().getLocalName());
         if(inform.getContent().equals("1"))
            successFlag = 1;
         else if(inform.getContent().equals("0"))
            successFlag = 0;
         else if(inform.getContent().equals("2"))
            successFlag = 2;
      }

      protected void handleRefuse(ACLMessage refuse)
      {
         System.out.println(oh.getLocalName()+": Refuse recieved from
"+refuse.getSender().getLocalName());
         successFlag = 0;
      }

      protected void handleFailure(ACLMessage failure)
      {
         System.out.println(oh.getLocalName()+": Failure received from
"+failure.getSender().getLocalName());
         System.out.println(oh.getLocalName()+": Retrying request");
         successFlag = 0;
      }

      public int onEnd()
      {
         return successFlag;
      }
}
```

```
class skipState extends OneShotBehaviour
{
   @Override
   public void action()
   {
      System.out.println("State 12");
   }

   public int onEnd()
   {
      return 1;
   }
}
```

A.2: Interface agent code

```
public class InterfaceAgent extends Agent
{
   //Globals
   private InterfaceAgent hwh = this;
   private String
interfaceID,IPAddress,servicesStr,currentWorker,previousWorker;
   private Socket acs;

   //status variables
   private boolean busy = false;
   private boolean fail = false;
   private boolean registered = false;
   private boolean breakStatus =false;
   private String currentStation = "0";

   //queue variables
   private int actionQueueNo = 0;
   private int currentAction = 0;

   //Setup method
   public void setup()
   {
      //=============================================================
      // general setup
      //=============================================================
      System.out.println(hwh.getLocalName()+": "+"Human worker holon
started");

      //extract data from arguments
      interfaceID = (String)hwh.getArguments()[0];
      IPAddress = (String)hwh.getArguments()[1];
      servicesStr = (String)hwh.getArguments()[2];

      //=============================================================
      // establish connection with interface app
      // =============================================================
      try
      {
         //open socket and establish connection
         System.out.println("CONNECTING TO: "+IPAddress);
         acs = new Socket(IPAddress,6004);//change
         sendToInterface("connection request "+interfaceID);

         //wait for listener on app to start
```

```
         try{Thread.sleep(5000);}
         catch (InterruptedException e){e.printStackTrace();}

         //start local listener
         new Thread(new TCPIPListenThread1(hwh, acs)).start();

         //wait for listener on app to start
         try{Thread.sleep(5000);}
         catch (InterruptedException e){e.printStackTrace();}
      }
      catch (IOException e){e.printStackTrace();}

      //=============================================================
      // SS responder dispatcher
      //=============================================================
      MessageTemplate template = MessageTemplate.and(

   MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_CONTR
ACT_NET),
      MessageTemplate.MatchPerformative(ACLMessage.CFP));

      //add behaviour to launch a Responder behaviour for every incoming
request
      addBehaviour(new SSResponderDispatcher(this,template)
      {
         @SuppressWarnings("serial")
         @Override
         public Behaviour createResponder(ACLMessage request)
         {
            System.out.println(getLocalName()+": REQUEST received from
"+request.getSender().getLocalName());
            System.out.println(getLocalName()+": Creating a Responder for
received request");

            ThreadedBehaviourFactory tbf = new ThreadedBehaviourFactory();

            //create responder
            SSContractNetResponder responder = new
SSContractNetResponder(hwh, request)
            {
               @Override
               protected ACLMessage handleCfp(ACLMessage cfp) throws
NotUnderstoodException, RefuseException
               {
                  System.out.println(hwh.getLocalName()+": CFP Recieved
from "+cfp.getSender().getLocalName());

                  // Decode message
                  String content = cfp.getContent();

                  // Evaluate action and form response
                  int proposalScore = evaluateAction(content);
                  if (proposalScore > 0)
                  {
                     // Provide a proposal
                     System.out.println(hwh.getLocalName()+": Proposing
"+proposalScore);
                     ACLMessage propose = cfp.createReply();
                     propose.setPerformative(ACLMessage.PROPOSE);
                     propose.setContent(Integer.toString(proposalScore));
                     return propose;
                  }
                  else
```

```
                {
                   // Refuse to provide a proposal
                   System.out.println(hwh.getLocalName()+": Refusing
proposal");
                   ACLMessage refuse = cfp.createReply();
                   refuse.setPerformative(ACLMessage.REFUSE);
                   refuse.setContent("refuse");
                   return refuse;
                }
             }

             @Override
             protected ACLMessage handleAcceptProposal(ACLMessage cfp,
ACLMessage propose, ACLMessage accept)throws FailureException
             {
                System.out.println(hwh.getLocalName()+": Accept Recieved
from "+cfp.getSender().getLocalName());

                //get content
                String content = cfp.getContent();

                //perform action
                ACLMessage inform = null;
                if(performAction(content))
                {
                   //create inform to send to order holon
                   inform = accept.createReply();
                   inform.setPerformative(ACLMessage.INFORM);
                   inform.setContent("1");
                }
                else
                {
                   //create failure to send to order holon
                   inform = accept.createReply();
                   inform.setPerformative(ACLMessage.FAILURE);
                   inform.setContent("0");
                }

                //wait is necessary
                try {Thread.sleep(1000);}
                catch (InterruptedException e){e.printStackTrace();}

                return inform;
             }

             protected void handleRejectProposal(ACLMessage cfp,
ACLMessage propose, ACLMessage reject)
             {
                System.out.println(hwh.getLocalName()+": Reject Recieved
from "+cfp.getSender().getLocalName());
             }
          };

          //terminate behavior when the current session ends
          //responder.closeSessionOnNextReply();
          return tbf.wrap(responder);
       }
    });
  }

  public int evaluateAction(String content)
  {
     System.out.println(hwh.getLocalName()+": Action evaluating");
```

```
      //Determine proposal score
      int score = 0;
      //random component
      score = score + (int) Math.random()*99+1;

      System.out.println(hwh.getLocalName()+": Action evaluated. Score is
"+score);

      return score;
   }

   public boolean performAction(String instructionsStr)
   {
      System.out.println(hwh.getLocalName()+": Action performing");

      //set busy flag
      busy = true;

      //decode
      String taskNo = instructionsStr.substring(0, 1);
      String workstationNo = instructionsStr.substring(1, 2);
      String instStr = instructionsStr.substring(2);

      //send instructions to interface
      String msgStr = "newInstructions"+"//n"+instStr;
      sendToInterface(msgStr);

      //send execution start command and operation id number to
performance tracker

   SendToPerformanceTracker("executionStart"+"//n"+currentWorker+"//n"+in
terfaceID+"//n"+instructionsStr.substring(0, 1));

      //wait for busy flag to be reset via message from interface
      try {Thread.sleep(1000);}
      catch (InterruptedException e){e.printStackTrace();}

      while(true)
      {
         if(busy==false)
         {
            System.out.println(hwh.getLocalName()+": Action completed");
            currentAction++;

            //send execution stop command and operation id number to
performance tracker

   SendToPerformanceTracker("executionStop"+"//n"+currentWorker+"//n"+int
erfaceID+"//n"+"Completed");

            return true;
         }
         else if(registered==false)
         {
            System.out.println(hwh.getLocalName()+": Action aborted");
            currentAction++;

