CONTROL OF THE FEEDER FOR A RECONFIGURABLE ASSEMBLY SYSTEM

by
Karel Kruger

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

Supervisor: Prof. Anton Basson

March 2013

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: 2013/02/25

Abstract

Control of the feeder for a reconfigurable assembly system

K. Kruger
Department of Mechanical and Mechatronic Engineering
Stellenbosch University
Private Bag X1, 7602 Matieland, South Africa
Thesis: MSc.Eng (Mechatronics)
March 2013

This thesis documents the research conducted into the control of the feeder subsystem of a Reconfigurable Assembly System (RAS). The research was motivated by a new set of modern manufacturing requirements associated with an aggressive and dynamic global market. The motivation can be more specifically attributed to the need for selective automation, through the installation of reconfigurable systems, in the South African manufacturing industry.

The objective of the research was to implement and evaluate Multi-Agent Systems (MASs) and IEC 61499 function block systems as potential control strategies for reconfigurable systems. The control strategies were implemented for the control of the feeder subsystem of an experimental RAS at Stellenbosch University. The subsystem's hardware consisted of a singulation unit with a machine vision camera, part magazines and a six DOF pick-'n-place robot.

The structure of the control strategies is based on the ADACOR holonic reference architecture. The mapping of the subsystem holons to the structures of the control strategies is explained. The development and implementation of the control strategies, along with the accompanying lower level software, is described in detail.

A system reconfigurability assessment was performed and the results are discussed. The assessment was performed at two levels – the Higher Level Control (HLC) (where the control strategies were implemented) and the low level control and hardware. The assessment was done through four reconfiguration experiments. The evaluation of the HLC was done through both quantitative and qualitative performance measures. The implications of the reconfiguration, involved in each of the respective experiments, on the low level software and hardware are discussed.

The experimental results show that agent-based control adds more reconfigurability to the feeder subsystem than IEC 61499 function block control, and that agents have more advantages regarding customizability, convertibility and scalability than IEC 61499 function blocks. Also, the ability of agent-based control to implement reconfiguration changes during subsystem operation makes it more suitable to the case study application.

Uittreksel

Beheer van die voerder vir ‘n herkonfigureerbare monteringstelsel

K. Kruger
Departement van Meganiese en Megatroniese Ingenieurswese
Universiteit Stellenbosch
Private Sak X1, 7602 Matieland, Suid-Afrika
Tesis: MSc.Ing (Megatronies)
Maart 2013

Hierdie tesis dokumenteer die navorsing gedoen in die beheer van die voerder sub-stelsel vir ‘n herkonfigureerbare monteringstelsel. Die navorsing was gemotiveer deur ‘n nuwe stel vereistes vir moderne vervaardiging wat met ‘n aggresiewe en dinamiese globale mark geassosieer word. Die motivering kan meer spesifiek toegeskryf word aan die behoefte tot selektiewe outomatisasie, deur middel van die implimentering van herkonfigureerbare stelsels, in the Suid-Afrikaanse vervaardigingsnywerheid.

Die doel van die navorsing is om multi-agent stelsels en IEC 61499 funksie-blok stelsels, as potensiële beheerstrategieë vir herkonfigureerbare stelsels, te implementer en evalueer. Die beheerstrategieë was geïmplementeer vir die voerder sub-stelsel van ‘n eksperimentele herkonfigureerbare monteringstelsel by Universiteit Stellenbosch. Die hardeware behels ‘n skeier-eenheid (*singulation unit*) met ‘n masjienvisie kamera, onderdeelmagasyne en ‘n ses-vryheidsgraad gearktikuleerde optel-en-plaas robot.

Die struktuur van die beheerstrategieë is gebaseer op die ADACOR holoniese verwysingsargitektuur. Die afbeelding van die sub-stelsel holons na die struktuur van die beheerstrategieë word verduidelik. Die ontwikkeling en implementering van die beheerstrategieë, asook die gepaardgaande laer-vlak programmatuur, word in detail beskryf.

Die stelsel se herkonfigureerbaarheid was geassesseer en die resultate daarvan word bespreek. Die assessering was op twee vlakke gedoen – die hoër-vlak beheer (waar die beheerstrategieë geimplementeer was) en die lae-vlak beheer en hardeware. Die assessering was gedoen deur middel van vier herkonfigurasie eksperimente. Die hoër-vlak beheer was geëvalueer deur beide kwalitatiewe en kwantitatiewe metings. Die implikasies van die herkonfigurasie, betrokke by die onderskeie eksperimente, op die lae-vlak beheer en hardeware word beskryf.

Die eksperimentele resultate wys dat agent-baseerde beheer meer herkonfigureerbaarheid tot die voerder sub-stelsel toevoeg as IEC 61499 funksie-blok beheer. Dit is geïdentifiseer dat agente meer voordele inhou ten opsigte van aanpasbaarheid, skakelbaarheid en skaalbaarheid as IEC funksie-blokke. Agent-baseerde beheer laat ook toe dat herkonfigurasieveranderinge tydens sub-stelsel werking geïmplimenteer kan word – dus is dit meer geskik vir aanwending in die gevallestudie.

Aan my familie,

vir al jul liefde, ondersteuning en inspirasie.

"en op die dag sien ek die nag

daar anderkant gaan oop

met 'n bars wat van my beitel af

dwarsdeur die sterre loop."

– N.P. van Wyk Louw

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:

- Prof. Basson, for your willingness to share your vast knowledge and experience with me. Your continual guidance has been invaluable and your passion for research has truly been contagious.

- My fellow members of the reconfigurable automation research group. Anro le Roux and Chibaye Mulubika, for your opinions, advice and enthusiasm. Reynaldo Rodriguez, for aiding me with your technical expertise.

- Mr. Ferdi Zietsman and the workshop staff, for all your patience and hard work.

- All of my friends, for all the support and inspiration you provided me. Even in the hardest of times, I never felt alone.

- My family, for always believing in me – even when I myself am doubtful. Your love and support continues to carry me through every day. I cannot express my gratitude for all that you have given me.

Above all, I thank our heavenly Father – without whom nothing would be possible.

Table of contents

List of tables

List of figures

List of abbreviations

ACL	-	Agent Communication Language
CC	-	Cell Controller
DAQ	-	Data Acquisition
FBDK	-	Function Block Development Kit
HLC	-	Higher Level Control
JADE	-	Java Agent Development framework
LLC	-	Lower Level Control
MAS	-	Multi-Agent System
PC	-	Personal Computer
PLC	-	Programmable Logic Controller
RAS	-	Reconfigurable Assembly Systems
RMS	-	Reconfigurable Manufacturing Systems
SU	-	Singulation Unit
XML	-	eXtensible Markup Language

1. Introduction

1.1 Background

The modern assembly and manufacturing environment is characterized by dynamic change and aggressive global competition. This dynamic environment is subject to rapid change in economical, technological and customer trends (Leitao and Restivo, 2006). A new set of requirements is thus applied to the modern manufacturing paradigm. Bi *et al.* (2008) describe some critical requirements for modern manufacturing systems:

- Short lead times for the introduction of new products into the system. This involves the rapid adjustment of existing functions and processes, as well as the integration of new ones.
- The ability to produce more product variants. This involves the addition of versatility and customization to production to satisfy customer demands.
- The ability to handle low and fluctuating production volumes in order to be competitive in unpredictable markets.
- Low product prices to compete in global markets.

The manufacturing and assembly environment in South Africa (SA) is no different to that described above. However, some additional challenges exist for South African companies. The first of which is the dependency on manual labour. The cost of manual labour in SA is higher than that of other global competitors (World Minimum Wages, [S.a.]). This additional cost, as well as the unpredictability of a manual workforce (strikes, occupational safety risks, etc.), is making it difficult for SA to be competitive in the global market. The second challenge deals with the automation of processes in SA industries. There are many small to medium sized factories in SA producing a large variety of products. This variety in production means that automation cannot be achieved by Dedicated Manufacturing Systems (DMSs), as is described in section 2.1. The expected revenue of these companies does not allow them to automate their processes by Flexible Manufacturing Systems (FMSs) (described in section 2.1).

The economic constraints faced by factories in SA limit the extent to which automation can be introduced to production activities. It is then only possible to automate certain production processes – an approach referred to as selective automation. The selection of which processes 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. This value can be measured in different ways, e.g. a decrease in production cost, an increase in throughput or an elimination of safety risks. This selective automation, incorporating the implementation of reconfigurable systems, can potentially solve some of the problems involved in local production environments.

The concept of reconfigurable manufacturing and assembly systems is a promising solution to the modern challenges. The selective implementation of such systems can solve the problems faced by SA companies. This implementation will decrease production costs and increase production reliability and product quality.

The research presented in this thesis forms part of a collective research effort into reconfigurable systems at Stellenbosch University. The research builds on previous studies which focussed on the conceptualization, design and control of an experimental Reconfigurable Assembly System (RAS). The RAS is based on the requirements of many factories in SA, especially those of CBI Electric – a global supplier of a high variety of quality trip switches. The products and processes of CBI Electric were used as case study for the experimental RAS.

Sequeira (2008) identified the spot-welding process, involved in the production of CBI Electric, as a suitable process for automation by means of a reconfigurable system. The process entails the welding of individual trip switch parts to create a variety of sub-assemblies. It was identified that automating this process would reduce the dependence on skilled manual labour and the necessary training programmes. The conceptual design of the RAS included subsystems for the following functions: storage, transport, feeding, welding and inspection and removal. At this stage all the subsystems, except the inspection and removal subsystem, have been developed.

Recent research at Stellenbosch University has placed emphasis on the control and coordination of the subsystems of the RAS. Parts of the presented research can be viewed as an advancement of the research performed by Sequeira (2008) and Adams (2010). The presented thesis places emphasis on the implementation and evaluation of proposed strategies for control of RASs. The feeder subsystem of an experimental RAS at Stellenbosch University was used as case study for the control implementation. This research was done in parallel with two other studies - Le Roux (2013) evaluated and implemented the control for the transport and storage subsystem and Mulubika (2013) designed and controlled the welding subsystem and developed a Cell Controller for the RAS.

1.2 Motivation

The feeder subsystem of the RAS had to incorporate mechanisms for part feeding, part manipulating and part fixturing. The feeding of parts involves the need for singulation actions – individual parts have to be singulated from bulk containers. This is followed by moving and manipulating the parts by a pick-‘n-place robot, and then placing the parts in a fixture, which holds them in fixed positions for the welding process. Conventional systems for the feeding of parts are specifically designed for a specific set of parts - the variety of parts involved in the production of CBI Electric requires the feeder subsystem to be reconfigurable in the mentioned actions. The feeder subsystem then has to be a reconfigurable system itself.

The research presented in this thesis focuses on the control of the feeder subsystem of the experimental RAS. The thesis evaluates suitable control strategies for implementation in the feeders of RASs. The thesis aims to give an indication regarding the best means of control for reconfigurable feeders, thus contributing towards the implementation of RASs in industry.

1.3 Objective

The objective of this research was to evaluate the IEC 61499 standard function block and agent-based control technologies as possible methods for implementing holonic control in feeder subsystems of Reconfigurable Assembly Systems (RASs).

The control strategies were implemented on a feeder subsystem of an automated welding RAS. The evaluation of the control strategies were based on the results of different experiments. These experiments provided performance measurements of the two control strategies according to the characteristics of RASs (described in section 2.2).

2. Literature review

This section starts with a discussion of classic manufacturing paradigms and conventional control strategies of manufacturing systems. The motivation for reconfigurable manufacturing systems, along with its inherent concepts and characteristics, is discussed. The holonic approach to system control, which is often associated with reconfigurable systems, is described, with specific reference to the existing architectures for holonic control. The concepts of agent-based and IEC 61499 function block control, as strategies for implementing holonic control, are discussed in depth.

2.1 Classic manufacturing paradigms

The manufacturing and assembly environment is evolving continuously. This evolution is driven by changes in technology and economic trends. The major paradigms in manufacturing and assembly, as presented by Mehrabi *et al.* (2000), are discussed in the following paragraphs.

The Machining System paradigm entailed the installation of one or more metal removing machine tools. These machine tools were accompanied by auxiliary equipment for material handling, control and communications. The operation of the machines was then coordinated to produce a fixed amount of parts. This paradigm pursued *mass production* as a strategy to reduce product cost.

The need for higher part quality and reduction in production costs brought about the Dedicated Machining System (DMS) paradigm. With DMSs, machining systems with fixed tooling and functions were designed for specific parts. The DMS paradigm was driven by the *lean production* ideology, where production costs were reduced by eliminating production waste.

A market demand for increased product variety led to the Flexible Manufacturing System (FMS) paradigm. FMSs were based on automation configurations of fixed hardware with programmable software. Flexibility refers to the ability of the system to switch to new families of components by changing the manufacturing or assembly processes and functions (Martinsen *et al.*, 2007). These systems were thus capable of handling changes in work orders and production schedules, and producing several types of parts with short changeover times. ElMaraghy (2006) identified several types of flexibility:

- *Machine flexibility* – the execution of various operations without changing the machine set-up.
- *Material handling flexibility* – the existence of various paths for the transfer of materials between machines.
- *Operation flexibility* – the availability of different operation plans for part processing.
- *Process flexibility* – the ability to produce different sets of part types without major set-up changes.
- *Product flexibility* – the agility to handle the introduction of new products.

- *Routing flexibility* – the existence of several feasible routes for the various product types.
- *Volume flexibility* – the ability to vary production volumes profitably within the current system capacity.
- *Expansion flexibility* – the ease in which system capability and capacity can be added to the system through physical changes.
- *Control program flexibility* – the ability of the control system to run virtually uninterrupted during production or system changes.
- *Production flexibility* – the ability to produce a number of product types without adding major capital equipment.

There have been several investigations into the shortcomings of FMSs with regard to implementation in industry. Raj *et al.* (2007) identified high costs, the difficulty of design and the lack of inherent product flexibility (relative to volume flexibility) in FMSs as barriers to industrial implementation. Mehrabi *et al.* (2002) adds to this list a lack of software reliability, the need for highly skilled personnel, high support costs and a lack of support from machine tool manufacturers. They also mention that FMSs tend to be designed with excess features and capacity, which remain unutilized in many cases.

2.2 Reconfigurable manufacturing systems

The concept of reconfigurable manufacturing systems (RMSs) is a solution to the requirements of modern systems discussed in section 1.1. RASs are the specific application of RMSs in assembly processes.

It is important to discuss the exact meaning of reconfigurability in this context. Martinsen *et al.* (2007) describes reconfigurability as the ability of a manufacturing or assembly system to switch, with minimal delay and effort, between a particular family of parts by adding or removing functional elements. These functional elements can form part of the system hardware or software (Vyatkin, 2007).

RMSs and FMSs are often confused because of their similarity – each system can be adapted and is capable of handling production variety. It is important to consider the differences between the abilities of RMSs and FMSs. Mehrabi *et al.* (2000) mention that the key difference between RMSs and FMSs is that the capacity and functionality of RMSs are not fixed – RMSs are designed for rapid adjustment, through rearrangement or change of their components, in response to production demands. Wiendahl (2007) identified two more differences:

1. The diversity of the workpieces that can be handled by the system. RMSs can be switched to accommodate different families of products, while FMS can only handle similar products.
2. The extent to which the system is changed. With RMSs, the changes can be made through the addition or removal of components. FMSs are designed to only allow for changes in the production processes and the flow of material.

Mehrabi *et al.* (2000) identified five key characteristics of RASs. A sixth characteristic was identified by ElMaraghy (2006). The characteristics are then as follows:

1. **Modularity** of the hardware and software system components.
2. **Integratability** of the system and the system components for both ready integration of existing technology and the introduction of new technology in the future.
3. **Convertibili**ty for the fast changeover between existing products and fast adaptability of the system for future products.
4. **Diagnosibility** for fast identification of the sources of quality and reliability errors in the system.
5. **Customization** of the system capability and flexibility to match specific applications.
6. **Scalability** of the system capacity.

RMSs satisfy all the requirements of modern assembly mentioned in section 1.1. Mehrabi *et al.* (2000) explain that RMSs permit reduction in lead times and quick integration of new technology and/or functionality. Bi *et al.* (2008) recognised that RMSs have the ability to reconfigure hardware and control resources, at all functional levels, to rapidly adjust the production capacity and functionality in response to sudden changes. Bi *et al.* (2007) is in agreement with this statement, identifying that with RMSs “the system and its components have adjustable structure that enables system scalability in response to market demands and system adaptability to new products”.

Rooker *et al.* (2007) explain that there are two different types of reconfiguration which can occur in RMSs: basic and dynamic reconfiguration. Basic reconfiguration requires the system to be stopped. The system is then restarted after the necessary software and hardware changes have been implemented. With dynamic reconfiguration, the changes can be made while the system is still in operation.

There exist several issues which have hampered the development and implementation of RMSs. Bi *et al.* (2007a) explain the key issues regarding RMS development:

- The separation of RMS design from product design. Most RMSs are developed separate from the product design, which complicates the optimization of the system.
- RMSs are perceived as a premature technology. Developers are still dealing with unresolved issues, which prohibit full automation through RMSs.
- Indifferent attitudes toward RMSs. Many companies are uncertain of the advantages that reconfigurable automation holds for their production.
- The use of RMSs as a wrong solution. RMSs should be implemented in production scenarios where the necessary production requirements exist

and a sufficient level of technical competence is available. The RMS concept is not a suitable solution for all production scenarios.

2.3 Control of manufacturing systems

This section describes some of the commonly used classifications and approaches for the control of manufacturing systems.

2.3.1 Types of control architectures

Three different types of control architectures are discussed by Meng *et al.* (2006): centralized, hierarchical and heterarchical. The organizational structures of the control architectures are depicted in Figure 1. The architectures are described in the following paragraphs.

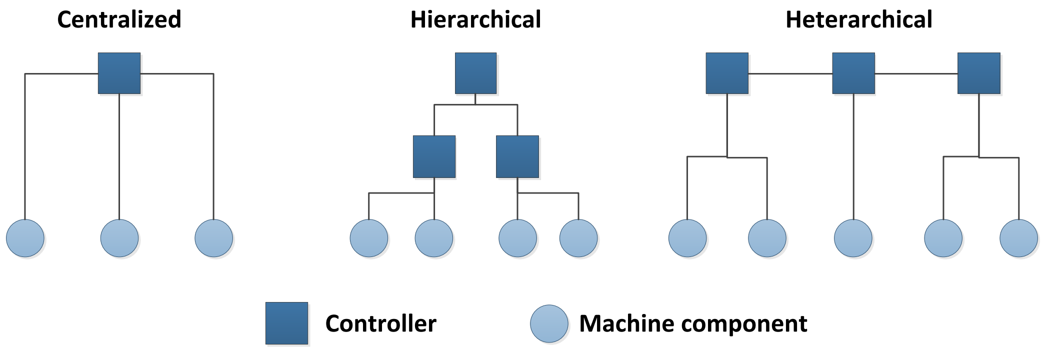


Figure 1: Types of control architectures (adapted from Meng *et al.* (2006)).

The centralized control architecture achieves system control by means of one central controller. This controller is then responsible for the execution of all the automated processes in the system. The architecture is typically implemented in conventional control systems (discussed in section 2.3.2).

The hierarchical control architecture implements the hierarchical arrangement of multiple controllers in a system. Different levels of control exist within the system. This implementation sees the passing of instructions in a downward direction and feedback in an upward direction. The hierarchical architecture is typically implemented in conventional control systems, while mixed architectures (combinations of hierarchical and heterarchical architectures) are often implemented in distributed control systems like holonic control (discussed in section 2.3.3).

Heterarchical control architectures apply no hierarchical levels of control. The control of the system is achieved by several independent controllers. These controllers each have their own goals and specific functionality. Communication and coordination between these independent controllers enable complex system functionalities and the pursuing of the system goals. Mixed or strict heterarchical control architectures are typically implemented in holonic control systems.

2.3.2 Conventional control

The control of manufacturing systems is conventionally done through centralized control systems or Petri nets, for the control of distributed processes.

2.3.2.1 Centralized control

Conventional manufacturing control systems are typically large, centralized applications which are developed and adapted on a case-by-case basis (Leitao and Restivo, 2008). These control systems implement centralized or strict hierarchical architectures (as was described in section 2.3.1). These control systems exist within the concept of Computer Integrated Manufacturing (CIM), which utilises large central databases to support the system information (Scholz-Reiter and Freitag, 2007). Conventional control hardware and programming techniques greatly rely on Programmable Logic Controllers (PLCs) (Black and Vyatkin, 2009).

Leitao and Restivo (2008) explain that conventional control systems do not efficiently satisfy the requirements of modern manufacturing and assembly (such as those specified in section 1.1). These control systems require expensive and time-consuming efforts to implement, maintain or reconfigure the control application. Scholz-Reiter and Freitag (2007) noticed that “the complexity of the control system grows rapidly with the size of the underlying manufacturing system”. Meng *et al.* (2006) explains that conventional control is not reconfigurable-friendly due to shortcomings such as structural rigidity, lack of flexibility and convertibility and inability to tolerate faults or disturbances. The monolithic nature of general PLC software increases the difficulty of system modification and maintenance, and reduces the scalability of the system. This centralized approach also cannot be appropriately applied to applications of wide physical dispersion of hardware (Black and Vyatkin, 2009).

2.3.2.2 Petri nets

Petri nets are a graphical and mathematical tool for describing system processes. This approach is very advantageous when the processes are distributed, asynchronous and/or nondeterministic (Murata, 1989). Since being introduced in the late 1970s, Petri nets have seen numerous implementations in all types of manufacturing systems.

Murata (1989) explains that Petri nets are a particular kind of directed graph, which consists of two types of nodes: *places* and *transitions*. These nodes relate to that of *events* and *conditions* used in system modeling. *Arcs* are used to connect *places* to *transitions* or vice versa. A *transition* (an event) has a certain number of input and output *places* – these *places* represent the pre- and post-conditions for the event. The state of the conditions is represented in Petri nets as a *token* which is assigned to a *place*. This assignment is then representative of a “true” condition for the *place*. The firing of system *transitions* can then be controlled by implementing certain rules concerning the presence of *tokens* in the relative input and output *places*.

The popularity of Petri net implementation in manufacturing systems is based on the ease of which it can be converted into computer control mechanisms (Zhou *et al.*, 1992). Petri nets can "concisely represent the activities, resources and constraints of a system in a single coherent formulation" (Lee and DiCesare, 1994). The graphical representation inherent in the Petri net approach also aids the understanding and formulating of system problems.

2.3.3 Holonic control

The term holon was first introduced by Koestler in 1967 (Paolucci and Sacile, 2005). The term comes from the Greek words "holos" (meaning "the whole") and "on" (meaning "the particle"). Holons are then "any component of a complex system that, even when contributing to the function of the system as a whole, demonstrates autonomous, stable and self-contained behaviour or function" (Paolucci and Sacile, 2005). When this concept is applied to manufacturing or assembly systems, a holon is an autonomous and cooperative building block for transforming, transporting, storing or validating information of physical objects. A Holonic Manufacturing System (HMS) is then "a holarchy (a system of holons which can cooperate to achieve a goal or objective) which integrates the entire range of manufacturing activities" (Paolucci and Sacile, 2005).

The distributed holonic model represents an alternative to the traditional centralization of functions (Paolucci and Sacile, 2005). Holonic control usually combines the best features from both hierarchical and heterarchical control architectures (Kotak *et al.*, 2003). Kotak *et al.* (2003) explain that individual holons have at least two basic parts: a functional component and a communication and cooperation component. The functional component can be represented purely by a software entity or it could be a hardware interface represented by a software entity. The communication and cooperation component of a holon is implemented by software.

The implementation of holonic control in assembly systems holds many advantages. Holonic systems are attractive because they are resilient to disturbance and adaptable in response to faults (Black and Vyatkin, 2009). Holonic systems have the ability to organise production activities in a way that they meet the requirements of scalability, being robust and being fault-tolerant (Kotak *et al.*, 2003). Scholz-Reiter and Freitag (2007) describe advantages of holonic control systems due to the incorporation of heterarchical control. These advantages are:

- Reduced system complexity due to the localization of information and control.
- Reduced software development costs by the elimination of supervisory control levels.
- Higher maintainability and modifiability due to system self-configurability abilities and system modularity.
- Improved reliability due to a fault-tolerant approach as opposed to a fault-free approach.

The two reference architectures for holonic control that are most often encountered in the literature are PROSA and ADACOR. These two architectures are discussed in the remainder of the section.

The first proposed holonic control architecture is **PROSA** (**P**roduct-**R**esource-**O**rder-**S**taff **A**rchitecture), which is comprehensively described by van Brussel *et al.* (1998). PROSA defines four classes of holons: product, resource, order and staff.

The first three classes of holons can be classified as basic holons. This is because their existence is based on that of three independent manufacturing concerns:

i. Product related technological aspects, such as the management of process sequence and the product life cycle. Product holons hold the product and process information required for the production of system products. These holons contain the various "product models" and can provide the other holons in the system with product information.
ii. Resource aspects, such as optimizing the performance of machines and the maximizing of machine capacity. Resource holons contain the physical hardware, accompanied by the control software, for production line components. These holons then offer their functionality and capacity to the other holons in the system.
iii. Logistical aspects, such as those concerning customer demands and production deadlines. The order holons can be represented as tasks within the manufacturing system. These holons manage the logistical information related to the product being produced. Order holons contain the "product state model" and can thus provide production information to the other holons in the system.

The basic holons can interact with each other by means of knowledge exchange, as is shown in Figure 2. The process knowledge, which is exchanged between the product and resource holons, is the information and methods describing how a certain process can be achieved through a certain resource. The production knowledge is the information concerning the production of a certain product by using certain resources – this knowledge is exchanged between the order and product holons. The order and resource holons exchange process execution knowledge, which is the information regarding the progress of executing processes on resources.

Staff holons are considered to be special holons. This is because staff holons are added to the holarchy to operate in an advisory role to basic holons. The addition of staff holons aim to reduce work load and decision complexity for basic holons, by providing them with expert knowledge. The staff holons consider some aspects of the problems faced by the basic holons, and provide sufficient information such that the correct decision can be made to solve the problem.

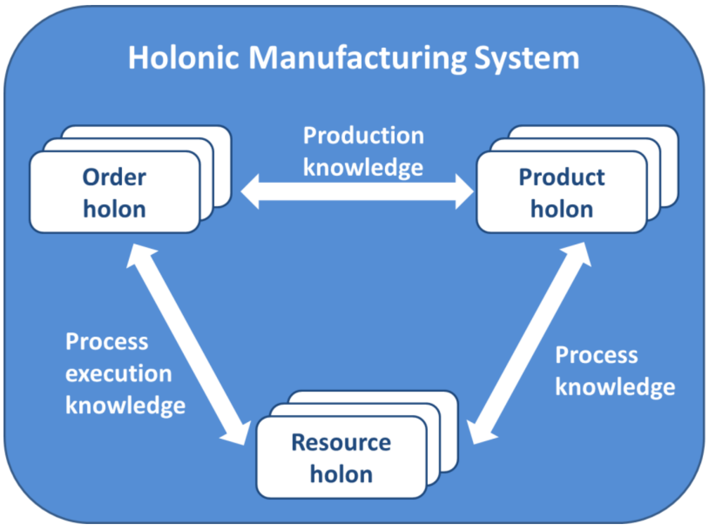


Figure 2: Structure of PROSA architecture (adapted from van Brussel *et al.* (1998)).

The holonic characteristics of PROSA contribute to the different aspects of reconfigurability. The ability to decouple the control algorithm from the system structure and the logistical aspects from the technical aspects adds integratability and modularity. Modularity is also added by the similarity that is shared by holons of the same type, since this allows holons to be interchanged easily.

Another proposed control architecture for holonic systems is that of **ADACOR** (**ADA**ptive holonic **CO**ntrol a**R**chitecture for distributed manufacturing systems). Within ADACOR, each holon represents a physical resource or logic entity. ADACOR defines four holon classes according to their roles and functionalities: product holons (PH), task holons (TH), operational holons (OH) and supervisor holons (SH). The structure of the ADACOR architecture is shown in Figure 3.

The product, task and operational holons are similar to the product, order and resource holons of the PROSA architecture. The product holons represent the products available for production – these holons have access to all the relevant product information. The task holons represent the processes, along with the necessary resources, required to satisfy the production orders. The operational holons represent the physical shop floor resources. The supervisor holon is quite different to the staff holon. Supervisor holons are capable coordinating groups of holons and optimizing their collective actions. The supervisor holons thus introduce some hierarchy into the decentralized system.

The ADACOR holons comprise a Logical Control Device (LCD) and a physical resource (if it exists for the specific holon). The LCD has three functional components: a communication component for inter-holon communication, a decision component for regulating holon behaviour and an interface component for integrating with the physical resources.

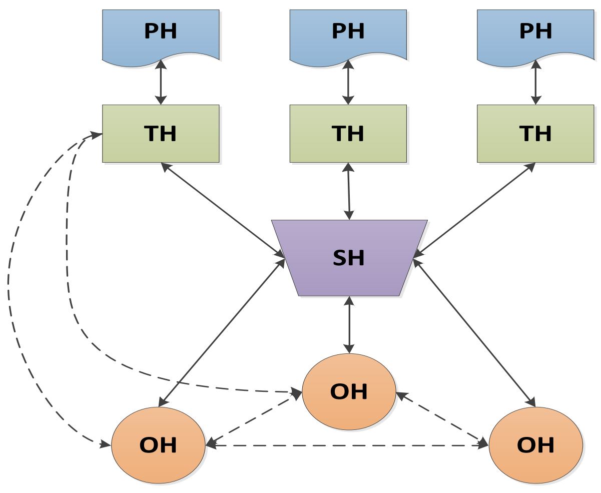


Figure 3: Structure of ADACOR architecture (adapted from Leitao and Restivo (2006)).

According to Leitao and Restivo (2008), ADACOR addresses the improvement of flexibility and response to change of manufacturing control systems operating in volatile environments. ADACOR is suited to dealing with control problems in a distributed manner by being "as centralized as possible and as decentralized as necessary". An ADACOR control system can be formally specified and modelled using Petri nets. ADACOR is "built upon a community of autonomous and cooperative entities, designated by holons, to support the distribution of skills and knowledge, and to improve the capability of adaption to changing environments".

Two possible strategies for implementing holonic control are agent-based control and IEC 61499 function block control, discussed in sections 2.4 and 2.5.

2.4 Agent-based control

The use of agent-based software to control manufacturing systems, i.e. agent-based control, has received much attention in the research community – particularly in combination with holonic and reconfigurable systems.

2.4.1 Definition of agents and agent systems

An agent can be defined as a computational system with goals, sensors and effectors, which can autonomously decide which actions to take, in a given situation, to maximize its progress towards its goals (Paolucci and Sacile, 2005). With reference to a multi-agent system, Xie *et al.* (2007) define an agent as "a software system that communicates and cooperates with other software systems to solve a complex problem beyond their individual capabilities".

Paolucci and Sacile (2005) explain that an agent is different to a holon in the sense that a holon can consist of other holons, while an agent cannot include other agents. With this said, agents can practically be equivalent to holons in some cases. This is usually the case with agents which directly control a physical device. Here the agent then represents the software component of the holon introduced to decentralize the control system at the lowest level.

According to Paolucci and Sacile (2005) three different classes of agents can be identified:

- Agents that execute tasks based on predetermined rules and assumptions.
- Agents that execute well-defined tasks at the request of a user.
- Agents that volunteer information or services to a user whenever it is deemed appropriate.

The main characteristics of these agents are then as follows:

- **Autonomy**. Agents should be able to perform most of their tasks without user intervention.
- **Social ability**. Agents should be able to interact with other agents and users.
- **Responsiveness**. Agents should be able to respond to changes in their environment.
- **Proactiveness**. Agents should exhibit opportunistic and goal-orientated behaviour.
- **Adaptability**. Agents should be able to modify their behaviour in response to changes in their environment.
- **Mobility**. Agents should possess the ability to change physical location to improve their performance.
- **Veracity**. Agents should communicate reliable information.
- **Rationality**. Agents should act in a manner as to achieve their goals.

Agents of different classes, performing different roles and functions, can cooperate and communicate within a Multi-Agent System (MAS) to achieve their individual goals and the goals of the system. MASs can be understood as societies of autonomous entities that, by their own convenient interaction and coordination, attempt to achieve local and global goals. MASs can then be summarized as "flexible networks of problem solvers that tackle problems that cannot be solved using the capabilities and knowledge of the individual solver" (Paolucci and Sacile, 2005).

2.4.2 Design methodologies for MASs

Paolucci and Sacile (2005) discuss three design methodologies for the design of MASs: problem-oriented, architecture-oriented and process-oriented MAS design.

The problem-oriented MAS design process is guided by the identification of the reasons for which the MAS is needed. This usually involves obtaining an MAS solution to an existing problem or enhancing certain aspects of a system. The types of problems are then identified and transformed into tasks, which can be performed by agents. Two promising approaches to problem-oriented MAS design are the GAIA approach and the Multi-agent Systems Engineering (MaSE) approach.

The architecture-oriented MAS design process is oriented by the requirements and implications of the design on the system architecture. The architecture determines the capabilities of the agent system. The Synthetic-Ecosystems approach is proposed for architecture-oriented MAS design.

Process-oriented MAS design is guided by the definition of time constraints imposed by the different processes in the manufacturing system. The real-time behaviour is an important aspect of MASs as they have to deal with internal and external asynchronous signals, along with the necessary time constraints. A proposed approach to process-oriented MAS design involves a four-layer, real-time holonic control architecture.

2.4.3 Standards and platforms for MASs

The establishment of methodologies and techniques for MAS design and operation are required to increase the amount of practical applications of MASs in industry. “The Foundation for Intelligent Physical Agents (FIPA) is an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies” (FIPA, 2010). FIPA was founded in 1996 and became an official IEEE standards organization in 2005. FIPA has thus begun to establish standards for the development and communication of agent-based systems. The most significant of the FIPA standards is the agent communication standard (FIPA, 2010). Paolucci and Sacile (2005) explain that the standard formalizes the conversations between agents with two concepts: the communicative act and the Agent Interaction Protocol (AIP). The communicative act associates a predefined semantic to the content of messages to allow messages to be univocally understood by all agents. The AIP defines which communicative acts must be used in a conversation and also the sequence of messages to allow meaningful communication between agents. Other FIPA standards deal with issues surrounding the specification of the agent communication language and the mandatory components for agent platform architectures.

The FIPA standards mainly focus on specifications regarding agent interoperability. FIPA thus only describes an abstract architecture with little detail regarding aspects of the implementation platforms (Paolucci and Sacile, 2005). Despite the lack of detailed standards, several agent implementation platforms have been developed. The most widely used platforms are FIPA-OS, JADE and ZEUS. JADE (Java Agent DEvelopment framework) was developed by Telecom Italia Lab, in collaboration with the University of Parma, Italy. JADE was fully developed in Java language and runs in the Java run-time environment. JADE is also fully FIPA compliant.

Several platforms have also been developed for the simulation of MASs, of which the most renowned are Swarm, RePAST and MAST. The Swarm project was started to create a standard support tool for the management of swarms of objects – a concept necessary for handling MASs. Swarm is based on an object-oriented framework for the definition of agent behaviour and interaction. RePAST (Recursive Porous Agent Simulation Toolkit) was initially viewed as a set of libraries intended to simplify the use of Swarm, but was later redesigned as a completely new framework. RePAST provides a library of classes to create, perform, view and collect data from agent simulations (Paolucci and Sacile, 2005). Research by Vrba (2003) brought about a simulation tool for agent-based systems in the form of MAST (Manufacturing Agent Simulation Tool). MAST is entirely devoted to the simulation of manufacturing processes. It has been implemented to simulate the material-handling activities of a manufacturing system. MAST is also based on the JADE platform and is also fully FIPA compliant.

2.4.4 Agent communication

The cooperation of agents in an MAS is dependent on effective agent communication. The agent platform must thus provide structures to ensure that agents can communicate easily and reliably. The Agent Content Language (ACL) is one such structure specified by FIPA.

Agent communication is based on ACL messaging. The ACL encapsulates and describes the message content by setting several message parameters. Paolucci and Sacile (2005) list the following parameters:

- *Performative* – description of the communicative action involved in the message.
- *Sender* and *Receiver* – the identification of the respective communicating agents.
- *Language* – the specific encoding of the message content.
- *Ontology* – the vocabulary to be used to understand the message.
- *Protocol* – the set of rules on which the communication is based.

MASs often employ ontologies to ensure that communicating agents fully understand the content of messages. An ontology is a vocabulary used to describe the terms and relationships entities in a specific domain (Paolucci and Sacile, 2005). This description can be viewed as an explicit specification of conceptualizations. Ontologies provide a useful means to facilitate the access and re-use of knowledge – especially in multi-actor environments (Gruber, 1991). The use of an ontology allows agents to have a shared understanding of certain concepts inherent in the MAS, and specifies which type of manipulation and reasoning can be performed on them (Paolucci and Sacile, 2005).

Nikraz *et al*. (2006) explain that the interaction between agents, sharing a common ontology, depends on three interpretations: *Concepts*, *Predicates* and *Actions*. Concepts are structured templates for the exchange of complex information regarding entities in the agent environment. These templates then have slots to specify the necessary information needed for the interaction. The example of an address as a Concept, with the required slots, can be shown as follows:

Address:

- *City (String)*
- *Street (String)*
- *Number (Int)*

Entities within an environment are typically connected by means of relations. These relations can then also be complex structures which are defined by templates. These templates, which specify the relations between entities, are called Predicates. The Predicates contain slots to specify the entities that are related. An example of a Predicate, as implemented in the scheduling of an appointment, is as follows:

IsScheduled:

- *What (Meeting)*
- *Where (Address)*
- *When (Scheduled Time)*

Lastly, the actions that agents can perform must be represented by complex descriptions. These descriptions are contained in structured templates called Actions. As with the other templates, Actions also contain slots for specifying the information involved in performing the action. The Action template is shown below, where an agent must contact the attendee of a meeting:

ContactAttendee:

- *Number (Int)*
- *Email (String)*

2.4.5 Advantages of MASs

MASs hold several advantages for implementation in RASs. MASs have high modularity and reconfigurability. The addition or modification of resources can be achieved by simply inserting a new agent into the system or modifying the behaviour of an existing agent (Paolucci and Sacile, 2005). Vrba *et al.* (2009) recognised that due to its modular and decentralized characteristics, MASs are a way to reduce complexity and increase flexibility in a system. MASs can allow the simultaneous production of different products and improve the integration of legacy equipment (Candido and Barata, 2007). Xie *et al.* (2007) also recognised that MASs can respond quickly to dynamic changes in the manufacturing or assembly environment. Furthermore, agent-based technologies are capable of dealing with autonomy, distribution, scalability and disturbance (Bi *et al.*, 2008). The distributed and redundant nature of agent-based control systems minimizes the effect of local failure on the overall functionality of the system (Vrba and Marik, 2009). This is also confirmed by simulations performed by Lepuschitz *et al.* (2009), showing that agent-based control is "extremely robust against disturbances of machines, as well as failure of control units".

