Development and Evaluation of an Erlang Control System for Reconfigurable Manufacturing Systems

by
Karel Kruger

Dissertation presented for the degree of Doctor of Engineering in the Faculty of Engineering at Stellenbosch University

Supervisor: Prof Anton Herman Basson

March 2018

Plagiarism Declaration

By submitting this dissertation 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.

This dissertation includes four original papers published in peer-reviewed journals or books and three unpublished publications. The development and writing of the papers (published and unpublished) were the principal responsibility of myself and, for each of the cases where this is not the case, a declaration is included in the dissertation indicating the nature and extent of the contributions of co-authors.

Date: March 2018

UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Plagiaatverklaring / Plagiarism *Declaration*

1 Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2 Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
I agree that plagiarism is a punishable offence because it constitutes theft.

3 Ek verstaan ook dat direkte vertalings plagiaat is.
I also understand that direct translations are plagiarism.

4 Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.

5 Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

15127303 **Studentenommer / *Student number***	**Handtekening / *Signature***
K. Kruger **Voorletters en van / *Initials and surname***	29 January 2018 **Datum / *Date***

Abstract

Development and Evaluation of an Erlang Control System for Reconfigurable Manufacturing Systems

K. Kruger

Department of Mechanical and Mechatronic Engineering
Stellenbosch University
Private Bag X1, 7602 Matieland, South Africa
Dissertation: Ph.D. (Mechatronic Engineering)
March 2018

The dynamic and highly competitive nature of the modern manufacturing environment has introduced a new set of challenges, urging researchers and industry to formulate new and innovative solutions. The concepts of holonic and reconfigurable manufacturing systems showed great promise to address the challenges. While these concepts could not achieve significant industry adoption, they will play an important role in the latest emerging paradigm in manufacturing – the fourth industrial revolution, or Industry 4.0.

Industry 4.0, can potentially have a significant impact on all aspects of the manufacturing industry, aiming to enhance individualization of products through highly flexible production, extensively integrate customers and businesses in value-added processes and link production and high-quality services to deliver hybrid products. To achieve these goals, Industry 4.0 relies on Cyber-Physical Production Systems (CPPSs) to enhance the connectedness throughout all levels of the manufacturing enterprise. CPPSs aim to enhance the intelligence, connectedness and responsiveness of manufacturing systems. These goals closely resemble those of holonic and reconfigurable manufacturing systems, indicating the relevance of research on these topics to the development and implementation of CPPSs.

The objective of this dissertation is to evaluate the suitability of the Erlang programming language as an alternative for the implementation of holonic control in manufacturing systems. The dissertation presents an Erlang-based holonic control implementation for a manufacturing cell. The Erlang implementation is evaluated through a comparison with an equivalent implementation using Multi-Agent Systems (MASs), which is considered as the *status quo* for holonic control implementation in manufacturing systems research.

To accomplish the evaluation of the holonic control implementations, evaluation criteria is formulated. The evaluation criteria focusses on both the development of control implementations and the adoption of the implementations by industry. The criteria identifies a set of quantitative and qualitative performance measures that are indicative of seven critical requirements for holonic control implementations. The Erlang and MAS implementations are evaluated and compared according to these performance measures and requirements.

The comparison shows that the Erlang implementation matches the functionality of the MAS implementation and even offers some advantages for the desired characteristics for the holonic control of manufacturing systems. The advantages in availability and supportability can be attributed to the enhanced modularity and fault tolerance of the Erlang implementation. The Erlang implementation also allows for increased development productivity through a reduction in software complexity and simplification of software verification.

The findings of the evaluation confirms the inherent suitability of the Erlang programming language for the implementation of holonic control. It is recommended that further research be conducted on the refinement of the architecture and the development of a framework for holonic control implementations in Erlang.

Uittreksel

Ontwikkeling en Evaluering van ‘n Erlang Beheerstelsel vir Herkonfigureerbare Vervaardigingstelsels

K. Kruger

Departement van Meganiese and Megatroniese Ingenieurswese
Universiteit Stellenbosch
Privaatsak X1, 7602 Matieland, Suid-Afrika

Proefskrif: Ph.D. (Megatroniese Ingenieurswese)
Maart 2018

‘n Nuwe stel uitdagings, wat na vore gebring is deur die dinamiese en hoogs-kompeterende aard van die moderne vervaardiging omgewing, spoor navorsers en die bedryf aan om nuwe en innoverende oplossings te formuleer. Die konsepte van holoniese en herkonfigureerbare stelsels het beloof om hierdie uitdagings aan te spreek. Alhoewel hierdie konsepte nie beduidende bedryfsaanneming kon behaal nie, het dit ‘n belangrike rol om te speel in die nuutste ontluikende paradigma in vervaardiging – die vierde industriële revolusie, of *Industry 4.0*.

Industry 4.0 het die potensiaal om ‘n beduidende impak te hê op alle aspekte van die vervaardigingsbedryf, deur die individualisering van produkte te verbeter met hoogs-buigsame produksie, breedvoerig kliënte en besighede in waarde-toevoegingsprosesse te integreer en produksie met hoë-kwaliteit dienste te verbind om hibriede produkte af te lewer. Om hierdie doelwitte te bereik maak *Industry 4.0* staat op Kuber-Fisiese Produksiestelsels (KFPs) om verbondenheid tussen al die vlakke van ‘n vervaardigingsonderneming te verbeter. KFPs beoog om die intelligensie, verbondenheid en responsiwiteit van vervaardigingstelsels te verbeter. Die doelwitte van KFPs stem ooreen met die van holoniese en herkonfigureerbare vervaardigingstelsels, wat die relevansie van die benaderings op die ontwikkeling en implementering van KFPs aandui.

Die doelwit van hierdie proefskrif is om die geskiktheid van die Erlang programmeringstaal, as ‘n alternatief vir die implementering van holoniese beheer in vervaardigingstelsels, te evalueer. Die proefskrif beskryf ‘n Erlang-gebaseerde beheerimplementering vir ‘n vervaardigingsel. Die Erlang implementering is evalueer deur middel van ‘n vergelyking met ‘n ekwivalente implementering wat gebruik maak van ‘n Multi-Agent Stelsel (MAS), wat beskou word as die *status quo* vir holoniese beheerimplementering in vervaardigingstelsel navorsing.

‘n Evalueringkriteria vir holoniese beheerimplementering is geformuleer om die evaluering te vervul. Die evalueringkriteria fokus op beide die ontwikkeling van beheerimplementerings en die aanneming daarvan deur die bedryf. Die kriteria identifiseer ‘n stel kwantitatiewe en kwalitatiewe prestasiemaatreëls wat aanduiding gee vir sewe kritiese vereistes vir holoniese beheerimplementerings.

Die Erlang en MAS implementerings is ge-evalueer en vergelyk volgens die prestasiemaatreëls en vereistes.

Die vergelyking wys dat die funksionaliteit van die Erlang implementering ooreenstem met die van die MAS implementering, en selfs voordele inhou vir die gewenste eienskappe vir holoniese vervaardigingstelsels. Die voordele van Erlang, ten opsigte van beskikbaarheid en ondersteunbaarheid, kan toegeskryf word aan verbeterde modulariteit en fout-verdraagsaamheid. Die Erlang implementering maak ook voorsiening vir verhoogde ontwikkelingsproduktiwiteit, deur die kompleksiteit van die sagteware te verminder en die verifikasie daarvan te vereenvoudig.

Die bevindinge van die evaluering bevestig die Erlang programmeringstaal se inherente geskiktheid vir die implementering van holoniese beheer. Dit word voorgestel dat verdere navorsing gedoen word op die verfyning van die argitektuur en die ontwikkeling van ‘n raamwerk vir holoniese beheerimplementering in Erlang.

Aan Jolise, Riejed, my familie en vriende –

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 the realization of this dissertation – the following people deserve special recognition for their contributions:

Prof. Anton Basson, for your continuous support, guidance and mentorship over the years. I have learned so much from you – not just from your advice, but also your example. I am truly grateful for the part that you played in the completion of this dissertation, and I hope that we can continue to do interesting work together in the years to come.

I am grateful for the support of the management of the Department of Mechanical and Mechatronic Engineering – for granting me the opportunity and funding to pursue a Ph.D. degree and for creating a stimulating working environment for me and many other students and staff members.

I am thankful for the love and support of my family and friends. My wife, for her love, patience and unwavering belief in me. My parents, for the passion and perseverance that they instilled in me and for continuing to be an example and inspiration. For everyone else, for all your positivity and encouragement.

Above all, I thank our heavenly Father for the infinite blessings that I am grateful for every day – none more so than the blessing of our daughter, Riejed.

Table of contents

List of Tables

List of Figures

List of Acronyms

ACL	-	Agent Communication Language
ADACOR	-	Adaptive Holonic Control Architecture
AMS	-	Agent Management System
CIM	-	Computer Integrated Manufacturing
CFP	-	Call For Proposal
CNP	-	Contract Net Protocol
CPS	-	Cyber-Physical System
CPPS	-	Cyber-Physical Production System
DF	-	Directory Facilitator
FBDK	-	Function Block Development Kit
FIPA	-	Foundation for Intelligent Physical Agents
FMS	-	Flexible Manufacturing System
FSM	-	Finite State Machine
HLC	-	High Level Control
HMS	-	Holonic Manufacturing System
IDE	-	Integrated Development Environment
IOT	-	Internet Of Things
JADE	-	Java Agent Development framework
LLC	-	Low Level Control
MAS	-	Multi-Agent System
OS	-	Operating System
OTP	-	Open Telecom Platform
PC	-	Personal Computer
PLC	-	Programmable Logic Controller
PROSA	-	Product-Resource-Order-Staff Architecture

RE	-	Rational Effect
RMS	-	Reconfigurable Manufacturing System
SLOC	-	Source Lines Of Code
TSS	-	Telecommunications Switching System
XML	-	Extensible Markup Language
WCF	-	Windows Communication Foundation

1. Introduction

This section provides the background and context for the presented research. The objectives of the research and the contributions of the dissertation are outlined, the importance of this endeavour is motivated and the methodology that was followed is described. Finally, the section presents an overview of the dissertation structure.

1.1. Background

The modern 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 functionality and technology.
- The ability to produce more product variants. This involves the enhancement of production versatility and customization 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 concepts of Reconfigurable Manufacturing Systems (RMSs) and Holonic Manufacturing Systems (HMSs) presented promising solutions to the modern challenges. Recently, ideas like Industry 4.0, Cyber-Physical Production Systems (CPPSs) and the Industrial Internet of Things (IIOT) promise to address the challenges of future manufacturing.