            //send execution stop command and operation id number to
performance tracker

   SendToPerformanceTracker("executionStop"+"//n"+previousWorker+"//n"+in
terfaceID+"//n"+"Aborted");
```

```
            return false;
         }

         try {Thread.sleep(100);}
         catch (InterruptedException e){e.printStackTrace();}
      }
   }

   public void sendToInterface(String msgStr)
     {
         //get socket
         Socket s1 = acs;

         //send string over socket
         try
         {
            PrintWriter pw1 = new PrintWriter(s1.getOutputStream());
            pw1.println(msgStr);
            pw1.flush();
         }
         catch (IOException e){e.printStackTrace();}
     }

   public void messageReceived(String msgStr)
   {
      //read string into custom scanner
         Scanner scan1 = new Scanner(msgStr).useDelimiter("//n");
         String command = scan1.next();

      if(command.equals("instructionsCompleted"))
      {
         System.out.println(hwh.getLocalName()+" : Message received from
inteface: Instructions completed");
         busy = false;
      }
      else if(command.equals("login"))
      {
         System.out.println(hwh.getLocalName()+" : Message received from
inteface: Login");
         logIn(scan1.next());
      }
      else if(command.equals("logout"))
      {
         System.out.println(hwh.getLocalName()+" : Message received from
inteface: Logout");
         logOut();
      }
   }

   public void SendToPerformanceTracker(String s)
   {
      AchieveREInitiator init = new AchieveREInitiator(hwh, null)
      {
         protected Vector prepareRequests(ACLMessage request)
         {
            System.out.println(hwh.getLocalName()+" : Sending command to
performance tracker");

            //search directory fasilitator
            DFAgentDescription[] result = null;
            DFAgentDescription template = new DFAgentDescription();
            ServiceDescription sd = new ServiceDescription();
```

```
            sd.setType("performance tracker");
            template.addServices(sd);
            try {result = DFService.search(hwh,template);}
            catch (FIPAException fe){fe.printStackTrace();}
            AID reciever = result[0].getName();

            //Prepare the request message
            ACLMessage rmsg = new ACLMessage(ACLMessage.REQUEST);
            rmsg.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
            rmsg.addReceiver(reciever);
            rmsg.setContent(s);
            Vector v = new Vector(1);
            v.addElement(rmsg);
            return v;
         }

         protected void handleInform(ACLMessage inform)
         {
            System.out.println(hwh.getLocalName()+" : Inform recieved from
"+inform.getSender().getLocalName());
         }

         protected void handleRefuse(ACLMessage refuse)
         {
            System.out.println(hwh.getLocalName()+" : Refuse recieved from
"+refuse.getSender().getLocalName());
         }

         protected void handleFailure(ACLMessage failure)
         {
            System.out.println(hwh.getLocalName()+" : Failure recieved
from "+failure.getSender().getLocalName());
         }
      };
      hwh.addBehaviour(init);

      //wait for communications to complete
      try {Thread.sleep(500);}
      catch (InterruptedException e){e.printStackTrace();}
   }

   public void SendToSafteyMonitor(String s)
   {
      AchieveREInitiator init = new AchieveREInitiator(hwh, null)
      {
         protected Vector prepareRequests(ACLMessage request){
            //search directory fasilitator
            DFAgentDescription[] result = null;
            DFAgentDescription template = new DFAgentDescription();
            ServiceDescription sd = new ServiceDescription();
            sd.setType("safety monitor");
            template.addServices(sd);
            try {result = DFService.search(hwh,template);}
            catch (FIPAException fe){fe.printStackTrace();}
            AID reciever = result[0].getName();

            //Prepare the request message
            ACLMessage rmsg = new ACLMessage(ACLMessage.REQUEST);
            rmsg.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
            rmsg.addReceiver(reciever);
            rmsg.setContent(s);
            Vector v = new Vector(1);
            v.addElement(rmsg);
```

```
            return v;
         }

         protected void handleInform(ACLMessage inform){
            System.out.println(hwh.getAID().getName()+" : Inform recieved
from "+inform.getSender().getName());
         }

         protected void handleRefuse(ACLMessage refuse) {
            System.out.println(hwh.getAID().getName()+" : Refuse recieved
from "+refuse.getSender().getName());
         }

         protected void handleFailure(ACLMessage failure) {
            System.out.println(hwh.getAID().getName()+" : Failure recieved
from "+failure.getSender().getName());
         }
      };
      hwh.addBehaviour(init);
   }

   public void logIn(String workerID)
   {
      currentWorker = workerID;

      //send session start command to performance tracker

   SendToPerformanceTracker("sessionStart"+"//n"+currentWorker+"//n"+inte
rfaceID);

      registerService();
   }

   public void logOut()
   {
      if((busy==false)&&(registered==true))
      {
         //send session stop command to performance tracker

   SendToPerformanceTracker("sessionStop"+"//n"+currentWorker+"//n"+inter
faceID);

         //De-register from the DF
         previousWorker = currentWorker;
         currentWorker = null;
         deRegisterService();
      }
      else if((busy==true)&&(registered==true))
      {
         //send session stop command to performance tracker

   SendToPerformanceTracker("sessionStop"+"//n"+currentWorker+"//n"+inter
faceID);

         //De-register from the DF
         previousWorker = currentWorker;
         currentWorker = null;
         deRegisterService();

         //wait for failures to send
         try {Thread.sleep(5000);}
         catch (InterruptedException e){e.printStackTrace();}
      }
```

```
    }

    public void terminate()
    {
        if((busy==false)&&(registered==true))
        {
            logOut();

            //kill agent
            System.out.println(hwh.getLocalName()+" : killing agent");
            hwh.doDelete();
        }
        else if((busy==true)&&(registered==true))
        {
            logOut();

            //wait for failures to send
            try {Thread.sleep(5000);}
            catch (InterruptedException e){e.printStackTrace();}

            //kill agent
            System.out.println(hwh.getLocalName()+" : killing agent");
            hwh.doDelete();
        }
        else if((busy==false)&&(registered==false))
        {
            logOut();

            //kill agent
            System.out.println(hwh.getLocalName()+" : killing agent");
            hwh.doDelete();
        }
    }

    public void registerService()
    {
        System.out.println(hwh.getLocalName()+": "+"Registering service with
DF");

        //Register all services of the user in the DF
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(getAID());

        Scanner sc1 = new Scanner(servicesStr).useDelimiter("//n");
        while(sc1.hasNext())
        {

            ServiceDescription sd = new ServiceDescription();
            String serviseDescriptionStr = sc1.next();
            sd.setType(serviseDescriptionStr);
            sd.setName(getLocalName()+"-"+serviseDescriptionStr);
            dfd.addServices(sd);
        }

        try {DFService.register(this, dfd);}
        catch (FIPAException fe) {fe.printStackTrace();}
        registered = true;
    }

    public void deRegisterService()
    {
        System.out.println(hwh.getLocalName()+": "+"De-registering service
with DF");
```

```
      //De-register from the DF
      try{DFService.deregister(hwh);}
      catch (FIPAException fe){fe.printStackTrace();}
      registered = false;