2.4.6 Implementations of MASs

There have been several practical implementations of agent-based control. The ADACOR architecture (described in section 2.3.3) was implemented on a test production system, using multi-agent technology, by Leitao and Restivo (2008). The production system consisted of a manufacturing cell, an assembly cell, a storage and transportation cell and a maintenance and setup cell. The control system was then integrated with PLCs and PCs (running different software platforms), various robots and vision sensors and an Automatic Guided Vehicle (AGV). Candido and Barata (2007) implemented a multi-agent control system for the NovaFlex shop floor assembly case study. The NovaFlex system is composed of two assembly robots, an automatic warehouse and a transport system connecting all the modules. DaimlerChrysler's Production 2000+ project implemented an agent-based control system for a flexible cylinder head production system. This production system is composed of modules, each consisting of a CNC machine, three conveyors, two switches and a shifting table (Marik *et al.*, 2010). Marik *et al.* (2010) also reported an agent-based control solution which added flexibility to a steel rod bar mill for BHP Billiton. A multi-agent control system was also implemented in the holonic packing cell of the Centre for Distributed Automation and Control (CDAC) at the University of Cambridge.

Even though there have been several test cases and some industrial implementations, the manufacturing and assembly industry is still hesitant to apply agent-based technologies. Candido and Barata (2007) give four reasons for this hesitation and a fifth is mentioned by Marik *et al.* (2010):

- A paradigm misunderstanding still exists due to a lack of practical test cases.
- Members of the industry are still unaware about the changes in modern manufacturing and assembly requirements.
- There is a lack of experience in agent-based technology by actual system integrators.
- There is a pioneering risk involved in investing in an unproven technology.
- The unpredictability of emergent behaviour in agent-based systems complicates the quantitative comparison to other technologies.

2.5 IEC 61499 Function Block control

The IEC 61499 standard specifies a framework for distributed and embedded control using function blocks. The ability to control distributed systems, makes this approach a candidate for use in reconfigurable systems.

2.5.1 The IEC 61499 standard

The IEC 61499 standard is a successor to the IEC 1131 standard, which later became IEC 61131. The IEC 1131 standard is aimed at control applications in PLCs. The standard provides specifications ranging from PLC programming to the fieldbus communication of applications in PLCs. The standard is divided into several parts dealing with the various aspects concerning PLCs. The IEC 61131-3 part of the standard deals with the programming of PLCs. According to Lewis (1998), this part of the standard aims to improve the following aspects of PLC programming:

- **Capability** of a system to perform its intended design functions.
- **Availability** of a system during its life cycle when it is available for its intended design functions.
- **Usability**, which indicates the ease with which a specified set of users can acquire and exercise the ability to interact with the system in order to perform its intended design functions.
- **Adaptability**, which refers to the ease with which a system may be changed in various ways from its initial intended design functions.

The IEC 61131 standard has had implied limitations when dealing with complex computations, knowledge processing, advanced network messaging and service orientation (Vrba and Marik, 2009). The IEC 61499 standard addresses these limitations and extends the IEC 61131 standard by adding event-driven execution. The IEC 61499 standard was also developed, according to Rooker *et al.* (2007), to address the following shortcomings of its IEC 61131 predecessor:

- Non-deterministic switching points – this is due to the cyclic execution policy which is implemented by the IEC 61131 standard.
- Lack of task level granularity[1] complicates communication and re-initialization.
- Jittering effects – a change in one system task influences the other tasks in the system.
- The possibility of entering inconsistent states during system reconfiguration, which may lead to deadlocks.

The IEC 61499 standard is then a developed set of specifications for distributed processes and control systems (Wang *et al.*, 2007). Black and Vyatkin (2009) mention that the IEC 61499 standard "provides an architectural framework for the design of distributed and embedded control systems" and has "undoubted advantages concerning distributed automation" (Vrba *et al.*, 2009). The IEC 61499 standard defines a component-based modelling approach using function blocks. The standard enables the development of new technologies which aim to reduce design efforts and enhance reconfiguration. The goal of the IEC 61499 standard is "to offer an encapsulation concept that allows the efficient combination of legacy representation forms (such as ladder logic) with the new object and component-orientation realities" (Vyatkin, 2007). The IEC 61499 standard uses a bottom-up approach in implementing decentralized control. This approach then starts at the shop floor level, where it effectively prepares for the distributed placement of holons (Paolucci and Sacile, 2005). The requirements for

[1] Presumably the extent to which control programs can be subdivided into smaller modules.

holonic control are thus inherent in the IEC 61499 specification (Black and Vyatkin, 2009).

The function block of the IEC 61499 standard can be understood as an abstraction that represents a component. This component can be implemented and controlled by the function block software (Vyatkin, 2007). The function block concept is applicable to the data encapsulation and adaptive process plan execution involved in the assembly or manufacturing processes. The event-driven model of the function blocks then adds intelligence and autonomy to the resources of the system, increasing its decision-making ability (Wang *et al.*, 2007).

2.5.2 Advantages of function block control

Function blocks provide an advance from established ladder logic and structured text programming languages, but its application extends past the simple replacement of these systems. This is due to the inherent support for distributed applications and the ability to provide a modelling and simulation platform with well-defined interfaces (Black and Vyatkin, 2009). Rooker *et al*. (2007) mention that the distributive properties of IEC 61499 function blocks hold several advantages. The programmed function block networks are directly mapped to the real system controllers and devices, where the execution takes place. This facilitates the movement of functionality amongst controllers and devices. This support of distribution then also facilitates the implementation of component-based information. Another benefit of using the IEC 61499 function blocks is that, as a modeling language, it is directly executable and is thus ready for simulation. This allows the testing of the control system prior to deployment. This simulation model can then be seamlessly substituted by the hardware interface to real sensors and actuators. The use of function blocks also greatly increases the modularity of the system and enables the reusability of software components in the system (Black and Vyatkin, 2009). Function blocks also have a robust character which makes it appropriate for implementation in the broader embedded systems domain (Vyatkin, 2007).

2.5.3 Platforms for function block control

There exists a few platforms and tools for the design of function block control systems. The Function Block Development Kit (FBDK) is the most widely-used design platform (Black and Vyatkin, 2009). The model-view-control design pattern for function blocks is also applied in FBDK. This platform also includes the Function Block Run-Time (FBRT) environment. The entire platform is based on Java programming structures. Another commercial support tool is that of the ISaGRAF industrial control design software, which can also support the IEC 61499 function blocks (Black and Vyatkin, 2009).

2.5.4 Implementations of IEC 61499 function block control

Due to the predominant presence of the IEC 1131-3 standard in industry and relatively recent development of the IEC 61499, it has seen very few practical implementations. IEC 61499 function block control was implemented in the automation of a baggage handling system by Black and Vyatkin (2009). Vyatkin

(2007) describes the first factory installation of an IEC 61499 function block control system by Tait Control Systems in New Zealand.

3. Case study description

This thesis uses, as a case study, the control system of the feeder subsystem hardware of the experimental RAS at the Department of Mechanical and Mechatronic Engineering at Stellenbosch University. The RAS is an automated implementation of the spot-welding process involved in the production activities of CBI Electric.

3.1 Product description

A complete trip switch assembly, as produced by CBI Electric, consists of several sub-assemblies. The experimental RAS of this research was set up to produce one of these sub-assemblies, which consists of six parts that are attached through a spot-welding process. The sub-assembly is shown in Figure 4. The sub-assembly consists of six parts: the moving contact, the coil, the load terminal, the handle frame assembly and the long and short pigtails. The five spot-weld points are encircled.

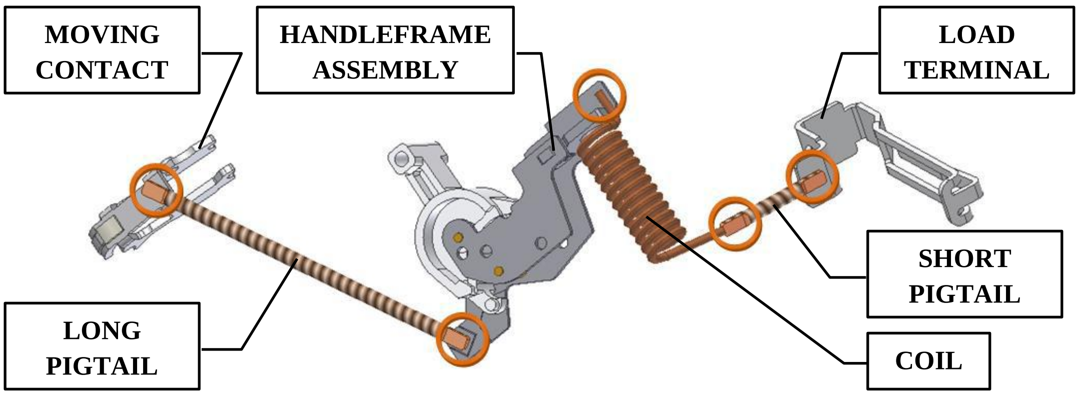


Figure 4: The case study sub-assembly with the spot weld points indicated.

3.2 System overview

The case study used in this research contributes to the development of an RAS for an automated spot-welding process. The experimental RAS consists of four subsystems: the transport subsystem, the storage subsystem, the feeder subsystem and the welding subsystem.

The transport subsystem uses a modular conveyor system to move pallets to designated stations. These pallets are stored in the storage subsystem. The storage subsystem utilizes a large pallet magazine which can store, dispense and retrieve pallets. Different fixtures, for the various system products, are mounted on these pallets. The pallet magazine can store three different pallet types (according to the mounted fixtures) separately. The pallet magazine can then dispense or retrieve a specified pallet to or from the appropriate storage area.

The welding subsystem uses a three-axis Cartesian robot fitted with a simulated welding head. This robot manipulates the welding head to simulate the spot-welding process required to produce the sub-assembly of this case study.

The removal station shares the services of the pick-‘n-place robot of the feeder subsystem. The robot removes the completed sub-assemblies from the fixtures and places them in an output bin. In the case of defective sub-assemblies, the robot removes the parts individually and places them in a recycling bin.

The feeder subsystem is described in detail in section 3.3. The layout of the experimental RAS is shown in Figure 5.

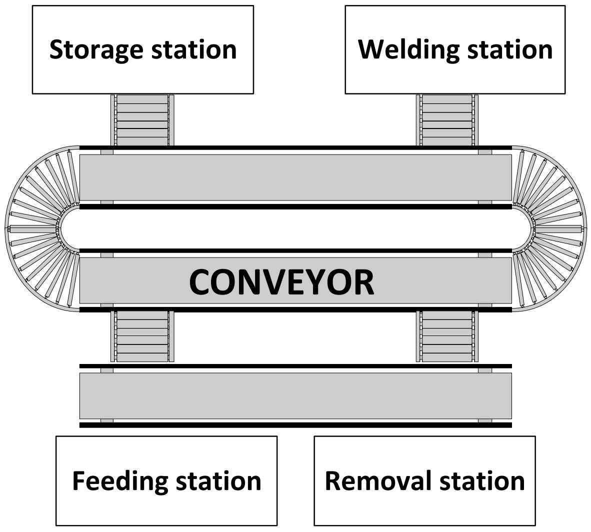


Figure 5: Schematic layout of the experimented RAS.

3.3 Feeder subsystem

The feeder subsystem is responsible for the loading of individual parts, which make up the sub-assembly, onto the transport subsystem. The parts are placed in fixtures which are mounted on the pallets of the conveyor system. The feeder subsystem consists of several singulation units (SUs) or part magazines and a pick-‘n-place robot. The layout of the feeder subsystem is shown in Figure 6 and Figure 7.

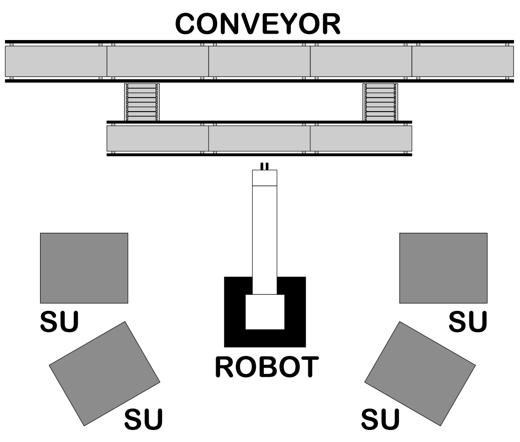


Figure 6: Schematic layout of the feeder subsystem.

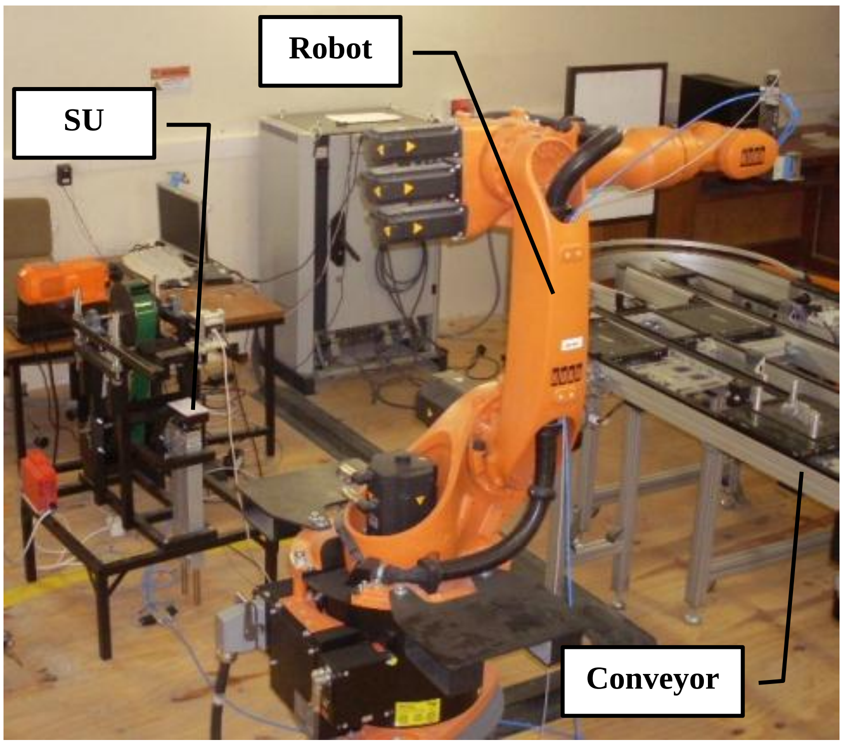


Figure 7: Hardware of the feeder subsystem.

3.3.1 Singulation units

The function of the singulation units (or sometimes referred to as feeders) is to present a single part – to be picked up by the pick-'n-place robot – from bulk container. The parts must be presented in a collectable pose, i.e. the parts must be in an orientation in which the robot can pick them up and place them in the fixture. This process can be described as singulation.

The singulation unit used in this case study is based on the "stepped-conveyor" concept – it is shown in Figure 8. The singulation unit has a conveyor belt, fitted with scoops, which pick up individual parts from its input bin. At the top of the conveyor cycle, the individual parts fall through a gateway mechanism which channels the parts to either the presentation platform or the rejection shoot. The singulation unit is fitted with a camera for the detection and inspection of presented parts. When in operation, the camera continuously checks the presentation platform for the presence of a part. When a part is presented, the camera sends feedback to the subsystem controller to change the direction of the gateway mechanism (so that parts are channelled to the rejection shoot). After a part is detected, the camera performs an inspection to determine whether the part is in a collectable pose. If the part cannot be picked up by the robot, or multiple parts are present, the platform is lowered and the part(s) are rejected back to the bin. In the case of a collectable part, the camera responds to the control program with the pickup coordinates. The presentation platform is then lifted to a level above that of the camera – this is to ensure that there will be no interference during the pickup action of the robot.

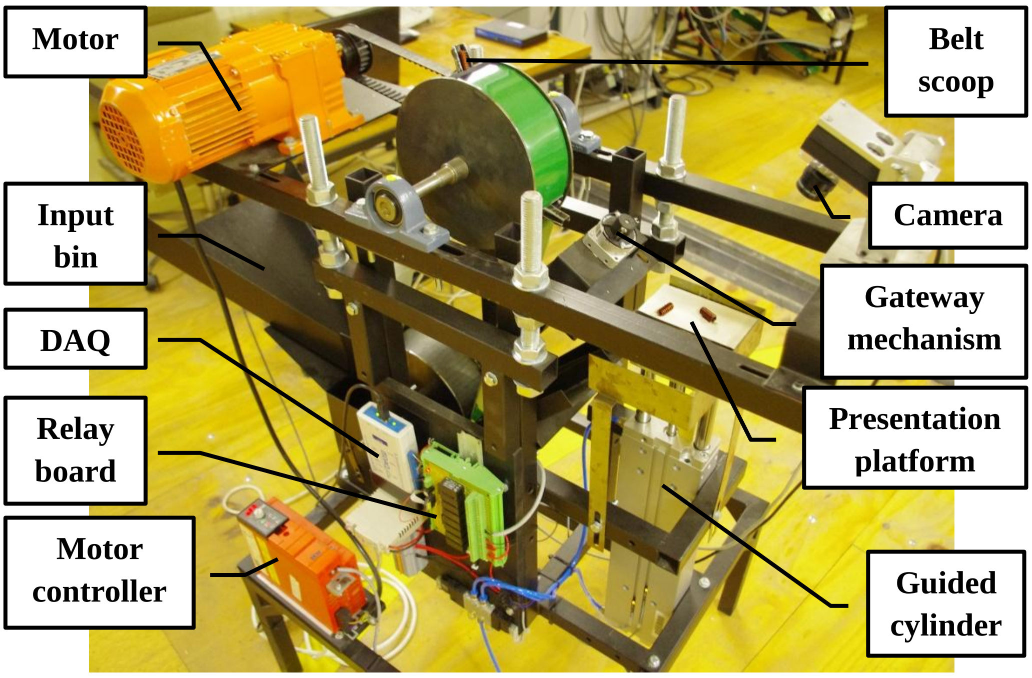


Figure 8: Stepped-conveyor singulation unit.

The conveyor of the singulation unit is driven by an AC motor. The torque is transmitted through a timing belt to the conveyor pulleys. The gateway mechanism uses a pneumatic swivel unit – this actuator causes the rotation of a deflector plate. The motion of the presentation platform is provided by a guided linear pneumatic cylinder. The position of the platform is monitored by using two proximity switches – one for the *home* position (the position of the platform when awaiting parts from the conveyor) and one for the *rejection* position (the position of the platform when a part is rejected). The rejection action of the presentation platform is done by tilting the platform as it is lowered. The platform is attached to the cylinder through a pivot support. A tilt pin is mounted beneath the platform to force tilting as the platform is lowered.

The actuators of the singulation unit are controlled via the digital outputs of an Eagle µDAQ-lite device. The 5V digital outputs of this device are used to switch the relays of an Eagle relay board, to which the actuator inputs are connected. The digital outputs of the DAQ device control the motor, the direction of the gateway mechanism and the motion of the guided cylinder. The digital inputs of the DAQ device are used to read the status of the proximity switches, thus monitoring the position of the guided cylinder.

3.3.2 Part magazines

Since the singulation units were still under development, several part magazines were designed and manufactured to allow for the complete production simulation of the feeder subsystem. These part magazines present parts for pickup at predefined coordinates. The parts are placed into the magazines manually.

Part magazines were designed and manufactured for the load terminal, short and long pigtail, handle frame assembly and moving contact parts (shown in Figure 9). The part magazines are specific to their respective parts. The parts are held in position by several supports, which constrain the parts in all the degrees of freedom. The moving contact, handle frame assembly and load terminal parts are held in position by dowel pins. The long and short pigtails are placed in the manufactured grooves on the part magazine.

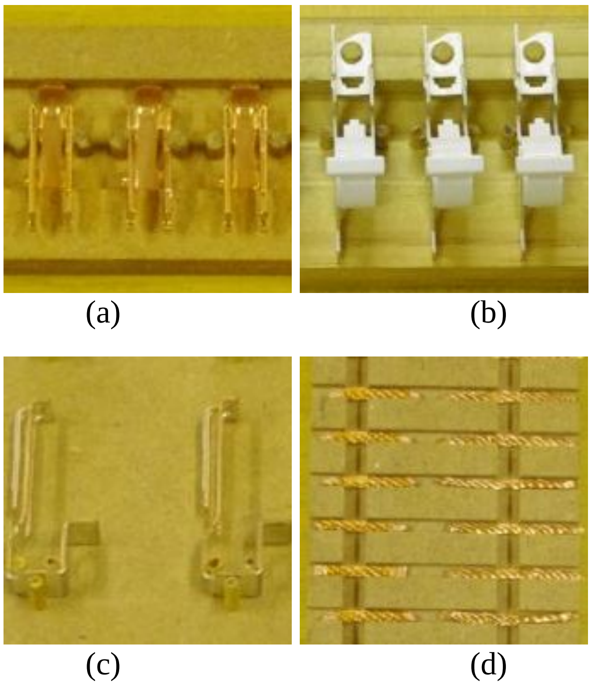


Figure 9: Part magazines for the (a) moving contact, (b) handle frame assembly, (c) load terminal and (d) long and short pigtail parts.

3.3.3 Camera

The feeder subsystem requires the installation of a camera to obtain the coordinates of parts presented by the singulation units. The position of the camera is shown in Figure 10. The camera performs inspections which return the coordinates of the part pickup position. These coordinates are used by the robot to pick up the parts and place them in the fixtures. A DVT Legend 530 series camera was mounted on the singulation unit.

The camera is accompanied by DVT Intellect software which is installed on a PC. The connection between the camera and the PC is done over an Ethernet connection. This software is used to set up the machine vision inspection, using built-in functions and customized script programs, and loading the setup into the flash memory of the camera. The use of the DVT Intellect software is described in section 5.2.

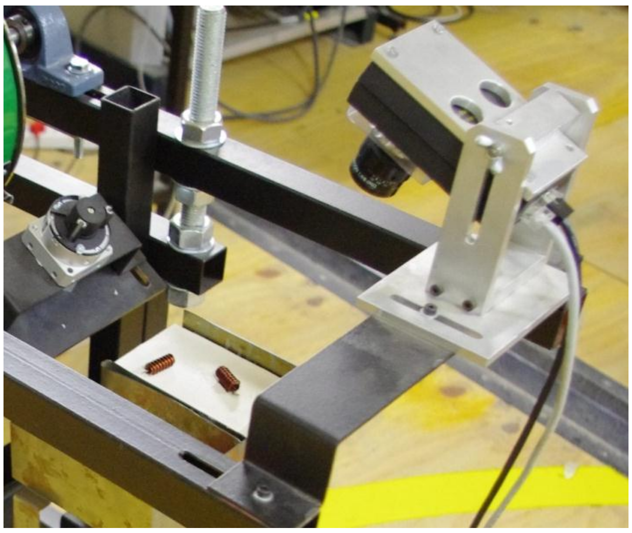


Figure 10: The camera mounted on the singulation unit.

3.3.4 Robot

The pick-‘n-place robot of the feeder subsystem is used to pick up parts from the singulation units and part magazines, and place them in their appropriate fixture positions. The robot is fitted with a pneumatic gripper equipped with customized gripper fingers. The pickup coordinates of the parts presented by the singulation units are passed on to the robot from the camera inspection.

The feeder subsystem is equipped with a six degree of freedom, articulated KUKA KR16 robot (shown in Figure 11) for all pick-‘n-place actions. The robot is accompanied by a controller, which uses an industrial PC with a Windows XP operating system. The robot motion can be controlled either through a control pendant or by customized programs developed in the KUKA Robot Language (KRL) software platform. The latter was predominantly used in the motion programming of the robot. The KRL control software has several built-in functions to accommodate and simplify the calibration and motion programming of the robot. The controller is also equipped with a serial communication port, with built-in functions to send and receive information. The robot is not equipped with any analogue outputs, which means that the gripper tool must be controlled by an external source.

Two useful functions included in the KUKA control are that of tool calibration and workspace definition. These functions contribute greatly to the reconfigurability of the feeder subsystem, as they ease the change of gripper components and the positional recalibration after the relocation of subsystem hardware components. The steps involved with performing these functions are presented in Appendix E.

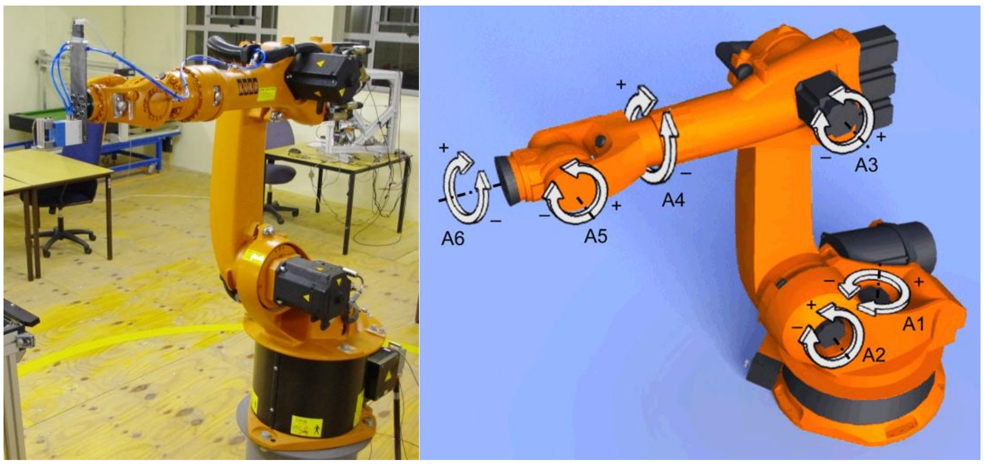


Figure 11: Robot with axis movement indicated (KUKA Robot Group, 2007).

A gripper is required to pick up and place the parts in the operation of the feeder subsystem. A plate, on which a pneumatic gripper is mounted, is attached to the tool interface of the robot. The gripper, as it is mounted on the robot, is shown in Figure 12. The control valve of the gripper is also mounted on the attachment plate.

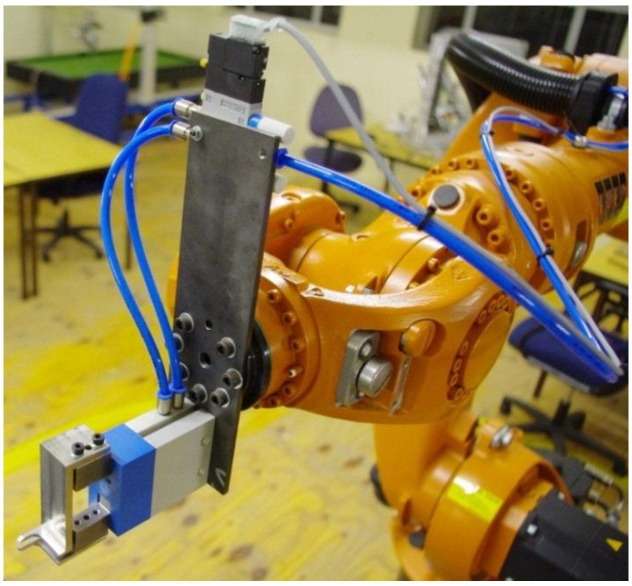


Figure 12: The gripper as it is mounted on the robot.

The gripper is equipped with two custom-designed fingers for the effective picking and placing of the sub-assembly parts. The fingers are designed to be large enough to ensure a sufficient gripping area, but small enough to allow for gripping inside some of the parts. The design allows for parts to be picked and placed in different orientations. The fingers are also designed to minimize the potential interference of the gripper with the parts, part magazines and the fixtures. The gripper fingers are machined from stainless steel, which allows the fingers to withstand the fatigue demand of the gripping actions. The detail design of the gripper is presented in Appendix B.

3.3.5 Fixture

A fixture was designed to keep the parts in their specified positions during the welding and transport activities. The fixtures were mounted on the pallets of the transport system, as shown in Figure 13.

The fixture was designed to be modular. The support of the individual parts is done by interchangeable support components, which are attached to a base plate. The base plate is then mounted onto the conveyor pallet. The base plates were used to add fixture modularity and to minimize the number of holes to be made in the conveyor pallet. The standard interface between the base plate and the pallet allows for the interchanging of base plates, and thus fixtures. The fact that the support components can be removed from the base plate means that base plate fixtures can be adjusted to allow for other types of sub-assemblies, and that the components can be re-used on other base plates in the construction of new fixtures. A change in the type of sub-assembly can thus be accommodated by installing the appropriate supports on the appropriate base plate.

The fixture was designed to be used without the aid of a clamping mechanism (which secures the parts during the welding process). This was achieved by changing the orientation of the welding process from the vertical plane to horizontal plane, i.e. instead of having the welding electrodes weld from above and below, they weld from the sides of the fixture. The sub-assembly is then fixed in the upright position, allowing the individual parts to be located by the supports against any movement in the horizontal plane. This independence of a clamping mechanism improves the reconfigurability of the fixtures, since a change in the fixture supports does not entail the changing of a clamping mechanism as well.

The supports were designed to simplify the feeding and welding processes by allowing easy entrance for both the gripper fingers and the welding electrodes. The supports for the pigtail parts are designed with slots for the gripper fingers – this allows the pigtails to be placed securely into the supports. The supports are spaced from one another where clearance was needed for the welding electrodes. The placement and welding of the individual parts are depicted in Appendix C.

To allow for the stacking of pallets on top of each other in the pallet magazine, the fixture design included four columns at the base plate corners. These columns press against the bottom of the stacked pallet on top, giving enough clearance to provide for the fixture supports and the pallet RFID tag. The columns are chamfered at the top to allow for the easy alignment during the stacking process.

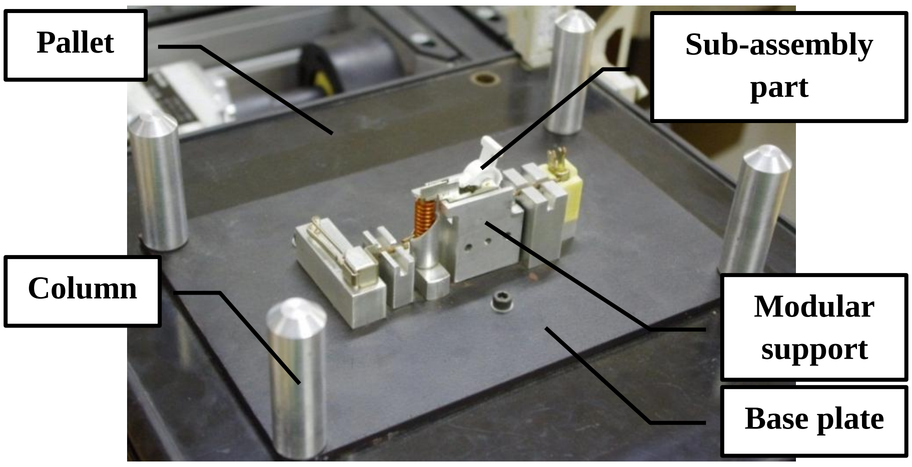


Figure 13: The fixture mounted on a pallet.

3.4 Development and testing of the singulation unit

The stepped-conveyor singulation unit (mentioned in section 3.3.1) is the Stellenbosch research group's second research concept for reconfigurable feeders. The initial design of the singulation unit was done by Poletti (2011), but further refinement was required to get the machine to a working state. These refinements are as follows:

- The input bin was redesigned to allow for the effective scooping of parts. The design had to maximize the potential singulations by the scoops – the number of parts in the bin, along with their position and movement, were the main design considerations.
- The design of the scoops (steps) which are attached to the conveyor belt was refined to increase the effectiveness of the scooping of parts. Experimentation was done concerning the size and shape of the scoops, which affect the frequency of successful singulations from the bin.
- The presentation platform was enclosed in the "home" position (the level of the platform when awaiting parts from the scoops). Without the enclosure parts would often slide or bounce off the platform. The enclosure was designed to ensure parts would remain on the platform, whilst not impeding the motion of the platform or the inspection of the camera.
- The PC control of the actuators and sensors was added.

A series of experiments were performed to evaluate the singulation unit in terms of throughput and reconfigurability – the results are given in Appendix A. The results of the throughput analysis are summarised in Figure 14. The probability of achieving a successful singulation is plotted against time, for different speeds of the conveyor motor (measured in steps per minute, spm). The results show that the singulation unit performs best at a speed of 63 spm, at which speed there exists a 90% probability of achieving a successful singulation within 3 seconds. The graph is a plot of discrete events (as indicated by the markers) and the information is not continuous – the lines connecting the symbols are only shown to aid the interpretation of the results.

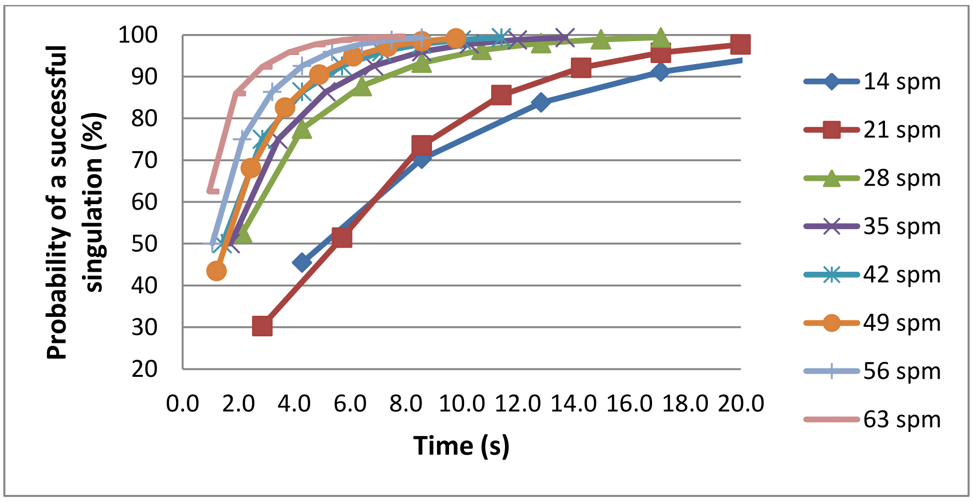


Figure 14: Singulation probability vs. time experimental results for the stepped-conveyor singulation unit.

A subjective evaluation was also conducted to determine the reconfigurability and reliability of the concept. The following remarks can be made:

- The scoops of the singulation unit are specific in terms of part size, i.e. the scoops are able to pick up parts from the bin which are of similar dimensions to the coils used for most of the experiments. This means that the scoops will be able to singulate a family of coil parts. For other types of parts (differing in size and shape), the belt can be replaced with one having appropriate scoops. Since this is the only part/size specific element of the design, the singulation unit retains good reconfigurability characteristics.
- The unpredictability of the pickup action from the input bin reduces the consistency, and thus the throughput rate, of the singulation unit. A great deal of refinement to the input bin and the scoops was required to make the singulation unit work effectively.
- The pickup action of the scoops moving through the input bin causes the occasional deformation of delicate parts. This may be a prohibitive problem if the parts in question are subjected to tight tolerances.
- The location of the camera in the current configuration does not allow an optimal inspection setup. The design requires the camera to be at an angle to the presentation platform (as opposed to being perpendicular). This adds complexity to the reliable identification and location of parts, and it requires longer calibration times during reconfiguration. Reliable machine vision inspection also requires consistent lighting – this is usually achieved by housing the camera inside a box. To address these two concerns would require the addition of further actuators or mechanisms to the singulation unit.

4. Holonic control architecture

The holonic control approach involves the mapping of the subsystem hardware and software components to holons. A holon may consist of only a software component or of both software and hardware components. The mapping of holons was done according to an adaptation of the ADACOR reference architecture (described in section 2.3.3). The ADACOR reference architecture was chosen over PROSA because of two reasons:

1. The ADACOR reference architecture meets the requirements for modern manufacturing systems and specifically addresses challenges not met by PROSA. These challenges include the formal specification of the dynamic behaviour and the achievement of global optimization of holonic systems.
2. A successful and comprehensive implementation was done by Leitao and Restivo (2008) using ADACOR in a similar experimental RMS.

The ADACOR reference architecture had to be adapted for implementation in the feeder subsystem. This adaptation was required due to the level of architecture implementation. The entire feeder subsystem would be mapped to one Operational holon (OH) according to Figure 3, since ADACOR is conventionally implemented at system level. The greatest adaptation is noticeable in the Product holon – the Product holon of the feeder subsystem merely accesses received product information, as opposed to being the primary structure for the storage of system product information.

The implementation of the ADACOR reference architecture was done to increase the reconfigurability of the feeder subsystem control. The decision is motivated by the inherent advantages of ADACOR regarding modularity and the reduction of system complexity.

In accordance with the ADACOR reference architecture, the parts of the feeder subsystem were mapped to a Supervisor holon, Product and Task holons and Operational holons. These holons are described in the remainder of this section.

The subsystem contains the following Operational holons: Singulation unit, Camera, DAQ and Robot. Except for the Singulation unit holon, all the Operational holons comprise hardware and software components. This means that the holons consist of the physical hardware entity, as well as the accompanying software control entities. As an example, the structure of the robot holon is depicted in Figure 15. The structure shows the division of the software entity into Higher Level Control (HLC) and Lower Level Control (LLC) – these control levels are discussed in sections 5 and 6. The Singulation unit holon, on the other hand, consists of only a software entity, since it only coordinates the actions of the other Operational holons.

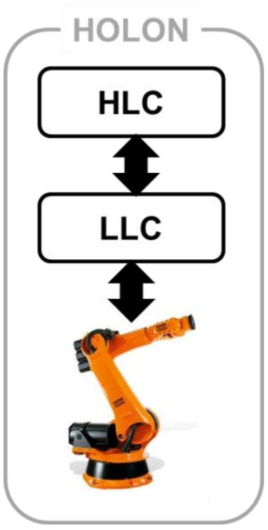


Figure 15: Schematic representation of a holon consisting of both software and hardware entities.

The information of every product to be produced by the subsystem is sent by the Cell Controller (CC) to the feeder subsystem, where it is stored locally. The retrieval and interpretation of this information is mapped to a Product holon specific to the product. The Product holon has access to the information regarding the coordination of subsystem tasks, along with the necessary coordinate and part data to be used in performing them. The creation of a Product holon for each product was done due to initial considerations of containing all the product information within the subsystem. When the product information is contained only in the CC, the information regarding all the products could be handled by one generic Product holon.

Each task that the subsystem can perform is mapped to a Task holon. The Task holons possess the necessary information and decision-making functionality to coordinate the actions of the Operational holons to perform a specified task. For example, a Task holon is created for the control of picking up a specific part from the singulation unit and placing it into the appropriate fixture position – it thus has to control the functions of the singulation unit, camera, DAQ and robot holons.

Finally, the ADACOR reference architecture requires the addition of a Supervisor holon. This holon consists of only a software entity, which has the information and capability required to coordinate the other holons in the subsystem to perform a desired sequence of actions. The Supervisor holon also interfaces with the control of the other subsystems.

5. Lower Level Control and interfacing

As shown in Figure 15, holons contain a Lower Level Control (LLC) layer for interfacing and controlling their hardware component. LLC programs were developed to control the subsystem hardware, or interface with the hardware-specific control programs. The LLC programs also have a communication interface with the Higher Level Control (HLC) programs. This intermediate layer was included to reduce the complexity of the HLC programs by separating it from the hardware interfaces.