The application of the holonic systems architecture has been a popular approach to organize and implement the control of modern manufacturing systems. The implementation of holonic control is fundamental to HMSs, but also proved to be effective in enabling control reconfigurability in RMSs. Holonic control architectures offer several advantages – increased modularity, scalability and robustness, while reducing overall system complexity and cost.

Holonic control architectures have been most often implemented using Multi-Agent Systems (MASs) – to the extent where MASs implementations (specifically using the Java Agent Development (JADE) framework) have become the *status quo* in academic studies. MASs originated from the agent-oriented programming paradigm, which brought the theories and concepts of artificial intelligence into the realm of distributed systems (Bellifemine *et al.*, 2007). The similarity between a holon and a software agent was the initial driving factor for the use of MASs to implement holonic control architectures.

The Mechatronic, Automation and Design (MAD) Research Group, at the Department of Mechanical and Mechatronic Engineering of Stellenbosch University, has conducted research into modern manufacturing systems since 2006. Initial studies focussed on the conceptualization, design and control of RMSs, while further studies placed emphasis on the control and coordination of the subsystems in HMSs and RMSs. The studies formed part of two research projects on the design and control of a reconfigurable manufacturing system for electrical circuit breaker production. The first project (2006-2012) focussed on the development of an automated assembly and welding cell, while the subsequent project (2013-2015) considered an assembly and quality assurance cell.

This dissertation builds on the knowledge and experience obtained through the above-mentioned studies and contributed to the second research project, with the assembly and quality assurance cell being used as the case study for the presented research. However, it is the first study within the research group to focus on the use of the Erlang programming language for implementing holonic control in manufacturing systems.

1.2. Objectives and Contributions

The objective of the dissertation is to evaluate the suitability of the Erlang programming language for the implementation of holonic control in manufacturing systems. To this end, an Erlang-based holonic control implementation is performed for a manufacturing case study. The Erlang implementation is evaluated through a comparison with an equivalent MAS implementation, which is the academic standard for holonic control implementation in manufacturing systems research.

Holonic control can be implemented at several levels within a manufacturing enterprise – from high-level logistics and scheduling, to low-level machine control. The presented research focusses on the control implementation at the manufacturing cell level, where specified production orders are executed through the coordination of individual workstations. This dissertation considers the following definitions:

- A *workstation* is a collection of actuators and devices which work together to perform a specific task – e.g. feeding or welding. The control at this level involves the coordination of the various hardware actions to perform the desired task.
- A *cell* is a collection of workstations responsible for performing a specific set of production tasks. Control at cell-level involves the coordination of the various workstations, and the flow of material and information between them, to accomplish the production tasks.

A manufacturing cell resembles – and in some cases can be equivalent to – a manufacturing system. The manufacturing execution system entails the coordination of different manufacturing cells, and the flow of material and information between them, to produce complete product. Due to this similarity, the results and findings obtained from the manufacturing cell implementation can be extended to manufacturing execution systems. As a case study, the research considers an assembly and quality assurance cell for the production of electrical circuit breakers.

The development of an Erlang holonic control implementation requires a well-defined architecture. The implementation is thus based on the well-established Product-Resource-Order-Staff Architecture (PROSA). For the implementation of the PROSA holons, an internal architecture, that is specifically suited to implementation with Erlang, is developed. The internal architecture is based on the generic internal holon architectures presented in literature, but incorporates and exploits the inherent features of Erlang.

To evaluate the suitability of the Erlang holonic control implementation, a comparison is performed with an equivalent implementation based on a MAS. The MAS approach is regarded as the academic *status quo* for the implementation of holonic control in manufacturing systems. To ensure that the implementations exhibit equivalent functionality, which is required for a fair comparison, both implementations are based on the PROSA holonic reference architecture. The control implementations are evaluated and compared by means of evaluation criteria specifically formulated for the implementation of holonic control in manufacturing systems.

In order to achieve the above-mentioned objective, this dissertation offers the following original contributions:

- The formulation of an internal architecture for the implementation of a holon using Erlang.
- The implementation of holonic control for a manufacturing cell, using Erlang.
- The formulation of criteria to facilitate the evaluation of holonic control implementations in manufacturing systems.
- The evaluation and comparison of two equivalent holonic control implementations, using Erlang and a MAS, respectively.

1.3. Motivation

HMSs and RMSs have received wide academic attention for over two decades, with research performed in many aspects – from hardware design and configuration, to control and optimization. Several aspects of HMSs and RMSs have become well established within academic research, especially the implementation of control architectures using MASs. Even though MAS control implementation promise to realise the advantages of HMSs and RMSs, there exists very few industrial implementations to support its great academic acclaim. Almeida *et al.* (2010) identify several barriers to the industry adoption of MASs, of which the most relevant to this dissertation are:

- The complexity involved in the design and implementation of such systems.
- A shortage of quality measures to aid the design, validation and evaluation of such systems.
- The lack of support for MAS implementation in industrial controllers.

- Concerns regarding the scalability of MASs for large-scale implementations.
- The lack of standards and commercial products for the development of MASs for manufacturing system control.

Considering the above-mentioned issues, along with the emergence of Industry 4.0, it is a good time to reconsider the *status quo* and investigate alternatives. The manufacturing environment is currently undergoing another paradigm shift – the new paradigm promises to address modern manufacturing requirements (as mentioned in section 1.1) through enhanced system intelligence, connectedness and responsiveness (Monostori *et al.*, 2016). It is thus essential that the suitability of established and alternative approaches and technologies be evaluated for use in Industry 4.0.

This dissertation embraces this opportunity to evaluate an alternative to MASs for the implementation of holonic control in manufacturing systems. The use of Erlang, a highly concurrent, functional programming language, is presented as a possible alternative. Implementation using Erlang has the potential to satisfy the requirements of both the previous and emerging manufacturing paradigms, and can narrow the gap between academic research and industrial implementation. This is due to several advantages offered by the Erlang language (which are described in detail in section 2.4.3):

- **Industrial acceptance** – Erlang was developed by the research laboratory at Ericsson, and was subsequently implemented in some of Ericsson's products. Erlang is currently used in many leading software and telecommunication applications.
- **High productivity** – there are reports in literature that indicate that software development can be achieved much faster, and with fewer errors, with Erlang than other well-known languages (such as Java, C or C++).
- **High reliability** – the initial requirements for which Erlang was developed specified very high reliability and robustness, leading to the inclusion of important mechanisms at the architectural level of Erlang.
- **High maintainability** – Erlang allows for the updating of code without having to disturb the operation of a running program, allowing for bug fixes, updates and code changes to be performed without any system downtime.
- **High adaptability** – Erlang is characterised by advanced modularity and distribution, which are two enabling factors in achieving adaptability in system control.

These advantages can take academic research towards industrial implementation in different ways. Industrial system integrators will feel more confident to implement a technology that was developed, tested and used by a large, respected corporation such as Ericsson. Their confidence will be boosted even more by Erlang's renown for reliability. The increased development productivity will allow for faster software development – which, together with the maintainability, will decrease the lead times involved with the implementation and reconfiguration of control systems. Furthermore, the potential of Erlang to fulfil the reconfigurability requirements will improve the perception of HMSs and RMSs as feasible solutions for industry and as enabling technologies for Industry 4.0.

1.4. Methodology

To evaluate the suitability of Erlang for implementing holonic control, the dissertation performs a comparison of the Erlang implementation with a MAS implementation. Specifically, the Erlang implementation is done using standard Erlang that is supplemented with the Open Telecom Platform (OTP) – the software development is done using the Eclipse Integrated Development Environment (IDE), with the ErlIDE plugin for Erlang. The MAS implementation is performed using Java, with the inclusion of the Java Agent Development (JADE) framework – Eclipse is also used as the IDE. From here on, the *programming language* of the implementations will refer either to Erlang with OTP, or MAS developed with JADE.

To perform a comparison of the two implementations is a challenging task – the implementations are different in not just the programming language, but also in programming paradigm (imperative and functional). While several studies have attempted comparisons of different programming languages (e.g. Harrison *et al.* (1996), Prechelt (2000) and Cesarini *et al.* (2008)), assessments based on generic, objective and quantitative measures are hard to come by. Aiming to avoid this treacherous terrain, the comparison presented in this dissertation has a specific focus: the suitability of the Erlang programming language as a *tool* for implementing holonic control. The comparison thus pays less attention to the philosophical and semantic differences between the programming languages, and rather compares the provisions of each programming language to facilitate the implementation of holonic control. This methodology is similar to that adopted by Chirn and McFarlane (2005) in evaluating the effectiveness of a holonic system design.

The implementation of the same architecture in the two programming languages forms the basis for the comparison. The Product-Resource-Order-Staff Architecture (PROSA) (described in section 2.2.3) is used as the foundation for the development of both the Erlang and MAS holonic control implementations. The use of a common reference architecture allows for comparable functionality in the two implementations – the equivalence is verified through a series of verification experiments, as presented in section 6.2.

As a case study, the implementations are performed for an assembly and quality assurance cell for electrical circuit breakers (discussed in chapter 3). While a case study implementation limits the extent to which the results can be generalised, it does facilitate an evaluation and comparison based on quantitative and qualitative performance measures.

For the implementations, the software was developed according to common practices for each programming language – i.e. provided libraries were used as far as possible, and the development followed the principles outlined in literature

(Logan *et al.* (2011), Armstrong (2007) and Anonymous (s.a. (b)) for Erlang and Bellifemine *et al.* (2007) for JADE).

To improve the reliability and validity of the proposed comparison, both implementations were developed by the author. The premise of using a common reference architecture relies on consistency in the developer's understanding and interpretation of the architecture. Additionally, even though the code is significantly different, the developer follows a similar approach in both implementations. The author possesses the following relevant expertise and experience:

- Undergraduate degree in mechatronic engineering.
- Master's degree in mechatronic engineering, of which the thesis focussed on the development and evaluation of two holonic control implementations – one being a MAS developed in JADE and the other an IEC61499 application using the Function Block Development Kit (see Vyatkin (2007)) – further details can be found in Kruger and Basson (2013).
- Completed an online course on Erlang programming (prior to which the developer was unfamiliar with Erlang programming).

In line with the objective of this research, the evaluation criteria is set up from the perspective of the developers and consumers of holonic control implementations, as opposed to that of computer scientists. The performance measures are thus derived from the requirements for holonic manufacturing systems and the evaluation aims to emphasise the extent to which each implementation satisfy these requirements.