      //reset current station
      currentStation = "0";
   }

   protected void TakeDown()
   {
      logOut();
      deRegisterService();
   }
}

class TCPIPListenThread1 implements Runnable
{
   InterfaceAgent hh;
   Socket acs;

   public TCPIPListenThread1(InterfaceAgent hh1, Socket acs1)
   {
      hh = hh1;
      acs = acs1;
   }

   public void run()
   {
      while(true)
      {
         try
         {
            BufferedReader br = new BufferedReader(new
InputStreamReader(acs.getInputStream()));
            System.out.println(hh.getAID().getLocalName()+" : Listening
for message from interface");
            String str1 = br.readLine();

            if(str1==null)
            {
               break;
            }
            else
            {
               hh.messageReceived(str1);
            }
         }
         catch (IOException e){e.printStackTrace();}
      }
   }
}
```

A.3: Worker agent code

```
public class WorkerAgent extends Agent
{
   //Globals
   private WorkerAgent hwh = this;
```

```
   private String workerID,IPAddress,servicesStr;
   private Socket acs;

   //status variables
   private boolean busy = false;
   private boolean fail = false;
   private boolean registered = false;
   private boolean breakStatus =false;
   private String currentStation = "0";

   //queue variables
   private int actionQueueNo = 0;
   private int currentAction = 0;

   //Setup method
   public void setup()
   {
      // ================================================================
      // general setup
      // ================================================================
      System.out.println(hwh.getLocalName()+": "+"Human worker holon
started");

      //extract data from arguments
      workerID = (String)hwh.getArguments()[0];
      IPAddress = (String)hwh.getArguments()[1];
      servicesStr = (String)hwh.getArguments()[2];

      //Register service in the DF
      registerService();

      //send session start command to performance tracker
      SendToPerformanceTracker("sessionStart"+"//n"+workerID+"//n"+"n/a");

      // ================================================================
      // establish connection with interface app
      // ================================================================
      try
      {
         //open socket and establish connection
         acs = new Socket(IPAddress,6002);
         sendToInterface("connection request "+workerID);

         //wait for listener on app to start
         try{Thread.sleep(5000);}
         catch (InterruptedException e){e.printStackTrace();}

         //start local listener
         new Thread(new TCPIPListenThread(hwh, acs)).start();

         //wait for listener on app to start
         try{Thread.sleep(5000);}
         catch (InterruptedException e){e.printStackTrace();}
      }
      catch (IOException e){e.printStackTrace();}

      // ================================================================
      // SS responder dispatcher
      // ================================================================
      MessageTemplate template = MessageTemplate.and(

   MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_CONTR
ACT_NET),
```

```
      MessageTemplate.MatchPerformative(ACLMessage.CFP));

      //add behaviour to launch a Responder behaviour for every incoming
request
      addBehaviour(new SSResponderDispatcher(this,template)
      {
         @SuppressWarnings("serial")
         @Override
         public Behaviour createResponder(ACLMessage request)
         {
            System.out.println(getLocalName()+": REQUEST received from
"+request.getSender().getLocalName());
            System.out.println(getLocalName()+": Creating a Responder for
received request");

            ThreadedBehaviourFactory tbf = new ThreadedBehaviourFactory();

            //create responder
            SSContractNetResponder responder = new
SSContractNetResponder(hwh, request)
            {
               @Override
               protected ACLMessage handleCfp(ACLMessage cfp) throws
NotUnderstoodException, RefuseException
               {
                  System.out.println(hwh.getLocalName()+": CFP Recieved
from "+cfp.getSender().getLocalName());

                  // Decode message
                  String content = cfp.getContent();

                  // Evaluate action and form response
                  int proposalScore = evaluateAction(content);

                  // Provide a proposal
                  System.out.println(hwh.getLocalName()+": Proposing
"+proposalScore);
                  ACLMessage propose = cfp.createReply();
                  propose.setPerformative(ACLMessage.PROPOSE);
                  propose.setContent(Integer.toString(proposalScore));
                  return propose;
               }

               @Override
               protected ACLMessage handleAcceptProposal(ACLMessage cfp,
ACLMessage propose, ACLMessage accept)throws FailureException
               {
                  System.out.println(hwh.getLocalName()+": Accept Received
from" +cfp.getSender().getLocalName());

                  //get content
                  String content = cfp.getContent();

                  //book action
                  int queueNo = bookAction();

                  //perform action
                  ACLMessage inform = null;
                  while(true)
                  {
                     if(queueNo==currentAction)
                     {
                        if(performAction(content))
```

```
                        {
                           //create inform to send to order holon
                           inform = accept.createReply();
                           inform.setPerformative(ACLMessage.INFORM);
                           inform.setContent("1");
                        }
                        else
                        {
                           //create failure to send to order holon
                           inform = accept.createReply();
                           inform.setPerformative(ACLMessage.FAILURE);
                           inform.setContent("0");
                        }
                        break;
                     }

                     //wait is necessary
                     try {Thread.sleep(1000);}
                     catch (InterruptedException e){e.printStackTrace();}
                  }

                  return inform;
               }

               protected void handleRejectProposal(ACLMessage cfp,
ACLMessage propose, ACLMessage reject)
               {
                  System.out.println(hwh.getLocalName()+": Reject Recieved
from "+cfp.getSender().getLocalName());
               }
            };

            //terminate behavior when the current session ends
            //responder.closeSessionOnNextReply();
            return tbf.wrap(responder);
         }
      });
   }

   public int evaluateAction(String content)
   {
      System.out.println(hwh.getLocalName()+": Action evaluating");

      //Determine proposal score
      int score = 0;
      //random component (100)
      score = (int) (score + Math.round(Math.random()*100));
      //workstation component (1000)
      System.out.println(hwh.getLocalName()+": content
"+content.substring(1,2));
      if(currentStation.equals(content.substring(1,2)))
      {
         System.out.println(hwh.getLocalName()+": at workstation");
         score = score + 2000;
      }
      else if(currentStation.equals("0"))
      {
         System.out.println(hwh.getLocalName()+": not at workstation");
         score = score + 1000;
      }

      //queue component (-100*queue)
      score = score - 100*(actionQueueNo-currentAction);
```

```
      System.out.println(hwh.getLocalName()+": Action evaluated. Score is
"+score);

      return score;
   }

   public int bookAction()
   {
      int queueNo = actionQueueNo;
      actionQueueNo++;

      return queueNo;
   }

   public boolean performAction(String instructionsStr)
   {
      System.out.println(hwh.getLocalName()+": Action performing");

      //set busy flag
      busy = true;

      //decode
      String taskNo = instructionsStr.substring(0, 1);
      String workstationNo = instructionsStr.substring(1, 2);
      String instStr = instructionsStr.substring(2);

      //add move to workstation instruction
      if(!(workstationNo.equals(currentStation)))
      {
         instStr = "Move to workstation "+workstationNo+"//n"+instStr;

         //set new current station
         currentStation = workstationNo;
      }

      //send instructions to interface
      String msgStr = "newInstructions"+"//n"+instStr;
      sendToInterface(msgStr);

      //send execution start command and operation id number to
performance tracker

   SendToPerformanceTracker("executionStart"+"//n"+workerID+"//n"+worksta
tionNo+"//n"+instructionsStr.substring(0, 1));

      //wait for busy flag to be reset via message from interface
      try {Thread.sleep(1000);}
      catch (InterruptedException e){e.printStackTrace();}

      while(true)
      {
         if(busy==false)
         {
            System.out.println(hwh.getLocalName()+": Action completed");
            currentAction++;