The LLC programs communicate with the HLC programs through TCP/IP sockets in XML (eXtensible Markup Language) format. The LLC programs act as the servers to the sockets and the HLC programs then connect as clients. Both control levels are equipped with XML building functions, to construct messages, and XML parsing functions, to extract message information. The LLC programs receive commands from the HLC programs, perform the desired hardware actions and then respond with completion messages.

The LLC programs were developed in the Microsoft Visual Studio C# platform. The C# platform was chosen because of its robustness and ease of use – specifically in accommodating communication through TCP/IP and serial RS232. C# was chosen as opposed to Java (in which the HLC programs are programmed) because of two reasons:

1. The Java library for supporting serial communication (such as RS232) has not been updated since 2006 and has been criticised by software developers for its unreliability. On the other hand, C# is renowned for its reliability – especially due to its use of the .NET framework.
2. Since C# is commonly used for lower level PC-based control, drivers for hardware devices are more easily available. This was the case for the Eagle DAQ device.

The XML standard was chosen due to the following advantages (as mentioned by Exforsys Inc. (2007)):

- XML is a text-based language. This means that the messages are readable by humans, which allows for easy understanding and debugging by the software developer.
- XML is extendable. The specification allows for the unrestricted creation of customized message tags.
- XML is platform, system and vendor independent – this is very beneficial when used in distributed applications.

5.1 DAQ LLC

The DAQ LLC program directly controls the functions of the Eagle µDAQ-lite device (using the device driver) via the USB interface. The actions of the singulation unit components (guided pneumatic cylinder, pneumatic swivel unit, AC motor and proximity switches), as well as the robot gripper, are controlled by setting the digital outputs and reading the digital inputs of the DAQ device. The functionality of the DAQ LLC program is illustrated in Figure 16.

The DAQ LLC program starts by initializing the required variables and then creating a TCP/IP socket. The DAQ LLC program acts as server to the socket, while the HLC program connects as a client. Upon connection, the DAQ LLC program receives a command from the HLC program in the format of an XML string. This string is then parsed to extract the command information, which will be used to trigger the appropriate DAQ function. After the desired function is performed, a confirmation message is compiled in the form of a XML string. This message is sent to the HLC program through the TCP/IP socket. The socket connection is then closed and the next connection of the HLC program is awaited.

The command received by the DAQ LLC program entails an integer number to which a predefined DAQ function is allocated. The number is extracted and, by means of a *switch* function, the desired function is triggered. The functions are implemented in the form of methods. The methods which directly access the digital outputs and inputs are summarized in Table 1.

Table 1: The DAQ LLC methods and the respective DAQ control functions.

Method	**DAQ function**	**Control action**
startMotor( )	Starts the conveyor motor.	Write to digital outputs.
stopMotor( )	Stops the conveyor motor.	Write to digital outputs.
liftPlatform( )	Switches the valve port to initiate upward motion of the guided cylinder.	Write to digital outputs.
lowerPlatform( )	Switches the valve port to initiate downward motion of the guided cylinder.	Write to digital outputs.
stopPlatform( )	Switches both valve ports on to stop the motion guided cylinder.	Write to digital outputs.
openGripper( )	Switches the valve port to open the gripper fingers.	Write to digital outputs.
closeGripper( )	Switches the valve port to close the gripper fingers.	Write to digital outputs.
readSensor( )	Monitors the switching of the proximity sensors.	Read digital inputs.

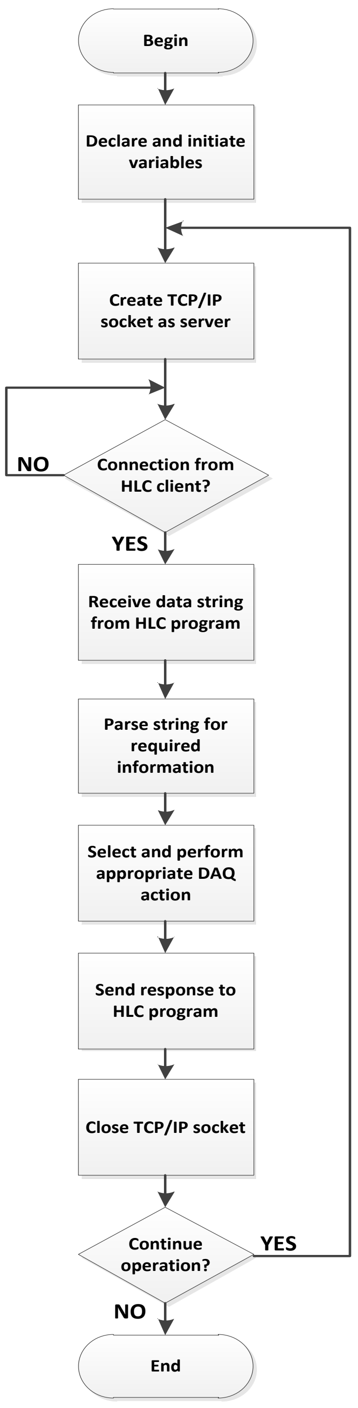

Begin
Declare and initiate variables
Create TCP/IP socket as server
NO
Connection from HLC client?
YES
Receive data string from HLC program
Parse string for required information
Select and perform appropriate DAQ action
Send response to HLC program
Close TCP/IP socket
Continue operation?
YES
NO
End

Figure 16: Flow diagram of the DAQ LLC functionality.

These methods are combined to accomplish more complex functions. The method *homePlatform( )* returns the presentation platform to the "home" position (the level at which the presentation platform can receive parts from the conveyor). The current position of the presentation platform is stored in a local variable. This variable is checked to determine whether the platform should move upwards or downwards. The motion is initiated by calling either the *lowerPlatform( )* or *liftPlatform( )* methods. The digital inputs, indicating the status of the proximity sensors, are then continuously monitored using the *readSensor( )* method. When a change in the digital input is received (indicating that the platform is at the "home" level), the *stopPlatform( )* method is called to stop the actuator motion.

The method *rejectPlatform( )* causes the presentation platform to be lowered to the "reject" position (the position where the platform is tilted and the parts slide down the rejection shoot), stop and return to the "home" position. This action is accomplished by calling the *lowerPlatform( )* method to initiate downward movement. The digital input, connected to the proximity sensor which indicates the "reject" position, is continuously monitoring by calling the *readSensor( )* method in a loop. Upon reaching the "reject" position, the cylinder is stopped by calling the *stopPlatform( )* method. The *homePlatform( )* method is then immediately called to return the platform to the "home" position.

Functions such as switching the motor on or off, opening or closing the gripper and changing the direction of the gateway actuator are purely controlled through the digital outputs of the DAQ device.

5.2 Camera LLC

The camera LLC has two parts: the PC-based C# LLC program and the DVT Intellect inspection control. The LLC program handles communication between the HLC and the camera, while the DVT Intellect inspection control controls the camera actions.

5.2.1 Inspection control

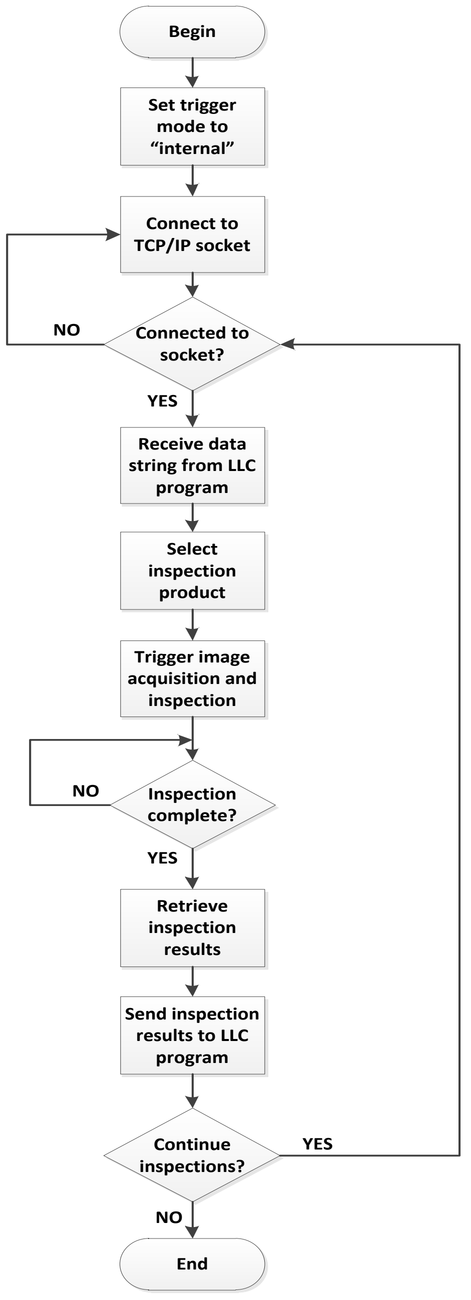
A machine vision inspection was set up for the camera using DVT Intellect software. A background script program, which runs continuously, handles the communication with the C# LLC program. This background script program also coordinates the camera inspections. A unique inspection was set up for every part to be singulated – this set-up is referred to as an inspection product. These products implement several built-in image processing software sensors and custom foreground script programs to determine the inspection result and to extract the necessary inspection information.

A background script program was created to monitor and execute certain functions continuously, without disrupting any triggered inspections – the flow diagram is shown in Figure 17 (a). The background script program connects as a client to the TCP/IP socket created by the C# LLC program. The command and part ID is passed on to the background script program in the form of a byte array. The elements of the array are then checked to determine that the command is indeed to inspect a part, and the appropriate inspection product is selected. The part ID is used to select the inspection product – this is done using the *prod.Select( )* function. The background script program then triggers the acquisition of an image and the succeeding inspection by the specified inspection product – this is done by using the function *SetInputs( )* to set the trigger bit in the registers of the camera. With the inspection triggered, the background script program waits for the softsensors and the foreground script program of the selected inspection product to finish the inspection. The foreground script program indicates the inspection completion by setting a bit in a specified register – this bit is checked by the background script program through the *RegisterReadByte( )* function. The foreground script program stores the inspection result string in a specified register - the background script program then reads the string from the register and replies to the LLC program via the TCP/IP socket. Unless the inspections are manually stopped, the program awaits the arrival of the next command from the LLC program.

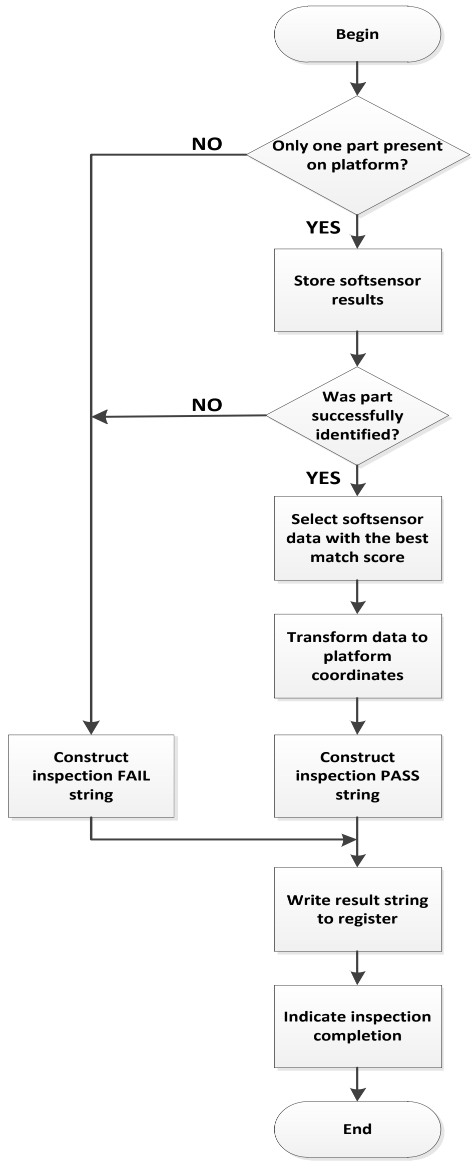
A foreground script program was included in the inspection product to generate an inspection result from the softsensor data. The program is triggered with each inspection, after all the softsensors have completed their analysis – the flow diagram is presented in Figure 17 (b). The foreground script program first declares and initiates all the variables to be used for the temporary storage of data. The first step is to determine if only one part is present on the presentation platform. This is done by evaluating the number of blobs detected by the blob identification softsensor. If more than one part is detected, a FAIL message is constructed and stored to the result register. With only one part present, the program now checks if the part was sufficiently identified by the feature detection softsensors. This entails the storage of the softsensor results to variables in an array and then evaluating the results in a loop. If none of the softsensors could sufficiently identify the part, a FAIL message is generated. Otherwise, the best identification must be determined by comparing the matching scores (relative to the learned models) of the softsensors. The coordinate results from the softsensor with the best identification are now transformed to the physical coordinates of the platform. This coordinate set is included in the generated PASS result, which is stored in the result register. The foreground script then indicates the completion of the inspection by setting a specified bit in a register using *RegisterWriteByte( )*.

The foreground scripts evaluate the data which is gathered by several softsensors in order to determine the inspection result. Each inspection product implements a different set of softsensors, according to the part that is being inspected. The inspections make use of edge detection, blob identification and feature location softsensors. The implementation of each softsensor is described in the following paragraphs.

The inspection product which is responsible for detecting the presence of a part on the platform implements "along a line" edge detection softsensors. These softsensors use differences in pixel intensity to detect edges, along a defined straight line through the image. The difference in contrast between the white background of the presentation platform and presented parts allows these softsensors to detect a part (by detecting an edge in the image). The inspection implements six of these line softsensors, so as to detect a part in every position on the presentation platform – the inspection setup is shown in Figure 18. This approach was selected because of the speed and robustness of the edge-detecting line sensors. The speed of the softsensor is an especially important measure, as the feedback from the inspection has to be quick enough to stop the singulation unit conveyor before multiple parts are delivered to the platform. The accumulated processing time required by all six line softsensors is less than implementing an alternative blob detection softsensor over the platform area. The edge detecting softsensors are also more robust against changes in light intensity than the blob-detecting alternative.


Begin
Set trigger mode to "internal"
Connect to TCP/IP socket
Connected to socket?
NO
YES
Receive data string from LLC program
Select inspection product
Trigger image acquisition and inspection
Inspection complete?
NO
YES
Retrieve inspection results
Send inspection results to LLC program
Continue inspections?
YES
NO
End

(a)


Begin
Only one part present on platform?
NO
YES
Store softsensor results
Was part successfully identified?
NO
YES
Select softsensor data with the best match score
Transform data to platform coordinates
Construct inspection FAIL string
Construct inspection PASS string
Write result string to register
Indicate inspection completion
End

(b)

Figure 17: Flow diagrams of the (a) background and (b) foreground script programs.

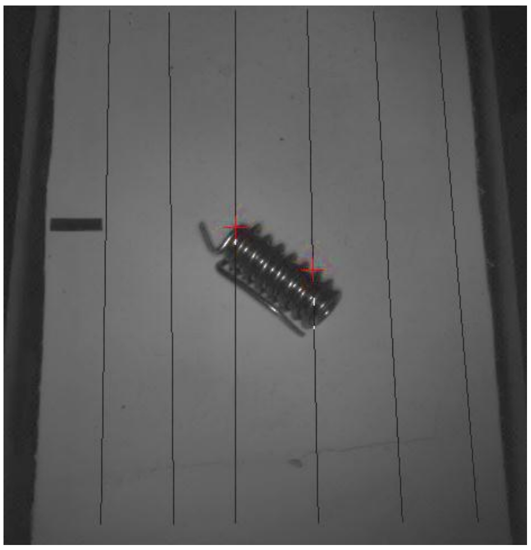


Figure 18: The setup of the inspection product for detecting parts on the presentation platform.

The inspection product for locating a coil part on the platform implements several types of softsensors. These softsensors have to gather information regarding:

- The number of parts present on the presentation platform.
- The identification of the part.
- The coordinates of the pickup position of the part.

The number of parts present on the platform is evaluated by a blob detection softsensor over the platform area. Blob detection involves the grouping of pixels of similar intensity into so-called "blobs". The intensity, size and shape of these blobs can then be analysed. Parts on the platform will thus appear as blobs, of which the number is counted. The blob detection softsensor was selected above the feature locating softsensor for two reasons. Firstly, the blob detection softsensor requires less processing time and, secondly, there is no need to extract details such as the part shape or position at this stage. The implementation of the blob detection softsensor is illustrated in Figure 19.

The part on the platform is identified by an object location softsensor. This softsensor searches for a learned model (a predefined pattern) in the image. During the setup of the inspection, an image of the part is used to calibrate the softsensor – the outline (perimeter) of the part is extracted and taught to the softsensor. The softsensor then scans the pixels in the image in search of this outline pattern. The perimeter of the part on the platform is compared to this model and the degree of similarity is calculated as a "match score". A higher match score indicates greater pattern similarity. The object locating softsensor was chosen over its blob detection counterpart because it extracts more detail and is more robust to changes in light intensity.

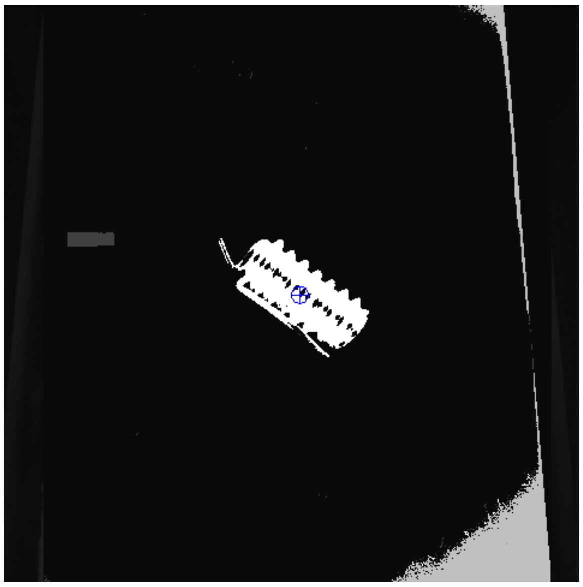
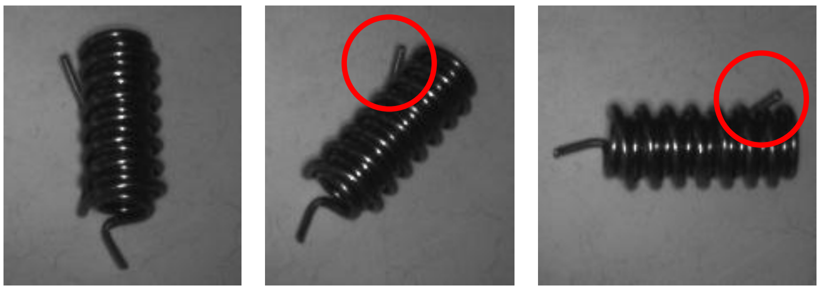


Figure 19: The implementation of the blob detection softsensor.

This inspection product requires the implementation of eight different object locating softsensors. This is due to the fact that the camera is positioned at an angle to the platform (as opposed to directly above). The angle causes the obscurity of part detail in the image with angular rotation of the part, as shown in Figure 20. This means that the shape of the part will be different to that of the learned model, causing the softsensors to not identify the part. The angular rotation of the part also causes a change of part shape (in terms of length and width) in the image – this is also noticeable in Figure 20. Multiple softsensors, each with a different learned model, is thus necessary to identify the part in any rotational position. Four softsensors were implemented, each with a learned model of the part at 0°, 90°,180° and 270° respectively. The coils could also have two possible orientations – the normal orientation or the flipped-over orientation (shown in Figure 21). Another set of four softsensors were implemented to identify the part when it is in the flipped-over orientation. It is thus clear that having the camera at an angle to the platform quadruples the number of object locating softsensors and the accompanying processing time required for identifying the coil parts. This situation is compared to one where the camera is positioned above the platform in Figure 22.


(a) (b) (c)

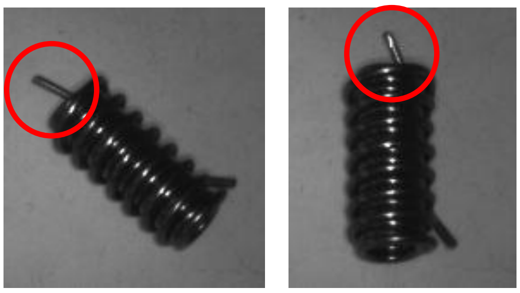

(d) (e)

Figure 20: The obscurity of part features with angular rotation: (a) 0°, (b) 45°, (c) 90°, (d) 135° and (e) 180°.

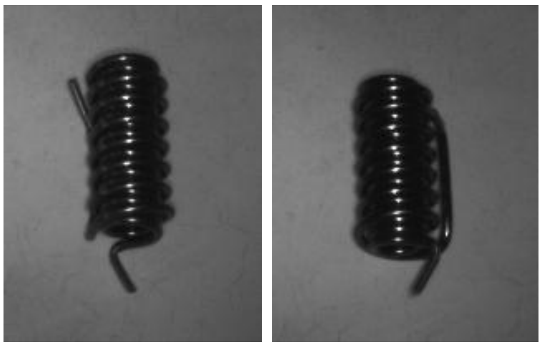


Figure 21: The coil parts in the two possible orientations.

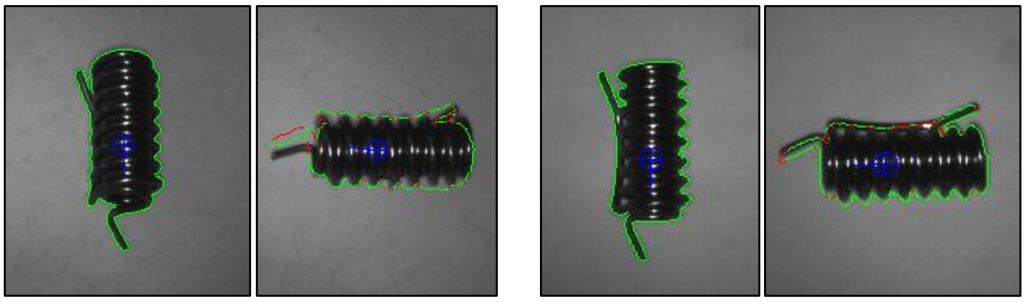

(a) (b)

Figure 22: Variation in inspection results between having the camera at an angle (a) and having the camera directly above (b).

The object location softsensors also return the coordinates of the part pickup position. The desired pickup position is calibrated along with teaching the part model. The softsensor automatically locates the centroid of the model – the pickup position is calibrated by means of an offset to the centroid. Along with this coordinate offset, the pickup angle can also be specified – the softsensor can thus return the X-, Y- and Z-axis pickup coordinates, along with the pickup angle. Initially, the coordinates are returned relative to the origin of the image, and not the origin of the platform in the real world coordinate system. This problem was solved by using two edge detection softsensors which locate the origin of the platform in the image. The edge detection softsensors were implemented instead of pre-programming the origin coordinates – this was done to ease the recalibration of the inspection after a reconfiguration and to continuously monitor the position of the presentation platform during operation. The position of the platform origin is then used in the foreground script program to give the pickup coordinates as an offset from the platform origin.

The coordinates must be transformed to the real world coordinate system, so that the robot can accurately pick up the part. The coordinate transform was done using a standard DVT calibration grid (an asymmetric matrix of dots with equal spacing) and the built-in "coordinate system calibration" tool. The grid was placed on the presentation platform and an image was acquired. The calibration tool evaluated the grid to determine the correct transformation and scaling ratios to relate the pixel coordinates in the image to millimetres in the real world. The transformation and scaling ratios are then applied to the inspection product, allowing softsensors to return real world coordinates.

5.2.2 PC control

The camera LLC program handles all communication between the HLC camera programs and the DVT Intellect script program. When the camera LLC program receives commands (in XML format) from the camera HLC program, these commands are parsed and sent to the Intellect script through a TCP/IP socket. These inspection results are received and compiled into an XML string and passed on to the HLC program. The functionality of the camera LLC program is explained in Figure 23.

The camera LLC program declares and initializes the necessary variables at start-up. This is followed by the creation of a TCP/IP socket for communication with the camera HLC program. The HLC program connects to the socket as a client. When the LLC program receives a message from the HLC program, the relevant information is extracted by parsing the XML string.

The camera LLC program then creates another TCP/IP socket, to which the camera background script program connects as client. The message information is now stored as bytes in the command byte array – this array is sent to the camera background script program. The LLC program awaits the response from the background script program containing the result of the inspection. This inspection result is included in an XML string, which is sent to the camera HLC program.

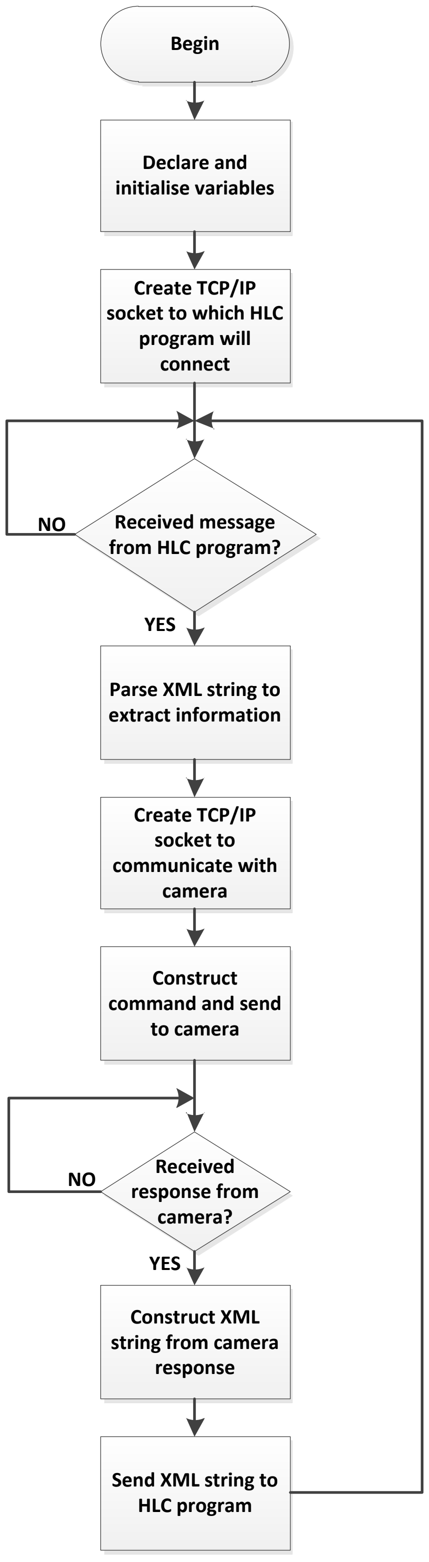

Begin
Declare and initialise variables
Create TCP/IP socket to which HLC program will connect
Received message from HLC program?
NO
YES
Parse XML string to extract information
Create TCP/IP socket to communicate with camera
Construct command and send to camera
Received response from camera?
NO
YES
Construct XML string from camera response
Send XML string to HLC program

Figure 23: Flow diagram of the camera LLC program.

5.3 Robot LLC

The LLC of the robot consists of two parts: a PC-based C# LLC program and the KUKA Robot Language (KRL) programs which reside on the KUKA controller. The roles of these control programs are described in this section.

5.3.1 KRL program control

Several KRL programs were constructed for the low level control of the robot actions – the code of some programs are included in Appendix E. These programs are run on the robot controller. The controller has a communication interface with the controlling PC through a RS232 connection. The KRL programs receive commands from the PC and perform the appropriate robot actions. The KRL platform allows for modular programming – programs can call other programs as subroutines. The functionality of the KRL programs is shown in Figure 24.

Upon start-up of the feeder subsystem, the *MAIN( )* program is run on the robot controller. The necessary variables are declared and initialised at start-up. The program then waits for the arrival of a command message (in the form of an ASCII string) from the robot LLC program. When a message is received, the command, part and coordinate information are extracted. The appropriate subroutine is then called according to the part that is to be handled. The subroutines are part-specific, since the nature of the part, part magazine or singulation unit affects the motion path and the approach position required for the operation.

A specific PICKUP_PART( ) subroutine exists for every part that the robot must pick up. The subroutine is passed the pickup coordinates, as received from the LLC program. The subroutine uses this coordinate information to determine the correct approach position for the robot, i.e. the appropriate position and angle of the gripper to allow for a successful pickup operation. From this position, the robot can be moved to the pickup position. When the pickup position is reached, the subroutine sends a “close gripper” message to the LLC program – this is sent through the same communication channel as used before. After the HLC coordinates the DAQ action to close the gripper, the LLC sends a “continue” message to the controller. The robot is then moved to an intermediate position and the subroutine returns to the MAIN( ) program.

The MAIN( ) continues the operation by calling the PLACE_PART( ) subroutine. The appropriate approach position is again determined for the placement operation. The robot is then moved to the place position, at which point an “open gripper” message is sent to the LLC. With the opening of the gripper, the part is placed and the robot is moved back to the home position. The MAIN( ) then sends a “done” message to the LLC program and awaits the arrival of a new command message.

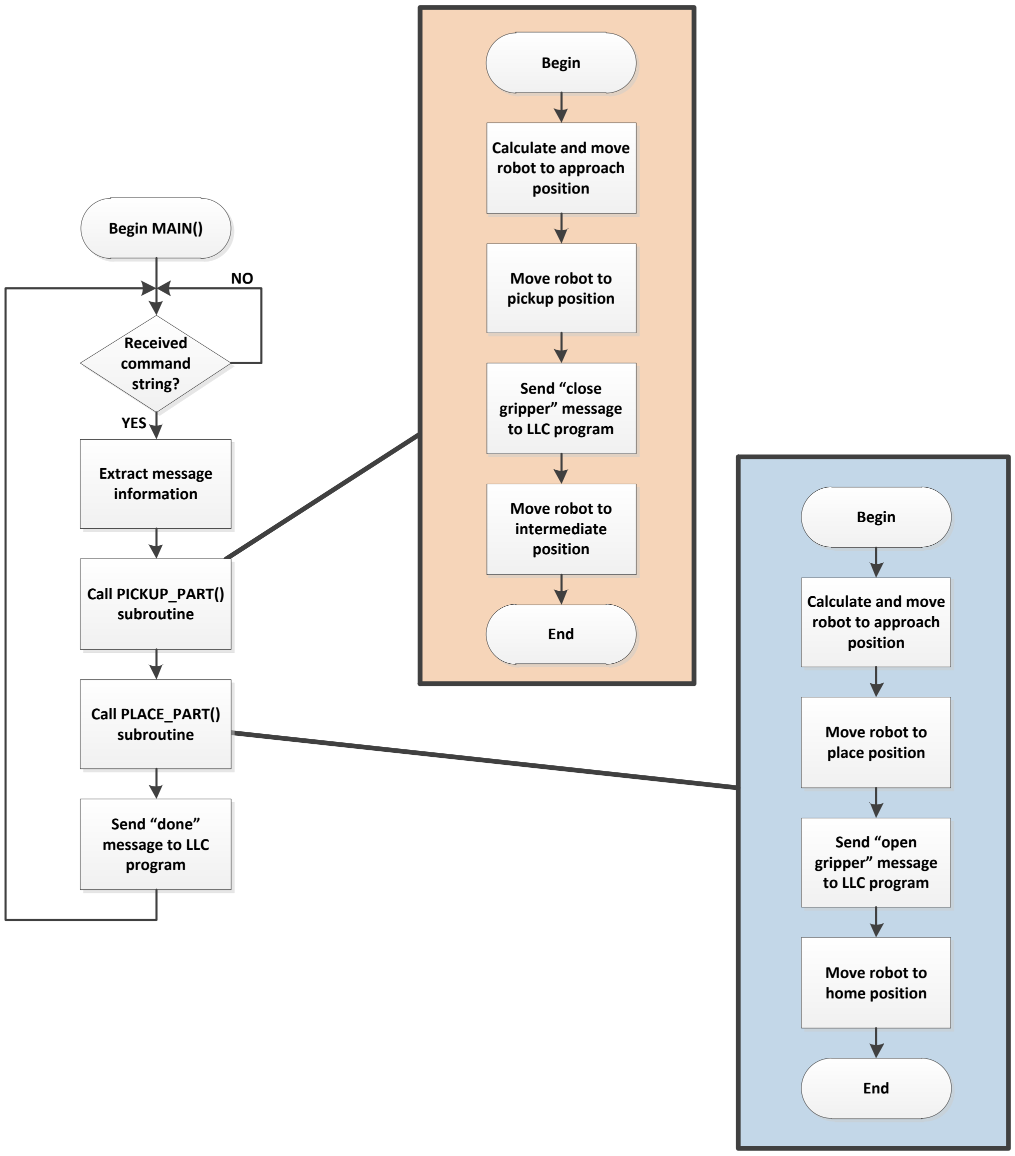

Begin MAIN()
NO
Received command string?
YES
Extract message information
Call PICKUP_PART() subroutine
Call PLACE_PART() subroutine
Send “done” message to LLC program
Begin
Calculate and move robot to approach position
Move robot to pickup position
Send “close gripper” message to LLC program
Move robot to intermediate position
End
Begin
Calculate and move robot to approach position
Move robot to place position
Send “open gripper” message to LLC program
Move robot to home position
End

Figure 24: Flow diagram of the KRL programs functionality.

5.3.2 PC control

All communication between the HLC robot program and the robot controller is handled by the robot LLC program. XML strings are received from the HLC program – these strings include the command, part type and relevant coordinates. The command information is compiled into an ASCII string and is sent to the robot controller via RS232. The working of the robot LLC program is shown in the flow diagram of Figure 25.

The robot LLC program starts by declaring and initialising the necessary program variables. The program creates a TCP/IP socket as a server, to which the robot HLC program will connect as client. With the socket created and a connection established, the LLC program awaits the arrival of a message from the HLC program. This received message is in XML format – the built-in C# functions are used to parse the message for the relevant information.

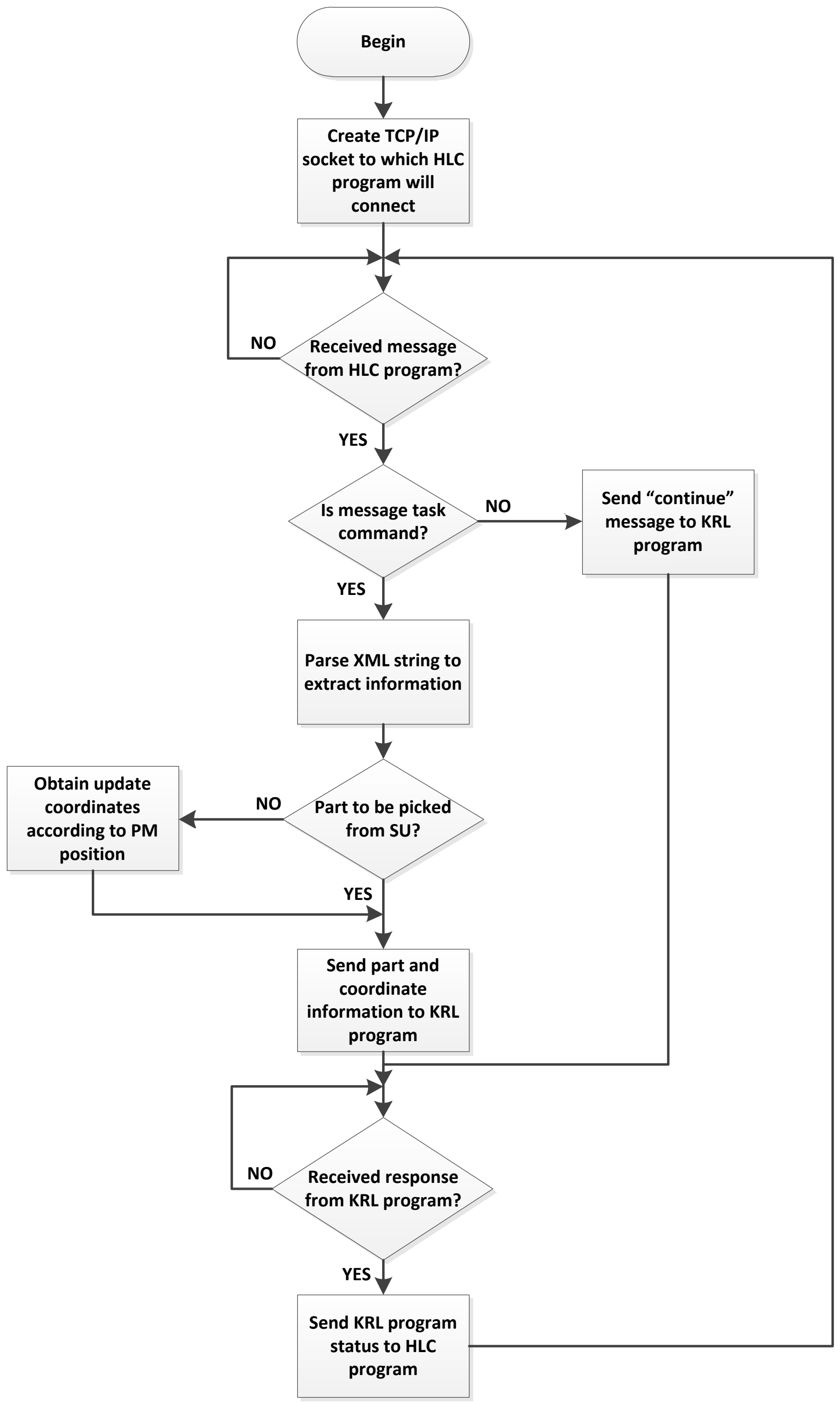

Begin
Create TCP/IP socket to which HLC program will connect
NO
Received message from HLC program?
YES
Is message task command?
NO
Send “continue” message to KRL program
YES
Parse XML string to extract information
Obtain update coordinates according to PM position
NO
Part to be picked from SU?
YES
Send part and coordinate information to KRL program
NO
Received response from KRL program?
YES
Send KRL program status to HLC program

Figure 25: Flow diagram of the robot LLC program.

The robot LLC program can receive two types of messages – a “task” message and a “continue” message. The “task” message represents a HLC command for the robot to perform a task, while the “continue” message indicates that the robot can continue with the current task. This is required due to the fact that the DAQ controls the gripper actuation (since the robot does not have on-board digital or analogue outputs). When the robot reaches a point where the gripper requires actuation, the LLC program sends a message to the HLC program. When the desired gripper action was coordinated by the HLC, a “continue” message is sent to the LLC program.

In the case of a “task” message being received, the robot LLC determines if the part is to be picked up from a singulation unit (SU) or a part magazine (PM). If the part is present at a singulation unit, the message coordinates are used. For parts available from part magazines, the message only specifies the coordinates of the first part in the magazine. As a part is picked up, the LLC program calculates an offset. This offset is stored and used to obtain the coordinates of the next part in the part magazine.

The part and coordinate information is then compiled into an ASCII string, with the “#” character used as separation token. This ASCII string is then sent via RS232 to the KRL programs on the robot controller. The LLC program then waits for a response from the KRL control program. This response if then forwarded to the HLC program in the form of an XML string.