Several aspects of the comparison involve impressions, experiences and philosophies, which are not suited to quantification, leading to criteria comprised of both quantitative and qualitative performance measures. Even though the evaluation is inherently subjective, the comparison strives to provide an unbiased reflection of the suitability of Erlang for holonic control implementation – this is enforced through reference to experimental data, examples from code and findings from literature, as far as possible.

1.5. Dissertation Structure

The dissertation is presented as a collection of papers and is organized into seven chapters. Each chapter provides an overview of the context, content and objective for the included paper(s). The dissertation structure is as follows:

A review of the relevant literature is presented in chapter 2. The review describes the relevant manufacturing system paradigms and, specifically, the research that has been performed on the control of manufacturing systems. The chapter also provides the necessary background for MASs and the Erlang programming language. Beyond chapter 2, a short review of relevant literature is provided in each of the papers included in the later chapters.

Chapter 3 describes the case study and testbed system that was used for the research. Using the case study as starting point, the development of the testbed system, which was used for performing experiments with the control implementations, is discussed.

The development of the Erlang and MAS holonic control implementations are described in chapters 4 and 5, respectively. Chapter 4 comprises two papers that describe the architecture and implementation of holonic control using Erlang, and a third that presents an additional case study implementation. A paper discussing the MAS implementation is presented in chapter 5. Appendix A presents source code for both implementations.

Two papers describing the evaluation of the Erlang control implementation is presented in chapter 6. The first paper describes the formulation of the evaluation criteria, and the second paper presents the comparison of the two implementations.

The dissertation concludes with chapter 7, wherein the important contributions and findings are summarised and some recommendations for future research are given. The reference list provided in chapter 8 contains all the references used in this dissertation, including those used in each of the papers.

The papers that are included in this dissertation are presented as they were submitted for publication – changes were only made to heading and paragraph numbering and formatting, to improve consistency throughout the dissertation. Each paper includes an abstract, an introduction of the context and relevant literature and a reference list, and there will therefore be considerable overlap between these parts of the papers.

All of the presented papers are co-authored by the academic supervisor of this research. However, the author is the main contributor in every paper, with the supervisor's contribution limited to advice on the structuring of arguments and reviewing the manuscripts.

2. Literature Review

This section starts with an overview of manufacturing system paradigms, including holonic and reconfigurable manufacturing systems and Industry 4.0. The review then focusses on the control of manufacturing systems – specifically the use of holonic systems concepts. The architectures and most prominent tools for the implementation of holonic control in manufacturing systems are discussed. Finally, an introduction and overview of the Erlang programming language is presented.

2.1. Manufacturing System Paradigms

2.1.1. Classic and Flexible Manufacturing Systems

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.

2.1.2. Holonic Manufacturing Systems

The concept of Holonic Manufacturing Systems (HMSs) came into being in the early to mid 1990s, aiming to address the requirements of the modern manufacturing environment (as listed in section 1.1). Early research into HMSs was driven by the HMS Consortium (Christensen, 1994), but numerous studies followed in the subsequent decades. The basic concepts and development of HMSs are summarized in this section.

The concept of holonic systems was developed by Koestler (1967) as an explanation of the self-organizing tendencies observed in social and biological systems. The term *holon* 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).

Holonic Manufacturing Systems (HMSs) result from the application of the holonic systems concept in manufacturing systems. In a manufacturing system context, a holon is an autonomous and cooperative building block for transforming, transporting, storing or validating information of physical objects. A 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).

Figure 1 shows the internal architecture for a holon in a HMS, as formulated by Leitao and Restivo (2002). The internal architecture makes provision for the two essential holon characteristics: cooperation and autonomy. The *communication* component enables cooperation with the other holons in the system through maintaining a communication interface and constructing, parsing and exchanging information. The *decision-making* component of the holon internal architecture facilitates the implementation of autonomy, where the logic can be added to control the behaviour of the holon.

The internal architecture in Figure 1 also includes an *interfacing* component. This component provides a mechanism to connect the software and hardware parts of the holon, so that process execution information can be exchanged. Even though it can be expected that holons in a HMS include a hardware resource on the shop floor, these systems also include holons that purely exist in software. Such holons implement a similar internal architecture, but without the need for an *interfacing* component.

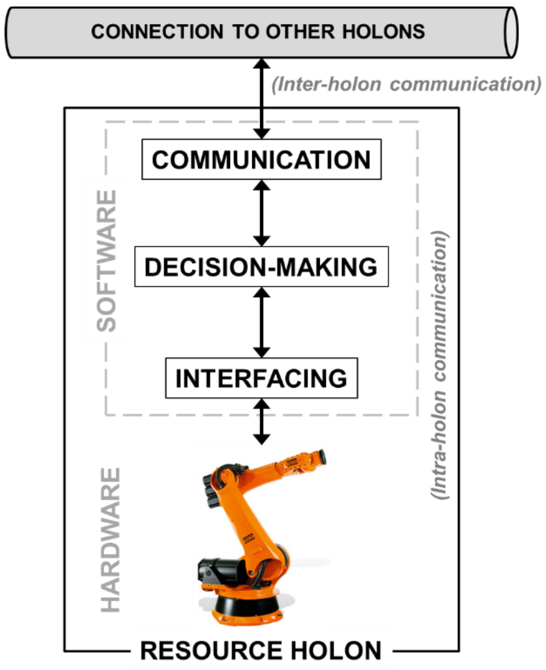

CONNECTION TO OTHER HOLONS
(Inter-holon communication)
COMMUNICATION
DECISION-MAKING
INTERFACING
SOFTWARE
HARDWARE
(Intra-holon communication)
RESOURCE HOLON

Figure 1: Internal architecture for a Resource holon (adapted from Leitao and Restivo (2002)).

It is clear that HMSs inherently consider the software aspects of manufacturing systems, along with the physical hardware. Extensive research has been done on the application of the holonic concept to organise the control of manufacturing systems, within and beyond the HMS paradigm – this is reviewed in detail in section 2.2.3.

2.1.3. Reconfigurable Manufacturing Systems

The concept of Reconfigurable Manufacturing Systems (RMSs) is another solution to the requirements of modern systems. The development of RMSs occurred in parallel with HMSs, mainly driven by the research of Koren (Koren and Ulsoy, 1997; Koren *et al.*, 1999). This section defines reconfigurability in the manufacturing context and presents an overview of the key aspects of RMSs.

It is important to discuss the exact meaning of reconfigurability in this context. Martinsen *et al.* (2007) describe reconfigurability as the ability of a manufacturing 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).

Rooker *et al.* (2007) explain two different types of reconfiguration that 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.

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 only designed to allow for changes in the production processes and the flow of material.

Koren and Ulsoy (2002) identified six key characteristics that must be exhibited by the mechanical, control and communication components of RMSs. The characteristics are as follows:

1. **Modularity** of the hardware and software system components, so that components can be replaced or rearranged to meet new requirements.
2. **Integrability** of the system and the system components for both integration of existing technology and the introduction of new technology in the future.
3. **Convertibility** of the system to allow for fast changeover between existing products and fast adaptability of the system for future products.
4. **Diagnosability** 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 products or production requirements.
6. **Scalability** of the system capacity through the addition of resources.

RMSs satisfy all the requirements of modern manufacturing mentioned in section 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 in RMSs "the system and its components have adjustable structure that enables system scalability in response to market demands and system adaptability to new products".

Several issues have hampered the development and implementation of RMSs. Bi *et al.* (2007) 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.

2.1.4. Industry 4.0 and Cyber-Physical Production Systems

Industry has experienced three revolutions: the first was brought about by the invention of the mechanical loom for use in the textile industry in 1764; the second was driven by Henry Ford's mass production assembly line for the T1 model in 1913; and the third was due to the introduction of the first Programmable Logic Controller (PLC) in 1968. It would seem that industry is currently on the brink of the fourth industrial revolution – often referred to as Industry 4.0.

Industry 4.0 is driven by an increased connectedness of the real and virtual worlds, forming the Internet of Things (IOT). The effect of IOT on production will be an enhanced individualization of products through highly flexible production, the extensive integration of customers and businesses in value-added processes and the linking of production and high-quality services to deliver hybrid products.

Industry 4.0 will be characterized by the individualization of products and services, new organization and control of the entire value chain and the formulation of new business models. These characteristics can be facilitated through the connection of humans, objects and systems, and the generation and use of information in real-time. Cyber-Physical Systems (CPSs) will play a key role in the connection of people, components/systems, information and services.

CPSs are systems of communicating computational entities, which are connected to the physical world, that simultaneously use and provide data and services, using the Internet. These entities can monitor, control, coordinate and integrate the operations of physical or engineered systems. The maturity model for CPS functionality is shown in Figure 2.

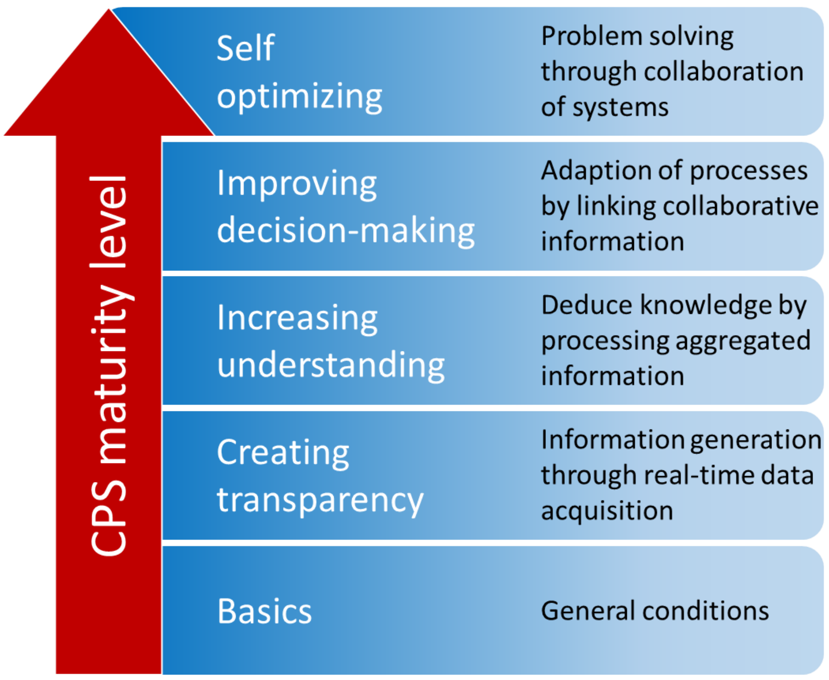

CPS maturity level
Self optimizing
Problem solving through collaboration of systems
Improving decision-making
Adaption of processes by linking collaborative information
Increasing understanding
Deduce knowledge by processing aggregated information
Creating transparency
Information generation through real-time data acquisition
Basics
General conditions

Figure 2: CPS maturity model (adapted from Monostori *et al.* (2016)).