            //send execution stop command and operation id number to
performance tracker

   SendToPerformanceTracker("executionStop"+"//n"+workerID+"//n"+workstat
ionNo+"//n"+"Completed");
```

```
            return true;
         }
         else if(registered==false)
         {
            System.out.println(hwh.getLocalName()+": Action aborted");
            currentAction++;

            //send execution stop command and operation id number to
performance tracker

   SendToPerformanceTracker("executionStop"+"//n"+workerID+"//n"+workstat
ionNo+"//n"+"Aborted");

            return false;
         }

         try {Thread.sleep(100);}
         catch (InterruptedException e){e.printStackTrace();}
      }
   }

   public void sendToInterface(String msgStr)
     {
         //get socket
         Socket s1 = acs;

         //send string over socket
         try
         {
            PrintWriter pw1 = new PrintWriter(s1.getOutputStream());
            pw1.println(msgStr);
            pw1.flush();
         }
         catch (IOException e){e.printStackTrace();}
     }

   public void messageReceived(String msgStr)
   {
      //read string into custom scanner
         Scanner scan1 = new Scanner(msgStr).useDelimiter("//n");
         String command = scan1.next();

      if(command.equals("instructionsCompleted"))
      {
         System.out.println(hwh.getLocalName()+" : Message received from
inteface: Instructions completed");
         busy = false;
      }
      else if(command.equals("logout"))
      {
         System.out.println(hwh.getLocalName()+" : Message received from
inteface: Logout");
         logOut();
      }
      else if(command.equals("break"))
      {
         System.out.println(hwh.getLocalName()+" : Message received from
inteface: Break");
         Break();
      }
   }

   public void SendToPerformanceTracker(String s)
```

```
   {
      AchieveREInitiator init = new AchieveREInitiator(hwh, null)
      {
         protected Vector prepareRequests(ACLMessage request)
         {
            System.out.println(hwh.getLocalName()+" : Sending command to
performance tracker");

            //search directory fasilitator
            DFAgentDescription[] result = null;
            DFAgentDescription template = new DFAgentDescription();
            ServiceDescription sd = new ServiceDescription();
            sd.setType("performance tracker");
            template.addServices(sd);
            try {result = DFService.search(hwh,template);}
            catch (FIPAException fe){fe.printStackTrace();}
            AID reciever = result[0].getName();

            //Prepare the request message
            ACLMessage rmsg = new ACLMessage(ACLMessage.REQUEST);
            rmsg.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
            rmsg.addReceiver(reciever);
            rmsg.setContent(s);
            Vector v = new Vector(1);
            v.addElement(rmsg);
            return v;
         }

         protected void handleInform(ACLMessage inform)
         {
            System.out.println(hwh.getLocalName()+" : Inform recieved from
"+inform.getSender().getLocalName());
         }

         protected void handleRefuse(ACLMessage refuse)
         {
            System.out.println(hwh.getLocalName()+" : Refuse recieved from
"+refuse.getSender().getLocalName());
         }

         protected void handleFailure(ACLMessage failure)
         {
            System.out.println(hwh.getLocalName()+" : Failure recieved
from "+failure.getSender().getLocalName());
         }
      };
      hwh.addBehaviour(init);

      //wait for communications to complete
      try {Thread.sleep(500);}
      catch (InterruptedException e){e.printStackTrace();}
   }

   public void SendToSafteyMonitor(String s)
   {
      AchieveREInitiator init = new AchieveREInitiator(hwh, null)
      {
         protected Vector prepareRequests(ACLMessage request){
            //search directory fasilitator
            DFAgentDescription[] result = null;
            DFAgentDescription template = new DFAgentDescription();
            ServiceDescription sd = new ServiceDescription();
            sd.setType("safety monitor");
```

```
            template.addServices(sd);
            try {result = DFService.search(hwh,template);}
            catch (FIPAException fe){fe.printStackTrace();}
            AID reciever = result[0].getName();

            //Prepare the request message
            ACLMessage rmsg = new ACLMessage(ACLMessage.REQUEST);
            rmsg.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
            rmsg.addReceiver(reciever);
            rmsg.setContent(s);
            Vector v = new Vector(1);
            v.addElement(rmsg);
            return v;
         }

         protected void handleInform(ACLMessage inform){
            System.out.println(hwh.getAID().getName()+" : Inform recieved
from "+inform.getSender().getName());
         }

         protected void handleRefuse(ACLMessage refuse) {
            System.out.println(hwh.getAID().getName()+" : Refuse recieved
from "+refuse.getSender().getName());
         }

         protected void handleFailure(ACLMessage failure) {
            System.out.println(hwh.getAID().getName()+" : Failure recieved
from "+failure.getSender().getName());
         }
      };
      hwh.addBehaviour(init);
   }

   public void logOut()
   {
      if((busy==false)&&(registered==true))
      {
         //De-register from the DF
         deRegisterService();

   SendToPerformanceTracker("sessionStop"+"//n"+workerID+"//n"+"n/a");

         //kill agent
         System.out.println(hwh.getLocalName()+" : killing agent");
         hwh.doDelete();
      }
      else if((busy==true)&&(registered==true))
      {
         System.out.println(hwh.getLocalName()+" : De-registering agent");

         //De-register from the DF
         deRegisterService();

   SendToPerformanceTracker("executionStop"+"//n"+workerID+"//n"+"n/a"+"/
/n"+"Aborted");

   SendToPerformanceTracker("sessionStop"+"//n"+workerID+"//n"+"n/a");

         //wait for failures to send
         try {Thread.sleep(5000);}
         catch (InterruptedException e){e.printStackTrace();}

         //kill agent
```

```
            System.out.println(hwh.getLocalName()+" : killing agent");
            hwh.doDelete();
        }
    }

    public void Break()
    {
        if(registered==true)
        {
            System.out.println(hwh.getLocalName()+" : De-registering agent");

            //De-register from the DF
            deRegisterService();

    SendToPerformanceTracker("breakStart"+"//n"+workerID+"//n"+"n/a");
        }
        else if(registered==false)
        {
            System.out.println(hwh.getLocalName()+" : Registering agent");

            //register from the DF
            registerService();
            SendToPerformanceTracker("breakStop"+"//n"+workerID+"//n"+"n/a");
        }
    }

    public void registerService()
    {
        System.out.println(hwh.getLocalName()+": "+"Registering service with
DF");

        //Register all services of the user in the DF
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(getAID());

        Scanner sc1 = new Scanner(servicesStr).useDelimiter("//n");
        while(sc1.hasNext())
        {

            ServiceDescription sd = new ServiceDescription();
            String serviseDescriptionStr = sc1.next();
            sd.setType(serviseDescriptionStr);
            sd.setName(getLocalName()+"-"+serviseDescriptionStr);
            dfd.addServices(sd);
        }

        try {DFService.register(this, dfd);}
        catch (FIPAException fe) {fe.printStackTrace();}
        registered = true;
    }

    public void deRegisterService()
    {
        System.out.println(hwh.getLocalName()+": "+"De-registering service
with DF");

        //De-register from the DF
        try{DFService.deregister(hwh);}
        catch (FIPAException fe){fe.printStackTrace();}
        registered = false;