6. Higher Level Control

The Higher Level Control (HLC) is implemented by both an agent-based controller and IEC 61499 function blocks. This control level is responsible for decision-making and coordination of the subsystem functions, and has communication interfaces with both the Cell Controller (CC) and the LLC programs.

6.1 Communication between HLC programs and the Cell Controller

In order to promote the reconfigurability, it was decided that most product information will reside with the CC. This centralization of product information simplifies the process of adding or altering a product – if the information was distributed, changes would have to be made in each subsystem.

The product information is communicated in the feeder subsystem via the Supervisor holon. The Supervisor holon indicates that the subsystem is ready by sending the status information to the CC, through a TCP/IP socket. When the CC requires action from the feeder subsystem, it sends an XML string containing the command and product information. The XML string is structured as follows (the variable tag information is shown in red):

*<CELLCONTROLLER><FEEDER><COMMAND>**LOAD**</COMMAND><PRODUCT>1*

*</PRODUCT><NUMOFTASKS>**6**</NUMOFTASKS><TASK1>**1**</TASK1><X1>**105.6**</X1>*

*<Y1>**150.3**</Y1><Z1>**27.8**</Z1><A1>**0.0**</A1><TASK2>…</A6></FEEDER><CELL-CONTROLLER>*

The string is structured so that all the command and product information is contained within the sender ("CELLCONTROLLER") and receiver ("FEEDER") tags. These tags are used to check if a received message is indeed at its intended destination. The command to be performed by the feeder subsystem is contained in the "COMMAND" tag, with the accompanying product information presented in the next tags. The product number is specified and the number of tasks which are involved in it (in the "NUMOFTASKS" tag). The part and coordinate information is presented in the order of which the tasks must be performed, i.e. the first task to be performed ("TASK1") is the loading of part *X* onto the fixture. The coordinates of this part, as it is to be placed in the fixture, is given in the X1, Y1, Z1 and A1 (referring to the rotation angle) tags.

6.2 Agent-based control

A Multi-Agent System (MAS) HLC was developed using the JADE (Java Agent DEvelopment framework) platform. The functionality, cooperation and communication of the various agents are described in this section.

6.2.1 Control system overview

The MAS is based on the ADACOR holonic architecture, as described in section 4. The holons of the system are embodied by the following agent types: Supervisor, Product, Task and Operational. The MAS implements one Supervisor agent and multiple Product, Task and Operational agents. The Supervisor agent handles all external communication, with the Cell Controller (CC) program, and coordinates the subsystem functions by launching the appropriate Product and Task agents. The Product agent holds all the information required to accomplish the product order, such as the required task sequence and relevant coordinates. The subsystem hardware actions are then coordinated by the Task agents, through communication with the respective Operational agents. The Operational agents interface with the hardware of the subsystem and are thus responsible for the execution of hardware actions. This MAS has an Operational agent for each of the hardware devices, i.e. a Singulation Unit (SU) agent, a Camera agent, a DAQ agent and a Robot agent. The structure of the MAS is depicted in Figure 26.

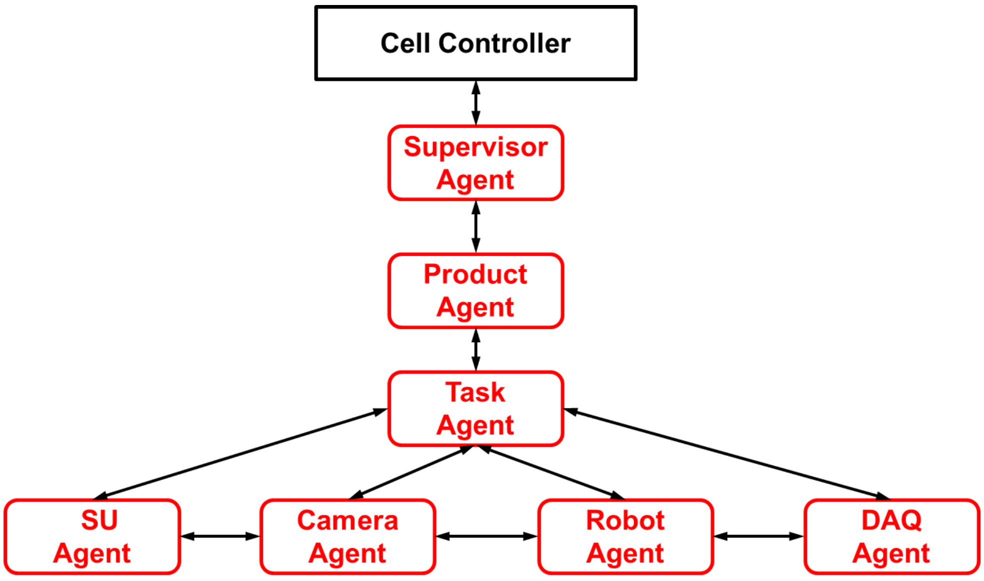


Figure 26: The structure of the Multi-Agent System.

6.2.2 Agent communication and coordination

The cooperation of the agents in the MAS is facilitated by several tools and functions. These tools and functions are explained in this section.

6.2.2.1 The Agent Management System

The FIPA standards require the existence of an Agent Management System (AMS) in an agent platform architecture. The AMS is responsible for the management of the agent platform, of which the main functions are the creation, deletion and life-cycle management of agents. The AMS maintains a physical identifier, referred to as Agent Identification (AID), for each agent residing in the MAS. The AID allows the unequivocal identification of every agent in the system (Paolucci and Sacile, 2005).

6.2.2.2 The Directory Facilitator

All the agents in the system register their services (i.e. the activities which they are able to perform) and address (AID) with the Directory Facilitator (DF). Agents query the DF for agents which provide a specific desired service. The DF then supplies the searching agent with a vector of addresses for the appropriate agents in the system. The searching agent can then initiate communication with the relevant agents in an attempt to contract their services – thus the Directory Facilitator can be related to a "Yellow Pages" service (Paolucci and Sacile, 2005).

6.2.2.3 Contract Net Protocol

The planning and scheduling, inherent in the cooperation the MAS agents, is achieved through an auction process. Auction processes are based on two features – decomposition and negotiation. The decomposition feature refers to the distribution of decision-making ability among all the agents. The negotiation which is involved in the process refers to the decisions which are made following the agent interaction. This auction process is implemented by the Contract Net Protocol (CNP) (Paolucci and Sacile, 2005).

The CNP entails that the subcontracting of agent services commence with a call for proposals (CFP). This CFP specifies the service which is required. Agents which are capable of providing the service reply with proposals to the CFP. These proposals are then handled in the same way as bids during an auction. The proposals are evaluated according to a specific parameter which is relevant to the service, such as completion time. The best proposal can thus be selected and the appropriate agent can be contracted.

6.2.2.4 Ontology

An ontology was used to simplify the intra-agent communication. This MAS implements an adaptation of the ontology developed by Adams (2010). The ontology defined several concepts, actions and predicates to allow for the common understanding between agents. For use in this MAS, some parts of the ontology were omitted as they were not used. The ontology uses several concepts and actions, but no specific need was found for the defined predicates.

The concepts defined for the MAS are presented in Table 2. The PART concept refers to the part involved in a certain task – only the name of the part is required. The POSITION concept refers to the coordinates of a part as given by the camera inspection. PLACE_POSITION refers to the pickup or placement coordinates of a specific part in the fixture. Both concepts require the coordinate slots to be filled with information. The DURATION concept is an indication of the time it will take for an agent to provide a service or perform an action – this time must be presented in the SECONDS slot.

The actions of the system which are represented in the ontology are shown in Table 3. The INSPECT action refers to the inspection of parts by the camera – the PART information is required for the selection of the inspection product. The LOAD and REJECT actions refer to the functions of the singulation unit. The robot receives commands in the form of PICKNPLACE actions – these actions require the PART information and two slots for the coordinate information. DURATION is used throughout as a measure of time involved in performing the actions.

Table 2: Concepts included in the MAS ontology.

CONCEPT	INFORMATION SLOTS
PART	NAME
POSITION	ANGLE X-POS Y-POS Z-POS
PLACE_POSITION	PLACE_ANGLE PLACE_X PLACE_Y PLACE_Z
DURATION	SECONDS

Table 3: Actions included in the MAS ontology.

ACTION	INFORMATION SLOTS
INSPECT	PART DURATION
LOAD	PART DURATION
REJECT	DURATION
PICKNPLACE	PART DURATION POSITION PLACE_POSITION

6.2.2.5 Communication between MAS and Cell Controller

The communication between the MAS and the CC is handled by the Supervisor agent. The agent sends status updates of the subsystem in XML strings, and receives the command and product information from the CC also in XML strings, as discussed in section 6.1.

The Supervisor agent uses the built-in Java XML parsing functions to extract the necessary data. The command information is stored to a local variable, as it is only used to select the appropriate Product agent. The product information must be accessed by the Product agents, thus it is stored in a public array. The storage of the product information is shown in Figure 27.

The tag information is extracted from the XML string and stored in specific array positions. The task information is stored in the order in which they will be performed, i.e. the first row of the array holds the information for the first task. This information consists of the part type involved in the task (stored in the first column) and the accompanying coordinates (stored in the succeeding columns). The coordinates include the X-position, Y-position, Z-position and a rotation angle (indicated as A*n* in Figure 27). The Product agent can then access the product information to select which task should be performed (according to the part type) and then pass the coordinate information on to the appropriate Task agent.

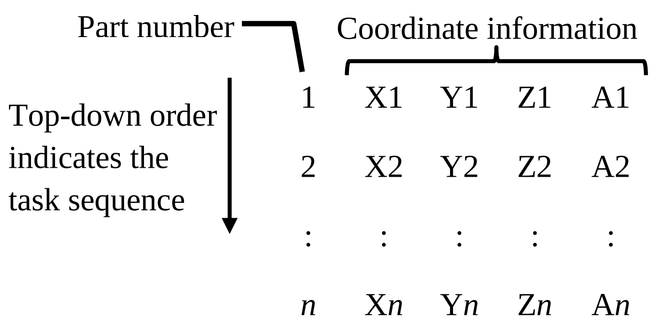


Figure 27: Storage of product information in the MAS.

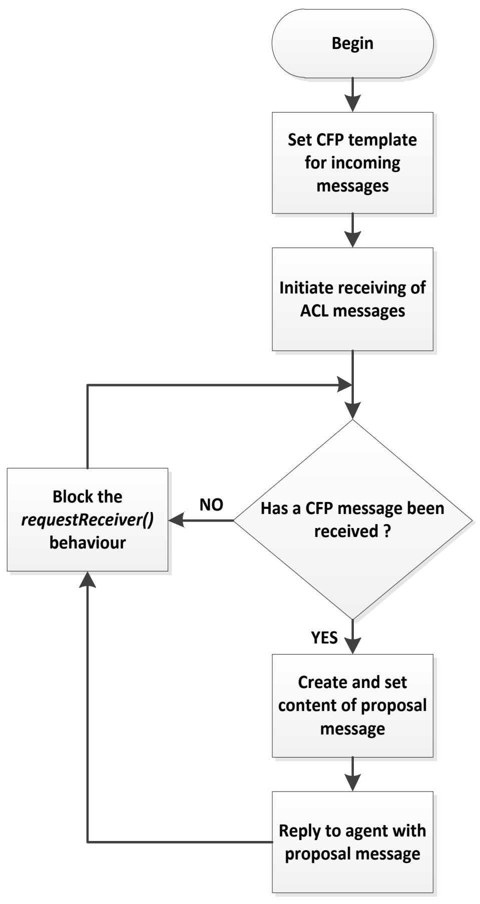
6.2.3 Agent behaviours

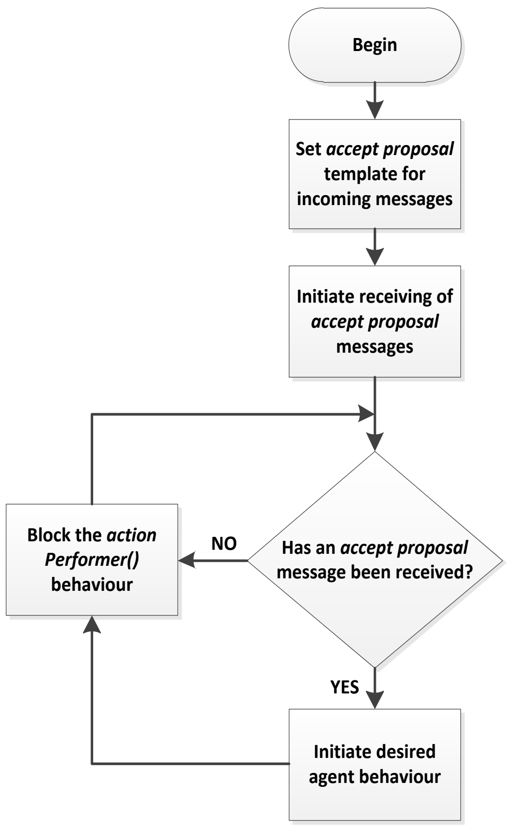
The functionality of JADE agents is constructed in special JADE classes called behaviours. This section describes the methods and behaviours which are implemented in several agents in the MAS.

A *Setup( )* method is performed upon the instantiation of an agent. This method starts by instantiating the ontology and language that will be used in the MAS. This is done by creating an instance of the MAS ontology, and then registering the ontology and the language with the Content Manager of the AMS. The next step is to register the services that the particular agent can provide with the Directory Facilitator. This registration requires the agent ID and agent name, along with the type of service the agent can provide. The final step of the method is to instantiate the behaviours of the agent which are required for its initial, basic operation.

To enable the utilization of the Contract Net Protocol, agents in the MAS must include the following behaviours: *requestReceiver( )*, *actionPerformer( )* and *requestAction( )*. The first two behaviours are used in agents providing a service – their flow diagrams are shown in Figure 28. The *requestAction( )* behaviour allows agents to acquire a desired service – the flow diagram is shown in Figure 29.

The cooperation characteristic of the MAS means that agents may require services of other agents in the system. In such a case, the CNP requires agents to send "Call for Proposal" (CFP) messages to all the agents in the MAS which provide the desired service (this list of agents is obtained from the Directory Facilitator). The *requestReceiver( )* behaviour is thus implemented, by agents which provide a service, to receive these CFP messages. This behaviour first sets the message template to that of CFPs and then awaits the arrival of messages. The received messages will be compared to the CFP template to ensure that they are correct and applicable. If the messages do not match the CFP template, they are discarded. The behaviour is then blocked until a new message arrives. If the message is indeed a CFP, a proposal is constructed. The proposal may contain a certain parameter on which the proposal will be judged - a predicted completion time or a convenience indicator (indicating how easy it would be for the agent to provide the service) are examples of a proposal parameter. The proposal is then sent to the contracting agent for evaluation.


Begin
Set CFP template for incoming messages
Initiate receiving of ACL messages
Has a CFP message been received ?
NO
Block the *requestReceiver()* behaviour
YES
Create and set content of proposal message
Reply to agent with proposal message


Begin
Set *accept proposal* template for incoming messages
Initiate receiving of *accept proposal* messages
Has an *accept proposal* message been received?
NO
Block the *action Performer()* behaviour
YES
Initiate desired agent behaviour

(a) (b)

Figure 28: Flow diagrams of the (a) *requestReceiver( )* and (b) *actionPerformer( )* behaviours.

If the proposal is selected by the contracting agent, the CNP requires confirmation with the sending of an “accept proposal” message. This message is then received by the *actionPerformer( )* behaviour of the contracted agent. This behaviour again sets the message template to that of “accept proposal”, which is compared to the received messages. If an “accept proposal” message is received, the appropriate behaviour of the contracted agent is initiated. In this case, and if the received message does not match the template, the *actionPerformer( )* behaviour is blocked until the arrival of a new message.

Figure 29 shows the behaviour which is exhibited by agents to acquire the services of another agent in the system, according to the CNP. The *requestAction( )* behaviour starts by declaring and initialising the required local variables. The next step involves the sending of CFPS to all the agents in the system which provides the desired service (the list is obtained from the Directory Facilitator). The behaviour is blocked while no proposals (or messages with an incorrect format) have been received. Upon the arrival of the proposal messages, the proposal parameters are evaluated to determine which proposal is the best option. An "accept proposal" message is then sent to the agent which issued the best proposal. The behaviour is then blocked until a reply message from the contracted agent arrives. If the reply is an "inform" message, it indicates that the service was successfully performed and the behaviour ends. If the reply is not an "inform" message, it means that the process was unsuccessful – the behaviour then repeats the CNP steps.

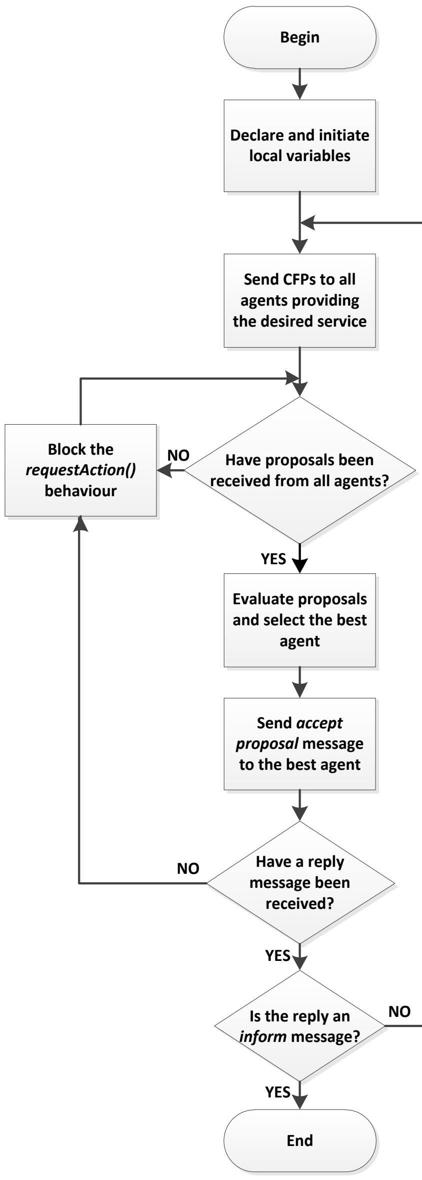

Begin
Declare and initiate local variables
Send CFPs to all agents providing the desired service
Block the *requestAction()* behaviour
NO
Have proposals been received from all agents?
YES
Evaluate proposals and select the best agent
Send *accept proposal* message to the best agent
NO
Have a reply message been received?
YES
Is the reply an *inform* message?
NO
YES
End

Figure 29: Flow diagram of the *requestAction( )* behaviour.

6.2.4 Supervisor agent

The CC would typically receive a production order from a defined production schedule and then coordinate the subsystems to accomplish the specified order. The Supervisor agent receives a command from the CC when actions are required from the feeder subsystem, and replies with a confirmation message upon completion. The behaviours of the Supervisor agent are depicted in Figure 30.

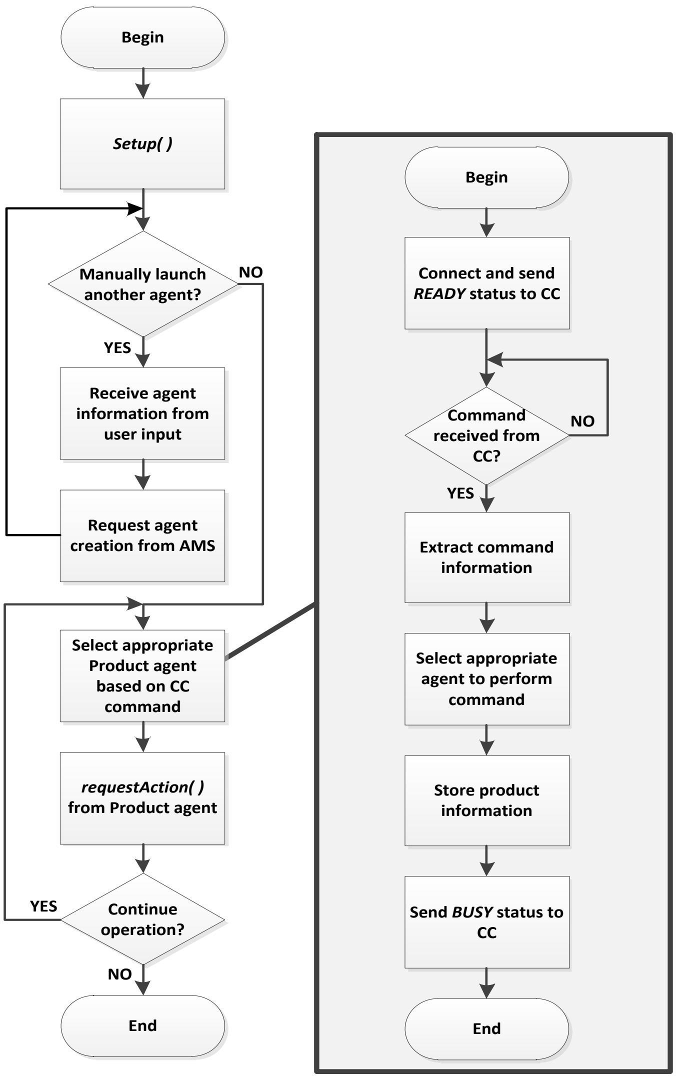

Begin
Setup( )
Manually launch another agent?
NO
YES
Receive agent information from user input
Request agent creation from AMS
Select appropriate Product agent based on CC command
requestAction( ) from Product agent
YES
Continue operation?
NO
End
Begin
Connect and send *READY* status to CC
Command received from CC?
NO
YES
Extract command information
Select appropriate agent to perform command
Store product information
Send *BUSY* status to CC
End

Figure 30: Flow diagram of the Supervisor agent functionality.

The Supervisor agent starts by performing the *Setup( )* method to initialise and register the agent services. The agent then implements the *createAgent( )* behaviour. This behaviour allows the creation of agents, in the agent container, by means of user input. This was done to ease the MAS control when a hardware reconfiguration has taken place. If a new hardware component is added, its HLC agent can be launched by the user (otherwise changes would have to be made to the *Main( )* class of the HLC). The creation of an agent is achieved through sending a request message to the AMS. The Supervisor agent then enter its operational state as it implements the *selectProduct( )* behaviour. This behaviour receives commands from the Cell Controller and selects the appropriate Product agent to perform the desired tasks. This selection is followed by implementing the *requestAction( )* behaviour to acquire the service of the selected Product agent.

The *selectProduct( )* behaviour is also shown in more detail in Figure 30. In the first step of this behaviour, the Supervisor agent connects to the CC via a TCP/IP socket. To indicate the readiness of the MAS, the Supervisor agent sends a *READY* status to the CC – the feeder subsystem is thus ready to receive commands. When a message is received from the CC (in the format of an XML string), it is parsed to extract the relevant information. The XML string will contain the command to be executed, the product type involved with the command and the necessary task sequence and coordinate information. The command and product type is then used to select the appropriate Product agent, while the task and coordinate information is stored for access at a later stage. The Supervisor agent then immediately responds to the CC with a *BUSY* status – this indicates to the CC that the feeder subsystem will not be able to perform any commands until the *READY* status is sent again upon completion.

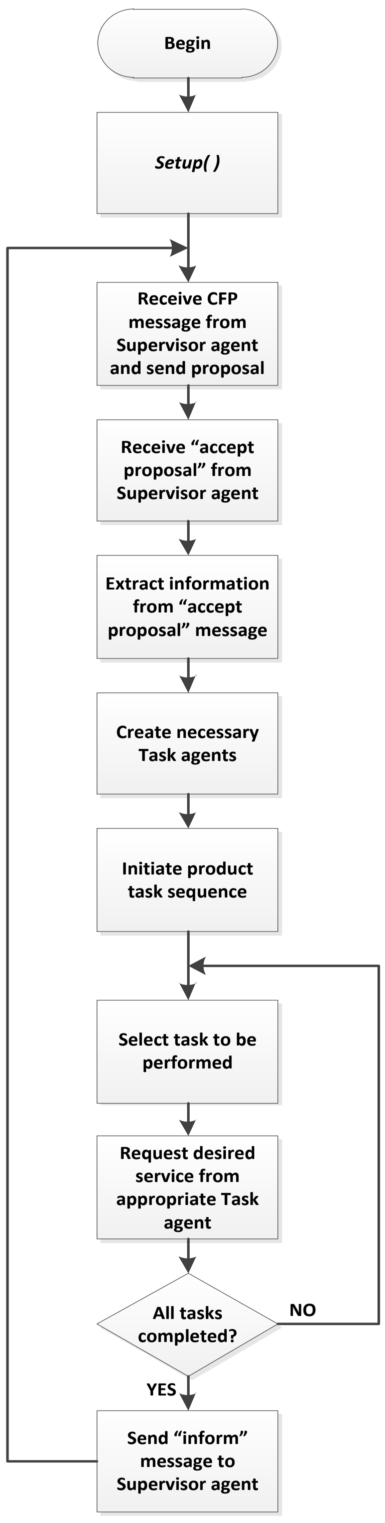
6.2.5 Product agents

When the Supervisor agent receives a command from the CC (e.g. to load the parts of a specific product onto the fixture), it launches the appropriate Product agent. The Product agent then accesses the relevant information concerning the tasks to be performed – such as the task sequence and part and coordinate data. One Product agent could handle all the products of the system (when the product information resides within the CC), but the holonic architecture was designed so that a Product agent can be created for each product type to allow for situations where the product information can reside in the feeder subsystem (e.g. if the future introduction of new products need not be provided for). The functionality of the Product agents is depicted in Figure 31 (a).

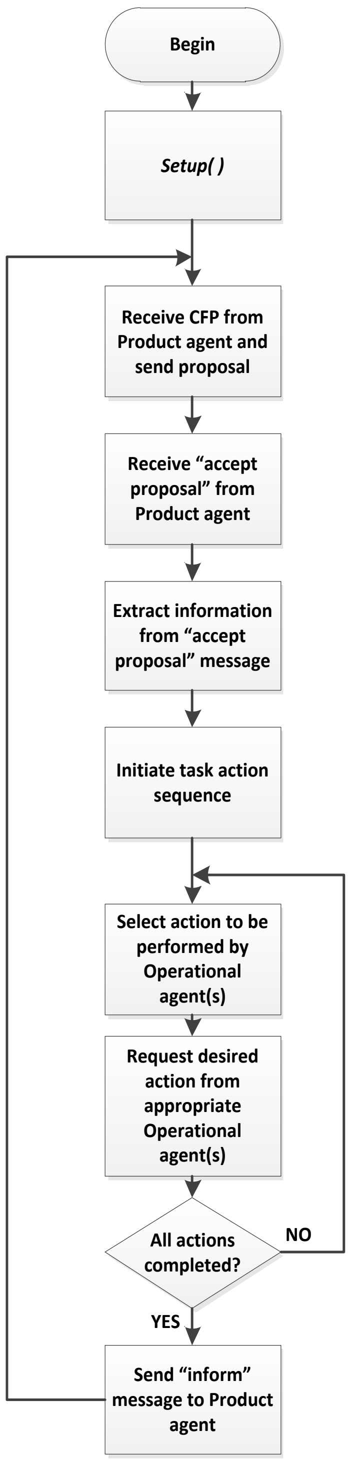
The ontology and language used by the Product agents, as well as the initial behaviours, are instantiated in the *Setup( )* method. The two behaviours are that of *requestReceiver( )* and *actionPerformer( )*, which await the arrival of respectively CFP and "accept proposal" messages from the Supervisor agent. Upon receiving the "accept proposal" message, the Product agent then requests the launching of the necessary Task agents (according to the tasks involved in completing the product) from the AMS. The Product agent then retrieves the task sequence and the relative part and coordinate information. The sequence of tasks is then initiated. The tasks are performed one at a time by acquiring the services of the appropriate Task agents through the *requestAction( )* behaviour. When a task is successfully completed, the Product agent moves on to the next one. A "inform" message is sent to the Supervisor agent when all the tasks of the product have been performed.

6.2.6 Task agents

The necessary Task agents are launched according to the information of the Product agent. The Task agents then drive the required hardware actions. A Task agent exists for every function inherent in the system, e.g. a specific Task agent is responsible for the loading of one of the required parts onto the fixture. A flow diagram of the workings of Task agents is presented in Figure 31 (b).


Begin
Setup( )
Receive CFP message from Supervisor agent and send proposal
Receive "accept proposal" from Supervisor agent
Extract information from "accept proposal" message
Create necessary Task agents
Initiate product task sequence
Select task to be performed
Request desired service from appropriate Task agent
All tasks completed?
NO
YES
Send "inform" message to Supervisor agent

(a)


Begin
Setup( )
Receive CFP from Product agent and send proposal
Receive "accept proposal" from Product agent
Extract information from "accept proposal" message
Initiate task action sequence
Select action to be performed by Operational agent(s)
Request desired action from appropriate Operational agent(s)
All actions completed?
NO
YES
Send "inform" message to Product agent

(b)

Figure 31: Flow diagram of (a) Product and (b) Task agent functionality.

The Task agent starts by running the *Setup( )* method. The ontology and initial behaviours are thus instantiated and the agent services are registered. Next the agent performs the *requestReceiver( )* behaviour, which awaits the arrival of a CFP from a Product agent. With the proposal sent, the agent awaits the arrival of the succeeding "accept proposal" message in the *actionPerformer( )* behaviour. The coordinate information which accompanies the message is extracted. According to the "accept proposal" message, the Task agent starts to perform the necessary subsystem action sequence. The first required action to be performed is achieved by acquiring the service of the appropriate Operational agent, through the *requestAction( )* behaviour. Upon the completion of this action, the next action is selected – this process continues until all the actions concerning the desired task are completed. The agent then returns to the idle state, where it awaits the next CFP message from a Product agent.

6.2.7 Operational agents

The Task agents coordinate the Operational agents to perform the desired hardware functions. The Operational agents send the necessary command, part type and coordinate information to the respective LLC programs. The Operational agents also interact with one another where cooperation is needed to perform a certain hardware function. The Operational agents in the MAS are described in this section.

6.2.7.1 Singulation unit agent

The Singulation unit (SU) agent is responsible for the control of the singulation unit actions. This agent represents an Operational holon which only consists of a software entity. This is because the actuators of the singulation unit are physically controlled by the DAQ device, which is represented by its own HLC and LLC control. The SU agent thus controls the actions of the singulation unit by coordinating the actions of the DAQ and Camera agents. The functionality of the Singulation unit agent is shown in Figure 32.

The agent initializes by performing the *Setup( )* method. With the agent services now registered in the DF, it awaits the arrival of a CFP message from a Task agent. The agent responds with a proposal. If the agent receives the "accept proposal" message, its services is contracted. The agent extracts the necessary command information from the "accept proposal" message and then initiates the required task sequence. The actions to be performed are selected and requested from the appropriate Operational agents by the *requestAction( )* behaviour. These actions are the control of the singulation unit actuators by the DAQ agent or the trigger of inspections by the Camera agent. These actions are then performed in the specified sequence until the operation is completed – at which point an "inform" message is replied to the Task agent.

6.2.7.2 DAQ agent

The DAQ agent controls the actions of the DAQ device by sending commands to the DAQ LLC program. The services of the DAQ agent are acquired by other Operational agents which require the DAQ to perform an action, such as:

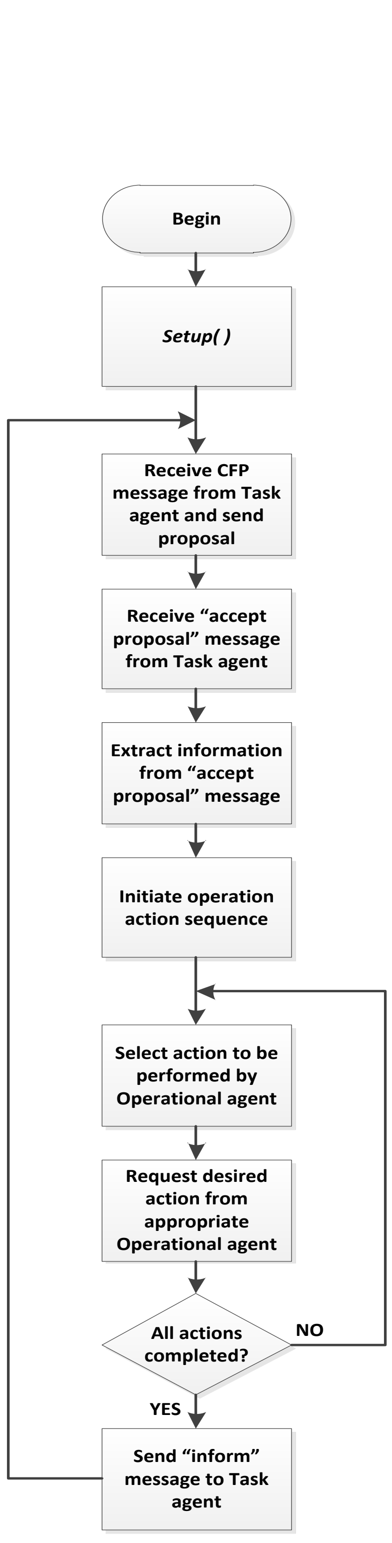
- The SU agent requires the actuation of the singulation unit components.
- The Robot agent requires the DAQ to actuate the gripper during picking and placing of parts. The Robot agent also commands the DAQ agent to lower the presentation platform after the part has been picked up.

The DAQ agent functionality is presented in Figure 32. The agent again starts with the initializing *Setup( )* method and awaits a CFP message from one of the Operational agents. The agent sends a proposal and, if the proposal is selected, receives an “accept proposal” message. The content of the “accept proposal” message is extracted to determine which action should be performed by the DAQ. The command is then constructed in the form of an XML string. This string is sent to the LLC program through a TCP/IP socket. The agent then awaits the completion message from the LLC program, which is also in the XML format. This string is parsed to extract the result of the operation. If the action was successful, an “inform” message is sent to the respective Operational agent. If not, a “failure” message is replied.

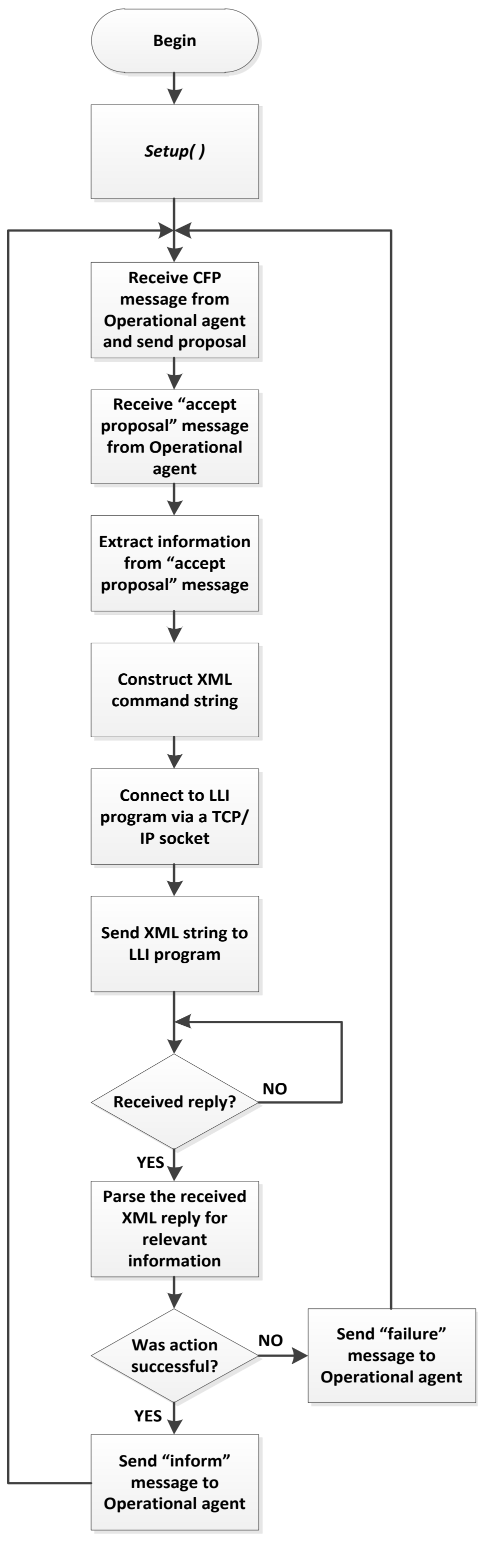
6.2.7.3 Camera agent

The Camera agent is responsible for controlling the inspections of the camera mounted on the singulation unit. The agent sends the command information to the camera LLC program in XML string format. The LLC program returns the inspection result, along with the coordinate information, in an XML string.

The functionality of the Camera agent is similar to that of the DAQ agent, as is presented in Figure 32. The Camera agent provides the inspection service, which is required by the Task agent. The Task agents thus send CFP messages to the Camera agent. The “accept proposal” message, which is sent by the Task agent, contains an ontological reference. The INSPECT action (explained in section 6.2.2.4), along with its information, is included in the message content. This INSPECT information is extracted from the content. The part type information is then included in the XML command string which is sent to the camera LLC program. The reply message, from the LLC program, contains the camera inspection result. In the case of a successful inspection, the pickup coordinates of the presented part is also included in the message. The coordinate information is extracted by parsing the incoming XML string, and is then stored to the slots of the POSITION concept of the ontology. This POSITION concept is then set as the content for the “inform” message which is sent to the Task agent. When the inspection is unsuccessful, a ‘failure” message is sent to the Task agent.


Begin
Setup( )
Receive CFP message from Task agent and send proposal
Receive "accept proposal" message from Task agent
Extract information from "accept proposal" message
Initiate operation action sequence
Select action to be performed by Operational agent
Request desired action from appropriate Operational agent
All actions completed?
NO
YES
Send "inform" message to Task agent

(a)


Begin
Setup( )
Receive CFP message from Operational agent and send proposal
Receive "accept proposal" message from Operational agent
Extract information from "accept proposal" message
Construct XML command string
Connect to LLI program via a TCP/IP socket
Send XML string to LLI program
Received reply?
NO
YES
Parse the received XML reply for relevant information
Was action successful?
NO
Send "failure" message to Operational agent
YES
Send "inform" message to Operational agent

(b)

Figure 32: Flow diagram of (a) Singulation unit and (b) DAQ agent functionality.

6.2.7.4 Robot agent

The Robot agent is implemented as the HLC for the robot holon of the subsystem. The agent controls the actions of the robot based on communication with the other agents in the MAS. The commands are constructed into XML strings and passed on to the robot LLC program, which communicates with the robot controller through RS232 serial communication.

The functionality of the Robot agent is depicted in Figure 33. The initial working of the Robot agent is similar to that of the DAQ and Camera agents. The initialization is done by the *Setup( )* method, and the *receiveRequest( )* and *actionPerformer( )* methods are added to handle the arrival of CFP and "accept proposal" messages. The "accept proposal" message contains the ontological action PICKNPLACE (described in section 6.2.2.4). This action contains the critical information regarding the part to be picked up and placed, as well as the coordinates involved with both operations. This information is extracted and included in the XML command string which is sent to the robot LLC program.