When the concept is applied to manufacturing, it is referred to as Cyber-Physical Production Systems (CPPSs). CPPSs then entail the convergence of the virtual and physical worlds of manufacturing – the first driven by developments in computer science and information and communication technologies, and the second by manufacturing science and technology.

CPPSs consist of autonomous and cooperative elements and subsystems that can be connected within and across all levels of production – from high-level enterprise resource planning and plant management, to the lower levels of process and hardware control. The 5C architecture, proposed by Lee *et al.* (2015), explains the role of CPPS implementation in different levels of automation:

- Smart **C**onnection level – the acquisition of accurate and reliable data from machines. The data is obtained directly from sensors or via controller or manufacturing execution systems.
- Data **C**onversion level – meaningful information is inferred from the acquired data using smart analytics.
- **C**yber level – information is gathered from all connected system components. The centralization of information allows for analysis based on historical data or through comparison between similar cases.
- **C**ognition level – knowledge is generated from the comparative information, which provides support for expert users in making decisions on corrective and predictive actions.
- **C**onfiguration level – corrective or predictive decisions are applied to the physical system, resulting in the adaption of machine/system configuration.

Through the approach and architecture described above, CPPSs aim to exhibit the following characteristics:

- Intelligence – system elements are capable of acquiring information and acting autonomously.
- Connectedness – connections exist between the system elements (including humans) and knowledge and service depositories (such as the Internet) to facilitate cooperation.
- Responsiveness – the system is capable of responding to internal and external changes.

The approach, architecture and characteristics described above closely resemble some of the aspects introduced in sections 2.1.2 and 2.1.3. In fact, Monostori *et al.* (2016) acknowledge that CPPS is not a novel, stand-alone concept, but rather a culmination of several preceding developments in manufacturing science and technology – including that of holonic and reconfigurable manufacturing systems. Furthermore, Wang and Haghighi (2016) believe that control implementation platforms, like multi-agent systems and function blocks, will play in important role in CPPSs.

2.2. Control of Manufacturing Systems

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

2.2.1. 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 3. The architectures are described in the following paragraphs.

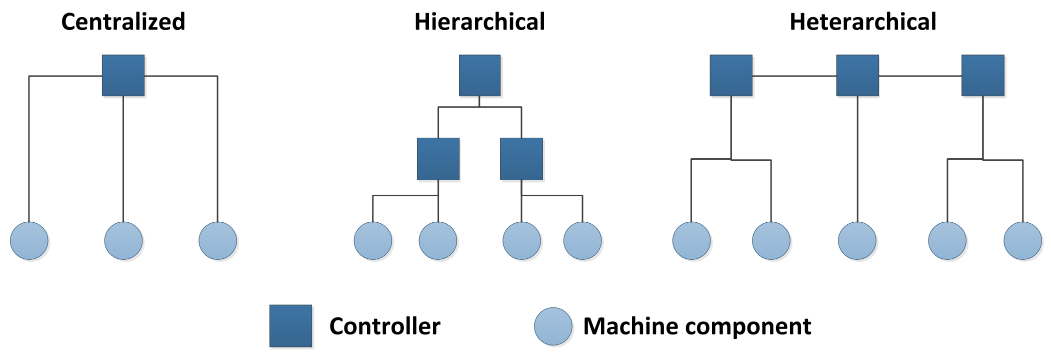


Figure 3: Types of control architectures (adapted from Meng *et al.* (2006)).

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 2.2.2).

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 2.2.3).

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.

2.2.2. Conventional Control

Conventional manufacturing control systems are typically large, centralized applications that 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 2.2.1). 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).

2.2.3. Holonic Control

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 disturbances 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, robustness and fault tolerance (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 4. 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, operating 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.

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 integrability 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.

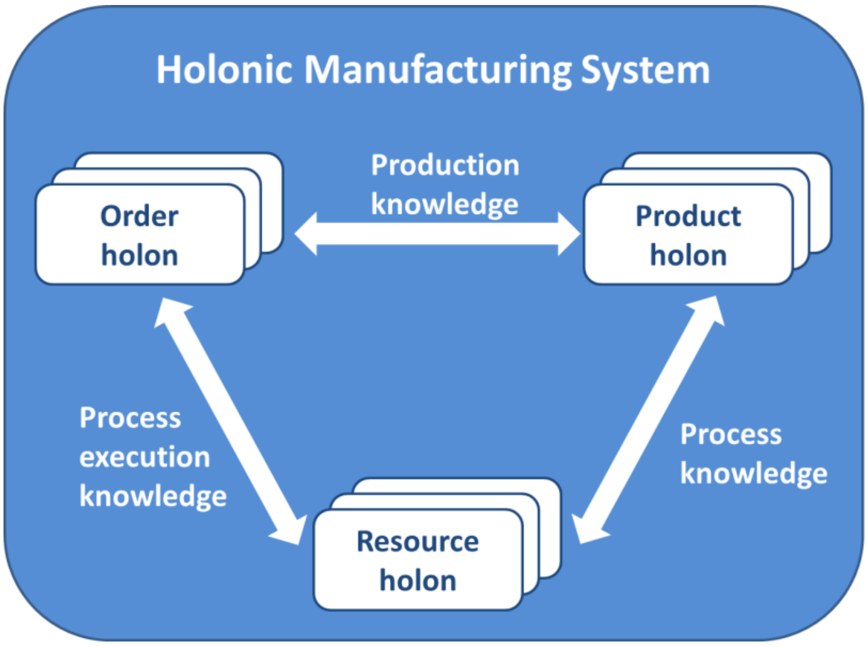


Figure 4: Structure of PROSA architecture (adapted from van Brussel *et al.* (1998)).

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 5.

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 can 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), as is shown in Figure 1. 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.

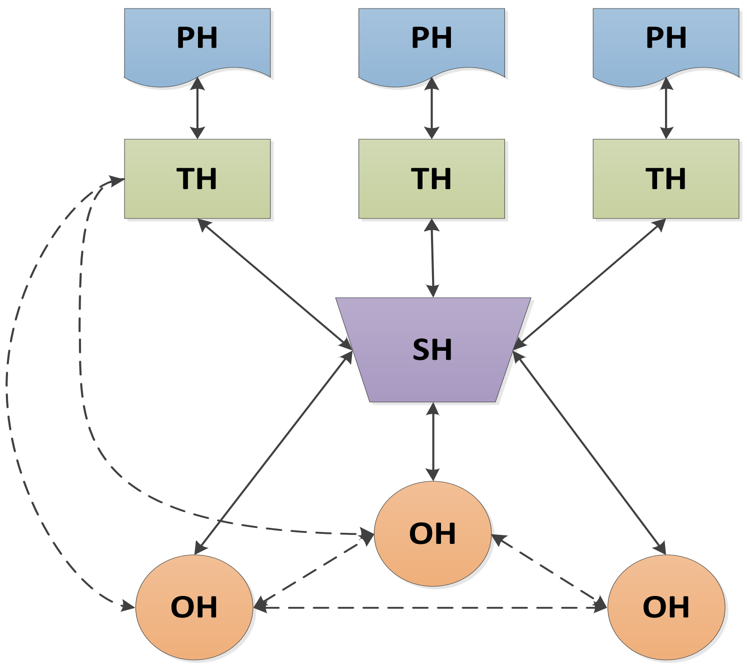


Figure 5: Structure of ADACOR architecture (adapted from Leitao and Restivo (2006)).

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".

2.3. Holonic Control Implementation

Two platforms have been regularly used to implement the holonic control architectures presented in section 2.2.3 – multi-agent systems and IEC 61499 function blocks. The basic concepts, advantages, standards and platforms for development and existing implementations are discussed for each platform.

2.3.1. Multi-Agent Systems

The use of agent-based software to control manufacturing systems has received much attention in the research community. MASs have become a popular choice for the implementation of holonic control architectures in both holonic and reconfigurable manufacturing systems.

2.3.1.1. 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).

2.3.1.2. 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.

2.3.1.3. 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 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 JADE, FIPA-OS 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 fully FIPA compliant.

2.3.1.4. Advantages of MASs

MASs hold several advantages for implementation in RMSs. 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".

2.3.1.5. Implementations of MASs

There have been several practical implementations of agent-based control. The ADACOR architecture (described in section 2.2.3) 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. Recently, MASs were implemented as a key technology in four European innovation projects focussed on the development of CPPSs (Leitao *et al.*, 2016).

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.

2.3.2. IEC 61499 Function Blocks

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

2.3.2.1. 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.

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

The standard enables the development of new technologies that 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 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).

2.3.2.2. 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).

2.3.2.3. 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 IEC 61499 function blocks is that, as a modelling 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 a 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 that makes it appropriate for implementation in the broader embedded systems domain (Vyatkin, 2007).

2.3.2.4. 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.

2.4. Erlang

Since its initial development for telecommunications switching systems (TSSs), the Erlang programming language has been implemented in a wide field of applications. This section provides a brief introduction of Erlang and OTP, discusses the advantages that are offered and presents an overview of significant implementations.

2.4.1. Erlang Programming Language

Erlang is a concurrent, functional programming language that was developed for programming concurrent, scalable and distributed systems. The language was developed at the Ericsson Computer Science Laboratory and implemented by Ericsson from 1986 to 1998 (Armstrong, 2003). The development of Erlang was inspired by an investigation into whether modern declarative programming paradigms could be used for programming large industrial TSSs (Armstrong *et al.*, 1996). Table 1 summarises the design requirements of TSSs, matched with the characteristics of Erlang.

Erlang owes its concurrency to the *process model* on which it is built. Processes, as the basic unit of abstraction, are extremely lightweight with memory requirements that can vary dynamically. Erlang processes are not operating system (OS) threads – processes are implemented by the Erlang runtime system, which facilitates and schedules the process execution within the OS (Logan *et al.*, 2011).