        //reset current station
        currentStation = "0";
```

```
   }

   protected void takedown()
   {
      logOut();
   }
}

class TCPIPListenThread implements Runnable
{
   WorkerAgent hh;
   Socket acs;

   public TCPIPListenThread(WorkerAgent hh1, Socket acs1)
   {
      hh = hh1;
      acs = acs1;
   }

   public void run()
   {
      while(true)
      {
         try
         {
            BufferedReader br = new BufferedReader(new
InputStreamReader(acs.getInputStream()));
            System.out.println(hh.getAID().getLocalName()+" : Listening
for message from interface");
            String str1 = br.readLine();

            if(str1==null)
            {
               break;
            }
            else
            {
               hh.messageReceived(str1);
            }
         }
         catch (IOException e){e.printStackTrace();}
      }
   }
}
```

A.4: Supervisor agent code

```
public class SupervisorAgent extends Agent
{
   //agent
   SupervisorAgent sh = this;

   //action queue variables
   int currentActionNo = 0;
   int queueActionNo = 0;

   @SuppressWarnings("serial")
   public void setup()
   {
      System.out.println(getLocalName()+": "+"Supervisor holon started");
```

```
      //Register service in the DF
      DFAgentDescription dfd = new DFAgentDescription();
      dfd.setName(getAID());
      ServiceDescription sd = new ServiceDescription();
      sd.setType("supervisor");
      sd.setName(getLocalName()+"-supervisor");
      dfd.addServices(sd);
      try {DFService.register(this, dfd);}
      catch (FIPAException fe) {fe.printStackTrace();}

      // ============================================================
      // SS responder dispatcher
      // ============================================================
      MessageTemplate template = MessageTemplate.and(

   MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_CONTR
ACT_NET),
      MessageTemplate.MatchPerformative(ACLMessage.CFP));

      //add behaviour to launch a Responder behaviour for every incoming
request
      addBehaviour(new SSResponderDispatcher(this,template)
      {
         @SuppressWarnings("serial")
         @Override
         public Behaviour createResponder(ACLMessage request)
         {
            System.out.println(getLocalName()+": REQUEST received from
"+request.getSender().getLocalName());
            System.out.println(getLocalName()+": Creating a Responder for
received request");

            ThreadedBehaviourFactory tbf = new ThreadedBehaviourFactory();

            DataStore ds;

            //create responder
            SSContractNetResponder responder = new
SSContractNetResponder(sh, request)
            {
               @Override
               protected ACLMessage handleCfp(ACLMessage cfp) throws
NotUnderstoodException, RefuseException
               {
                  System.out.println(sh.getLocalName()+": CFP Recieved from
"+cfp.getSender().getLocalName());

                  // Provide a proposal
                  ACLMessage propose = cfp.createReply();
                  propose.setPerformative(ACLMessage.PROPOSE);
                  propose.setContent(Integer.toString(1));
                  return propose;
               }

               protected void handleRejectProposal(ACLMessage cfp,
ACLMessage propose, ACLMessage reject)
               {
                  System.out.println(sh.getLocalName()+": Reject Recieved
from "+cfp.getSender().getLocalName());
               }
            };
```

```
            //register CNI to handle request state
            @SuppressWarnings("resource")
            Scanner scan1 = new
Scanner(request.getContent()).useDelimiter("///n");
            String service = scan1.next();
            String content = scan1.next();
            responder.registerHandleAcceptProposal(new
CNIsh(sh,service,content,responder));

            return tbf.wrap(responder);
         }
      });
   }

   public int bookAction()
   {
      int queueNo = 0;

      //add to booking list
      queueNo = queueActionNo;
      queueActionNo++;

      return queueNo;
   }
}

//============================================================
//Contract net initiator
//============================================================
class CNIsh extends ContractNetInitiator
{
   SupervisorAgent sh;
   String service, content;
   SSContractNetResponder responder;

   public CNIsh(SupervisorAgent a, String service1, String content1,
SSContractNetResponder responder1)
   {
      //invoke the constructor of the ContractNetInitiator class
      super(a,null);

      //save arguments to global variables
      sh = a;
      service = service1;
      content = content1;
      responder = responder1;
   }

   protected Vector prepareCfps(ACLMessage m)
   {
      //book action wait for place in action queue
      int queueNo = sh.bookAction();
      while(true)
      {
         //check if this action is next in line
         if(queueNo==sh.currentActionNo)
         {
            break;
         }

         //sleep for 1 second before retry
         try{Thread.sleep(500);}
         catch(InterruptedException e){e.printStackTrace();}
```

```
      }

      //====Prepare the request message====
      ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
      cfp.setProtocol(FIPANames.InteractionProtocol.FIPA_CONTRACT_NET);
      cfp.setContent(content);

      //====search directory facilitator and add receivers====
      DFAgentDescription[] result = null;
      DFAgentDescription template = new DFAgentDescription();
      ServiceDescription sd = new ServiceDescription();
      sd.setType(service);
      template.addServices(sd);

      while(true)
      {
         System.out.println(sh.getLocalName()+": "+"Searching DF for
"+service);

         try {result = DFService.search(sh,template);}
         catch (FIPAException fe){fe.printStackTrace();}

         if(result.length!=0)
         {
            break;
         }
         else
         {
            System.out.println(sh.getLocalName()+": "+"Retrying search for
"+service);
            try{Thread.sleep(2*1000);}
            catch (InterruptedException e){e.printStackTrace();}
         }
      }

      for(int i=0;i<result.length;i++)
         cfp.addReceiver(result[i].getName());

      //====return ACL message to be sent====
      Vector v = new Vector(1);
      v.addElement(cfp);
      return v;
   }

   protected void handlePropose(ACLMessage propose, Vector acceptances)
   {
      System.out.println(sh.getLocalName()+": "+"Propose recieved from
"+propose.getSender().getLocalName());
   }

   protected void handleRefuse(ACLMessage refuse)
   {
      System.out.println(sh.getLocalName()+": "+"Refuse recieved from
"+refuse.getSender().getLocalName());
   }

   protected void handleInform(ACLMessage inform)
   {
      System.out.println(sh.getLocalName()+": "+"Inform recieved from
"+inform.getSender().getLocalName());

      //put inform message in data store at reply key
```

```
      ACLMessage informOH =
((ACLMessage)getDataStore().get(responder.ACCEPT_PROPOSAL_KEY)).createRep
ly();
      informOH.setPerformative(ACLMessage.INFORM);
      informOH.setContent("1");
      getDataStore().put(responder.REPLY_KEY,informOH);
   }

   protected void handleFailure(ACLMessage inform)
   {
      System.out.println(sh.getLocalName()+": "+"Failure recieved from
"+inform.getSender().getLocalName());

      //put inform message in data store at reply key
      ACLMessage failureOH =
((ACLMessage)getDataStore().get(CFP_KEY)).createReply();;
      failureOH.setPerformative(ACLMessage.FAILURE);
      failureOH.setContent("1");
      getDataStore().put(REPLY_KEY,failureOH);
   }

   protected void handleAllResponses(Vector responses, Vector
acceptances)
   {
      //====message variables====
      String acceptContent = "accept";
      String rejectContent = "reject";

      //====Determine best proposal====
      ACLMessage bestResponse = null;
      int propVal = 0;
      int bestPropVal = -1000;