Since the robot itself does not control the actions of the gripper, the Robot agent must acquire the services of the DAQ agent during the pick-'n-place operations. When the robot reaches a point in the operation where the gripper must close (when picking up) or open (when placing), the controller program sends a message via RS232 to the LLC program, which passes it on to the Robot agent. The XML string which is received from the LLC program is parsed to determine if the robot action is complete or if a DAQ action is required. When a DAQ action is required, the Robot agent requests the services through the *requestAction( )* behaviour. The DAQ agent replies with a confirmation message once the DAQ action has been performed. The Robot agent then sends a "continue" message to the LLC program. When the pick-'n-place task is completed, the Robot agent sends an "inform" message to the Task agent.

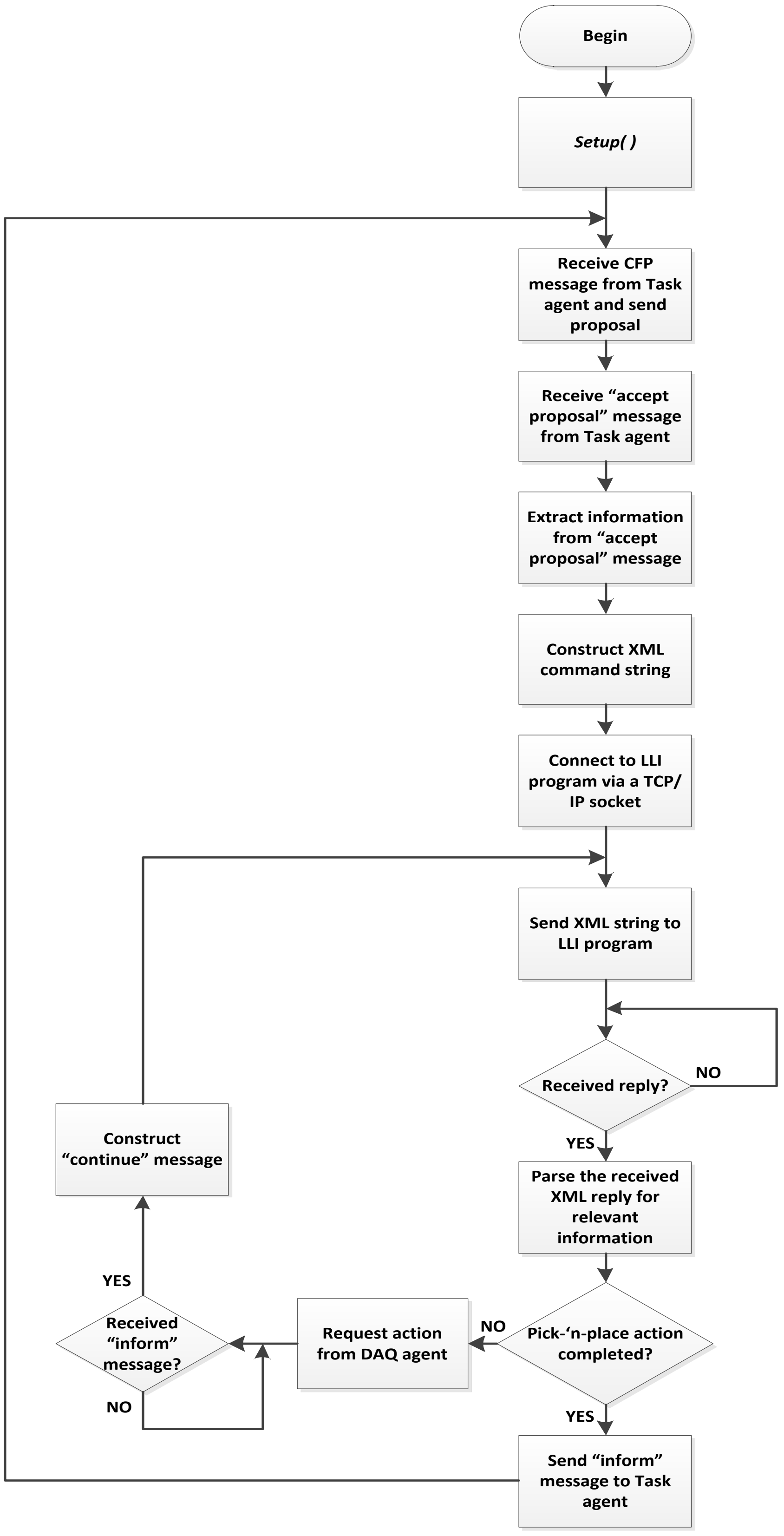

Begin
Setup( )
Receive CFP message from Task agent and send proposal
Receive "accept proposal" message from Task agent
Extract information from "accept proposal" message
Construct XML command string
Connect to LLI program via a TCP/IP socket
Send XML string to LLI program
Received reply?
NO
YES
Parse the received XML reply for relevant information
Pick-'n-place action completed?
NO
YES
Request action from DAQ agent
Received "inform" message?
YES
NO
Construct "continue" message
Send "inform" message to Task agent

Figure 33: Flow diagram of Robot agent functionality.

6.3 IEC 61499 function block control

The function block control was implemented on the FBDK (Function Block Development Kit) platform. FBDK is a prototype engineering software tool for IEC 61499 software development. FBDK provides an integrated development environment that supports the development of function blocks and systems, and their translation to Java classes. The Java classes are then executed using a Java Virtual Machine on the PC (Vyatkin, 2007).

6.3.1 Control system overview

The structure of the function block control is based on a distributed holonic approach. In FBDK, the holons of the subsystem are mapped to devices. A device can be understood as an abstract model that captures the information-processing properties of control devices. These devices are then hosts to resources, which contain the function block networks. FBDK also facilitates composite function blocks – these are function blocks which contain their own function block networks (Vyatkin, 2007).

The function block networks are where the control system is implemented. The subsystem devices are then as follows: FB_SUPERVISOR, COMMAND_EXECUTION, SINGULATION_UNIT, DAQ, CAMERA and ROBOT. The FB_SUPERVISOR, SINGULATION_UNIT, DAQ, CAMERA and ROBOT devices all contain one resource, which is given the same name as the device. The COMMAND_EXECUTE device, representing the Product holon, contains several resources: COMMAND_SELECT and a resource for each system product. The Task holon is not explicitly defined by a device or resource, as it is represented by the various function block network event and data connections, along with the intra-device communication function blocks. The structure of the control system is depicted in Figure 34. The respective device function block networks are given in Appendix G.

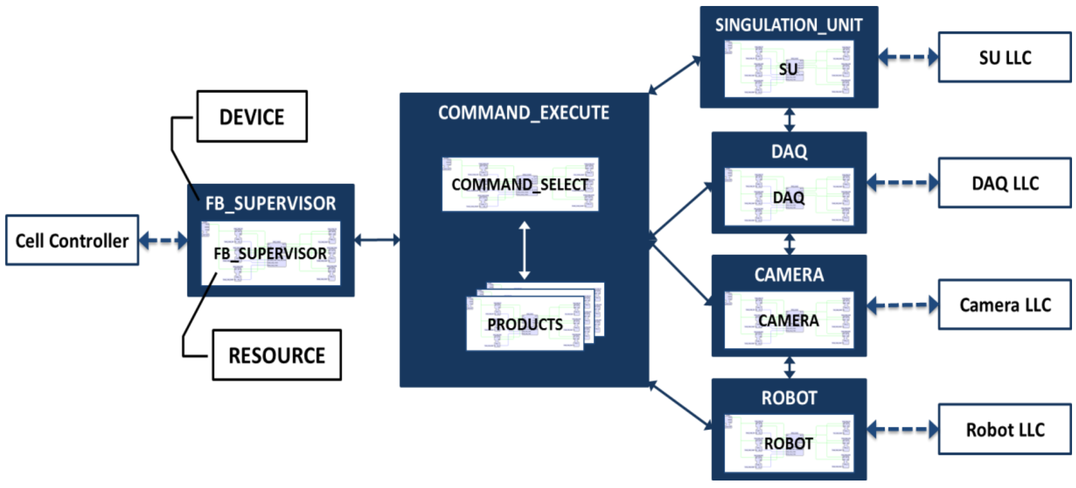


Figure 34: Structure of the IEC 61499 function block control system.

6.3.2 Function block communication and coordination

6.3.2.1 System communication

The communication between function blocks (and function block networks) comprises of two parts: the transfer of an event and the transfer of the accompanying event data. This communication, for function blocks residing in the same network, is done by event and data connections. The output event and data variables are connected to input variables by visual lines in the FBDK graphic user interface (GUI). The event connections are always connected to the top part of the function block shape and is indicated as green lines in the GUI. The data variable connections are connected to the bottom half of the function block shape and are indicated as blue lines. The data variables can be of various types – FBDK facilitates the standard types (STRING, WSTRING, INT, BOOL, etc.), as well as arrays and customized data structures.

When the communication occurs between the function blocks of different networks (contained in different resources or devices), PUBLISH and SUBSCRIBE function blocks are used (shown in Figure 35). The information that is to be sent is connected to a PUBLISH function block. When the input event of the function block is triggered, it sends the event and data to a specified SUBSCRIBE function block. The location to where the information must be sent is specified by using function block IDs. A unique ID is given to a SUBSCRIBE function block – this ID is then used by the PUBLISH function block. This use of IDs enables one PUBLISH function block to send information to different SUBSCRIBE function blocks, as the ID can be sent to the PUBLISH function block as a variable.

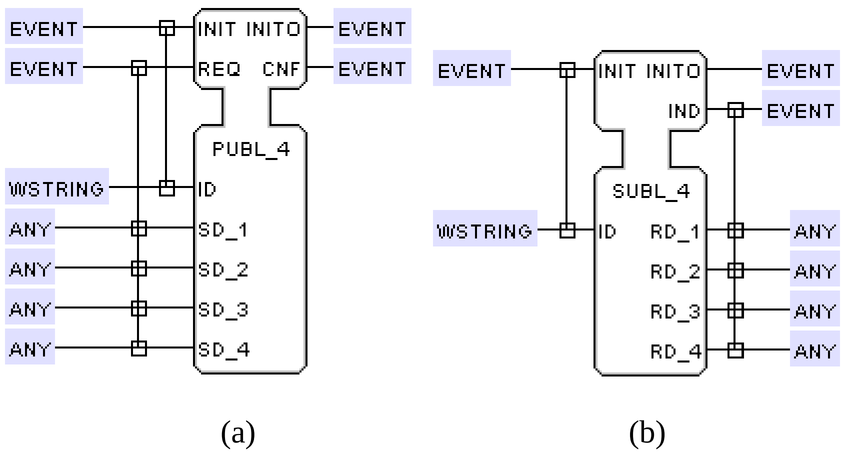


Figure 35: (a) PUBLISH and (b) SUBSCRIBE function blocks.

6.3.2.2 Communication with CC and LLC

The communication between the HLC and the CC and LLC is based on XML strings, sent through TCP/IP sockets. The function block control system thus requires function blocks for the building and parsing of XML string. A network segment showing the XML_BUILDER, COMMUNICATOR and XML_PARSER function blocks are shown in Figure 36. These function blocks use the Java functions (residing in imported packages) for building and parsing XML strings and communicating over TCP/IP sockets. The XML_BUILDER function block receives the command information through data connections. The functions of the algorithm then construct an XML string, which is passed on through an output data connection to the COMMUNICATOR function block. The COMMUNICATOR function block algorithm sends the received XML string to the LLC program through the TCP/IP socket. The predefined port number used for the communication is supplied to the function block as a constant. The algorithm then continuously monitors the socket for the arrival of a message from the LLC program. The LLC program replies with an XML string – this string is simply passed on to the XML_PARSER function block. The XML_PARSER function block algorithm parses the XML string for the relevant information. This information is stored to the respective output variables, which is emitted to the succeeding function blocks through data connections.

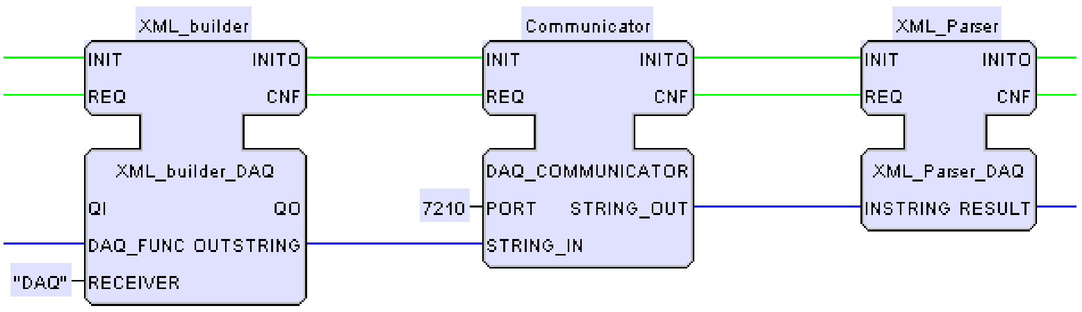


Figure 36: Function block network segment for XML communication.

6.3.3 FB_SUPERVISOR device

The FB_SUPERVISOR device contains a function block network which handles communication with the CC. The network of function blocks send the subsystem status to the CC and receive the command and product information. The received information is passed on to the COMMAND_EXECUTION device. The function block network is shown in Figure G 1.

The function block network instantiates a FB_SPVR_CONTROL composite function block, which contains the functionality of the device – the function block network is presented in Figure G 2. This composite function block is interfaced with the COMMAND_EXECUTION device by a PUBLISH function block, through which all the command and product information is communicated.

The FB_SPVR_CONTROL composite function block network implements the XML communication function blocks of section 6.3.2.2. These function blocks allow for communication with the CC program. When a XML command string (as explained in section 6.1) is received, the information must be extracted to data variables. The coordinate information must be stored in arrays, which is passed on the rest of the system. The STORE_TO_ARRAY function block stores the information to the arrays one element at a time. The output event variable of this function block is connected to the XML_PARSER function block input event variable. This causes the iteration of the parsing function block until all the data is stored in the arrays. These variables are then passed to the other devices when the output event is triggered.

6.3.4 COMMAND_EXECUTION device

The COMMAND_EXECUTION device receives the data from the FB_SUPERVISOR function block. The device holds a resource for each system product and a resource for selecting the specified PRODUCT resource. The function block networks of the COMMAND_SELECT and LOAD_1 resources are given in Figure G 3 and Figure G 4.

The command and product information is received from the FB_SUPERVISOR device. The information is received in the COMMAND_SELECT resource, which triggers the production of the desired product through an output event trigger to the appropriate PRODUCT resource (such as the LOAD_1 resource). The appropriate resource to be triggered is determined through an *if* statement in the algorithm of the COMMAND_SELECT function block. The algorithm compares the value of the product input data variable to predefined conditions. If the variable matches the condition, the respective resource is triggered. The product information is sent to the triggered PRODUCT resource via the INTERFACE function block. The INTERFACE function block merely passes the input information on as output information – this is needed because the output variables of a SUBSCRIBE function block cannot be directly connected to the input variables of a PUBLISH function block.

The functionality of the PRODUCT resource resides in the PRODUCT_CONTROL (labelled LOAD1_CONTROL in Figure G 4) function block. The function block triggers the required devices, according to the product information task sequence, by means of a *switch* statement in its algorithm. The *switch* statement compares the PART information to predefined conditions, which determine the device which must be triggered. The elements of the PART input array are used one at a time to trigger the desired product events. When all the tasks have been performed, a “completion” event is published to the COMMAND_SELECT resource.

6.3.5 SINGULATION_UNIT device

The SINGULATION_UNIT device contains the function block network for coordinating the actions of the singulation unit. The function block network is shown in Figure G 5.

The functionality of the device is contained in the SU_CONTROL function block. This function block controls the actions of the DAQ and CAMERA devices in the desired sequence by triggering the relevant output events. The decision making logic is contained in two function block algorithms – one for each of the input events. The algorithms trigger the output event variables.

The SINGULATION_UNIT device receives the input event indicating that a part is to be loaded. The SU_CONTROL function block starts the loading process by triggering an output event to the DAQ device. The DAQ device replies through the SUBSCRIBE function block. The output event is then triggered to start the camera detection. When a part is detected by the camera, the CAMERA device triggers an output event directed at the SU_CONTROL function block. The function block then activates the DAQ device to stop the conveyor motor and the CAMERA device to perform an inspection.

6.3.6 DAQ device

The DAQ device function block network controls the actions of the physical DAQ device. The network is shown in Figure G 6.

The functionality of the DAQ device resides in the DAQ_CONTROL composite function block, of which the network is shown in Figure G 7. The function block network of the DAQ_CONTROL function block contains the XML communication function blocks to communicate with the DAQ LLC program. The DAQ_CONTROL_IN and DAQ_CONTROL_OUT function blocks are responsible for the triggering of the correct output event variable.

The DAQ device receives commands through the input event variables from the connected devices. The DAQ_CONTROL function block compiles the received data into an XML string, sends it to the DAQ LLC program and awaits a reply. The reply from the DAQ LLC program, indicating completion, is relayed to the relevant system devices.

6.3.7 CAMERA device

The CAMERA device controls the functions of the Camera holon. The function block network of the device is shown in Figure G 8.

The communication function blocks of the CAMERA device are connected to a CAM_CONTROL composite function block, shown in Figure G 9. This composite function block contains the XML communication function blocks to achieve communication with the Camera LLC program. A CAM_CONTROL_OUT function block is also contained in the network. This function block is responsible for triggering the appropriate output event and data variables, according to the inspection tasks that the camera performed.

The CAMERA device is only activated through a command (event) from the SINGULATION_UNIT device. This event is accompanied by two data input variables – one indicating the inspection product to be triggered and the other specifying whether the camera should inspect or detect the parts. This information is compiled into an XML string and sent to the Camera LLC program. The inspection result string is received and parsed, and the coordinate information is stored to the various data output variables. The coordinates for the pick-‘n-place operation is sent to the ROBOT device.

6.3.8 ROBOT device

The ROBOT device embodies the HLC of the pick-‘n-place robot. Figure G 10 shows the function block network embedded in the device.

The functionality of the device is held within the ROBOT_CONTROL composite function block. The network residing in the ROBOT_CONTROL function block (shown in Figure G 11) employs the XML communication function blocks, for communication with the Robot LLC program, and also a ROBOT_CONTROL_OUT function block. The ROBOT_CONTROL_OUT function block triggers the appropriate event and data variables, according to the tasks performed by the robot.

The ROBOT device receives command events from the CAMERA device (if the part is to be picked up from the singulation unit) or the COMMAND_EXECUTION (if the part is to be picked up from a part magazine). These command events are accompanied by data input variables, which contain the coordinate information relevant to the task. The ROBOT_CONTROL function block compiles the XML string and sends it to the Robot LLC program. The LLC program replies with “open gripper” or “close gripper” messages during the operation. These messages cause the trigger of outputs events, which is published to the DAQ device. The DAQ device indicates the completion of the action by sending an event to the SUBSCRIBE function block of the ROBOT device. These events indicate that the pick-‘n-place activity can continue. When the operation is complete, the ROBOT device publishes the event to the COMMAND_EXECUTE device.

7. System reconfigurability assessment

This section evaluates the reconfigurability of the feeder subsystem at two levels – the reconfigurability of the HLC system and that of the low level subsystem software and hardware. The reconfigurability assessment is done by means of four reconfiguration experiments. The implications of the reconfiguration on both HLC strategies, as well as on the low level software and hardware, are described. The reconfigurability of the control strategies is compared by means of quantitative and qualitative measurements.

7.1 Experiment 1: Change in the task sequence

The first experiment involved the changing of the sequence in which tasks are performed to load a specified product onto a fixture. The sequence of tasks was changed in the CC program and was included in the product information sent to the HLC programs (as described in section 6.1). This experiment entails no changes to the low level software and hardware of the feeder subsystem.

7.1.1 MAS reconfiguration

The MAS receives the command and product information, sent from the CC program, via the Supervisor agent. The agent parses the XML string and stores the extracted product information in the element of a static array. This array is globally visible and accessible, granting all of the agents of the MAS access to the information.

The task information is stored in the sequence that they are to be performed. The Product agent then launches the required Task agents, and contracts their services, according to the sequence of the product information array. The coordinate information is also obtained from the array and stored to the PLACE_POSITION ontology concept. This concept is passed on to the Operational agents when their services are acquired.

The MAS HLC programs are thus not influenced by a change in product information – the changes can be made to the CC program without having to stop or restart the feeder subsystem.

7.1.2 Function block reconfiguration

The command XML string from the CC program is received by the SUPERVISOR device of the IEC 61499 function block control system. The string is parsed and the information is stored to data array variables. These arrays are sent to the COMMAND_EXECUTE device. The product information is sent to the selected PRODUCT resource.

The PRODUCT_CONTROL function block receives the product information arrays as input data variables. The array containing the parts to be loaded, in the correct sequence, is then used to determine which tasks should be performed. The tasks are then performed through the triggering of the PRODUCT_CONTROL function block output event variables. For each task, the respective coordinate information is extracted from the arrays and stored to individual coordinate data variable – these are passed on to the Operational devices.

The function block control system is thus also uninfluenced by any changes in the product information – the changes can also be implemented with the feeder subsystem remaining online.

7.2 Experiment 2: Addition of a new task

This reconfiguration experiment involved the addition of a Task holon to the feeder subsystem. The situation required an additional task to be performed in the loading of the sub-assembly. The added task is included in the product information contained in the CC program, which is passed on to the HLC programs. This new task did not entail the addition of new subsystem hardware.

In the event of reconfiguring the subsystem for an entirely new product, the addition of tasks for the new parts will be required. For this experiment, the additional task was the placement of a new part, the moving contact (which was previously not included), on the fixture. The moving contact parts were placed in a part magazine, from where the robot had to pick up the parts and place them in the fixture.

7.2.1 MAS reconfiguration

This reconfiguration entailed the addition of a new Task agent to the MAS. The Task agent contained the information for the necessary actions to perform the task, such as communicating with the relevant Operational agents and Product agents, and handling the part and coordinate information.

The new Task agent was created offline, using the same template as that of the other MAS Task agents. The sequence of Operational agent actions was defined in the behaviours of the Task agent. The use of the ontology (and potential additions to it) was also considered in the development of the Task agent.

The addition of a Task agent had to be recognised and utilized by the involved Product agents. The Product agents launch the Task agents which perform the desired services. The services are then acquired by searching the Directory Facilitator (DF). The Task agents are named according to their involved parts (such as “feedTask_1 Agent”), so they can be launched directly from the software package by the Product agent. The Product agent extracts the part information from the product information array and uses it to construct the names of the Task agents, as follows:

```
CreateAgent ca = new CreateAgent();
ca.setAgentName("feedTask_"+ part +"_Agent");
ca.setClassName("feedTask_"+ part +"_Agent");
```

The part information is then contained in the “part” string variable. This constructed name is then used to launch the Task agent by sending a request to the AMS. When the Task agents are launched, their services are acquired in a similar way through the DF.

The Task agent can thus be added to the MAS without having to stop or restart the system. The agent is created offline and then added to the JADE agent package – it is then launched and utilized by the Product agent automatically.

7.2.2 Function block reconfiguration

The addition of a Task holon means that the IEC 61499 function block control system requires the alteration of the COMMAND_EXECUTION device. The reconfiguration affects the relevant PRODUCT resource, since the Task holon is not explicitly embodied in the control system. The PRODUCT resource extracts the task information from the data array input variables and launches the execution of the tasks through output event variables.

The reconfiguration required the alteration of the algorithm of the PRODUCT_CONTROL function block, which resides in the PRODUCT resources of the COMMAND_EXECUTION device. The *if* statement of the algorithm was extended to facilitate the added task. When the task is to be performed, the appropriate SUBSCRIBE function block address is sent, along with the event trigger, to the PUBLISH function block. The event trigger is then sent to the desired device.

The alteration to the algorithm could not be done online. The feeder subsystem was stopped to perform the alteration and then restarted.

7.2.3 Low level software and hardware reconfiguration

The loading of new sub-assembly parts requires reconfiguration of the subsystem software and hardware. The necessary changes for each subsystem component are discussed in the following paragraphs.

The new parts may be placed in part magazines manually and presented to the robot, in which case a new part-specific part magazine must be designed and manufactured. Alternatively, it may be desired that the new part be singulated by an existing singulation unit – this singulation unit may then require some changes to enable effective part singulation. For the case of the stepped-conveyor singulation unit, the following changes may be necessary:

- Changing of the singulation unit’s conveyor belt. The scoops which are attached to the belt are designed to be part-size specific. A belt with appropriate scoops must be installed – this may require the design and manufacture of new scoops, which must be attached to a new belt.
- Adjusting the pulley positions. This may be required to ensure that the scoops perform effective singulation during their motion through the input bin.
- Adjusting the speed of the conveyor motor. The dropping of the parts from the scoops, through the gateway actuator, is also dependent on the properties of the part (mass, size and shape). The motor speed may require some tuning to ensure that the parts drop into the gateway actuator.

A new part to be singulated requires the setup of a new camera inspection product. The inspection product must be able to identify the part and return its pickup coordinates. This can be done by taking images of the part on the platform with the camera and using an emulator to set up the inspection product offline. The images can be used to generate models which must be taught to the object locating softsensors. These models are also specific with regards to the relative pickup position of the located shape. The new inspection product must then be added to the flash memory of the camera.

If the part is presented in a new part magazine, the new workspace must be calibrated by the robot – the *base* calibration procedure is explained in Appendix E. The correct pickup coordinate of the part from the magazine must then be entered into the robot LLC program. The robot may also be required to pick up the part from the singulation unit and place the new part in the fixture. This new pick-‘n-place activity requires the development of new KRL programs, which entail the following:

- Setting up appropriate motion paths to allow for effective picking and placing.
- Calculating the correct approach position and motion for both the picking and placing actions.
- Using the received coordinate data in performing the actions.

The size or shape of the part might also require the installation of a new gripper and/or gripper fingers – this addition of hardware is discussed in section 7.4.

7.3 Experiment 3: Addition of a new product

The addition of a Product holon to the HLC systems was required with this experiment. This holon represents a new product to be loaded by the feeder subsystem. Due to restrictions in time and hardware, a completely new product (with new parts) could not be implemented – instead, a new combination of the case study parts was used to simulate a new product sub-assembly.

The new product consisted of four of the case study parts - the load terminal, handle frame assembly and the long and short pigtails. The parts were to be picked up from the part magazines and placed in the fixture. The order of the parts was also specified in the product information sent by the CC.

7.3.1 MAS reconfiguration

A new Product agent was added to the MAS for this experiment. The Product agent had to be added to the JADE agent container and be able to provide the service to the Supervisor agent. This Product agent had to be responsible for the loading of the individual parts of the new product onto the fixture. The agent had to create the necessary Task agents and acquire their service to accomplish the loading of the product. The creation of the Task agents, along with their respective coordinate information, had to be done using the product information array.

The Product agent was developed offline, using a similar template to that of the existing Product agents. The necessary functionality for Product agent was implemented in the agent behaviours – such as the registration with the DF, the extraction of product information from the global array and the sequential execution of the product tasks. The agent accesses the product information array for the task information. This information is used to create the necessary Task agents which are involved in the loading of the product. The Task agents are created and their services are contracted, through the DF, in the sequence specified in the product information array.

The Product agent can be launched to the agent platform manually while the system remains online. This is done by using the "start new agent" function of the JADE GUI. The user provides the name of the Product agent (such as "Product_2_Agent") for which the function then searches in the agent package. The agent is then launched to the JADE agent container when found. The addition of a Product holon to the MAS can thus be achieved without disturbing the operation of the control system.

7.3.2 Function block reconfiguration

A new product resource had to be added to the COMMAND_EXECUTION device of the function block control system. The function block network of this resource had to also retrieve the relevant part and coordinate information from the data table and incorporate all the necessary communication channels to accomplish the loading of the product. This added resource had to contain the necessary functionality to initiate the tasks in the right sequence, by triggering the appropriate function block networks.

The development of the new PRODUCT resource was done offline. The resource function block network contains SUBSCRIBE and PUBLISH function blocks, and one PRODUCT_CONTROL function block. The resource subscribes to "command" information from the COMMAND_SELECT resource and "completion" information from the ROBOT device. The PRODUCT resource publishes event and data information to the ROBOT device (to trigger the task execution) and to the COMMAND_SELECT resource (to indicate product completion). The PRODUCT_CONTROL function block has the functionality to extract the part and coordinate information, along with the task sequence, from the input array data variables. This function block is also responsible for the execution of the tasks by setting the respective output events.

The functionality of the new PRODUCT_CONTROL function block can be tested individually (without being added to the control system) through the built-in FBDK testing interface. The output of the function block can be checked by manually triggering the respective input event variables with defined input data variables. This testing gives some assurance of the function block functionality before it is added to the system.

The new resource could not be added to the control system while it is operational. The control system was stopped while the resource was manually added to the

COMMAND_EXECUTION device. Some changes were also made to the COMMAND_SELECT device. The changes were made to the algorithm of the COMM_SEL function block. The algorithm is responsible for triggering the output event to the correct PRODUCT resource according to the product information received from the FB_SUPERVISOR device. The correct resource is triggered by publishing the event to the correct SUBSCRIBE function block. The algorithm implements a *switch* function to determine which WSTRING address (i.e. the ID of the SUBSCRIBE function block) must be sent to the PUBLISH function block. The address of the SUBSCRIBE function block, of the new resource, must thus be entered into the *switch* function of the COMM_SEL algorithm.

7.3.3 Low level software and hardware reconfiguration

When a new product is introduced, the procedures discussed in section 7.2.3 must be performed for each new part to be loaded by the feeder subsystem. No further low level software and hardware reconfiguration is otherwise needed for the introduction of a new product.

7.4 Experiment 4: Addition of new hardware

In this experiment an Operational holon is added to the feeder subsystem. This addition was achieved by adding a simulated singulation unit to the subsystem. The experiment could only be performed through simulation due to a shortage of functional singulation units. The singulation unit was simulated using a LLC program – the program created a user interface allowing the user to simulate the actions of the singulation unit. The added singulation unit must be controlled and utilised by the HLC programs.

The added singulation unit was chosen to be different, regarding its hardware control, to that of the existing stepped-conveyor singulation unit. The simulated singulation unit would be equipped with a local controller (such as a PLC), which controls all the actuators and the installed camera. This approach was chosen to allow for the addition of only one Operational holon, as opposed to the several Operational holons involved with the stepped-conveyor singulation unit concept.

7.4.1 MAS reconfiguration

The addition of a holon to the subsystem means that a new agent must be added to the MAS. The new Singulation unit agent had to exhibit the functionality of registering its services with the DF, receiving requests from Task agents and communicating with the LLC program.

The new agent was developed offline. The template of the existing Singulation unit agent was used, though the functionality concerning the coordination of the other Operational agents was not required. The required functionality was embedded in the behaviours of the agent. The agent sends command strings to the LLC to singulate a part. The LLC sends a reply to the agent when the part is successfully singulated.

The agent was once again added to the JADE agent platform while the feeder subsystem was online. If the new singulation unit is located in a new position, the position of the presentation platform must be calibrated by the robot. When the platform of the singulation unit is located in a previously calibrated position (as was the case for this experiment), the singulation unit can be added to the subsystem without disturbing the operation. The functionality of the singulation unit can then be seamlessly added to the production activities.

7.4.2 Function block reconfiguration

A new SINGULATION_UNIT device was added to the IEC 61499 function block control system. The device had to contain a function block network with the appropriate communication (PUBLISH and SUBSCRIBE function blocks and XML function blocks) and decision-making (embedded algorithms) functionality.

The function block network of the device was developed offline. The network subscribes to command information from the relevant PRODUCT resource of the COMMAND_EXECUTION device – the completion message after a successful singulation is then published to the same network. The SU_CONTROL function block, to which the SUBSCRIBE and PUBLISH function blocks are connected, contains the functionality to communicate with the LLC program (as described in section 6.3.2.2).

The SU_CONTROL function block was again tested individually to ensure that communication with the LLC program could be successfully achieved. The new device could again not be included to the control system while the subsystem was operational. A new device had to be created in the system when offline, to which the constructed network was imported. A change to the PRODUCT_CONTROL function block of the PRODUCT resource was also required – the address of the new SUBSCRIBE function block had to be added to the algorithm. The control system could then be restarted.

7.4.3 Low level reconfiguration

Additional or new hardware components may be installed in the feeder subsystem if a change in system capability is required. Apart from the reconfiguration implications to the HLC system, some low level reconfiguration actions also have to be performed.

The introduction of any new hardware component to the feeder subsystem will require the development of a LLC program to interface the hardware with the HLC programs and control the hardware’s actions.

When a new singulation unit or part magazine is added to the system, the position of the presentation platform or the magazine must be calibrated for the robot. This is done through the *base* calibration procedure explained in Appendix E. With this *base* calibrated, the robot is enabled to pick up parts from the added hardware.

The addition of a new gripper for the pick-‘n-place robot also requires calibration for the robot. The gripper tool calibration is done through the tool calibration procedure described in Appendix E. This calibration allows the robot to monitor its position according to the Tool Centre Point (TCP) of the gripper.

Any addition or relocation of hardware within the working envelope of the robot requires some recalibration of the robot operation. The calibration of software boundaries must be performed around each of the hardware components located within the reach of the robot. The software boundaries ensure that the robot TCP will never enter the specified space – this provides protection for the robot and the subsystem hardware.

7.5 Discussion of experimental results and observations

The performance of the control strategies during the reconfiguration experiments was compared for all of the reconfiguration experiments. The comparison was done through both quantitative and qualitative measurements.

7.5.1 Quantitative measurements

The quantitative measurements comprise of two sets of recorded times – that of development time and reconfiguration time. The development time refers to the time it took to develop the individual software for each experiment. The reconfiguration time then indicates the offline time (time for which operation was halted) required to introduce the software to the control system. The recorded times for each experiment are shown in Figure 37 and Figure 38. For the purpose of comparison, the respective reconfiguration and development times are added to give the total implementation time, which is presented in Figure 39.

The times shown in Figure 37 indicate the times required for the offline software development needed for each experiment. The figure shows that the development time increases with increasing software complexity. Both control strategies allow for the effective re-use of software components – this greatly shortens the required development time. For the MAS, it is evident that the added Task agent was more complex than the added Product agent. The complexity is due to the various actions and communications that have to be facilitated with the involved Operational agents. The development of the new Singulation unit agent took the most time, as it required some behaviour which was not included in the existing Singulation unit. The setup of the communication with the LLC program was also quite time consuming. As for the function block system, the increasing complexity resided with the creation of composite function blocks, which contain their own function block networks. The correct connection of event and data variables also takes up some development time.

The reconfiguration times, for the respective control strategies, for each experiment are shown in Figure 38. The fact that all the reconfigurations for the MAS could be implemented with the system online means that no reconfiguration time is required. For the function blocks, the feeder subsystem had to be stopped to implement the changes involved from experiment 2 onwards. The increasing complexity of the implementation of the changes is evident from the increasing reconfiguration times. This is because apart from the addition of the new software entity, changes to other devices are required to incorporate the new entity into the system.

It is evident from Figure 39 that, except for adding a task to the control system, the MAS requires less time to achieve reconfiguration. This result is a confirmation to the advantages that MAS exhibit towards reconfiguration.

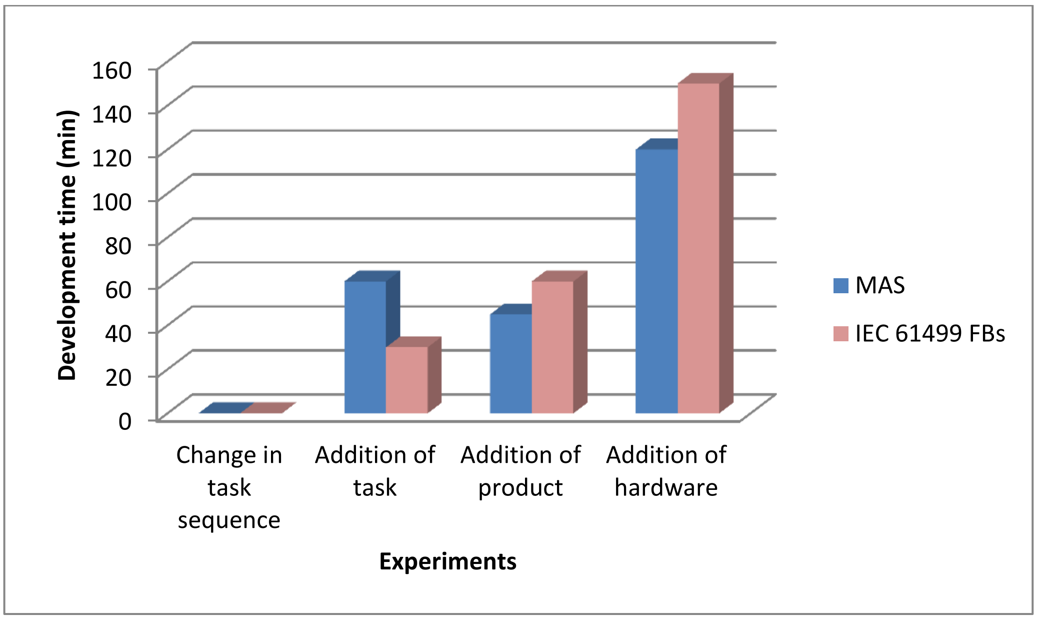


Figure 37: Recorded development times for the control strategies for the four experiments.

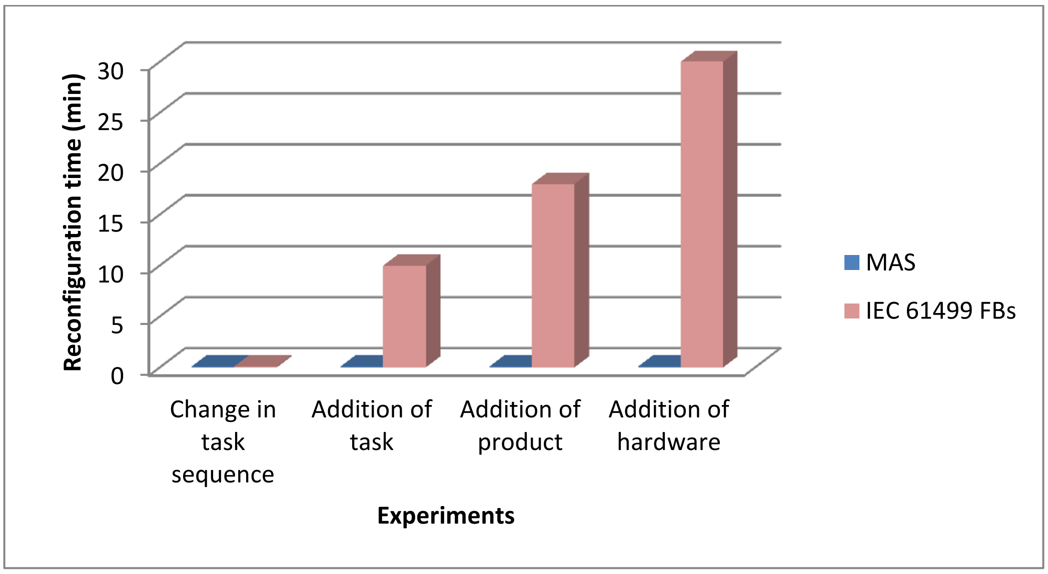


Figure 38: Recorded reconfiguration times for the control strategies for the four experiments.