Unlike OS threads, Erlang processes do not share a memory space. Process are strongly isolated, having no shared memory, and can only interact through the asynchronous sending and receiving of messages (Logan *et al.*, 2011). These characteristics not only allow many processes to work concurrently, but they can also be distributed across many devices (referred to as *nodes*).

Erlang provides simple mechanisms for inter-process data exchange through asynchronous message passing. To send a message, the "!" operator (called the *bang* operator) is used. For example, to send a message to another process can simply be done by:

```
ProcessID ! Message
```

Where `ProcessID` represents the unique process identifier or registered name of the recipient process and `Message` is a variable storing the message content. To receive and handle messages is equally simple. Every process has a mailbox that stores incoming messages as they arrive. Messages can then be searched and retrieved with the `receive` expression:

```
receive
      MessageTemplate1 -> Action1;
      MessageTemplate2 -> Action2
end.
```

The comparison of the received message to defined templates is called pattern matching. The action that the process executes depends on the template that matches to received message.

Table 1: Matching the requirements of TSSs to the characteristics of Erlang (adapted from Däcker (2000)).

Requirements of programming technology for TSSs	**Erlang characteristics**
Handling of a very large number of concurrent activities.	Concurrency is provided through a lightweight process concept which can be spread across nodes.
Actions to be performed at a certain point in time or within a certain time.	Erlang operates in soft real time (where response times in the order of milliseconds are required).
Systems distributed over several computers.	An Erlang system may contain nodes distributed over many computers running different operating systems, over a network.
Interaction with hardware.	Erlang can easily communicate with hardware drivers and programs written in other languages.
Very large software systems.	Erlang is based on the modularity concept, which allows for the expansion of the control program.
Complex functionality such as feature interaction[2].	Depends on the application.
Continuous operation for many years.	Erlang permits hot code loading, so that the system does not have to be stopped for any maintenance or reconfigurations.
Performing software maintenance, reconfiguration, etc. without stopping the system.	Erlang permits hot code loading, so that the system does not have to be stopped for any maintenance or reconfigurations.
Stringent quality and reliability requirements.	Depends on the application.
Fault tolerance to both hardware failures and software errors.	Erlang contains functions to catch and contain run-time errors, and to design supervision structures.

[2] *Feature interaction* is a euphemism for the concept that complicated systems have complicated behaviour, and that every time you add a feature to a system, it is likely to have unpredicted and unwelcome effects on the behaviour of existing features.

The process model enables Erlang as a concurrent programming language, but the language is also functional. Logan *et al*. (2011) summarise the main concepts of functional programming as:

- functions are treated as data – just like strings or integers;
- algorithms are expressed in terms of function calls, instead of loop constructs like *for* and *while*;
- and variables and values are not updated in place – a property termed *referential transparency*[3].

Erlang implements the above-mentioned concepts, but is not a "pure" functional language as Erlang relies on side effects. However, the reliance on side effects is limited to one operation – message passing. Each message represents an effect on a component of the program or outside world. Apart from this effect, each Erlang process essentially runs a functional program.

In order to aid the reader in understanding the Erlang code presented in this dissertation, a brief overview of the syntax is presented in Table 2. Erlang was initially implemented in the Prolog programming language and inherited many of its syntactical conventions.

Table 2: Erlang syntactical conventions.

Data type/ construct	Description	Syntactical convention	Example(s)

Atom	A special kind of string constant, similar to `enum` constants in C.	Starts with a lowercase letter	`ok`
			`error`
			`undefined`

Tuple	A fixed-length ordered sequence of other Erlang terms.	Written within curly brackets	`{1, two, 3}`
			`{nested, {structure}}`

List	An collection of Erlang terms with variable length.	Written within square brackets	`[1, two, 3]`
			`[list, {with, tuple}]`
			`[nested,[list]]`

Process identifier	A unique identifier for an Erlang process.	Three integers enclosed in angle brackets	`<0.35.0>`

Variable	Construct for storing Erlang terms.	Start with an uppercase letter	`Name`
			`SomeInfo`
			`Some_info`

Function	A collection of related Erlang expressions.	Function name starts with a lowercase letter, followed by input arguments within round brackets	`execute()`
			`add(1,2,3)`
			`get_message(Message)`
			`do_something()`

[3] Logan *et al* (2011) explain reference transparency as follows: if a process obtains some value or term, and assigns a name to it (i.e. assign the value to a variable), then it is guaranteed that the value of the variable will not change, even if a reference thereof is passed to some other part of the program.

2.4.2. OTP

The Erlang language is supplemented by the Open Telecom Platform (OTP). In 1995, Ericsson decided to restart a failed C++ project, using Erlang instead. The project was a success, largely due to the work of the Erlang language support department on the development of OTP (Logan *et al.*, 2011).

OTP includes a set of Erlang libraries and design principles, providing middleware to develop Erlang systems (Anonymous, s.a. (a)). OTP includes the following components (Armstrong, 2003):

- Compilers and development tools for Erlang.
- Erlang run-time systems for a number of different target environments.
- Libraries for a wide range of common applications.
- A set of design patterns for implementing common behavioural patterns.
- Educational material for learning how to use the system.
- Extensive documentation.

From a design and implementation perspective, the primary aim of OTP is to improve robustness and uniformity (Armstrong, 2003). The OTP behaviour libraries were developed and tested by expert programmers over several years. Furthermore, behaviours hide the complexity and exposes simple, generic interfaces to the developer. Behaviours also enforce a regular structure, leading to uniformity in the design and implementation of solutions. This uniformity allows for increased productivity and fewer errors in multi-programmer environments.

2.4.3. Advantages of Erlang/OTP

Traditionally, concurrency in a programming language has been achieved with threads. The execution of a program is split into concurrently running tasks, operating on shared memory. This leads to problems that can be hard to debug, such as the *lost update* problem. A solution to this is the use of *locks*, but this may lead to a *deadlock* problem. The Erlang processes have no shared memory, which eliminates the above-mentioned problems with threading. The Erlang processes are also very lightweight, making process creation orders of magnitude faster than thread creation in most programming languages (Armstrong, 2003).

Erlang holds a critical advantage over other programming languages when it comes to robustness. Erlang has improved fault-tolerance due to its inherent fault-isolation structure. Armstrong (2003) explains that processes act as abstraction boundaries, limiting the propagation of errors through the software. OTP contributes significantly to the robustness of Erlang applications in providing a reliable, stable code base in behaviours (Logan *et al.*, 2011). OTP also includes the *supervisor* behaviour, which facilitates the implementation of supervision trees to monitor processes and trap and handle errors.

The Erlang run-time environment is independent of the properties of the host operating system (Armstrong, 2003). The Erlang processes, and their concurrent operation, synchronization and interaction, are handled by the programming language and not by the operating system. Erlang makes use of very little operating system services, and can thus be ported to specialised environments (such as embedded systems) with relative ease.

Erlang/OTP has primitives that allow code to be replaced in a running system, enabling old and new versions of code to execute at the same time (Däcker, 2000). When a new module is loaded, newly started process will run the new version, while on-going processes will continue and finish undisturbed. This capability enables the uploading of bug fixes and upgrades in a running system without disturbing the current operation.

Since Erlang processes share no memory and interaction is only done through message passing, programs can very easily be distributed (Armstrong, 2003). Erlang programs that are designed for implementation as independent, concurrent processes can be implemented on a multi-processor or run on a distributed network of processors. This distributive characteristic is thus inherent in the Erlang design.

Wiger (2001) claims that comparisons, made between internal software development projects at Ericsson, have shown that Erlang allows for much higher productivity. When compared with C++, Erlang applications resulted in a ten-fold reduction in the number of uncommented source code lines – other comparisons have indicated a four-fold reduction. The same relationship tends to exist with code-error density (errors per line(s) of source code). The reuse of generic OTP behaviours further enhances productivity.

Erlang/OTP also allows for integration with software written in other programming languages. *Ports* allow programs to be called and interfaced to the Erlang application in such a way that they appear to the programmer as if they were written in Erlang (Armstrong *et al.*, 1996).

2.4.4. Erlang Implementations

Armstrong (2010) provides a short overview of the most significant implementation areas for Erlang/OTP:

- Switches – The largest implementation with Erlang, to date, is Ericsson's AXD301 asynchronous transfer mode switch. The switch contains 1.6 million lines of Erlang code implementing a modular, distributed architecture and achieving a scalable capacity between 10 Gbit/s to 160 Gbit/s.
- Instant messaging – Erlang's usefulness for developing instant messaging services for Internet applications is reflected in three projects: MochiWeb, Ejabberd and RabbitMQ. MochiWeb is an Erlang library for building HTTP servers with high-throughput, low-latency analytics and it used by Facebook Chat to serve 70 million users. Ejabberd is an Erlang implementation of the XMPP protocol and is amongst the most widely used open source XMPP servers. RabbitMQ is an implementation of the Advanced Message Queuing Protocol with Erlang, which provides reliable asynchronous message passing.

- Schema-free databases – Erlang is well suited for the creation of databases to store associative array or arbitrary tree-like data structures, as is reflected by the CouchDB (open source) and Amazon SimpleDB (commercial) implementations.

3. Case Study and Testbed System

This section describes the case study that is considered in the research, as well as the testbed system that was developed to facilitate the evaluation of the holonic control implementations. For the case study, the context and essential details are provided. The need for a testbed system is motivated and the development thereof is described – this description is followed by a paper that presents the use of an object-oriented simulation framework to create an emulation model for the testbed system.

3.1. Case Study Description

As case study, the research considers the proposed assembly and quality assurance cell for electrical circuit breakers for the production of a South African manufacturer, CBI Electric Ltd. The presented research formed part of larger research project conducted by the MAD research group at Stellenbosch University, which entailed the design and demonstration of a manufacturing cell to replace an existing manual labour production line.

The assembly and quality assurance cell poses all the challenges faced by modern manufacturing – the cell must be capable of handling product variation and fluctuating production volumes, with minimal changeover time and effort. The cell is thus considered suitable for design and implementation as a HMS.

The layout for the assembly and quality assurance cell is shown in Figure 6. A modular, palletized conveyor system transports the circuit breakers between the various automated and manual workstations, each performing a specified production task. The cell consists of the following workstations:

- Manual assembly station – the sub-components of circuit breakers are assembled and placed on empty carriers on the conveyor.
- Inspection station – a machine vision inspection is performed on the circuit breakers as the carriers are moved by the conveyor.
- Electrical test station – circuit breakers are picked up by a robot and placed into testing machines. The testing machines perform the necessary performance and safety tests on every breaker. When the testing is completed for a breaker, it is removed from the testing machine by the robot and placed on an empty carrier on the conveyor.
- Stacking station – multiple circuit breakers are stacked to produce multi-pole circuit breakers. The breakers are removed, stacked and placed on empty carriers by a robot.
- Riveting station – the casings of the circuit breakers are manually riveted shut.
- Removal station – the completed circuit breakers are removed from carriers. The breakers are then moved to the next cell for packaging.