      Enumeration e = responses.elements();
      while (e.hasMoreElements())
      {
         ACLMessage response = (ACLMessage) e.nextElement();
         if (response.getPerformative() == ACLMessage.PROPOSE)
         {
            propVal = Integer.parseInt(response.getContent());
            if(propVal>bestPropVal)
            {
               bestResponse = response;
               bestPropVal = propVal;
            }
         }
      }

      //====Respond to all proposals====
      e = responses.elements();
      while (e.hasMoreElements())
      {
         ACLMessage response = (ACLMessage) e.nextElement();
         if (response.getPerformative() == ACLMessage.PROPOSE)
         {
            if (bestResponse == response)
            {
               ACLMessage accept = response.createReply();
               accept.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
               accept.setContent(acceptContent);
               acceptances.addElement(accept);
            }
            else
```

```
            {
               ACLMessage reject = response.createReply();
               reject.setPerformative(ACLMessage.REJECT_PROPOSAL);
               reject.setContent(rejectContent);
               acceptances.addElement(reject);
            }
         }
      }
      //increment queue after accept is sent
      sh.currentActionNo++;

      //====If there were no proposals wait and retry====
      if(bestResponse == null){
         try
         {
             Thread.sleep(2*1000);
             System.out.println(sh.getLocalName()+": "+"Retrying CFP");
         }
         catch(InterruptedException
ex){Thread.currentThread().interrupt();}
      }
   }
}
```

Appendix B: Conveyor PLC sample code

```
CASE BYTE_TO_INT(IN:= %MB0) OF
    0:
        SG_0 := TRUE;          // STOP gate main- 3
        TC0_0 := FALSE;        // TC-main-low-4
        TC0_2 := FALSE;        // TC-main-high -5
        TC1_2 := TRUE;         // TC-parallel-high -6
        Motor := FALSE;        // Motor -7
        Motor_Main := FALSE;// main tracks -8

        T0_On := FALSE;
        T3_On := FALSE;
        T5_On := FALSE;

        Status_Byte.0 := FALSE;
        Status_Byte.1 := FALSE;
        Status_Byte.2 := FALSE;
        Status_Byte.3 := FALSE;
        Status_Byte.4 := FALSE;
        Status_Byte.5 := FALSE;
        Status_Byte.6 := FALSE;
        Status_Byte.7 := FALSE;

        Trigger := FALSE;

    1:  // Accept palate
    IF PS_0=TRUE THEN
        T0_On :=TRUE;
        SG_0 := FALSE;
        Motor_Main := TRUE;
    END_IF;

    IF T0_Q=TRUE THEN
        SG_0 := TRUE;
    END_IF;

    IF T1_Q=TRUE THEN
        Motor_Main := FALSE;
        TC0_2 := TRUE;
    END_IF;

    IF T2_Q=TRUE THEN
        Motor := TRUE;
    END_IF;

    IF RPS_0=TRUE THEN
        Motor := FALSE;
        Status_Byte.0 := TRUE;
        T5_On := TRUE;
    END_IF;
```

```
    IF T5_q=TRUE THEN
        Trigger := TRUE;
    END_IF;

    2:  // Release pallate
    IF RPS_0=TRUE THEN
        T3_On := TRUE;
        TC1_2 := FALSE;
        Motor_Main := TRUE;
    END_IF;

    IF T3_Q=TRUE THEN
        Status_Byte.0 := TRUE;
    END_IF;

    IF T4_Q=TRUE THEN
        Trigger := TRUE;
    END_IF;

END_CASE;

// Main conveyer SG prox switch
PS_0 := PS_0_Contact;
// Rocker switch contact
RPS_0 := RPS_0_Contact;

SG_0_Contact := NOT SG_0;
SG_1_Contact := FALSE;//NOT connected
TC0_0_Contact := TC0_0;
TC0_2_Contact := TC0_2;
TC1_2_Contact := TC1_2;
Motor_Contact := Motor;
Motor_Main_Contact := Motor_Main;

%MB1 := Status_Byte;

// timers
TON_0(IN:=(T0_On), PT:=(T#1s), Q=>(T0_Q));
TON_1(IN:=(T0_Q), PT:=(T#1.5s), Q=>(T1_Q));
TON_2(IN:=(T1_Q), PT:=(T#1s), Q=>(T2_Q));
TON_3(IN:=(T3_On), PT:=(T#2s), Q=>(T3_Q));
TON_4(IN:=(T3_Q), PT:=(T#1s), Q=>(T4_Q));
TON_5(IN:=(T5_On), PT:=(T#1s), Q=>(T5_q));

IF Trigger = TRUE THEN
    %MB0 := 0;
END_IF;
```

Appendix C: Machine vision code

The foreground script of the machine vision camera is given below. This code is responsible for interpreting the softsensors to determine the inspection results.

```
class Analysis
{
    public void inspect()
    {
        String output;

        //prepare result to be returned
        output = "";

        //P1
        if(P1_2.Result==PASS){//breaker
            if(P1.Result<FAIL){//empty
                output = output+"1";
            }
            else if(P1.Result==PASS){//filled
                output = output+"2";
            }
        }
        else if(P1_2.Result<FAIL){//no breaker
            output = output+"0";
        }

        //P2
        if(P2_2.Result==PASS){//breaker
            if(P2.Result<FAIL){//empty
                output = output+"1";
            }
            else if(P2.Result==PASS){//filled
                output = output+"2";
            }
        }
        else if(P2_2.Result<FAIL){//no breaker
            output = output+"0";
        }

        //P3
        if(P3_2.Result==PASS){//breaker
            if(P3.Result<FAIL){//empty
                output = output+"1";
            }
            else if(P3.Result==PASS){//filled
                output = output+"2";
            }
        }
```

```
        else if(P3_2.Result<FAIL){//no breaker
            output = output+"0";
        }

        //P4
        if(P4_2.Result==PASS){//breaker
            if(P4.Result<FAIL){//empty
                output = output+"1";
            }
            else if(P4.Result==PASS){//filled
                output = output+"2";
            }
        }
        else if(P4_2.Result<FAIL){//no breaker
            output = output+"0";
        }

        //P5
        if(P5_2.Result==PASS){//breaker
            if(P5.Result<FAIL){//empty
                output = output+"1";
            }
            else if(P5.Result==PASS){//filled
                output = output+"2";
            }
        }
        else if(P5_2.Result<FAIL){//no breaker
            output = output+"0";
        }

        //P6
        if(P6_2.Result==PASS){//breaker
            if(P6.Result<FAIL){//empty
                output = output+"1";
            }
            else if(P6.Result==PASS){//filled
                output = output+"2";
            }
        }
        else if(P6_2.Result<FAIL){//no breaker
            output = output+"0";
        }


        //write result to register
        RegisterWriteString(25, output);