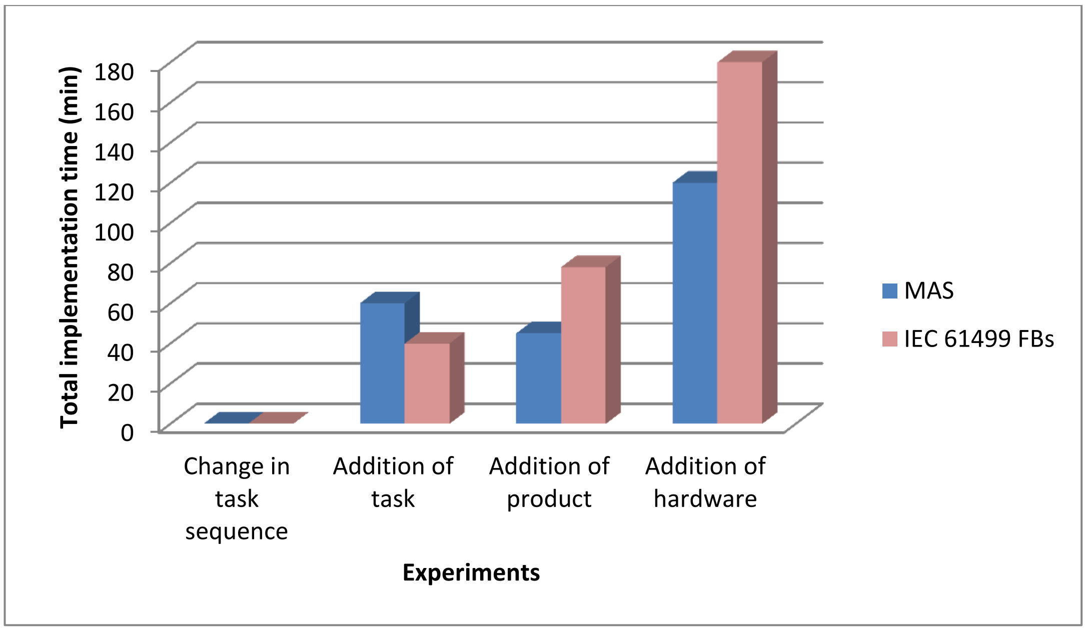


Figure 39: Total implementation times for the control strategies for the four experiments.

7.5.2 Qualitative measurements

The qualitative measurements were done according to the requirements set out in section 2.2, namely modularity, integratability, convertibility, diagnosibility, customizability and scalability. Subjective evaluations, according to the mentioned requirements, were constructed following the implementation of the control strategies during the experiments.

The first reconfigurability experiment, involving the change in the task sequence for a part, indicates the customizability of the control system. The control programs have to be customized to meet the desired production needs. The fact that both control strategies can facilitate the extraction of data from the XML strings and the storage of the information in accessible structures, make them equally customizable. In both cases the task sequence change is handled automatically and during runtime.

Experiment 2 presented an evaluation of the control strategy convertibility and customizability. The introduction of a new part (and so a new task) requires the adaptation of the control system to produce a new product – the ease of this adaptation indicates convertibility. Some control system customizations are then naturally included to meet the production needs. The online addition of the Task agent, which could automatically be used by the Product agent and coordinate the Operational agents is proof of the convertibility and customizability of the MAS. The function block system only requires an alteration to the algorithm of one function block, but it has to be done manually and offline. This hinders the performance of the function block control system concerning these reconfiguration requirements.

Convertibility and customizability are again measured through the reconfiguration of experiment 3. The ability of the MAS to add agents during runtime again gives it a clear advantage over function blocks, with regards to convertibility and customizability. The functionality of the DF within the MAS allows additional agents, which were not part of the initial MAS framework, to automatically be utilized by MAS agents and then use agents themselves. Not only does the addition of the new device to the function block system require the subsystem operation to be stopped, but additional programming to the COMMAND_SELECT resource is also required.

The ease of adding hardware, as is done in experiment 4, reflects the modularity, scalability and integratability of the control strategies. Modularity (i.e. the ability to have interchangeable system components with "plug and play" capabilities) is inherent in the architectural design of both control strategies. Both strategies employ architectural structures to distribute the system functionality in accordance with the holonic control approach. It also appears that both strategies are equally integratable, especially when used in collaboration with LLC programs (as is the case in this research). The function block control system can also employ service interface function blocks to interface with added new technology, though this was not required in this implementation. The scalability of the system is reflected in the capacity increase with a hardware addition, i.e. how easily, quickly and effectively a new hardware resource can be included in the production activities. This is more easily achieved with the MAS than the function block control. This is due to the functionality of the Directory Facilitator – it allows for the seamless introduction of agents to the system. The new agent can be utilized, to its full potential, by the control system components without any additional programming or alterations. On the other hand, the introduction of a new device to the function block control system requires some alteration to the function block of the PRODUCT resources.

The issue of diagnosibility was considered throughout all the experiments. It was found that the ease by which system error can be identified and diagnosed is largely dependent on the software platform. Then, in comparing the diagnostic functions of JADE (implemented in Eclipse) and FBDK, the MAS was found to be more diagnosable. This is due to the numerous built-in tools of the JADE and Eclipse platforms. The JADE GUI provides functions for monitoring the agent actions. The most significant of these functions is the JADE Sniffer function – this function graphically shows the communication between the agents of the MAS. All of the message information is then accessible to the software developer. This sort of functionality is lacking with the FBDK platform. The most significant shortcoming of the FBDK platform is that it has no inherent function for the monitoring of the function block system execution. This becomes especially noticeable when a network of function blocks does not behave as it is supposed to – it is hard to determine if the problem lies in the function blocks or the event/data that connect them.

It is also important to shed some light on the level of expertise required for developing a control system with each of the control strategies. For MAS based on JADE, a strong background in programming (with JADE, specifically Java) is required. A good understanding on the working of the MAS, with the JADE and FIPA specification, is also necessary. For the FBDK function blocks, simple applications can be developed without any expertise in programming – only a simple understanding of the FBDK platform is required. When dealing with more complex applications however, the level of expertise required increases dramatically. A good understanding of Java programming is necessary to implement algorithms in function blocks. In some cases, the Java files created by FBDK must be modified to allow for certain functionality – this then requires a high level of expertise.

8. Conclusion and recommendations

This thesis documents the research conducted into the control of the feeder subsystem of a Reconfigurable Assembly System (RAS). The research focused on the evaluation of an agent-based and an IEC 61499 function block control system, as possible control strategies for RASs. The objective of the research was to evaluate and compare the two strategies with regards to control system reconfigurability.

As a case study, the control strategies were implemented on the hardware of the feeder subsystem of an experimental RAS at Stellenbosch University. The experimental RAS simulates an automated spot-welding process for the production of trip-switch sub-assemblies. The RAS consisted of a singulation unit (which uses a machine vision camera), different part magazines and a six DOF articulated pick-‘n-place robot (fitted with a pneumatic gripper). The feeder subsystem interfaces with the rest of the system in loading individual sub-assembly parts onto a fixture, which is transported by the transport subsystem. The fixture was designed to be modular in an effort to increase the reconfigurability of the system.

The control strategies were implemented according to the ADACOR holonic control architecture. The ADACOR architecture specifies the mapping the subsystem entities to the following holons: Supervisor, Product, Task and Operational. These holons were embodied in the structure of both the control strategies.

For the agent-based control system, a Multi-Agent System (MAS) was developed to implement an agent for each of the subsystem holons. The Supervisor agent interfaces with the overall Cell Controller (CC) program and initiates the loading of the sub-assemblies. The Product agents access the product information and coordinate the various tasks involved in the loading of the parts. The Tasks agents are initiated by the Product agents and are responsible for coordinating the actions of the various Operational agents for the completion of the task. The Operational agents were created for each hardware entity of the feeder subsystem – these agents then control the actions of the hardware entities.

The IEC 61499 function block control system implements the subsystem holons as function block devices. The devices contain networks of function blocks in which the functionality is embedded. The FB_SUPERVISOR and PRODUCT devices have the same responsibilities as their agent counterparts. The Task holon is not explicitly embodied by a device, but is rather embedded in the various event and data connections between the function block networks. A device was developed for each Operational holon, each responsible for the actions of their respective hardware entities.

The Operational holons consist of two layers of control – the Higher Level Control (HLC) and Lower Level Control (LLC). The HLC is implemented through the MAS and function block control strategies. The LLC layers are implemented by C# programs and hardware-specific control programs (DVT Intellect programs for the camera and KRL programs for the robot). The C# programs acts as interface between the HLC programs and the hardware. The hardware-specific programs control the low level hardware actions of the hardware components.

The reconfigurability of the feeder subsystem was assessed in this thesis. The influence of the control strategies on the reconfigurability of the HLC were evaluated by means of four reconfiguration experiments. The evaluation was done through both quantitative and qualitative measurements. The quantitative measurements comprised of the recordings of the development and reconfiguration time required, for each control strategy, for each experiment. The qualitative evaluation was done according to the requirements of modern manufacturing system (identified by Bi *et al*. (2007)) – modularity, integratability, convertibility, customizability, diagnosibility and scalability. The results show that reconfiguration with the MAS can be implemented with the system remaining online. However, the function block control system requires the subsystem to be halted in order to implement the reconfiguration changes. This time increased with increasing complexity of the reconfigurations. In terms of development time (the time required for offline reconfiguration), the MAS reconfiguration required less time in all but one of the experiments.

The reconfiguration experiments provided the grounds for the qualitative evaluation of the control strategies. It was concluded that the MAS exhibits important advantages over the function block control regarding convertibility and customizability. These advantages are due to the ability of the MAS to introduce new control system components seamlessly at runtime and to automatically utilize the capabilities and capacity of the added component. This is the reason for the scalability advantages of the MAS over the function block system as well. It was noticed that the diagnosibility of the control strategy is dependent on the software platform. It was found that the JADE platform of the MAS provided more functions for the identification and solution of system problems than the FBDK platform of the function block system. The two control strategies have the same capabilities regarding modularity and integratability.

The following list of research recommendations were identified in the research performed for this thesis:

- Software platforms and tools for the simulation and testing of individual agent programs could be investigated.
- The implementation of the IEC 61499 function blocks in other software platforms should be assessed.
- It appears that neither the MAS nor function block control strategies have performed optimally in this research. The control level at which the strategies have been implemented does not allow either strategy to exhibit its full capability. The inherent characteristics of the MAS, such as autonomy and cooperation, make it more suitable for implementation at a

higher level. On the other hand, the IEC 61499 function block control strategy is possibly better suited to a lower level of control implementation. The implementation of Object-Oriented C# or Erlang control systems could be investigated.

- Research could be conducted into the use of OPC (OLE for Process Control) in the feeder subsystem.
- The addition of greater redundancy in the feeder subsystem will allow more comprehensive experimentation and comparison of the control strategies. This would be especially valuable for evaluating the performance of the MAS using CNP in agent cooperation.
- Different cell configurations can be implemented in the feeder subsystem to optimize production.
- Research should be conducted into ways to automatically calibrate the robot. The process should handle an initial recalibration after a subsystem reconfiguration, as well as continuous monitoring by the robot to ensure the system remains calibrated during operation.
- Experimentation can be performed with the camera mounted on the robot, as opposed to the installation of cameras on every singulation unit.

This thesis documents the implementation of holonic control, through both an agent-based and IEC 61499 function block control strategy, in the feeder subsystem of an experimental RAS. The reconfigurability of these control strategies were assessed by means of four reconfiguration experiments. The assessment showed that agent-based control is better suited for implementation in this case study. The presented results can however not be taken as a general indication – the selection of the appropriate control strategy will depend on the requirements and nature of the application.

9. References

Adams, A.O., 2010. *Control of a Reconfigurable Assembly System*. MSc.Eng. Thesis. Department of Mechanical and Mechatronic Engineering, Stellenbosch University

Bi, Z.M., Lang, S.Y.T., Shen, W. and Wang, L., 2008. Reconfigurable Manufacturing Systems: The State of the Art. *International Journal of Production Research*. Vol. 46, No. 4: 967 – 992

Bi, Z.M., Wang, L. and Lang, S.Y.T., 2007. Current Status of Reconfigurable Assembly Systems. *International Journal of Manufacturing Research, Interscience*. Vol. 2, No. 3: 303 – 328

Black, G. and Vyatkin, V., 2009. Intelligent Component-Based Automation of Baggage Handling Systems with IEC 61499. *IEEE Transactions on Automation Science and Engineering*. Vol. 7, No. 2: 337 – 351

Booch, G., 1986. Object-Oriented Development. *IEEE Transactions on Software Engineering*. Vol. SE-12, No.2: 211 – 221

Candido, G. and Barata, J., 2007. A Multiagent Control System for Shop Floor Assembly. *Proceedings of the 3rd International Conference on Industrial Applications of Holonic and Multi-agent Systems, HOLOMAS 2007*. Regensburg, Germany. pp. 293 – 302

ElMaraghy, H., 2006. Flexible and Reconfigurable Manufacturing System Paradigms. *International Journal of Flexible Manufacturing System*. Vol. 17:61–276

Exforsys Inc., 2007. *XML Advantages*. [Online]. Available: http://www.exforsys.com/tutorials/xml/xml-advantages.html. [2012, November 10]

FIPA (Foundation for Intelligent Physical Agents). 2010. [Online]. Available: http://www.fipa.org. [2011, July 20]

Gruber, T.R., 1991. The Role of Common Ontology in Achieving Sharable, Reusable Knowledge Bases in Allen, J.A., Fikes, R. and Sandewall, E. (eds). *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*. Cambridge. pp. 601 – 602

Heverhagen, T., Tracht, R. and Hirschfeld, R., 2003. A Profile for Integrating Function Blocks into the Unified Modeling Language. *Proceedings of the International Workshop on Specification and Validation of UML models for Real Time and Embedded Systems*. San Francisco, USA

Kotak, D., Wu, S., Fleetwood, M. and Tamoto, H., 2003. Agent-Based Holonic Design and Operations Environment for Distributed Manufacturing. *Computers in Industry*. Vol. 52: 95–108

KUKA Robot Group, 2007. *KUKA System Software 7.0: Operating and Programming Instructions for Systems Integrators*. Germany: KUKA Roboter GmbH

Lee, D.Y. and DiCesare, F., 1994. Scheduling Flexible Manufacturing Systems Using Petri Nets and Heuristic Search. *IEEE Transactions on Robotics and Automation*. Vol. 10, No. 2: 123 – 132

Leitao, P. and Restivo, F.J., 2006. ADACOR: A Holonic Architecture for Agile and Adaptive Manufacturing Control. *Computers for Industry*. Vol. 57, No. 2: 121–130

Leitao, P. and Restivo, F.J., 2008. Implementation of a Holonic Control System in a Flexible Manufacturing System. *IEEE Transactions on Systems, Man, and Cybernetics*. Vol. 38, No. 5: 699 – 709

Lepuschitz, W., Vrba, P., Vallee, M., Merdan, M., Kaindl, H., Arnautovic, E., 2009. An Automation Agent Architecture with a Reflective World Model in Manufacturing Systems. *2009 IEEE International Conference on Systems, Man and Cybernetics*. pp. 305 – 310

Lewis, R.W., 1998. *Programming Industrial Control Systems Using IEC 1131*. London: Institute of Electrical Engineers

Lewis, R.W., 2001. *Modeling Control Systems Using IEC 61499 Function Blocks: Applying Function Blocks to Distributed Systems*. London: Institute of Electrical Engineers.

Le Roux, A., 2013. *Control of a Conveyor System for a Reconfigurable Manufacturing Cell*. MSc.Eng. Thesis. Department of Mechanical and Mechatronic Engineering, Stellenbosch University

Marik, V., Vrba, P., Tichy, P., Hall, K.H., Staron, R.J., Maturana, F.P., Kadera, P., 2010. Rockwell Automation's Holonic and Multi-Agent Control Systems Compendium. *2010 IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*. Vol. 41, No. 1: 14 – 30

Martinsen, K., Haga, E., Dransfeld, S. and Watterwald, L.E., 2007. Robust, Flexible and Fast Reconfigurable Assembly System for Automotive Air-brake Couplings. *Intelligent Computation in Manufacturing Engineering*. Vol. 6

Mehrabi, M.G., Ulsoy, A.G., Koren, Y., 2000. Reconfigurable Manufacturing Systems: Key to Future Manufacturing. *Journal of Intelligent Manufacturing*. Vol. 13: 135 – 146

Mehrabi, M.G., Ulsoy, A.G., Koren, Y. and Heytler, P., 2002. Trends and Perspectives in Flexible and Reconfigurable Manufacturing Systems. *Journal of Intelligent Manufacturing*. Vol. 13: 135 – 146

Meng, F., Tan, D. and Wang, Y., 2006. Development of Agent for Reconfigurable Assembly System with JADE. *Proceedings of the 6th World Congress on Intelligent Control and Automation*. Dalian, China. pp. 7915 – 7919

Mulubika, C., 2013. *Evaluation of Control Strategies for Reconfigurable Manufacturing Systems*. MSc.Eng. Thesis. Department of Mechanical and Mechatronic Engineering, Stellenbosch University

Murata, T., 1989. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*. Vol. 77, No. 4.

Nikraz, M., Caire, G. and Bahri, P.A., 2006. A Methodology for the Development of Multi-Agent Systems Using the JADE Platform. *International Journal of Computer Systems Science & Engineering*. Vol. 21, No. 2: 99-116

Odell, J., 2002. Objects and Agents Compared. *Journal of Object Technology*. Vol. 1: 41 – 53

Panjaitan, S. and Frey, G., 2006. Combination of UML Modeling and the IEC 61499 Function Block Concept for the Development of Distributed Automation Systems. *IEEE Conference on Emerging Technologies and Factory Automation*. Vol. 1: 766-773

Paolucci, M. and Sacile, R., 2005. *Agent-Based Manufacturing and Control Systems*. London: CRC Press

Poletti, R., 2011. *Mechanical Design of a Stepped-Conveyor Singulation Unit*. Internal Design Report. Mechatronics, Automation and Design Research Group, Department of Mechanical and Mechatronic Engineering, Stellenbosch University

Raj, T., Shankar, R. and Suhaib, M., 2007. A Review of Some Issues and Identification of Some Barriers in the Implementation of FMS. *International Journal of Flexible Manufacturing System*. Vol. 19: 1 – 40

Rooker, M.N., Hummer, O., Sunder, C., Strasser, T. and Kerbleder, G., 2007. Downtimeless System Evolution: Current State and Future Trends. *5th IEEE Conference on Industrial Infomatics*. Vol. 2: 1077 – 1082.

Scholz-Reiter, B. and Freitag, M., 2007. Autonomous Processes in Assembly Systems. *Annals of the CIRP*. Vol. 56: 712 – 730

Sequeira, M.A., 2008. *Conceptual Design of a Fixture-Based Reconfigurable Spot Welding System*. MSc.Eng. Thesis. Department of Mechanical and Mechatronic Engineering, Stellenbosch University

Stefik, M. and Bobrow, D.G., 1985. Object-Oriented Programming: Themes and Variations. *AI Magazine*. Vol. 6, No. 4: 40 – 62

Van Brussel, H., Wyns, J., Valckenaers, P., Bongaerts, L. and Peeters, P., 1998. Reference Architecture For Holonic Manufacturing Systems: PROSA. *Computers in Industry*. Vol. 37: 255 – 274

Vrba, P., 2003. MAST: Manufacturing Agent Simulation Tool. *Proceedings of the IEEE Conference on Emergent Technology for Factory Automation.* Vol. 1: 282 – 287

Vrba, P., Lepuschitz, W., Vallee, M., Merdan, M., Resch, J., 2009. Integration of a Heterogeneous Low Level Control in a Multi-Agent System for the Manufacturing Domain. *2009 IEEE International Conference on Systems, Man and Cybernetics*. pp: 7 – 14

Vrba, P., Marik, V., 2009. Capabilities of Dynamic Reconfiguration of Multi-Agent Based Industrial Control Systems. *2009 IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*. Vol. 40, No. 2: 213 – 223

Vyatkin, V., 2007. *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design.* North Carolina: Instrumentation, Systems and Automation Society, ISA

Wang, L., Cai, N., Feng, H.Y., 2007. Dynamic Setup Dispatching and Execution Monitoring using Function Blocks. *Proceedings of the 2nd International Conference on Changeable, Agile, Reconfigurable and Virtual (CARV) Production.* pp. 699 – 708

Wiendahl, H. P., 2007. Changeable Manufacturing: Classification, Design and Operation. *Annals of CIRP*. Vol. 56: 783 – 809

World Minimum Wages. [S.a.]. [Online]. Available: http://www.minimum_wage.org/minwage/international. [2013, February 21]

Xie, H., Shen, W., Neelamkavil, J., Hao, Q., 2007. Simulation and Optimization of Mixed-Model Assembly Lines Using Software Agents. *Proceedings of the 2nd International Conference on Changeable, Agile, Reconfigurable and Virtual (CARV) Production.* pp. 340 – 347

Zhou, M., DiCesare, F. and Desrochers, A.A., 1992. A Hybrid Methodology for Synthesis of Petri Net Models for Manufacturing Systems. *IEEE Transactions on Robotics and Automation*. Vol. 8, No. 3: 350 – 361

Appendix A: Singulation unit throughput and reconfigurability investigation

The throughput of the stepped-conveyor singulation unit was investigated in two experiments – one to determine the optimal singulation speed and the other to determine the optimal number of parts to be present in the input bin. The reconfigurability experiment was used to determine the effectiveness of singulating a new part, without any hardware modifications.

The first experiment of the throughput analysis aimed to determine the optimal singulation speed for maximum throughput. The experiment had to consider two factors – the number of singulations within a time period and the success rate of those singulations. The time and number of singulations needed for ten successful singulations was recorded for eight singulation speeds – the recorded data is shown in Table A 1. The data is plotted in Figure A 1 and Figure A 2. The average singulation time indicates how long it took to achieve ten successful singulation, while the success rate refers to how many singulations were required to achieve ten successful ones. The results show a decrease of average singulation time with increasing singulation speed. While most of the speeds resulted in a singulation success rate of around 50%, the highest percentage (62%) was observed with the highest speed. For this experiment, the input bin was filled with one hundred coil parts for each speed setting.

The recorded data was used to calculate the probability of successful singulations, for specific time intervals, for each of the speed settings. This calculated data is shown in Table A 2 and is plotted in Figure 14.

Table A 1: Recorded data for the optimal singulation speed experiment.

	Conveyor motor speed (rpm)							
	100	**150**	**200**	**250**	**300**	**350**	**400**	**450**
	Belt speed (rpm)							
	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	Number of potential singulations per minute							
	14	**21**	**28**	**35**	**42**	**49**	**56**	**63**
Average singulation time (s)	10	8.7	4	2.5	2.1	2.2	1.8	1.2
Singulation success rate (%)	45.5	30.3	52.6	50.0	50.0	43.5	50.0	62.5

The second throughput experiment was done to determine the optimal number of parts in the input bin which would maximise the singulation success rate. A similar procedure was followed as in the first experiment, except that it was done at one speed and with different numbers of parts in the bin. The recorded data is shown in

Table A 3. The results show that the optimal number of parts in the input bin is eighty. The constant speed setting was chosen to be 400 rpm.

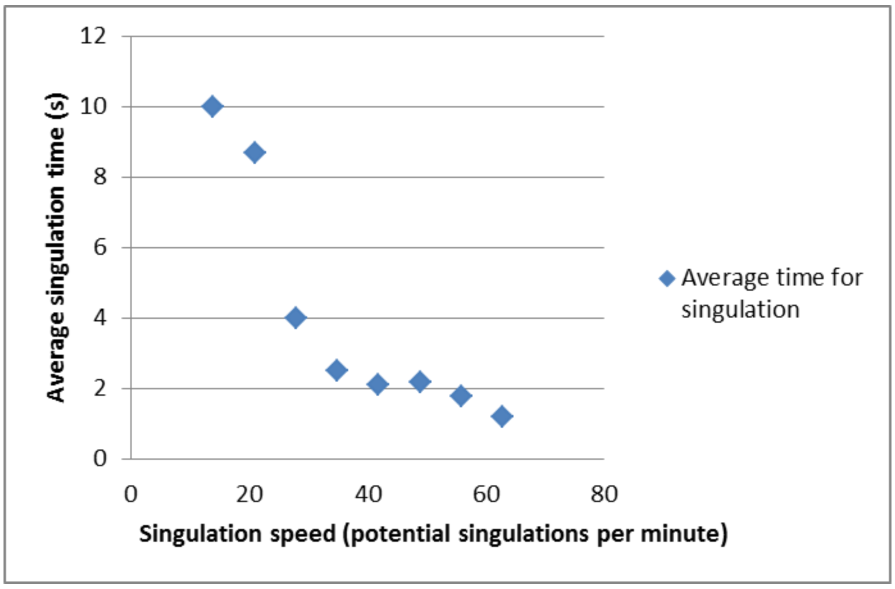


Figure A 1: Average singulation time for different singulation speeds.

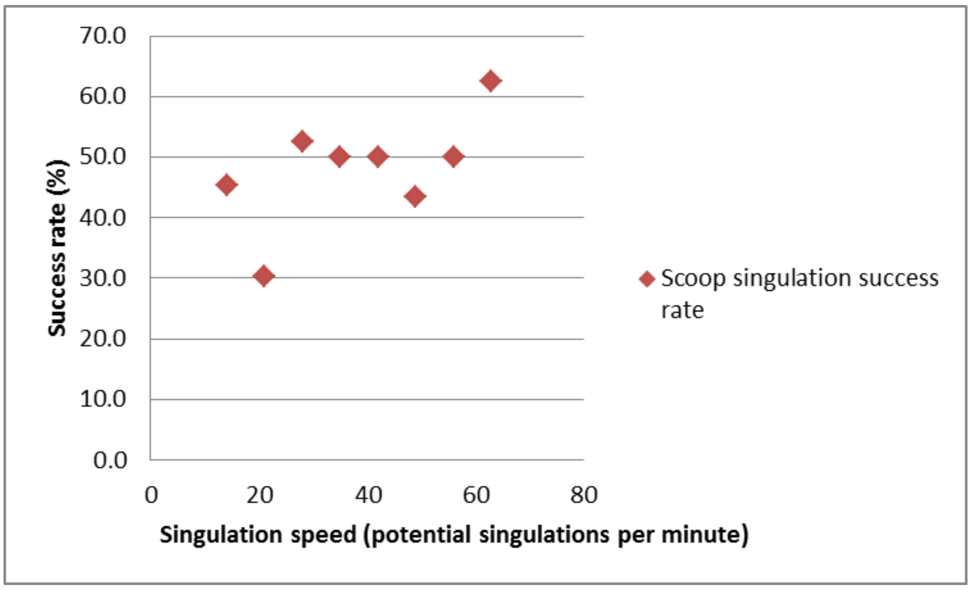


Figure A 2: Average success rates for different singulation speeds.

Table A 2: The calculated data for Figure 14.

				Singulations							
				1	2	3	4	5	6	7	8
14 spm	**Success rate**	**45.5**	**Probability of successful singulation**	45.5	70.3	83.8	91.2	95.2	97.4	98.6	99.2
	Minimum singulation time	**4.3**	**Time intervals**	4.3	8.6	12.9	17.1	21.4	25.7	30.0	34.3
21 spm	**Success rate**	**30.3**	**Probability of successful singulation**	30.3	51.4	73.5	85.6	92.1	95.7	97.7	98.7
	Minimum singulation time	**2.9**	**Time intervals**	2.9	5.7	8.6	11.4	14.3	17.1	20.0	22.9
28 spm	**Success rate**	**52.6**	**Probability of successful singulation**	52.6	77.5	87.8	93.3	96.4	98.0	98.9	99.4
	Minimum singulation time	**2.1**	**Time intervals**	2.1	4.3	6.4	8.6	10.7	12.9	15.0	17.1
35 spm	**Success rate**	**50**	**Probability of successful singulation**	50	75.0	86.4	92.6	96.0	97.8	98.8	99.3
	Minimum singulation time	**1.7**	**Time intervals**	1.7	3.4	5.1	6.9	8.6	10.3	12.0	13.7
42 spm	**Success rate**	**50**	**Probability of successful singulation**	50	75.0	86.4	92.6	96.0	97.8	98.8	99.3
	Minimum singulation time	**1.4**	**Time intervals**	1.4	2.9	4.3	5.7	7.1	8.6	10.0	11.4
49 spm	**Success rate**	**43.5**	**Probability of successful singulation**	43.5	68.1	82.6	90.5	94.8	97.2	98.5	99.2
	Minimum singulation time	**1.2**	**Time intervals**	1.2	2.4	3.7	4.9	6.1	7.3	8.6	9.8
56 spm	**Success rate**	**50**	**Probability of successful singulation**	50	75.0	86.4	92.6	96.0	97.8	98.8	99.3
	Minimum singulation time	**1.1**	**Time intervals**	1.1	2.1	3.2	4.3	5.4	6.4	7.5	8.6
63 spm	**Success rate**	**62.5**	**Probability of successful singulation**	62.5	85.9	92.3	95.8	97.7	98.8	99.3	99.6
	Minimum singulation time	**1.0**	**Time intervals**	1.0	1.9	2.9	3.8	4.8	5.7	6.7	7.6

Table A 3: Success rates for different numbers of parts in the input bin.

	Number of parts in the input bin				
	40	**60**	**80**	**100**	**120**
Singulation success rate (%)	45	45	52	50	48

The reconfigurability investigation required a similar throughput experiment, but with a new part to be singulated. The results could then be compared to determine how part or part-size specific the singulation process is. The input bin was thus filled with one hundred moving contact parts. The moving contacts part was chosen as it is almost the same size as the coils. The experiment revealed that the singulation of the moving contact part had a success rate 10% lower than for the coil parts. This result shows that the design of the scoops is more part specific than part-size specific – this indicates that further refinement must be done to the scoop design.

Appendix B: Gripper design

B.1 Design requirements

The following requirements were considered in the gripper selection and finger design:

1. The gripper must be attached to the tool interface of the KUKA robot.
2. The stroke of the gripper jaws must be large enough to grip the outside of some parts, but small enough to allow the accurate calibration of gripper finger position.
3. The force induced by the gripper jaws must be large enough to firmly hold the parts, but must not cause any damage to the parts.
4. The gripper must be equipped with removable fingers.
5. The gripper fingers must be able to withstand the force of the gripping action for infinite life cycles.
6. The gripper fingers must be small enough to allow entrance to the inside of some parts.
7. The gripper fingers must allow the picking up of parts in different orientations.

B.2 Design specifications

This list of requirements of section B.1 was considered in the formulation of the following set of design specifications:

i. The gripper, along with the accompanying attachments, must weigh less than 16kg.
ii. The stroke of the individual gripper jaws must be greater than 1.5mm and less than 6mm.
iii. The force that the gripper exerts must be greater than 10N and less than 50N.
iv. The gripper fingers must be small enough to comfortably enter a 3mm diameter hole.

B.3 Static and fatigue analysis

Finger dimensions:

$l_1 := 0.01 \cdot m$	$w_1 := 0.024m$	$t_1 := 0.003m$
$l_2 := 0.055m$	$w_2 := w_1$	$t_2 := t_1$
$l_3 := 0.005m$	$w_3 := w_1$	$t_3 := t_1$
$l_4 := 0.015m$	$w_4 := 0.003m$	$t_4 := 0.003m$

$$d_{4\_corners} := \left(t_4^{\,2} + w_4^{\,2}\right)^{0.5} = 4.243 \times 10^{-3}\, m$$

Cross-sectional properties:

$A_1 := w_1 \cdot t_1 = 7.2 \times 10^{-5} m^2$ $\quad A_2 := w_2 \cdot t_2 = 7.2 \times 10^{-5} m^2$

$A_3 := w_3 \cdot t_3 = 7.2 \times 10^{-5} m^2$ $\quad A_4 := w_4 \cdot t_4 = 9 \times 10^{-6} m^2$

$I_2 := \frac{w_2 \cdot t_2^3}{12} = 5.4 \times 10^{-11} m^4$ $\quad I_3 := \frac{w_3 \cdot t_3^3}{12} = 5.4 \times 10^{-11} m^4$

$I_4 := \frac{w_4 \cdot t_4^3}{12} = 6.75 \times 10^{-12} m^4$

<u>Material Properties: AISI 1040 Cold-drawn steel</u>

Density:

$\rho := 7800 \frac{kg}{m^3}$

Elastic modulus:

$E := 207 \cdot 10^9 \cdot Pa$

Maximum allowable tensile stress:

$\sigma_{allow} := 568 \cdot MPa$

Yield strength:

$\sigma_y := 276 \cdot MPa$

Mass properties:

$m_1 := A_1 \cdot l_1 \cdot \rho = 5.616 \times 10^{-3} kg$ $\quad m_2 := A_2 \cdot l_2 \cdot \rho = 0.031 kg$

$m_3 := A_3 \cdot l_3 \cdot \rho = 2.808 \times 10^{-3} kg$ $\quad m_4 := A_4 \cdot l_4 \cdot \rho = 1.053 \times 10^{-3} kg$

$m_{finger} := m_1 + m_2 + m_3 + m_4 = 0.04 kg$

$w_{finger} := m_{finger} \cdot g = 0.396 N$

Static analysis:

Force:

$F_{grip} := 25 \cdot N$

Moment:

$$M_A := F_{grip} \cdot l_4 = 0.375\,N \cdot m$$

Bending stress:

$$y_{max\_A} := \frac{t_4}{2} = 1.5 \times 10^{-3}\,m$$

$$\sigma_{max\_A} := \frac{M_A \cdot y_{max\_A}}{I_4} = 8.333 \times 10^{7}\,Pa$$

Safety factor:

$$n_{bend\_A} := \frac{\sigma_y}{\sigma_{max\_A}} = 3.312$$

Shear stress:

$$\tau_{max\_A} := \frac{3 \cdot F_{grip}}{2 \cdot A_4} = 4.167 \times 10^{6}\,Pa$$

Safety factor:

$$n_{shear} := \frac{\sigma_y}{2 \cdot \tau_{max\_A}} = 33.12$$

Deflection analysis:

Tip deflection:

$$d_4 := \frac{F_{grip} \cdot \left(l_4\right)^3}{3 \cdot E \cdot I_4} = 2.013 \times 10^{-5}\,m$$

$$d_3 := \frac{\left[F_{grip} \cdot \left(l_4\right)\right] \cdot l_3^{\,2}}{2 \cdot E \cdot I_3} = 4.194 \times 10^{-7}\,m$$

$$\Phi_2 := \frac{\left[F_{grip} \cdot \left(l_3 + l_4\right)\right] \cdot l_2}{E \cdot I_2} = 2.46 \times 10^{-3}$$

$$d_{tip} := \Phi_2 \cdot \left(l_3 + l_4\right) + d_3 + d_4 = 6.975 \times 10^{-5}\,m$$

Fatigue analysis:

Endurance limit:

$$S_{e\_prime} := \frac{\sigma_{allow}}{2} = 2.84\times 10^{8}\,Pa$$

Endurance limit modification factors:

Surface factor: (Assumed cold-drawn)

$$a := 4.51$$

$$b := -0.265$$

$$k_a := a\cdot\sigma_{all}{}^{b} = 0.84$$

Size factor:

$$d_e := 0.808\left(t_4\cdot w_4\right)^{0.5} = 2.424\times 10^{-3}\,m$$

$$k_b := 1.24\,d_{e\_mod}{}^{-0.107} = 1.128$$

Loading factor:

$$k_c := 1$$

Temperature factor:

$$k_d := 1$$

Reliability factor: (99.9% reliability)

$$k_e := 0.753$$

Miscellaneous factor:

$$k_f := 1$$

Modified endurance limit:

$$S_e := k_a\cdot k_b\cdot k_c\cdot k_d\cdot k_e\cdot k_f\cdot S_{e\_prime} = 202.612\,MPa$$

Nominal fluctuating stress components:

$$\sigma_{ao} := \frac{\sigma_{max\_A}}{2}$$

$\sigma_{mo} := \sigma_{ao}$

Stress concentration factors: (3mm radius and 600 MPa ultimate tensile strength)

$q := 0.82$

$K_t := 1.4$

$K_f := 1 + q \cdot (K_t - 1) = 1.328$

Fluctuating stress components:

$\sigma_a := K_f \cdot \sigma_{ao} = 5.533 \times 10^7 \, Pa$

$\sigma_m := K_f \cdot \sigma_{mo} = 5.533 \times 10^7 \, Pa$

Fatigue factor of safety:

$$n_f := \frac{1}{\frac{\sigma_a}{S_e} + \frac{\sigma_m}{\sigma_{allow}}} = 2.699$$

B.4 Gripper pickup actions

The gripper fingers were designed to pick up parts from the singulation unit and the part magazines, and place them in the fixture. The pickup actions of the gripper are shown in Figure B 1 and the place actions in Figure C 1.

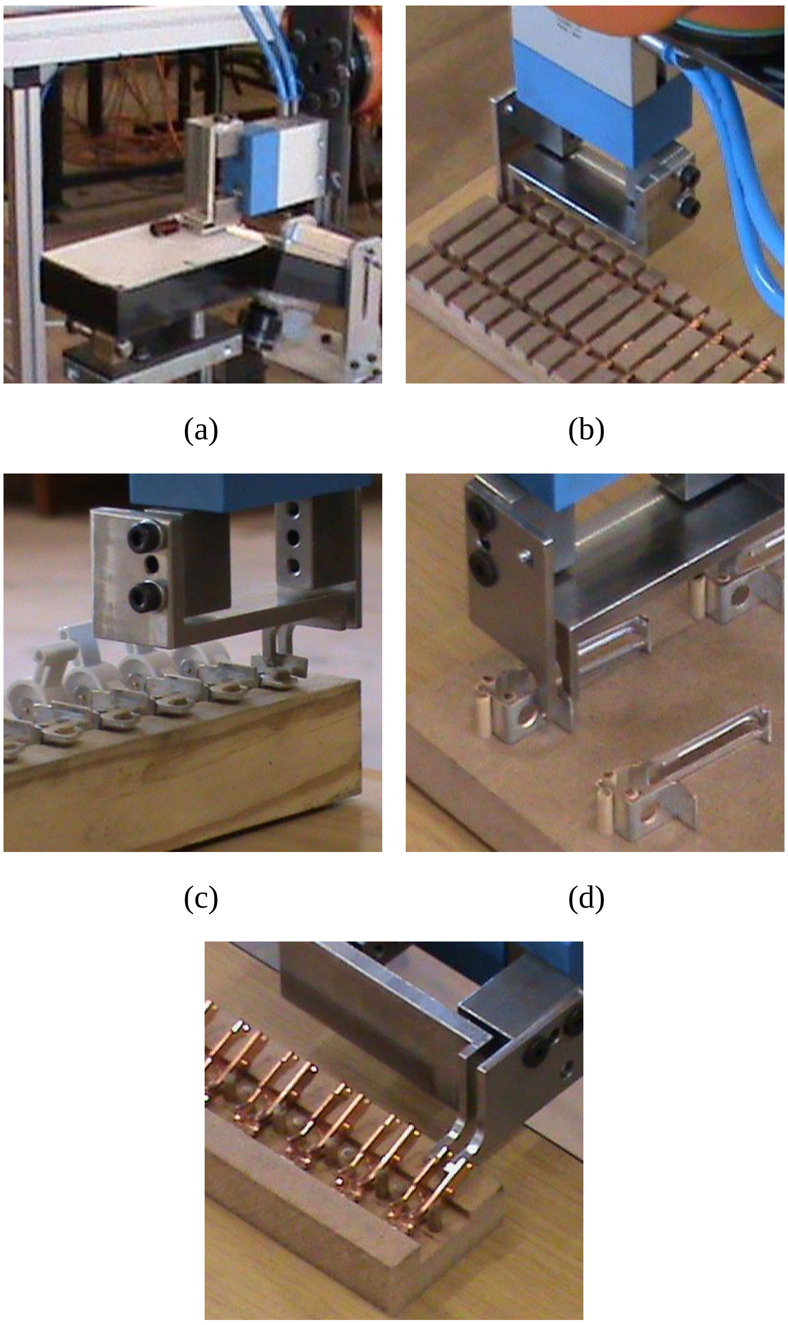


Figure B 1: Gripper pickup actions of the various parts – (a) coil, (b) long and short pigtails, (c) handle frame assembly, (d) load terminal and (e) moving contact.