The conveyor moves product carriers between the various workstations. The conveyor is equipped with stop gates and lifting stations at every workstation. The carriers are fitted with Radio Frequency Identification (RFID) tags and RFID readers are placed at multiple positions along the conveyor, to provide feedback of carrier location.

3.2. Testbed System

In order to investigate the case study and facilitate the implementation of holonic control, a testbed system was developed. The testbed system design followed the layout of the assembly and quality assurance cell, as shown in Figure 6.

The assembly and quality assurance cell was only proposed as a design and thus the hardware was not available to construct a real testbed system. Instead, the testbed system was constructed as an emulation model, representative of the cell's low level control and hardware, using the Simio simulation framework.

The model was developed to provide a realistic emulation of the production processes of the manufacturing cell. The emulation entailed the following:

- The provision of a mechanism for receiving inputs from the control software.
- The execution of the manufacturing processes in reaction to received inputs.
- The provision of feedback from the execution of manufacturing processes to the control software.
- The visualization of the processes of the manufacturing cell.

The emulation was designed with a TCP/IP interface that replicated those that would be used when using a real testbed system. In previous implementations, the MAD research group used TCP/IP sockets as a generic communication interface between the high and low level control programs.

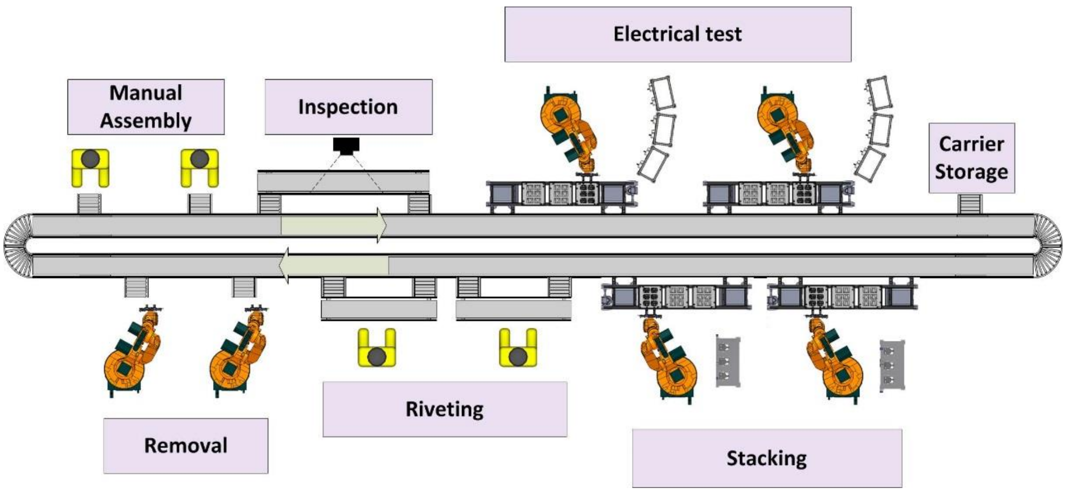


Figure 6: Layout of the assembly and quality assurance cell.

The use of Simio to construct an emulation model for the testbed system is presented as a paper in the next section – "Validation of a Holonic Controller for a Modular Conveyor System using an Object-oriented Simulation Framework" (Kruger and Basson, 2017 (c)). The paper describes the use of the Simio simulation framework to emulate manufacturing processes by receiving execution commands as inputs and providing output information on execution status and events. A discussion of the Interpreter application, which provides a communication interface between the holonic control software and the Simio emulation model, is also included. The paper was presented at the sixth international workshop on Service Orientation in Holonic and Multi-Agent Manufacturing (SOHOMA) in Lisbon, Portugal, in 2016.

3.3. Validation of a Holonic Controller for a Modular Conveyor System using an Object-Oriented Simulation Framework

Karel Kruger [a,*] and Anton Basson [a]
[a] Dept of Mechanical & Mechatronic Eng, Stellenbosch Univ, Stellenbosch 7600, South Africa
[*] Corresponding author. Tel: +27 21 808 4258; Fax: +27 21 866 155 206;
kkruger@sun.ac.za

Abstract

This paper presents the use of a commercial, object-oriented simulation framework to facilitate the validation process of a holonic controller. The case study involves a holonic controller for a modular conveyor system. The holonic control is implemented using Erlang and thus exploits the scalability and concurrency benefits it has to offer – the simulation model and necessary interfacing was then customized to accommodate the nature of the implementation. The simulation model interface, incorporating TCP communication and Windows Communication Foundation services, was designed to mirror that of the conveyor hardware to allow for the seamless changeover between emulated and real operation.

Keywords: Emulation; Holonic Manufacturing Systems; Reconfigurable Manufacturing Systems; Manufacturing Execution Systems

3.3.1. Introduction

Modern markets have enforced a new set of requirements on the manufacturing industry – increased adaptability to accommodate market trends and fluctuations, shorter lead times, increased product variation and customizability [1], [2]. This was already anticipated more than two decades ago [3] and, since then, research has been done on many aspects concerning the transformation of modern manufacturing systems.

A popular approach used in several studies and implementations, is that of holonic systems. This idea, originally presented by Koestler [4], can be understood within the manufacturing system environment as the division of a system into autonomous, cooperating entities which work together to accomplish the system functions [5].

The holonic approach to manufacturing systems have provided many benefits – enhanced system scalability, customizability and fault tolerance, which lead to increased system reconfigurability and reliability, and reduced complexity [6]. As can be expected, holonic systems have encountered some challenges – of which the most relevant to this paper is that of system validation.

The validation of manufacturing systems can be understood as the means to test the system and obtain assurance that the system functions as desired. The validation of holonic systems can be difficult since the system functions are distributed over several processes and/or controllers. Since holonic systems are based on holon cooperation and is often distributed, it becomes harder to validate a control application [7].

Regarding the validation of holonic systems, there has been some research done into tools to aid in the endeavour. One example is that of the Multi-Agent Simulation Tool (MAST) presented in [8]. MAST uses a multi-agent system to control a graphical simulation. A discussion of the use of simulation in holonic systems is presented in [9].

There are several simulation software packages available which have been used in many different fields and applications. One such package is Simio, a modelling framework based on object-oriented principles [10]. The inherent architecture of Simio fits well with the idea of holonic systems and is discussed further in section 3.3.2.

This paper describes the use of the Simio simulation framework to validate a holonic control implementation through hardware emulation. The hardware emulation is configured to extend the holonic principles of the higher level control architecture by facilitating control distribution and modularity within the simulation framework.

For a case study implementation the control and emulation of a modular, palletized conveyor system is used. Conveyor systems entail complex interactions and logic, and require significant programming and testing efforts during commissioning and reconfiguration activities. These challenges motivate the need for a simulation tool to validate the routing and control logic – especially for systems large in size and complexity – to decrease lead and ramp-up times.

This paper presents a discussion of the holonic architecture, with focus on the inclusion and use of the Simio emulation model to facilitate the control validation. The validation process and the useful tools provided by Simio are also discussed.

3.3.2. Simio Modelling Framework

[11] presents Simio as a graphical modelling framework which implements object-oriented principles in both the simulation logic programming and the construction of simulation models. Simio provides the developer with the infrastructure to build up a simulation model with customizable objects. The behaviour of the Simio objects can be customized by adding processes that define the execution logic. Processes are sequences of steps that are executed in a thread of execution. Steps perform some specified function, such as handling or triggering events that influence the state of the object.

Furthermore, Simio is programmed in Microsoft Visual C# - this opens the framework for incorporation with powerful tools like .NET and Windows Communication Foundation (WCF). Simio also explicitly provides an API for C#, which provides several useful functions for the construction and running of Simio models.

3.3.3. Holonic Cell Control

At cell control level, a holonic architecture was implemented in accordance with the PROSA [12] reference architecture. As is clear from Figure 7, the modular conveyor system is represented as a Resource holon at cell control level. The conveyor holon is comprised of three components – High Level Control (HLC), Low Level Control (LLC) and the physical hardware. The HLC component represents the holon in the virtual cell control environment. This component handles all communication with the holons in the cell, such as service bookings, service cancellations, etc. The HLC activates execution of a desired service through communication with the LLC component. The LLC has interfaces with the physical actuators and sensors of the hardware and can thus coordinate the sequence of hardware actions required to perform a desired service.

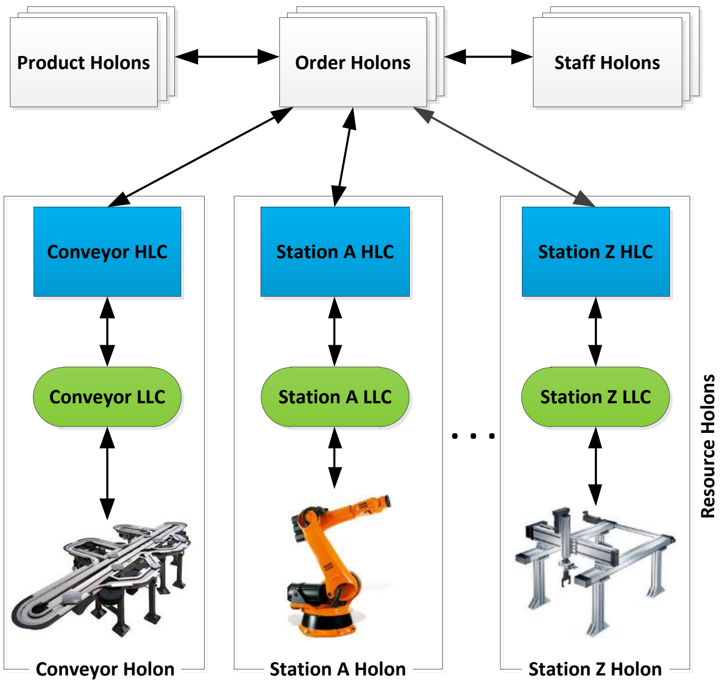


Figure 7: Manufacturing cell control architecture.

3.3.4. Conveyor Holon

3.3.4.1. Holonic Controller

The Conveyor holon component which forms part of the PROSA cell control application is implemented using Erlang. Erlang is a functional programming language with inherently strong scalability, concurrency and fault-tolerance characteristics.