        //indicate inspection completion
        byte b = 1;
        int stat = RegisterWriteByte(110,b);
    }
}
```

Appendix D: Experimental results sample

The tables below shows the results of one of the experiments that were performed with the testbed cell. The three tables represent the order records, session records and operation records respectively. This experiment was conducted with the 2W3S scenario and the IHA was used. Similar data from all the other experiments were used to generate the result summary tables in section 7.4. All the results could not be added here because of the page restrictions of this thesis.

Table C1: Sample experimental results: Orders.

ORDERS			
Or. No.	Start Date	Completion Date	Or. Time
0	Tue Oct 24 13:46:36 CAT 2017	Tue Oct 24 13:56:33 CAT 2017	597,389
1	Tue Oct 24 13:46:55 CAT 2017	Tue Oct 24 14:03:55 CAT 2017	1019,542
2	Tue Oct 24 13:47:14 CAT 2017	Tue Oct 24 14:09:04 CAT 2017	1310,481
3	Tue Oct 24 13:47:57 CAT 2017	Tue Oct 24 14:14:24 CAT 2017	1587,21

Table C2: Sample experimental results: Sessions.

SESSIONS					
Ses. No	W ID	WS ID	Start Date	Completion Date	Ses. Time
0	333333	1	Tue Oct 24 13:46:27 CAT 2017	Tue Oct 24 13:52:55 CAT 2017	388,18
1	222222	3	Tue Oct 24 13:50:01 CAT 2017	Tue Oct 24 13:54:56 CAT 2017	295,19
2	333333	2	Tue Oct 24 13:53:19 CAT 2017	Tue Oct 24 13:54:27 CAT 2017	68,02
3	333333	1	Tue Oct 24 13:54:46 CAT 2017	Tue Oct 24 13:55:44 CAT 2017	58,225
4	222222	2	Tue Oct 24 13:55:07 CAT 2017	Tue Oct 24 13:56:51 CAT 2017	103,919
5	333333	3	Tue Oct 24 13:56:00 CAT 2017	Tue Oct 24 13:56:38 CAT 2017	38,253
6	333333	1	Tue Oct 24 13:56:48 CAT 2017	Tue Oct 24 13:59:54 CAT 2017	185,35
7	222222	3	Tue Oct 24 13:57:01 CAT 2017	Tue Oct 24 13:59:32 CAT 2017	151,246
8	222222	2	Tue Oct 24 13:59:42 CAT 2017	Tue Oct 24 14:00:43 CAT 2017	61,118
9	333333	3	Tue Oct 24 14:00:04 CAT 2017	Tue Oct 24 14:00:28 CAT 2017	24,645
10	333333	1	Tue Oct 24 14:00:34 CAT 2017	Tue Oct 24 14:01:33 CAT 2017	58,286
11	222222	3	Tue Oct 24 14:01:18 CAT 2017	Tue Oct 24 14:09:10 CAT 2017	472,034
12	333333	2	Tue Oct 24 14:01:47 CAT 2017	Tue Oct 24 14:02:38 CAT 2017	50,717
13	333333	1	Tue Oct 24 14:02:50 CAT 2017	Tue Oct 24 14:04:47 CAT 2017	116,818
14	333333	2	Tue Oct 24 14:05:05 CAT 2017	Tue Oct 24 14:06:09 CAT 2017	63,981
15	333333	1	Tue Oct 24 14:06:32 CAT 2017	Tue Oct 24 14:08:30 CAT 2017	117,832
16	333333	2	Tue Oct 24 14:08:46 CAT 2017	Tue Oct 24 14:09:27 CAT 2017	41,007
17	222222	1	Tue Oct 24 14:09:20 CAT 2017	Tue Oct 24 14:10:18 CAT 2017	57,921
18	333333	3	Tue Oct 24 14:09:35 CAT 2017	Tue Oct 24 14:14:25 CAT 2017	289,227
19	222222	2	Tue Oct 24 14:11:08 CAT 2017	Tue Oct 24 14:11:47 CAT 2017	39,869

Table C3: Sample experimental results: Operations.

OPPERATIONS							
Opp. No.	W ID	WS ID	Opp.	Start Date	Completion Date	Opp. Time	Opp. Result
0	333333	1	1	Tue Oct 24 13:46:36 CAT 2017	Tue Oct 24 13:46:52 CAT 2017	16,28	Completed
1	333333	1	2	Tue Oct 24 13:46:54 CAT 2017	Tue Oct 24 13:47:23 CAT 2017	29,02	Completed-Correct
2	333333	1	3	Tue Oct 24 13:47:28 CAT 2017	Tue Oct 24 13:48:16 CAT 2017	48,00	Completed-Correct
3	333333	1	1	Tue Oct 24 13:48:29 CAT 2017	Tue Oct 24 13:48:44 CAT 2017	15,13	Completed
4	333333	1	2	Tue Oct 24 13:48:45 CAT 2017	Tue Oct 24 13:49:20 CAT 2017	34,78	Completed-Correct
5	333333	1	3	Tue Oct 24 13:49:26 CAT 2017	Tue Oct 24 13:50:17 CAT 2017	50,87	Completed-Correct
6	222222	3	5	Tue Oct 24 13:50:03 CAT 2017	Tue Oct 24 13:54:42 CAT 2017	279,01	Completed
7	333333	1	1	Tue Oct 24 13:50:29 CAT 2017	Tue Oct 24 13:50:44 CAT 2017	14,98	Completed
8	333333	1	2	Tue Oct 24 13:50:46 CAT 2017	Tue Oct 24 13:51:17 CAT 2017	30,85	Completed-Correct
9	333333	1	3	Tue Oct 24 13:51:22 CAT 2017	Tue Oct 24 13:52:19 CAT 2017	56,53	Completed-Correct
10	333333	1	1	Tue Oct 24 13:52:31 CAT 2017	Tue Oct 24 13:52:46 CAT 2017	14,17	Completed
11	333333	1	2	Tue Oct 24 13:52:47 CAT 2017	Tue Oct 24 13:52:56 CAT 2017	8,95	Aborted
12	333333	2	4	Tue Oct 24 13:53:20 CAT 2017	Tue Oct 24 13:54:16 CAT 2017	55,51	Completed