Appendix C: Fixture design

C.1 Design requirements

The fixture was designed according to the following requirements:

1. The fixture should fit on the conveyor pallets. No part of the fixture may extend over the edges of the pallet.
2. The fixture must allow for the stacking of several pallets on each other inside the pallet magazine.
3. The fixture must accommodate the entire range of trip switch parts, in their specified locations.
4. The fixture must provide access to the gripper fingers of the pick-‘n-place robot to allow for appropriate part placement.
5. Access should be allowed for the welding electrodes of the welding robot at the spot-weld locations.
6. The fixture should exhibit some reconfigurability characteristics.
7. The fixture must secure the parts during the transportation process, as the stoppages and direction changes may cause part movement.
8. The fixture must secure the parts during the welding process, as the electrodes may stick to the parts after welding.

C.2 Design specifications

In considering the requirements of section C.1, the following specifications were formulated:

i. The maximum fixture dimensions (according to the dimensions of the conveyor pallets): 300 mm x 300 mm
ii. The maximum height of the parts when held in the fixture (according to the entrance dimensions of the pallet magazine): 50 mm
iii. Alignment tolerances: 0.1 mm
iv. The minimum clearance radius around the spot-weld locations: 5 mm
v. The maximum fixture weight (according to the specification of the lifting pneumatic cylinder of the conveyor and pallet magazine): 20kg

C.3 Gripper place actions in the fixture

The fixture was designed to allow for the placement of parts, into the supports, by the gripper. The placement of the parts is shown in Figure C 1.

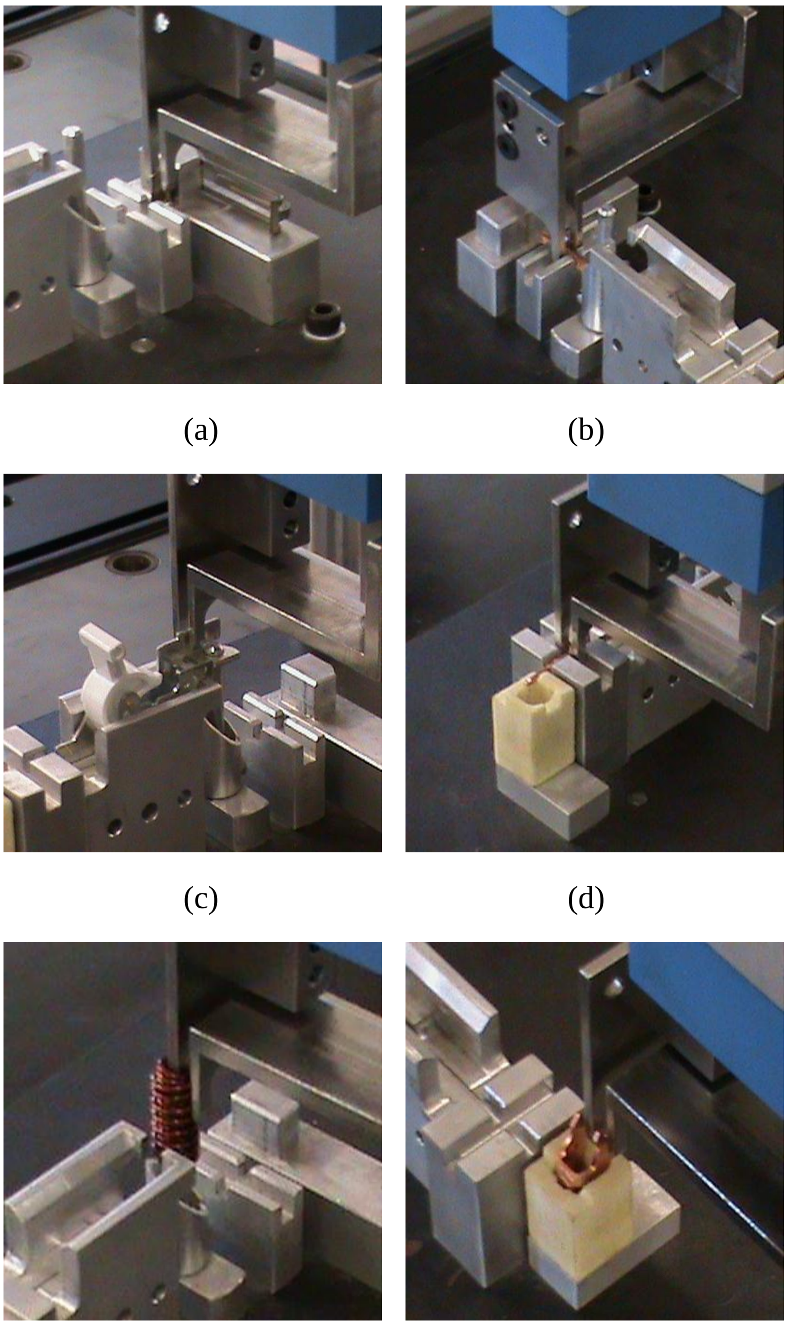

(a) (b)
(c) (d)
(e) (f)

Figure C 1: The placement of parts in the fixture by the gripper – (a) load terminal, (b) short pigtail, (c) handle frame assembly, (d) long pigtail, (e) coil and (f) moving contact.

Appendix D: DVT Intellect script programs

D.1 Background script program

```
class NewScript
{
    public static void main(String args[])
    {
        while(true)
        {
         int len = 2; //length of data to be recieved
         int port = 3248; //port number for connection
         int conStatus = -1; //to detect an accepted connection
         byte data[] = new byte[len]; //array for incoming data
         //sockets needed for communication
         Socket mySocket = new Socket();
         Socket sock = new Socket();

         //reset the external trigger mode bit to 0 (internal mode)
         SetInputs(0L,(1L<<7));
         //reset the inspection trigger bit
         SetInputs(0L,1L);

         int status = mySocket.Bind(port);     //returns binding status

         DebugPrint("socket connection status is " + status);
         if (status==0)
         {
              DebugPrint("socket bound to port");

              status = mySocket.Listen();

              if (status == 0)
              {
                     while (conStatus != 0)
                     {
                            //check socket connection
                            conStatus = mySocket.Accept(sock);
                     }

                     //receive and store data
                     status = sock.Recv(data, 0, len);

                     //check system status and wait until it is idle
                     long Bit = 1;
                     //check system busy bit
                     while((GetOutputs() & (Bit<<8)) != 0)
                     {
                     }
                     //inspection command
                     if (data[0] == 1)
                     {
                            DebugPrint("inspection product = " + data[1]);

                            //background script resets completion indicator bit
                            byte b;
                            b=0;
                            int stat = RegisterWriteByte(110,b);
                            //set to external trigger mode
                            SetInputs((1L<<7),0L);
```

```
                           //get inspection product by ID
                           Product prod = GetProductById((short) data[1]);
                           //sets the inspection product
                           prod.Select();

                           SetInputs(1L,0L); //trigger inspection

                           //delay to ensure inspections are complete
                           sleep(1000);
                           byte quick = 0;  //local storage variable

                           //wait for foreground script completion
                           while(quick != 1)
                           {
                                 quick = RegisterReadByte(110);

                                 //if part detection is triggered
                                 if(data[1] == 10){
                                        SetInputs(0L,1L); //stop inspection
                                        sleep(50);
                                        SetInputs(1L,0L); //start inspection
                                 }
                           }
                           DebugPrint("Part script ended its own job");

                           //reset the trigger bit to 0 to stop the inspection
                           SetInputs(0L,1L);

                           //background script resets this bit
                           b=0;
                           stat = RegisterWriteByte(110,b);

                           //read the inspection result at register number 25
                           String toSend;
                           toSend = RegisterReadString(25);

                           DebugPrint("According to background:");

                           DebugPrint(toSend);

                           //extract bytes from string
                           byte sendData[] = toSend.getBytes();
                           //send the extracted bytes
                           status = sock.Send(sendData,0,sendData.length);

                           SetInputs(0L,(1L<<7));//reset external trigger mode
                    }
             }
       }
       // Short delay before next iteration
       sleep(10);
    }
  }
```

D.2 Foreground script program

```
class COIL_LOCATE
{
       public static double transform_X(double PosX)
       {
             double PosX_real = 0;      //real x-position to return

             PosX_real = PosX - findOrigin_X.EdgePoint.X;
```

```
        return PosX_real;
}

public static double transform_Y(double PosY)
{
        double PosY_real = 0;       //real x-position to return

        PosY_real = PosY - findOrigin_Y.EdgePoint.Y;

        return PosY_real;
}

public void inspect()
{
        String output;

        double posX[];
        double posY[];
        double ang[];
        double score[];

        posX = new double[9];
        posY = new double[9];
        ang = new double[9];
        score = new double[9];

        double PosX = 0;
        double PosY = 0;
        double PosZ = 0;
        double Ang = 0;

        //check the number of parts present on the platform
        DebugPrint("Blobs found: "+ num_of_parts.ObjectCount);
        if(num_of_parts.ObjectCount != 1){

                COIL_LOCATE.Result = -1; //set inspection result to FAIL

                //prepare failure result to be returned
                output = "pass" + "false" + "end";
                //print failure message
                DebugPrint("Failure - more than one part on platform.");
        }
        else{
                //store values from the object locate softsensors
                if(coil_locate_1.Result == 0){
                        posX[1] = coil_locate_1.PickPoint.X;
                        posY[1] = coil_locate_1.PickPoint.Y;
                        ang[1] = coil_locate_1.PickPoint.Angle;
                        score[1] = coil_locate_1.MatchScore;
                }
                if(coil_locate_2.Result == 0){
                        posX[2] = coil_locate_2.PickPoint.X;
                        posY[2] = coil_locate_2.PickPoint.Y;
                        ang[2] = coil_locate_2.PickPoint.Angle;
                        score[2] = coil_locate_2.MatchScore;
                }
                if(coil_locate_3.Result == 0){
                        posX[3] = coil_locate_3.PickPoint.X;
                        posY[3] = coil_locate_3.PickPoint.Y;
                        ang[3] = coil_locate_3.PickPoint.Angle;
                        score[3] = coil_locate_3.MatchScore;
                }
```

```
                    if(coil_locate_4.Result == 0){
                           posX[4] = coil_locate_4.PickPoint.X;
                           posY[4] = coil_locate_4.PickPoint.Y;
                           ang[4] = coil_locate_4.PickPoint.Angle;
                           score[4] = coil_locate_4.MatchScore;
                    }
                    if(coil_locate_5.Result == 0){
                           posX[5] = coil_locate_5.PickPoint.X;
                           posY[5] = coil_locate_5.PickPoint.Y;
                           ang[5] = coil_locate_5.PickPoint.Angle;
                           score[5] = coil_locate_5.MatchScore;
                    }
                    //store values from sensor 6
                    if(coil_locate_6.Result == 0){
                           posX[6] = coil_locate_6.PickPoint.X;
                           posY[6] = coil_locate_6.PickPoint.Y;
                           ang[6] = coil_locate_6.PickPoint.Angle;
                           score[6] = coil_locate_6.MatchScore;
                    }
                    //store values from sensor 7
                    if(coil_locate_7.Result == 0){
                           posX[7] = coil_locate_7.PickPoint.X;
                                  posY[7] = coil_locate_7.PickPoint.Y;
                           ang[7] = coil_locate_7.PickPoint.Angle;
                           score[7] = coil_locate_7.MatchScore;
                    }
                    //store values from sensor 8
                    if(coil_locate_8.Result == 0){
                           posX[8] = coil_locate_8.PickPoint.X;
                           posY[8] = coil_locate_8.PickPoint.Y;
                           ang[8] = coil_locate_8.PickPoint.Angle;
                           score[8] = coil_locate_8.MatchScore;
                    }
                    //find the best matchScore
                    int best = 1;
                    double bestScore = score[1];
                    for(int count = 2;count < 8;count++){

                           if(score[best] < score[count]){
                                  best = count;
                                  bestScore = score[count];
                           }
                    }
                    DebugPrint("Best Score = "+bestScore+" by coil_locate_"+best);

                    if(bestScore >= 70){
                           //store best coordinates
                           PosX = posX[best];
                           PosY = posY[best];
                           Ang = ang[best];

                           PosX = transform_X(PosX);
                           PosY = transform_Y(PosY);
                           PosZ = 6.85; //vertical pick position of the coil
                           Ang = Ang*-1.00; //transform angle

                           //prepare successful result to be returned
                           output = "pass" + "true" + "x" + toString(PosX) + "y" +
toString(PosY) + "z" + toString(PosZ) + "angle" + toString(Ang) + "end";

                           //print result
                           DebugPrint("Pickup Position --> X: "+ PosX + " Y: "+
PosY + " Z: " + PosZ + " Angle: "+ Ang);
```

```
                        COIL_LOCATE.Result = 0;    //result is a PASS

                }
                else
                {
                        //case where all sensors failed to locate the part
                        //set inspection result to FAIL
                        COIL_LOCATE.Result = -1;

                        //prepare failure result to be returned
                        output = "pass" + "false" + "end";
                        //print failure message
                        DebugPrint("Pickup Position not found.");
                }
        }
        RegisterWriteString(25, output); //write result to register
        //indicate inspection completion
        byte b = 1;
        int stat = RegisterWriteByte(110,b);
    }
}
```

Appendix E: KUKA robot functionality

E.1 Calibration functions

The KUKA robot controller provides built-in functions for the calibration of tools and workspaces. These calibration functions are useful for hardware relocations during reconfiguration. The use of the functions is described in this section.

The tool calibration function allows for the easy calibration of the robot motion for a tool attached at the tool interface. This entails the definition of the Tool Centre Point (TCP) by manually moving the tool to a specified point from different directions, as is shown in Figure E 1. The robot position can then be given in terms of the position of the TCP. This calibration was done to allow the monitoring of the position of the gripper fingers. This tool also allows for the storage of calibration information of different tools. Different gripper configurations can then be calibrated in advance, which means that a manual gripper reconfiguration only requires the appropriate tool to be selected in the control software. This approach could decrease subsystem ramp-up time significantly.

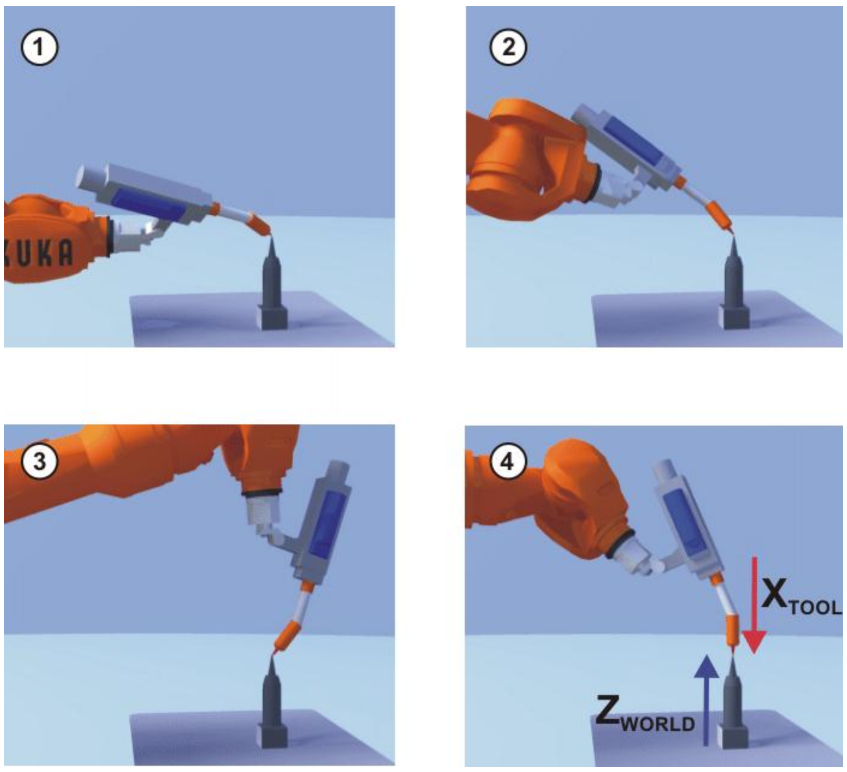


Figure E 1: The sequence of steps required for the calibration of a new tool (KUKA Robot Group, 2007).

The definition of workspaces (referred to as *bases*) is also very useful in pick-‘n-place applications. The origin and orientation information of a specified area can be calibrated by manually moving the TCP of the robot to certain positions in the area (depicted in Figure E 2). This allows the specification of a coordinate system to a *base* - eliminating the dependence on global coordinates. This was used to specify pick-‘n-place coordinates on the presentation platform of the singulation unit, the part magazines and the fixtures mounted on the pallets. The definition of *base* coordinate systems simplifies subsystem reconfiguration involving the relocation of hardware positions, as only the origin coordinates of the hardware workspace needs to be updated.

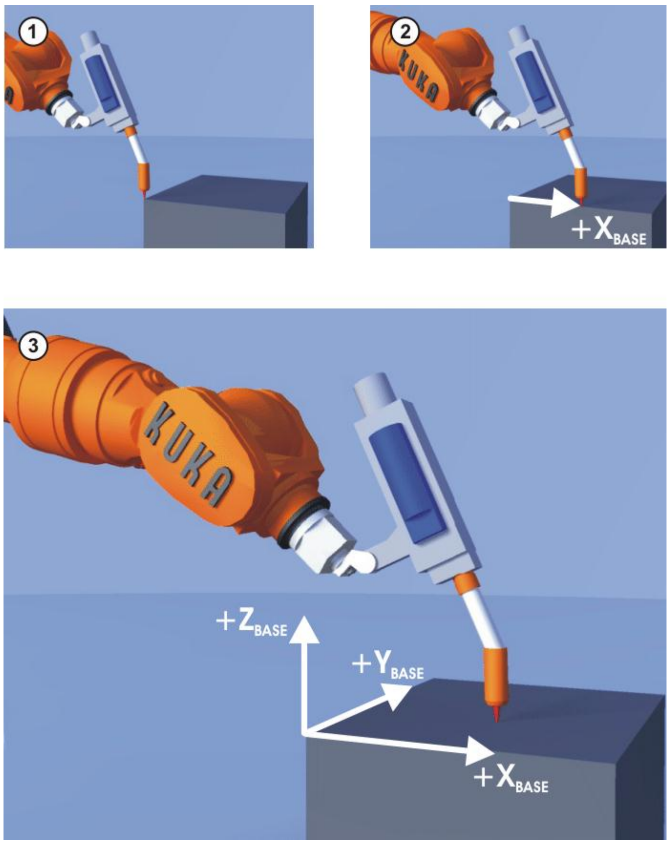


Figure E 2: Sequence of steps required for the calibration of a workspace (KUKA Robot Group, 2007).

E.2 KUKA KRL programs

The KUKA controller allows for the construction of customised control programs in the KRL software platform. The code of three constructed programs is presented in this section.

E.2.1 *MAIN( )*

```
DEF MAIN( )
;--------------------------------------------
; this program controls the pick and place
; actions of the robot by obtaining coordinate
; data through serial communication and then
; selecting appropriate robot motion sets.
;--------------------------------------------
;---initialization---
   MW_T=#ASYNC   ;---ASYNC -> does not wait for empty buffer
   MR_T=#ABS    ;---not sure between ABS or COND
   TIMEOUT = 30.0
   REC_DATA[] = "          "
   POSX[] = "000000"
   POSY[] = "000000"
   POSZ[] = "000000"
   ANG[] = "000000"

   PLACE_POSX[] = "000000"
   PLACE_POSY[] = "000000"
   PLACE_POSZ[] = "000000"
   PLACE_ANG[] = "000000"

   POS_X = 0.0
   POS_Y = 0.0
   POS_Z = 0.0
   ANG_R = 0.0

   PLACEPOS_X = 0.0
   PLACEPOS_Y = 0.0
   PLACEPOS_Z = 0.0
   PLACE_ANGLE = 0.0

   COMMAND[] = "    "

   LOAD = FALSE
   REMOVE = FALSE
   CLEAR = FALSE

   HANDLE = 0
   TEST = 1
   DONE[] = "FALSE"
;---------------------
;---main program---
;---------------------

   HANDLE = OPEN_CHNL(3)

   LOOP

   GET_COORDS()
   POS_X = CONVERT_S2R(POSX[])
   POS_Y = CONVERT_S2R(POSY[])
   POS_Z = CONVERT_S2R(POSZ[])
   ANG_R = CONVERT_S2R(ANG[])

   PLACEPOS_X = CONVERT_S2R(PLACE_POSX[])
   PLACEPOS_Y = CONVERT_S2R(PLACE_POSY[])
   PLACEPOS_Z = CONVERT_S2R(PLACE_POSZ[])
   PLACE_ANGLE = CONVERT_S2R(PLACE_ANG[])

   LOAD = STRCOMP(COMMAND[], "LOAD", #CASE_SENS)
   REMOVE = STRCOMP(COMMAND[], "RMVE", #CASE_SENS)

   IF LOAD == TRUE THEN

      PART_PICKUP(PART_ID,POS_X,POS_Y,POS_Z,ANG_R)

      PART_PLACE(PART_ID,PLACEPOS_X,PLACEPOS_Y,PLACEPOS_Z,PLACE_ANGLE)
      GOTO NEXT
   ENDIF

IF REMOVE == TRUE THEN

   PART_REMOVE(PART_ID,POS_X,POS_Y,POS_Z,ANG_R)

   GOTO NEXT
ENDIF
```

```
   NEXT:

   IF ($IN_HOME1 == TRUE) OR ($IN_HOME2 == TRUE) OR ($IN_HOME3 == TRUE) THEN
      DONE[] = "TRUE"
   ELSE
      HALT
   ENDIF

   SERIAL_WRITE(DONE[], HANDLE)
   CLEAR = STRCLEAR(REC_DATA[])
   ;CLEAR = STRCLEAR($DATA_SER3)
   WAIT FOR $DATA_SER3 == 0

   WAIT SEC 1

   ENDLOOP

   HANDLE = CLOSE_CHNL(3,HANDLE)

END

DEF GET_COORDS()
   ;---read coordinate string---
   OFFSET = 0   ;---read from first character
   WAIT FOR $DATA_SER3 > 0
      CREAD(HANDLE, SR_T, MR_T,TIMEOUT, OFFSET, "%S", REC_DATA[])
   IF (SR_T.RET1 <> #DATA_END) THEN
      HALT
   ENDIF
   ;--------------------------------
   ;---break up coordinate string---
   ;--------------------------------

   ;initialise counters
   COUNT = 1
   SPEC = 1
   OFFSET_1 = 0
   CMD = 0

   PX = 0
   PY = 0
   PZ = 0
   PA = 0

   PPX = 0
   PPY = 0
   PPZ = 0
   PPA = 0

   WHILE COUNT <= SR_T.LENGTH
      IF REC_DATA[COUNT] == 'H23' THEN
         SPEC = SPEC + 1
         COUNT = COUNT + 1
         OFFSET_1 = OFFSET_1 + 1
      ENDIF
      SWITCH SPEC
      CASE 1
         CMD = CMD + 1
         SREAD(REC_DATA[],STAT,OFFSET_1,"%01s", COMMAND[CMD])
      CASE 2
         SREAD(REC_DATA[],STAT, OFFSET_1,"%01d", PART_ID)
      CASE 3
         PX = PX + 1
         SREAD(REC_DATA[],STAT, OFFSET_1,"%01s", POSX[PX])
      CASE 4
         PY = PY + 1
         SREAD(REC_DATA[],STAT, OFFSET_1,"%01s", POSY[PY])
      CASE 5
         PZ = PZ + 1
         SREAD(REC_DATA[],STAT, OFFSET_1,"%01s", POSZ[PZ])
      CASE 6
         PA = PA + 1
         SREAD(REC_DATA[],STAT, OFFSET_1,"%01s", ANG[PA])
      CASE 7
         PPX = PPX + 1
         SREAD(REC_DATA[],STAT, OFFSET_1,"%01s", PLACE_POSX[PPX])
      CASE 8
         PPY = PPY + 1
         SREAD(REC_DATA[],STAT, OFFSET_1,"%01s", PLACE_POSY[PPY])
      CASE 9
         PPZ = PPZ + 1
         SREAD(REC_DATA[],STAT, OFFSET_1,"%01s", PLACE_POSZ[PPZ])
      CASE 10
         PPA = PPA + 1
         SREAD(REC_DATA[],STAT, OFFSET_1,"%01s", PLACE_ANG[PPA])
      CASE 11
         ;reached end of the string
      DEFAULT
         HALT
      ENDSWITCH
      COUNT = COUNT + 1
   ENDWHILE

END
```

E.2.2 *PICKUP_PART( )*

```
DEF PICKUP_PART1(POS_X: IN,POS_Y: IN,POS_Z: IN,ANG_R:IN)

;---declaration---
REAL POS_X,POS_Y,POS_Z,ANG_R
REAL S_PREP  ;specified offset distance
REAL X_PREP, Y_PREP, ANG_PREP
EXT BAS(BAS_COMMAND :IN, REAL :IN)
DECL FRAME PICK_POS
DECL FRAME PREP_POS
DECL FRAME ORIENT
;---------------------
;---initialization---
;---------------------
BAS(#INITMOV,0)

PICK_POS = {X 0,Y 0,Z 0,A 0,B 0,C 0}
PICK_POS.X = POS_X
PICK_POS.Y = POS_Y
PICK_POS.Z = POS_Z

S_PREP = 40.0
ANG_PREP = 90 - ANG_R  ;calculate entry angle
X_PREP = S_PREP*SIN(ANG_PREP)  ;calculate x offset
Y_PREP = S_PREP*COS(ANG_PREP)  ;calculate y offset

PREP_POS = {X 0,Y 0,Z 0,A 0,B 0,C 0}
PREP_POS.X = PICK_POS.X + X_PREP
PREP_POS.Y = PICK_POS.Y - Y_PREP
PREP_POS.Z = PICK_POS.Z

ORIENT = {X 0,Y 0,Z 0,A 0,B 0,C 0}
ORIENT.A = ANG_PREP

$TOOL = TOOL_DATA[2]
$ORI_TYPE = #CONSTANT
DONE[] = "TRUE"
;-----------------------------
PTP $AXIS_HOME[1]

PTP {AXIS: A1 -99.5, A2 -74, A3 110,A4 0, A5 -36,A6 0}

LIN BASE_DATA[2]:PREP_POS
PTP_REL ORIENT

LIN BASE_DATA[2]:PICK_POS

SERIAL_WRITE(DONE[], 3)
GET_CONFIRM(3)

WAIT SEC 2

LIN BASE_DATA[2]:{X 0,Y 0,Z 100,A 0,B 0,C 0}

PTP $AXIS_HOME[1]

END
```

E.2.3 *PLACE_PART( )*

```
DEF PLACE_PART1(PLACEPOS_X: IN,PLACEPOS_Y: IN, PLACEPOS_Z: IN, PLACE_ANG: IN )

;---declaration---
REAL PLACEPOS_X,PLACEPOS_Y,PLACEPOS_Z,PLACE_ANG
REAL S_PREP  ;specified offset distance
REAL X_PREP, Y_PREP, ANG_PREP
EXT BAS(BAS_COMMAND :IN, REAL :IN)
DECL FRAME PLACE_POS
DECL FRAME PREP_POS
DECL FRAME ORIENT
;---------------------
;---initialization---
;---------------------
BAS(#INITMOV,0)

PLACE_POS = {X 0,Y 0,Z 0,A 0,B 0,C 0}
PLACE_POS.X = PLACEPOS_X
PLACE_POS.Y = PLACEPOS_Y
PLACE_POS.Z = PLACEPOS_Z

PREP_POS = {X 0,Y 0,Z 0,A 0,B 0,C 0}
PREP_POS.X = PLACE_POS.X
PREP_POS.Y = PLACE_POS.Y
PREP_POS.Z = PLACE_POS.Z + 80

ORIENT = {X 0,Y 0,Z 0,A 0,B 0,C 0}
ORIENT.A = PLACE_ANG
$TOOL = TOOL_DATA[2]
$ORI_TYPE = #CONSTANT
DONE[] = "TRUE"
;-----------------------------
```

```
PTP_REL ORIENT

LIN BASE_DATA[3]:PREP_POS

LIN BASE_DATA[3]:PLACE_POS

SERIAL_WRITE(DONE[], 3)
GET_CONFIRM(3)

LIN BASE_DATA[3]:PREP_POS

PTP $AXIS_HOME[2]

END
```

```
CHAR WRITE_STRING[]

INT HANDLE
INT COUNT
;---initialization---
MW_T = #ASYNC
COUNT = 1

;---program---
CWRITE(HANDLE,SW_T,MW_T,"%s",WRITE_STRING[])

IF (SW_T.RET1 <> #CMD_OK) THEN
   HALT
ENDIF

END
```

```
DEF GET_CONFIRM(HANDLE :IN)

   INT HANDLE
   COUNT = 1
   MR_T=#ABS    ;---not sure between ABS or COND
   TIMEOUT = 30.0
   CONFRM[] = "     "
   TRUESTRING[] = "TRUE"

   ;---read confirm string---
   OFFSET = 0   ;---read from first character
   WAIT FOR $DATA_SER3 > 0
      CREAD(HANDLE, SR_T, MR_T,TIMEOUT, OFFSET, "%S", CONFRM[])
   IF (SR_T.RET1 <> #DATA_END) THEN
      HALT
   ENDIF

   WHILE (COUNT < 5)
      IF (CONFRM[COUNT] <> TRUESTRING[COUNT]) THEN
         HALT
      ENDIF
      COUNT = COUNT + 1
   ENDWHILE

END
```

Appendix F: JADE agent program example

As an example of a JADE agent program, the Java code of the Camera agent program is presented in this appendix.

```
//==========================//
//====== Camera Agent ======//
//==========================//

//Imports
import java.io.IOException;
import java.io.StringReader;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.logging.Level;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.*;
import org.xml.sax.*;
import org.apache.ecs.xml.*;

import jade.content.*;
import jade.content.lang.Codec.CodecException;
import jade.content.lang.sl.*;
import jade.content.lang.*;
import jade.content.lang.xml.*;
import jade.content.onto.Ontology;
import jade.content.onto.OntologyException;
import jade.content.onto.basic.*;
import jade.core.*;
import jade.core.behaviours.CyclicBehaviour;
import jade.core.behaviours.OneShotBehaviour;
import jade.domain.DFService;
import jade.domain.FIPAException;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.util.Logger;

import ontology.impl.*;
import ontology.FeedingMultiagentOntology;

public class CameraAgent extends Agent {

       // definition of codecs and ontology
       private Codec slCodec= new SLCodec();
       private Codec xmlCodec=new XMLCodec();
       private Ontology ontology;
       // definition of network communication
       private InetAddress IPaddress = null;
       private int port = 0;
       private Socket clientSocket = new Socket();

       private Duration duration = new Duration();
       private Integer time = 10; //dummy time variable

       protected void setup(){

              System.out.println("New camera agent created.");
```

```
             // ontology instantiation
             try{
            ontology=FeedingMultiagentOntology.getInstance();
        }
        catch (Exception oe){
            oe.printStackTrace();
        }

             // register codecs and ontology
             getContentManager().registerLanguage(slCodec);
             getContentManager().registerOntology(ontology);

             // register agent services with the Directory Facilitator
             DFAgentDescription dfd = new DFAgentDescription();
             dfd.setName(getAID());
             ServiceDescription sd = new ServiceDescription();
             sd.setType("Camera");
             sd.setName(getLocalName()+"Camera");
             dfd.addServices(sd);
             try{
                    DFService.register(this, dfd);
             }
             catch (FIPAException fe){
                    fe.printStackTrace();
             }

             // get IP address and port number of camera effector
             try {
            IPaddress = InetAddress.getLocalHost(); //IP address of Camera LLC
program
            port = 7220; //listening port of Camera LLC program

        } catch (UnknownHostException ex) {
            Logger.getLogger(CameraAgent.class.getName()).log(Level.SEVERE, null,
ex);
        }

             //=== Agent Behaviour ===
        addBehaviour(new requestReceiver());
        addBehaviour(new actionPerformer());

       }

       protected void takeDown(){
             // Deregister from the yellow pages
        try {
            DFService.deregister(this);
            }
        catch (FIPAException fe) {
            fe.printStackTrace();
            }

        System.out.println("Camera "+getAID().getName()+" terminating.");
       }

private class requestReceiver extends CyclicBehaviour{

             public void action(){

                    MessageTemplate mt =
MessageTemplate.MatchPerformative(ACLMessage.CFP);
                    ACLMessage msg = myAgent.receive(mt);
                    if(msg != null){
                          System.out.println("CFP received.");
                          ACLMessage reply = msg.createReply();

                          if(time != null){
                                reply.setPerformative(ACLMessage.PROPOSE);
                                reply.setContent(String.valueOf(time.intValue()));
                          }
                          else{
                                reply.setPerformative(ACLMessage.REFUSE);
                          }
                          myAgent.send(reply);
                          System.out.println("Camera Agent sent Proposal to 
requesting Agent.");
                    }
                    else{
                          block();
                    }
              }
        }

        private class actionPerformer extends CyclicBehaviour{

              private String action = "";
              private String dataIn = null;

              public void action(){

                    MessageTemplate mt = 
MessageTemplate.MatchPerformative(ACLMessage.ACCEPT_PROPOSAL);
                    ACLMessage msg = myAgent.receive(mt);

                    if(msg != null){
                          System.out.println("Camera received ACCEPT_PROPOSAL from 
requesting Agent.");
                          addBehaviour(new taskInspect(msg));
                          block();
                    }
                    else{
                          block();
                    }
              }
        }

        private class taskInspect extends OneShotBehaviour{

              private Position pos = new Position();
              private PlacePosition placepos = new PlacePosition();
              ACLMessage msg = null;

              public taskInspect(ACLMessage inMsg){
                    super();
                    msg = inMsg;
              }

              public void action(){
                    if (msg == null){
                          this.done();
                    }

                    try{        //when message is !null
                          ContentElement ce = null;
                          //extract the content of the message
                          ce = getContentManager().extractContent(msg);
                          Action act = (Action) ce;

                          if (act.getAction() instanceof Inspect){        //confirms
that action is "Inspect"

                                if (msg.getPerformative() ==
ACLMessage.ACCEPT_PROPOSAL){      //if the proposal was accepted by the Task Agent

                                      ACLMessage reply = msg.createReply();

                                      Inspect task = (Inspect) act.getAction();
      //instantiation to get required part type
                                      Part pa = (Part) task.getPart();

                                      if (inspectPart(pa)){

      reply.setPerformative(ACLMessage.INFORM);
                                            //reply.setContentObject( (Position)
pos);
                                            try {
                                                  PicknPlace result = new
PicknPlace(); //initialize task with part type and duration
                                      result.setPart(pa);
                                      result.setDuration(duration);
                                      result.setPosition(pos);
                                      result.setPlacePosition(placepos);

                                      Action actn = new Action();
                                      actn.setAction(result);
                                      actn.setActor(myAgent.getAID());
                                      getContentManager().fillContent(reply,
actn);
                                     }
                                     catch (CodecException Ce) {
                                         Ce.printStackTrace();
                                     }
                                     catch (OntologyException oe) {
                                         oe.printStackTrace();
                                     }
                                            System.out.println("Inspection data
sent to requesting Agent by Camera Agent...");
                                      }
                                      else{

      reply.setPerformative(ACLMessage.FAILURE);
                                            System.out.println("Failure
notification sent by Camera Agent...");
                                      }
                                      send(reply);
                                }
                                else{  //cannot understand message -> wrong type!
                                      ACLMessage reply = msg.createReply();
//create reply
                        reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
                        send(reply);
                                }
                          }
                          else{  //an error occurred with the extraction of the
message content
                                System.out.println("No message could be
extracted!");
                          }
```

```
                    }
                    catch (Exception ce){
                          ce.printStackTrace();
                    }
              }

              private Boolean inspectPart(Part pa){
                    String dataIn = null;
                    //an xml message must now be sent to the camera to initiate the
inspection
                    //composing xml string using Jakarta ECS
                    XML task = new XML("TASK");
                  task.addElement("INSPECT");  //should be INSPECT

                  XML part = new XML("PART_TYPE");
                  part.addElement(String.valueOf(pa.getName()));

                  XML reciever = new XML("CAMERA");
                  reciever.addElement(task);
                  reciever.addElement(part);

                  XML inspect = new XML(getAID().getLocalName()); //creates xml that
starts with name of agent (based on xml standard in this program)
                  inspect.addElement(reciever);

                  XMLDocument doc = new XMLDocument();
                  doc.addElement(inspect);

                //check whether network socket is still connected
               try{
                   System.out.println("IP address: " + IPaddress.toString() +"
Port: " + port);

                   clientSocket = new Socket (IPaddress, port);
               }
               catch (Exception e){
                   e.printStackTrace();
               }

               while(!clientSocket.isConnected()){

               }

               System.out.println("Client socket connected...");

               //now send the composed message
               try{
                    System.out.println("Sending message to camera effector to
inspect part...");

                    byte[] outByteString = doc.toString().getBytes("UTF-8");
       //set format
                    clientSocket.getOutputStream().write(outByteString, 0,
outByteString.length);
                    System.out.write(outByteString); //trying to print what is sent
to camera
                    System.out.println("Message sent to camera effector...");

                   byte[] inByteString = new byte[300] ;
                   int numOfBytes =
clientSocket.getInputStream().read(inByteString);
                   String inString = new String(inByteString, 0, numOfBytes, "UTF-
8");
```

```
                dataIn = inString;

                System.out.println("Recieved string of length: " + numOfBytes);

                System.out.println(inString);

                clientSocket.close();
            }
            catch (IOException io){
                  io.printStackTrace();
            }

            String oc = ParseXMLString(dataIn, "DONE");   //parsing of inspection
result data
            Boolean inspected = Boolean.parseBoolean(oc);

            if (inspected){
                  float x = Float.valueOf(ParseXMLString(dataIn, "X")); //parses
xml string for value of X
                float y = Float.valueOf(ParseXMLString(dataIn, "Y")); //parses
xml string for value of Y
                float z = Float.valueOf(ParseXMLString(dataIn, "Z")); //parses
xml string for value of Z
                float angle = Float.valueOf(ParseXMLString(dataIn, "ANGLE"));
//parses xml string for value of ANGLE

                pos.setXPos(x);
                pos.setYPos(y);
                pos.setZPos(z);
                pos.setAngle(angle);
            }
            return inspected;
          }

          public String ParseXMLString (String xmlRecords, String findText){

        String xmlStart = "CAMERA";
        String stringToReturn = "";

        try {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            InputSource is = new InputSource();
            is.setCharacterStream(new StringReader(xmlRecords));
            Document doc = db.parse(is);
            NodeList nodes = doc.getElementsByTagName(xmlStart);

            // iterate the entries
            for (int i = 0; i < nodes.getLength(); i++) {
                Element element = (Element) nodes.item(i);
                NodeList name = element.getElementsByTagName(findText);
                Element line = (Element) name.item(0);
                System.out.println(findText +
getCharacterDataFromElement(line));

                stringToReturn = getCharacterDataFromElement(line).toString();

            }

        }
        catch (Exception e) {
            e.printStackTrace();
        }
```

```
            return stringToReturn;
        }

        public String getCharacterDataFromElement(Element e) {
            org.w3c.dom.Node child = e.getFirstChild(); //Node formula classes with
jade.core.Node
            if (child instanceof CharacterData) {
                CharacterData cd = (CharacterData) child;
                return cd.getData();
            }
            return "?";
        }        //method to parse xml strings ends here

       }

}
```

Appendix G: IEC 61499 function block networks

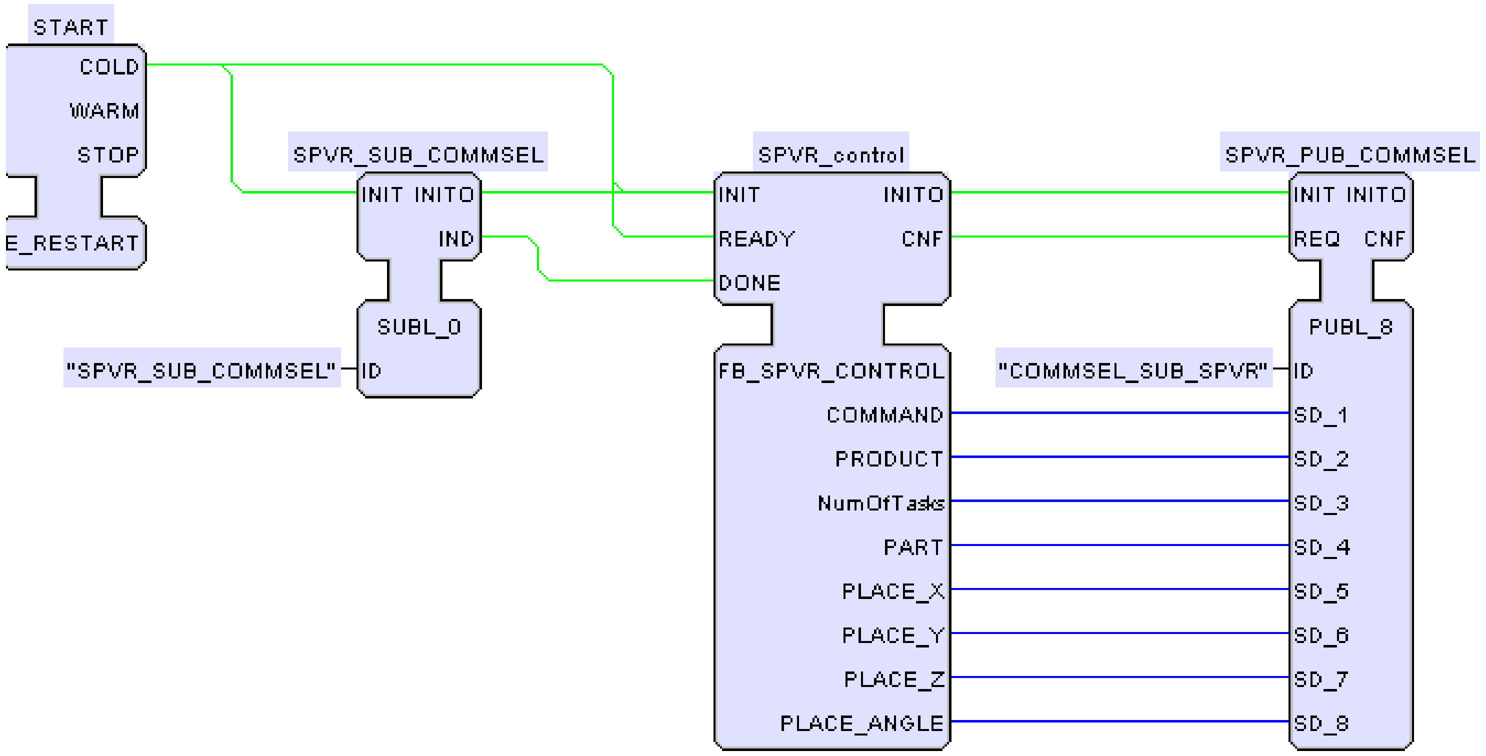


Figure G 1: Function block network of the FB_SUPERVISOR device.