The Conveyor holon HLC implementation was aimed at exploiting the modularity and scalability advantages that Erlang offers. The HLC component is itself implemented as a collection of holons which encapsulate, and through cooperation, constitute the Conveyor holon functionality. A detailed description is given in [13].

3.3.4.2. Interpreter

The Interpreter program provides a link between the holonic controller and the emulation model. The Interpreter maintains an interface to the holonic controller that is similar to that of the low level PLCs of the conveyor (shown in Figure 8) – this interface facilitates TCP communication over multiple ports (the same number as the number of PLCs used in real operation). The Interpreter program creates a link to the emulation model by making use of the Windows Communication Foundation (WCF) services. The two mentioned interfaces are discussed in the following sections.

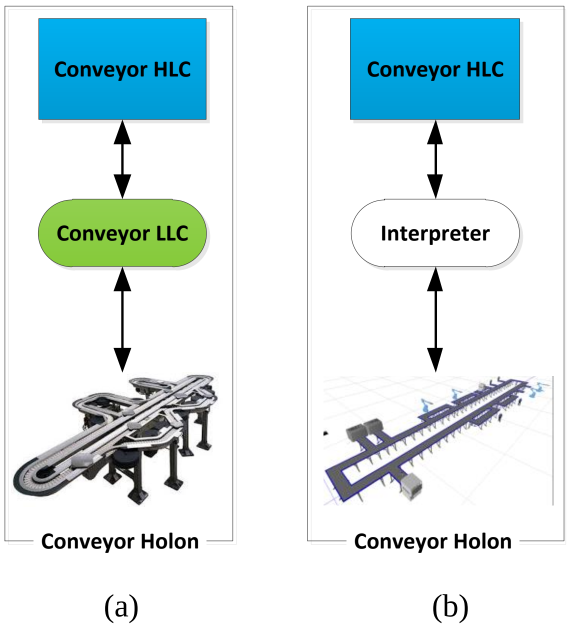


Figure 8: Conveyor holon architecture for (a) real and (b) simulated operation.

3.3.4.2.1. TCP Communication with HLC

As mentioned, the Interpreter program facilitates TCP communication which emulates the communication to the PLCs that control the conveyor hardware. To the Erlang-based holonic controller programs, there is no difference in the communication whether real operation or emulation is performed.

The Interpreter program maintains a port for every PLC that is installed on the conveyor. To communicate the information received from the holonic controller to the emulation model, the Interpreter program parses XML encoded strings received over the TCP ports. The parsing extracts the critical information that must be communicated to the emulation model. In the same way the PLCs will provide notifications based on the feedback of their connected sensors, the emulation model provides feedback based on events in the emulation model. This feedback information is then encoded into an XML string and is sent via the TCP port to the holonic controller.

3.3.4.2.2. WCF Interface with Emulation Objects

In order to interface the Interpreter C# program with the Simio objects during runtime, WCF was chosen to provide the infrastructure for communication. WCF is a software development kit for implementing services on the Windows operating system. Services, in this case, refer to units of functionality and coincide with that used in service-orientation principles. [14]

In the Interpreter program, WCF is used to host a service that exposes both events and event handlers. The service can be accessed by clients through bindings, which are configured in a service contract. The service contract allows the various EventInterface step object instances in Simio (discussed in section 3.3.5), which form part of the model processes, to trigger an event that will be handled by the Interpreter program. The contract also allows for the Interpreter program to trigger an event which is handled by the EventInterface objects.

Using the WCF service, the process step of each transfer node in the Simio model triggers a "notification" event that is handled by the Interpreter. This event is triggered whenever a carrier arrives at a transfer node and the carrier name is supplied as an event data parameter. This notification is forwarded as an XML string to the HLC.

With the notification received, the HLC must determine to which transfer node the carrier in the model must be directed next. This information is then sent to the Interpreter program, where an event is triggered (the name of the next transfer node is specified in the event information). This event is then handled by the process step of the relevant transfer node and the extracted information is used to direct the carrier in the desired direction.

3.3.5. Conveyor Model

The Simio emulation model for the conveyor is shown in Figure 9. As will be explained in the following sections, the model is constructed using standard Simio objects and the logic is implemented through Simio processes with customized process steps.

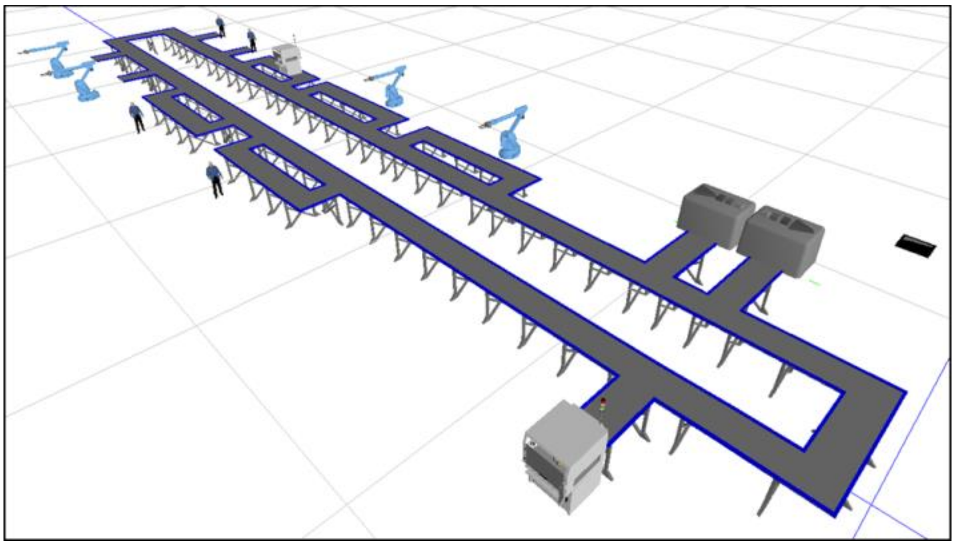


Figure 9: Conveyor emulation model.

3.3.5.1.1. Simio Model

The conveyor system is modelled as a network of nodes linked by transitions (also referred to as paths or links). Nodes are points on the conveyor where two or more transitions meet – on the physical system, nodes are implemented by stop gates (usually in combination with lifting stations or transverse conveyors, and are equipped with RFID readers), as is shown in Figure 10. These physical entities can be modelled in Simio by transfer node objects for node entities and either conveyor objects (for one-directional transitions) or path objects (for bi-directional transitions).

The model of the conveyor also includes means of carrier storage (i.e. a mechanism to unload or store carriers). For the conveyor used in this case study, this function is performed by an automated carrier magazine. The same functionality can be achieved in the emulation model by using the source and sink standard Simio objects. The source object unloads carriers for the conveyor and the sink model stores carriers.

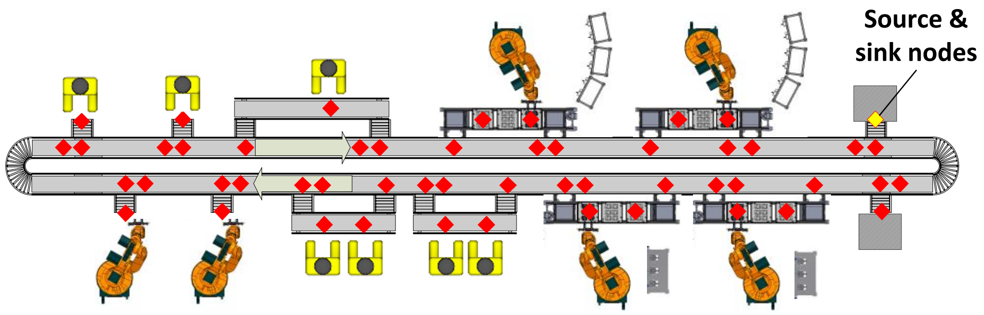


Figure 10: Schematic of the conveyor with all nodes indicated

3.3.5.1.2. Simio Processes

The behaviour of the standard Simio transfer node objects can be customized by adding processes to the object instance. Processes are constructed through a specified sequential execution of functional steps. The processes are executed when specific events occur – in the transfer node case, when an entity (carrier) enters the transfer node and the "entered" event is triggered. The process executed when the "entered" event is triggered is shown in Figure 11.

Figure 11 shows the process which is executed by transfer node objects when they are entered by an entity object. When an entity enters a transfer node, the first step executed in the process is NotifyReady. During this step, the notification event is triggered which is handled by the Interpreter. The step then also subscribes to the event that the Interpreter will trigger when it receives a message from the HLC specifying the next transfer node. When the event is triggered and handled by the NotifyReady step, the process next enters an Execute step – this Execute step then calls the SetNode process, which uses the obtained event information to specify the node to where the entity must be directed.

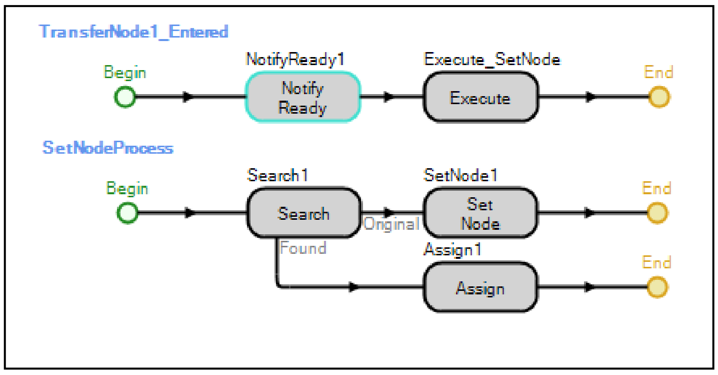


Figure 11: Simio processes for conveyor node objects.

3.3.5.1.3. Conveyor Emulation

During operation, the first task for the conveyor will be to unload a carrier from storage onto the conveyor – this unload task will be initiated from the controller. This unloaded carrier will be moved to some location, as controlled by a corresponding process in the holonic controller. After unloading, the carrier will arrive at the first transfer node and a notification will be sent to the Interpreter program, where it will be encoded into an XML string and be forwarded to the holonic controller. The controlling process can then react to this notification and send an XML string to the Interpreter which specifies the next transfer node to where the carrier must be moved. The transfer node currently occupied waits for this command to be received from the Interpreter and subsequently directs the carrier on the desired path towards the desired next transfer node.