13	222222	3	6	Tue Oct 24 13:54:44 CAT 2017	Tue Oct 24 13:54:57 CAT 2017	13,31	Aborted
14	333333	1	2	Tue Oct 24 13:54:48 CAT 2017	Tue Oct 24 13:55:31 CAT 2017	43,19	Completed-Correct
15	222222	2	4	Tue Oct 24 13:55:09 CAT 2017	Tue Oct 24 13:56:42 CAT 2017	93,66	Completed
16	333333	1	3	Tue Oct 24 13:55:37 CAT 2017	Tue Oct 24 13:55:45 CAT 2017	8,15	Aborted
17	333333	3	6	Tue Oct 24 13:56:01 CAT 2017	Tue Oct 24 13:56:31 CAT 2017	30,45	Completed
18	333333	1	3	Tue Oct 24 13:56:50 CAT 2017	Tue Oct 24 13:57:35 CAT 2017	44,21	Completed-Correct
19	222222	3	5	Tue Oct 24 13:57:02 CAT 2017	Tue Oct 24 13:59:26 CAT 2017	144,34	Completed
20	333333	1	1	Tue Oct 24 13:57:48 CAT 2017	Tue Oct 24 13:58:03 CAT 2017	15,05	Completed
21	333333	1	2	Tue Oct 24 13:58:04 CAT 2017	Tue Oct 24 13:58:41 CAT 2017	36,78	Completed-Correct
22	333333	1	3	Tue Oct 24 13:58:47 CAT 2017	Tue Oct 24 13:59:35 CAT 2017	48,65	Completed-Correct
23	222222	3	6	Tue Oct 24 13:59:28 CAT 2017	Tue Oct 24 13:59:32 CAT 2017	4,43	Aborted
24	222222	2	4	Tue Oct 24 13:59:43 CAT 2017	Tue Oct 24 14:00:20 CAT 2017	37,74	Completed
25	333333	1	1	Tue Oct 24 13:59:48 CAT 2017	Tue Oct 24 13:59:54 CAT 2017	5,81	Aborted
26	333333	3	6	Tue Oct 24 14:00:06 CAT 2017	Tue Oct 24 14:00:28 CAT 2017	22,35	Completed
27	333333	1	1	Tue Oct 24 14:00:36 CAT 2017	Tue Oct 24 14:00:48 CAT 2017	12,01	Completed
28	333333	1	2	Tue Oct 24 14:00:49 CAT 2017	Tue Oct 24 14:01:20 CAT 2017	30,23	Completed-Correct
29	222222	3	5	Tue Oct 24 14:01:19 CAT 2017	Tue Oct 24 14:03:39 CAT 2017	140,42	Completed
30	333333	1	3	Tue Oct 24 14:01:25 CAT 2017	Tue Oct 24 14:01:33 CAT 2017	8,07	Aborted
31	333333	2	4	Tue Oct 24 14:01:48 CAT 2017	Tue Oct 24 14:02:31 CAT 2017	42,03	Completed
32	333333	1	3	Tue Oct 24 14:02:51 CAT 2017	Tue Oct 24 14:03:19 CAT 2017	28,44	Completed-Correct
33	333333	1	1	Tue Oct 24 14:03:32 CAT 2017	Tue Oct 24 14:04:13 CAT 2017	40,43	Completed
34	222222	3	6	Tue Oct 24 14:03:41 CAT 2017	Tue Oct 24 14:03:53 CAT 2017	11,90	Completed
35	222222	3	5	Tue Oct 24 14:04:04 CAT 2017	Tue Oct 24 14:06:30 CAT 2017	146,26	Completed
36	333333	1	2	Tue Oct 24 14:04:14 CAT 2017	Tue Oct 24 14:04:36 CAT 2017	21,53	Completed-Correct
37	333333	1	3	Tue Oct 24 14:04:41 CAT 2017	Tue Oct 24 14:04:48 CAT 2017	6,18	Aborted
38	333333	2	4	Tue Oct 24 14:05:07 CAT 2017	Tue Oct 24 14:05:33 CAT 2017	26,19	Completed
39	222222	3	6	Tue Oct 24 14:06:31 CAT 2017	Tue Oct 24 14:07:00 CAT 2017	28,52	Completed
40	333333	1	3	Tue Oct 24 14:06:34 CAT 2017	Tue Oct 24 14:07:03 CAT 2017	29,81	Completed-Correct
41	222222	3	5	Tue Oct 24 14:07:10 CAT 2017	Tue Oct 24 14:08:49 CAT 2017	99,13	Completed
42	333333	1	1	Tue Oct 24 14:07:16 CAT 2017	Tue Oct 24 14:07:51 CAT 2017	34,22	Completed
43	333333	1	2	Tue Oct 24 14:07:52 CAT 2017	Tue Oct 24 14:08:19 CAT 2017	26,83	Completed-Correct
44	333333	1	3	Tue Oct 24 14:08:25 CAT 2017	Tue Oct 24 14:08:30 CAT 2017	5,89	Aborted
45	333333	2	4	Tue Oct 24 14:08:48 CAT 2017	Tue Oct 24 14:09:16 CAT 2017	28,50	Completed
46	222222	3	6	Tue Oct 24 14:08:51 CAT 2017	Tue Oct 24 14:09:02 CAT 2017	11,48	Completed
47	222222	1	3	Tue Oct 24 14:09:22 CAT 2017	Tue Oct 24 14:09:51 CAT 2017	28,86	Completed-Correct
48	333333	3	5	Tue Oct 24 14:09:37 CAT 2017	Tue Oct 24 14:10:48 CAT 2017	70,96	Completed
49	333333	3	6	Tue Oct 24 14:10:49 CAT 2017	Tue Oct 24 14:11:02 CAT 2017	12,84	Completed
50	222222	2	4	Tue Oct 24 14:11:09 CAT 2017	Tue Oct 24 14:11:33 CAT 2017	23,72	Completed
51	333333	3	5	Tue Oct 24 14:11:12 CAT 2017	Tue Oct 24 14:12:44 CAT 2017	91,81	Completed
52	333333	3	6	Tue Oct 24 14:12:45 CAT 2017	Tue Oct 24 14:12:58 CAT 2017	12,30	Completed
53	333333	3	5	Tue Oct 24 14:13:09 CAT 2017	Tue Oct 24 14:14:04 CAT 2017	55,85	Completed
54	333333	3	6	Tue Oct 24 14:14:06 CAT 2017	Tue Oct 24 14:14:22 CAT 2017	16,08	Completed

Appendix E: Pictures of the testbed cell

Figure E.1: The testbed cell.

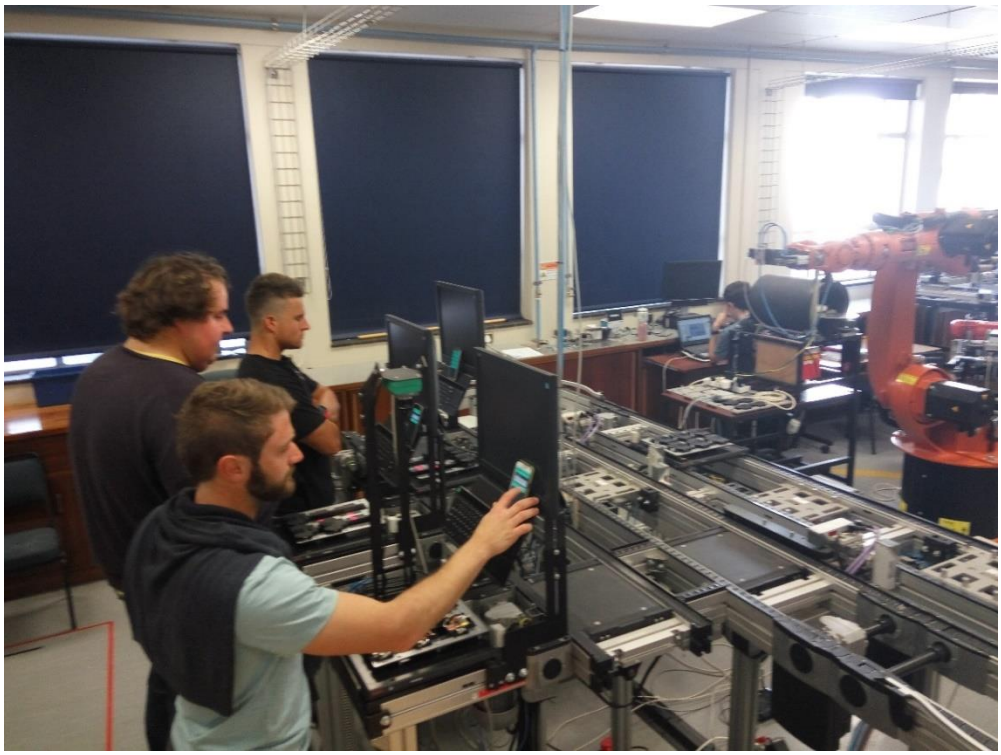


Figure E.2: Test workers at the workstations of the testbed cell.