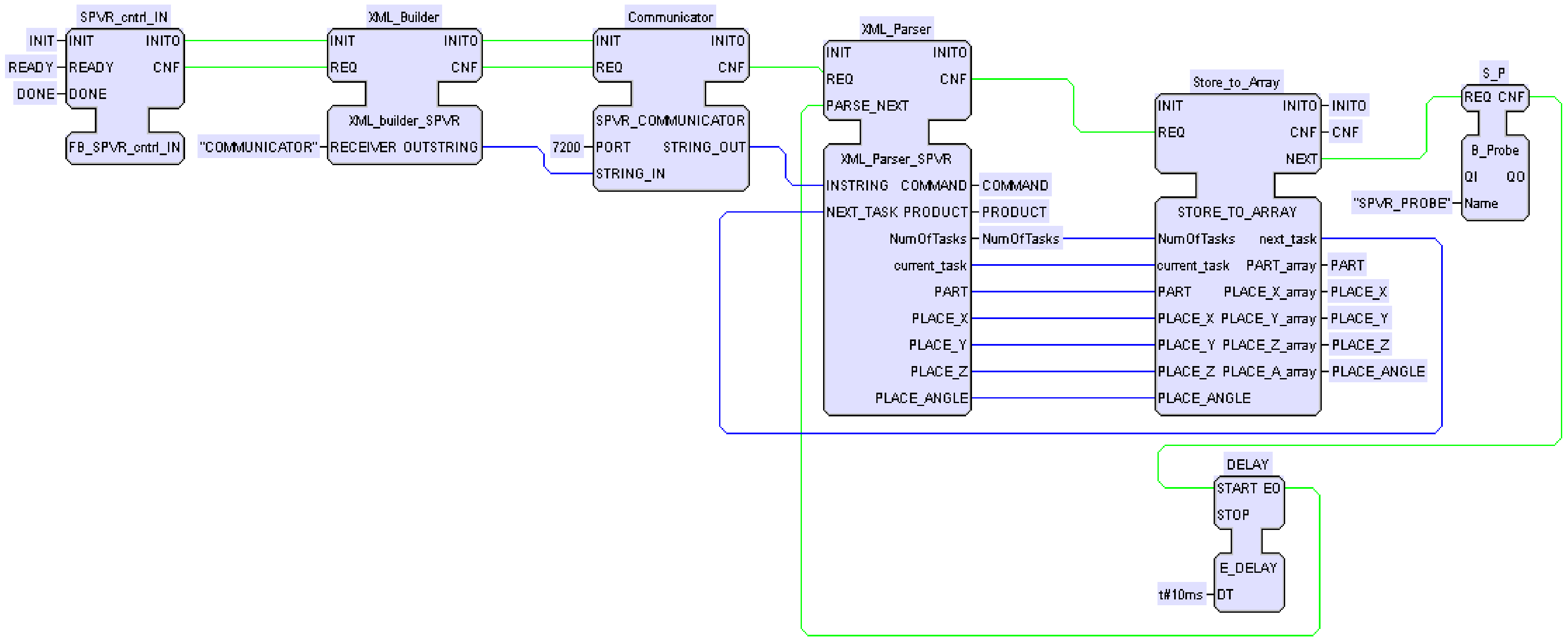


Figure G 2: Function block network of FB_SPVR_CONTROL composite function block.

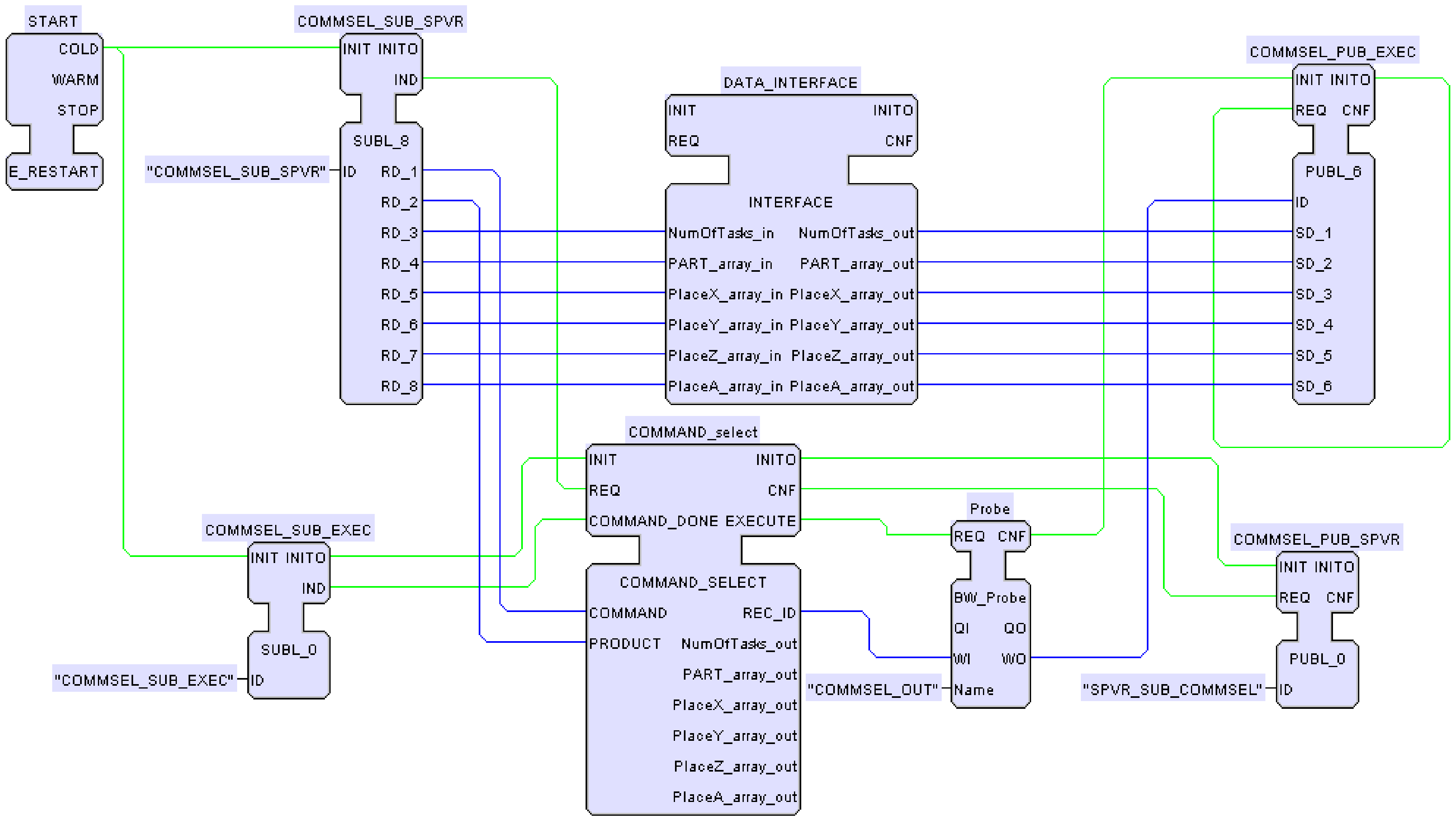


Figure G 3: Function block network of the COMMAND_SELECT resource.

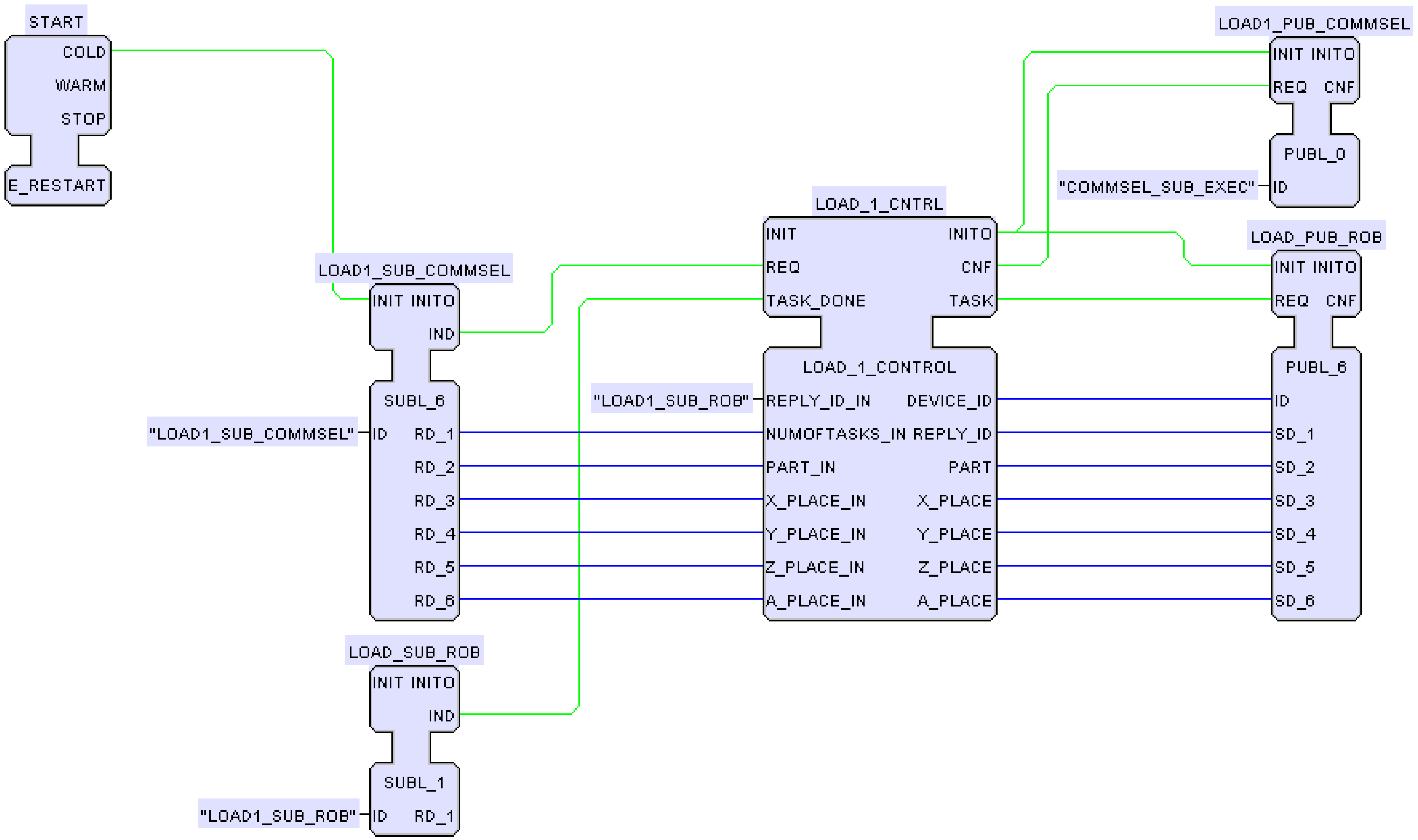


Figure G 4: Function block network of the LOAD_1 resource.

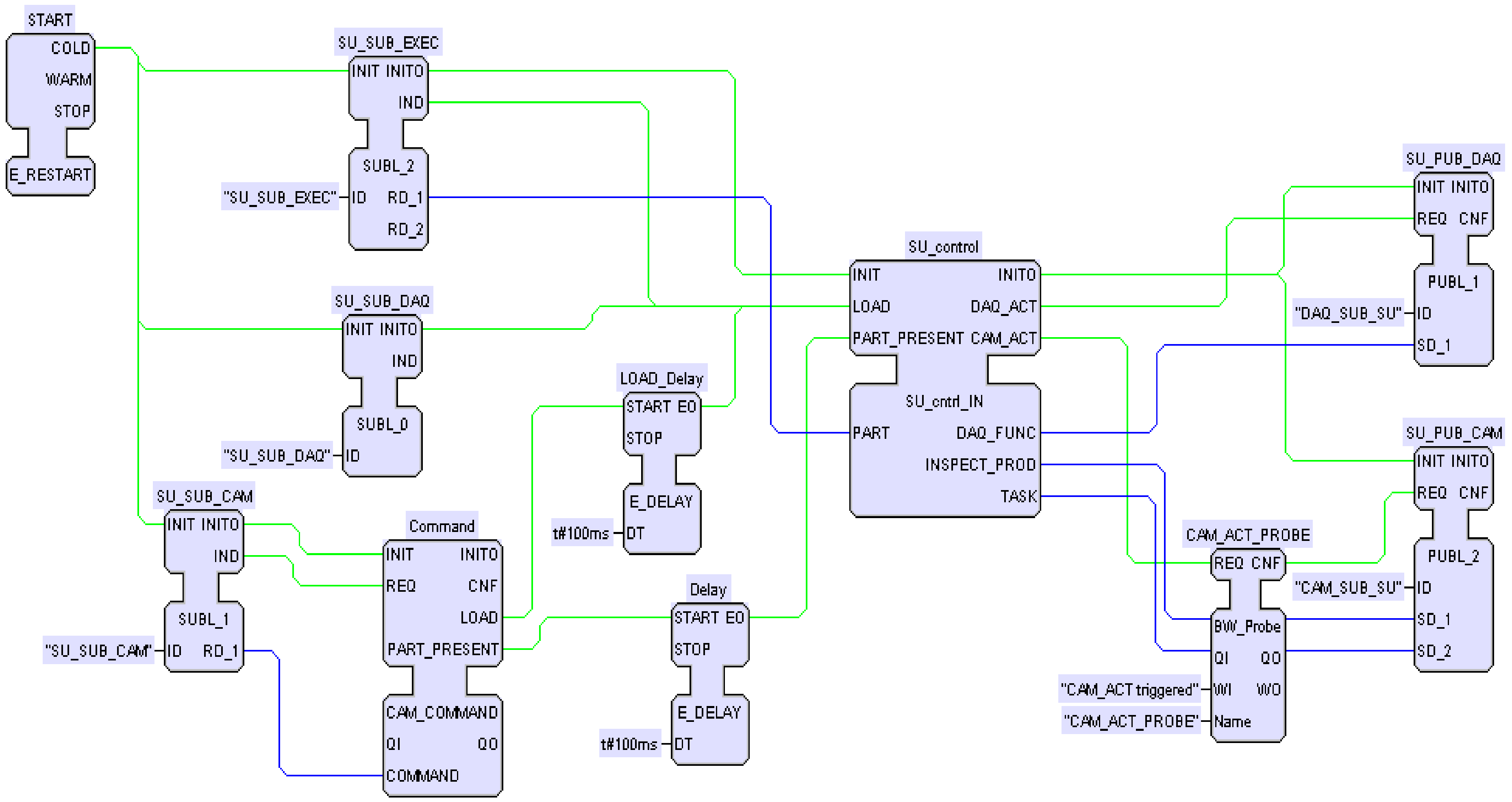


Figure G 5: Function block network of SINGULATION_UNIT device.

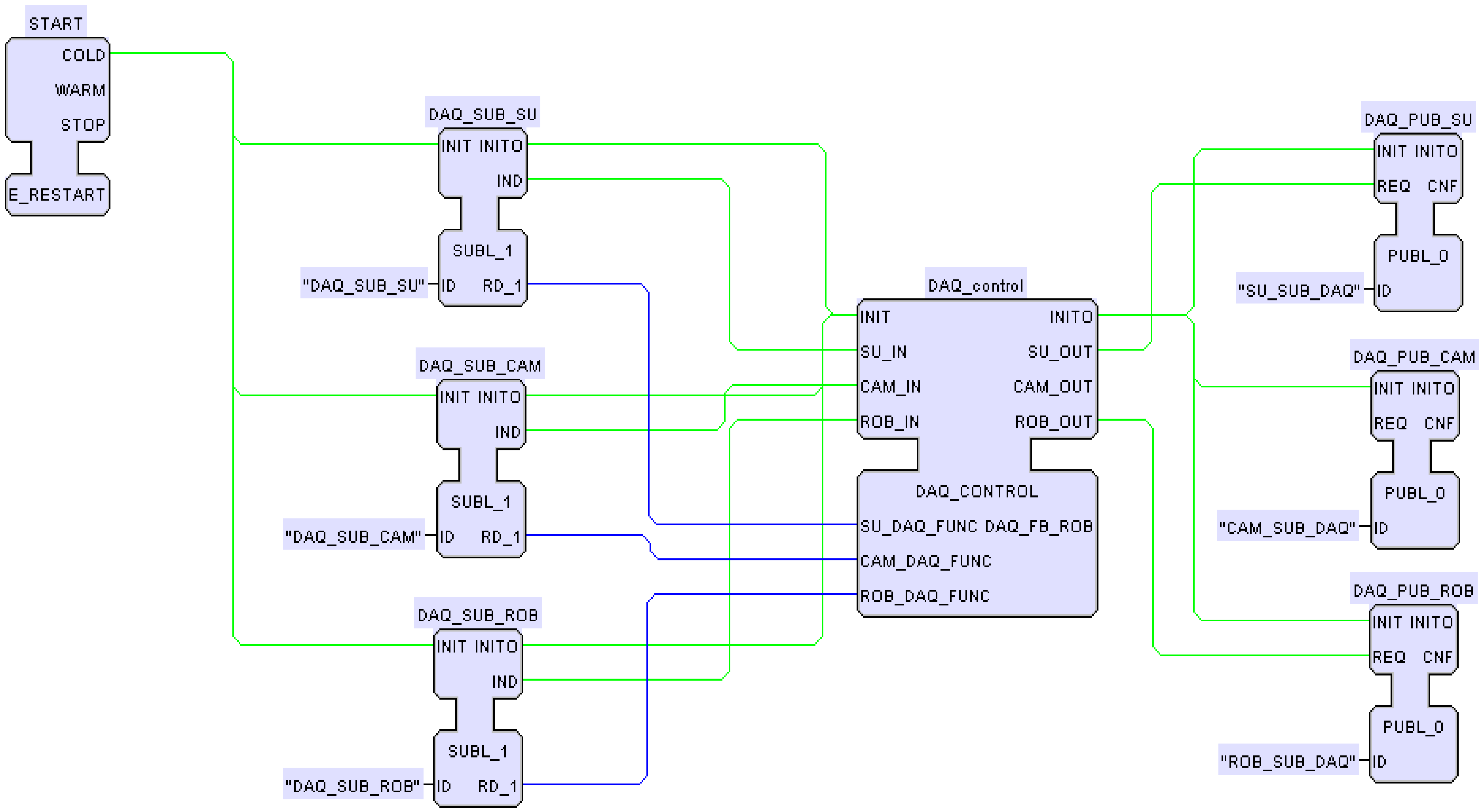


Figure G 6: Function block network of DAQ device.

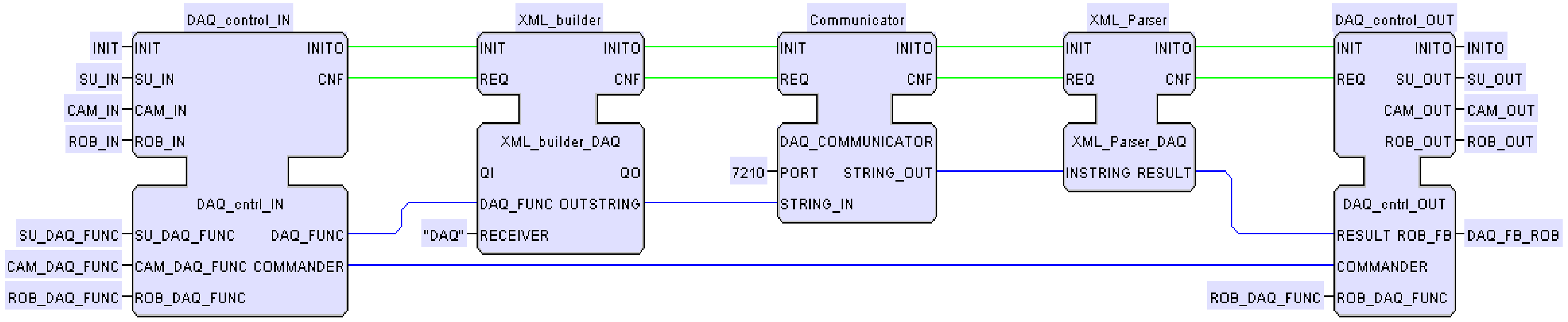


Figure G 7: Function block network of DAQ_CONTROL composite function block.

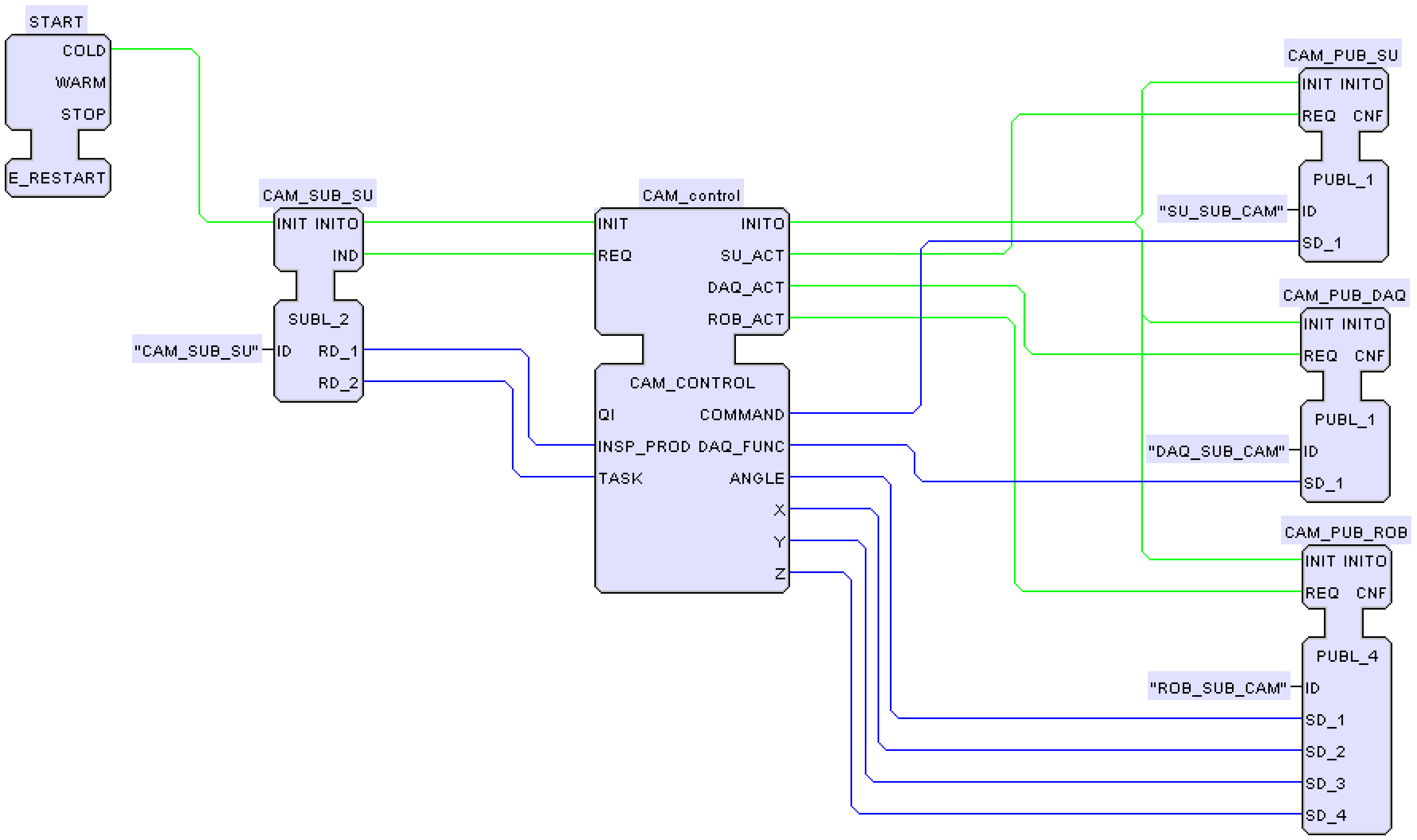


Figure G 8: Function block network of CAMERA device.

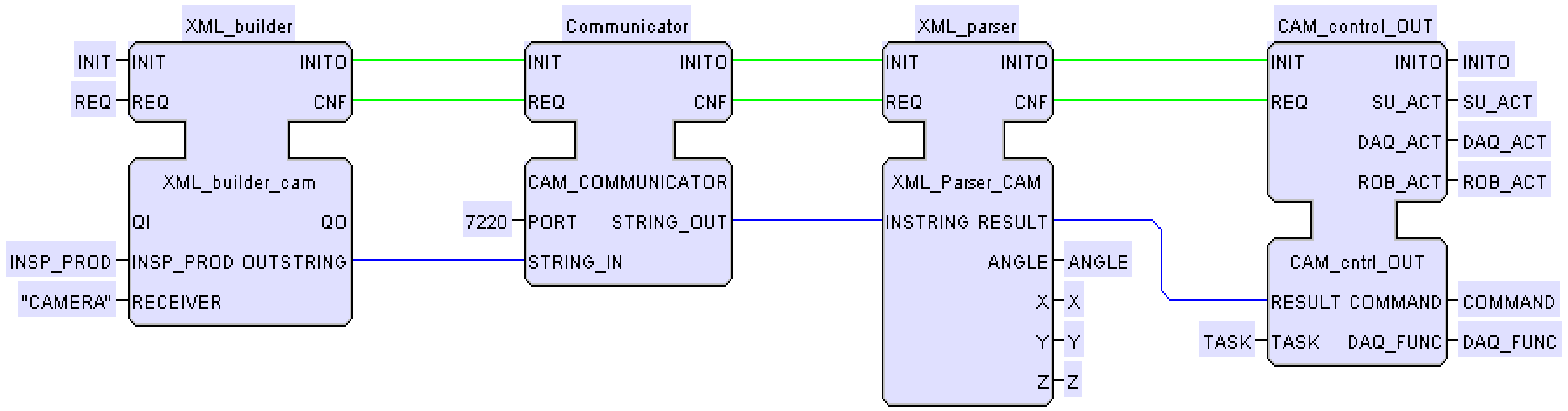


Figure G 9: Function block network of CAM_CONTROL composite function block.

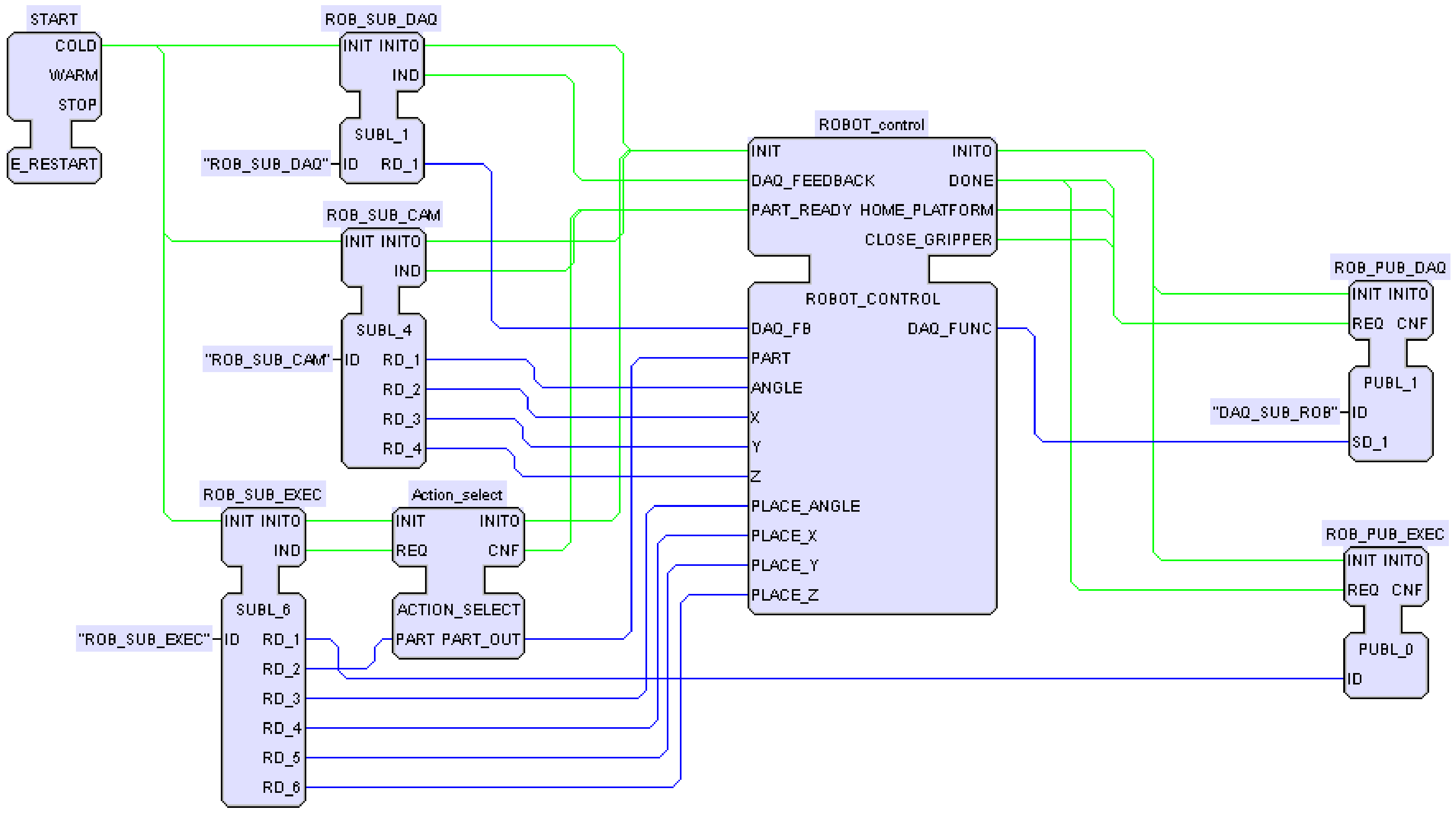


Figure G 10: Function block network of the ROBOT device.

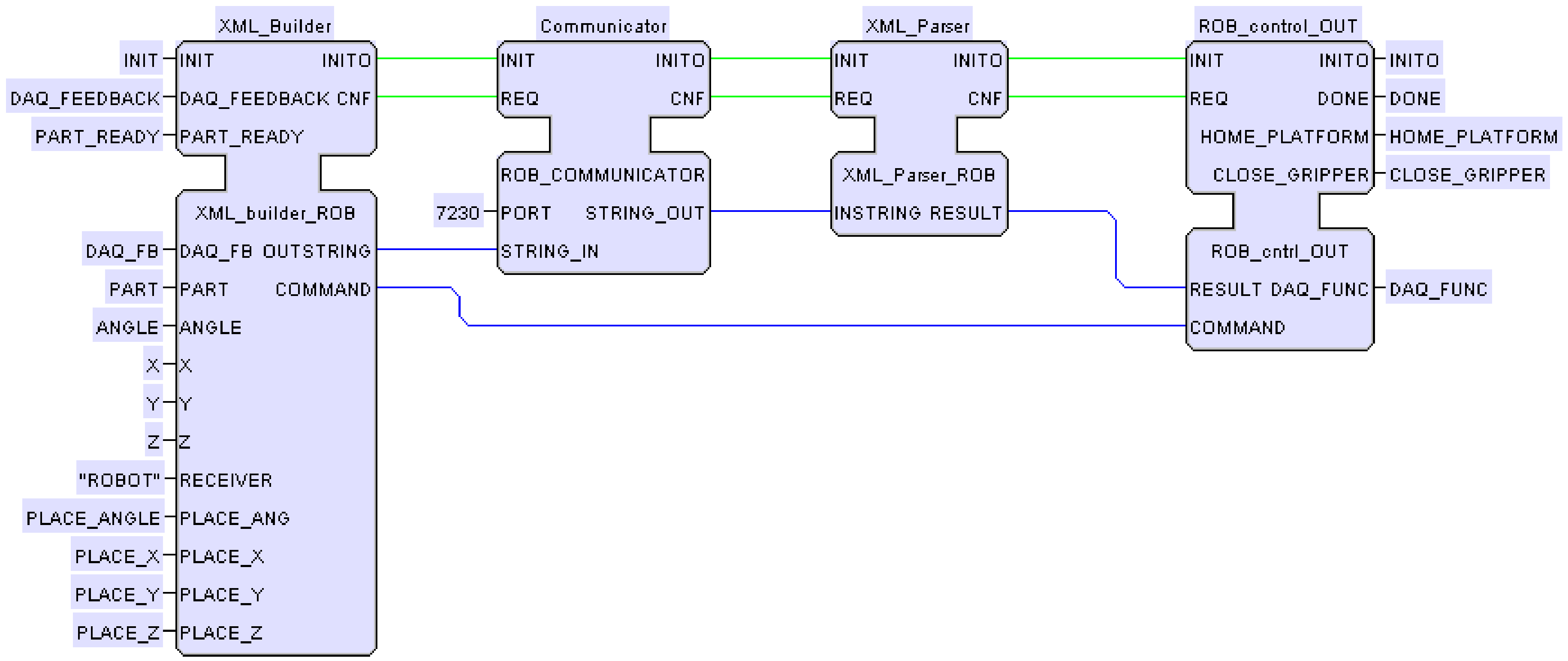


Figure G 11: Function block network of the ROBOT_CONTROL composite function block.