3.3.6. Control Validation

Validation, in this context, refers to the assurance that the holonic control application is performing the system functions as desired. The emulation of the conveyor system using a Simio model offers several advantages for the validation of the control logic.

An important advantage is the ability to perform long-running emulations in short times, as the execution speed of the emulation model can be controlled. Also, the use of Simio emulation allows for testing of specific production scenarios – this is especially useful in the event of reconfiguration. The initial conditions of the emulation environment can be customized to adhere to some HLC scenario – i.e. the conveyor emulation can begin with a “clean” startup, or with carriers in predefined locations. The combination of Erlang and Simio simplifies this process – the stateless nature of Erlang programs allow for the various system holons to be launched with specific state data, while the “open” nature of Simio, together with the C# API functions, provide the infrastructure to create custom scenarios.

Even though the research community is still striving toward standardized benchmarks for the performance of holonic systems, the collection of performance data is critical for the validation process. Simio incorporates the functionality to record and process diagnostic data from a performed emulation – this can include information on travelled carrier paths and times, collision detection and time-out errors.

Two quantitative measurements that are easy to obtain through this presented Erlang-Simio application, are throughput and resource utilization. For the conveyor, Simio reports data on each conveyor segment – e.g. the throughput, maximum and minimum carriers present at a given time and average time spent by carriers in that segment. When the other Resource holons are also integrated in the Simio emulation, Simio can report the time a specific resource was used during the total emulation time.

3.3.7. Conclusion and Future Work

This paper presented the use of simulation software in the validation of a holonic control implementation. The case study focused on the validation of an Erlang based holonic controller for a modular conveyor system, where Simio is used to provide a hardware emulation model.

To create an interface between the holonic controller and the emulation model, an Interpreter program was developed. The Interpreter program maintains an interface that emulates that of the physical conveyor system by handling TCP communication on multiple network sockets. The Interpreter also provides the means for communication with the emulation model using several instances of WCF services.

With further enhancement, the use of the Simio emulation model could prove to be valuable in the control validation process. The emulation of customized production scenarios is a great advantage in the context of reconfigurable manufacturing systems. The object-oriented nature of Simio also strongly resembles the principles of holonic systems and it thus interfaces well with higher level holonic control implementations.

Future work will entail the enhancement of the Simio emulation, with particular focus on enriched information flow between the control and emulation levels, and also the incorporation of measurement tools within Simio to capture and interpret diagnostic information from emulation experiments. Further work will be done on the construction of Simio models to accurately represent the real system components.

3.3.8. References

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

2. Y. Koren and M.Shpitalni, “Design of Reconfigurable Manufacturing Systems”, *Journal of Manufacturing Systems*, Vol. 29, pp. 130-141, 2010.

3. Y. Koren, U. Heisel, F. Jovane, T. Moriwaki, G. Pritschow, G. Ulsoy and H. Van Brussel, “Reconfigurable Manufacturing Systems”, *Annals of CIRP*, Vol. 48, No. 2:527-540, 1999.

4. A. Koestler, *The Ghost in the Machine,* London: Arkana Books, 1967.

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

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

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

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

9. P. Valckenaers and H. van Brussel, *Design for the Unexpected,* 1st edition, Butterworth-Heinemann, ISBN: 9780128036624, 2015.

10. C.D. Pegden, "Simio: A New Simulation System Based on Intelligent Objects", *Proceedings of the 2007 Winter Simulation Conference*, pp. 2294-2300, 2007.

11. C.D. Pegden, "Introduction to Simio", *Proceedings of the 2008 Winter Simulation Conference*, pp. 229-235, 2008.

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

13. K. Kruger and A.H. Basson, "Erlang-based Holonic Controller for a Modular Conveyor System", *SOHOMA'16 Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*, submitted for publication.

14. J. Lowy and M. Montgomery, *Programming WCF Services*, 4th edition, O'Reilly Media, 2015.

4. Erlang Holonic Control Implementation

This section describes the development of a holonic control implementation using Erlang. The development is documented in three papers, each presented in a dedicated subsection.

The first paper, “Implementation of an Erlang-based Resource Holon for a Holonic Manufacturing Cell” (Kruger and Basson, 2015), is presented in section 4.1 and describes the implementation of a Resource holon using Erlang. The holon internal architecture is described in terms of the functional components and inter- and intra-holon communication. The paper also presents the case study implementation of the Erlang-based Resource holon for a pick-‘n-place robot. The paper was presented at the fourth international workshop on Service Orientation in Holonic and Multi-Agent Manufacturing (SOHOMA) in Nancy, France, in 2014.

In section 4.2, the second paper presents a methodology for implementing holons using Erlang and is titled “Erlang-based Control Implementation for a Holonic Manufacturing Cell” (Kruger and Basson, 2017 (a)). The methodology provides guidelines for the implementation of holon functionality and the facilitation of communication in holonic control implementations. An implementation of the presented methodology is illustrated through the extension of the case study introduced in section 4.1. This paper was published in the International Journal of Computer-Integrated Manufacturing in March of 2017.

The third paper, “Erlang-based Holonic Controller for a Modular Conveyor System” (Kruger and Basson, 2017 (c)), is included in section 4.3 and describes the holonic control implementation for a conveyor system. The development of a holonic controller was required for the conveyor system that forms part of the testbed system, as discussed in section 3.2. The controller for the conveyor system was implemented using Erlang, providing an additional case study example. The Erlang holonic controller is also used by the MAS implementation (discussed in chapter 5) and is not included in the evaluation and comparison presented in chapter 6. The paper was presented at the sixth international workshop on Service Orientation in Holonic and Multi-Agent Manufacturing (SOHOMA) in Lisbon, Portugal, in 2016.

4.1. Implementation of an Erlang-based Resource Holon for a Holonic Manufacturing Cell

Karel Kruger [a] and Anton Basson [a,*]
[a] Dept of Mechanical & Mechatronic Eng, Stellenbosch Univ, Stellenbosch 7600, South Africa
[*] Corresponding author. Tel: +27 21 808 4250; Fax: +27 21 866 155 206;
ahb@sun.ac.za

Abstract

The use of holonic control in reconfigurable manufacturing systems holds great advantages, such as reduction in complexity and cost, along with increased maintainability and reliability. This paper presents an implementation of holonic control using Erlang, a functional programming language. The paper shows how the functional components of a PROSA Resource holon can be implemented through Erlang processes. The subjection of a case study implementation to a reconfigurability experiment is also discussed.

Keywords: Erlang/OTP; Holonic Manufacturing Systems; Reconfigurable Manufacturing Systems; Manufacturing Execution Systems; Automation

4.1.1. Introduction

Reconfigurable Manufacturing Systems (RMSs) are aimed at addressing the needs of modern manufacturing, which include [1]: short lead times for the introduction of new products into the system, the ability to produce a larger number of product variants, and the ability to handle fluctuating production volumes and low product prices.

RMSs then aim to switch between members of a particular family of products, by adding or removing functional elements, with minimal delay and effort [2, 3]. For achieving this goal, RMSs should be characterized by [4, 5]: modularity of system components; integrability with other technologies; convertibility to other products; diagnosability of system errors; customizability for specific applications; and scalability of system capacity. RMSs thus have the ability to reconfigure hardware and control resources to rapidly adjust the production capacity and functionality in response to sudden changes [1, 6].

A popular approach for enabling control reconfiguration in RMSs is the idea of holonic control. Holons are “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” [7]. Applied to manufacturing systems, a holon is an autonomous and cooperative building block for transforming, transporting, storing or validating information of physical objects.

Several experimental implementations of holonic control have been done using agent-based programming (such as in [8]), often using JADE as development tool. From our experiences with JADE, we believe there is room for improvement concerning complexity, industry acceptance, robustness and scalability.

This paper describes the implementation of holonic control using Erlang. Erlang is a functional programming language developed for programming concurrent, scalable and distributed systems [9]. The Erlang programming environment is supplemented by the Open Telecommunications Platform (OTP) - a set of robust Erlang libraries and design principles providing middleware to develop Erlang systems [10].

Erlang has the potential to narrow the gap between academic research and industrial implementation. This is due to several advantages offered by the Erlang language, such as increased productivity, reliability, maintainability and adaptability.

This paper describes an Erlang-based implementation of the control component for a PROSA Resource holon in a reconfigurable manufacturing cell, focusing on:

- The functional components of a Resource holon which must be incorporated by the Erlang implementation (section 4.1.2)
- A case study which demonstrates the Erlang-based holonic control (section 4.1.3)
- The implementation of the functional components of Resource holon control through Erlang/OTP processes (section 4.1.4)
- The reconfigurability of the Resource holon in reaction to changes in the holon’s service specification (section 4.1.5).

4.1.2. Holonic Control

4.1.2.1. Holonic Architecture

There are several existing reference architectures which specify the mapping of manufacturing resources to holons and to structure the holarchy (e.g. [11], [8]). Of these reference architectures, the most prominent is that of PROSA [12].

PROSA (Product-Resource-Order-Staff Architecture) defines four holon classes: Product, Resource, Order and Staff. The first three classes of holons are basic holons, which interact by means of knowledge exchange. The process knowledge, which is exchanged between the Product and Resource holons, is the information 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 which aid the basic holons by providing them with expert knowledge to reduce work load and decision complexity.

4.1.2.2. Resource Holon Internal Architecture

A Resource holon requires several capabilities, such as communication, execution control and hardware interfacing. The Resource holon model used for the implementation is described in this section.

[13] 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. [14] share a similar view of the internal architecture of a resource holon, as is illustrated in Figure 13 (a).

The communication component is responsible for the inter-holon interaction – i.e. the information exchange with other holons in the system. The decision-making component is responsible for the manufacturing control functions, and so regulates the behaviour and activities of the holon. The interfacing component provides mechanisms to access the manufacturing resources, monitor resource data and execute commands in the resource.

4.1.3. Case Study

The case study chosen for the presented Erlang-based holonic control implementation, as shown in Figure 12, entails the testing of circuit breakers. The station utilizes a pick-‘n-place robot to move circuit breakers from an incoming fixture to specified tester slots, in a specified sequence. Upon completion of the testing, the robot removes the circuit breakers and places them in the outgoing fixture. Breakers on the same fixture may require testing in different tester slots, which differ in testing parameters and times.

The robot utilized in the case study is a Kuka KR16 robot, fitted with two pneumatic grippers at the end effector (only one of the grippers is used in this implementation). A mock testing rack with four tester slots is used – the slots are fitted with a spring-loaded clamp to hold the breakers in place during testing.

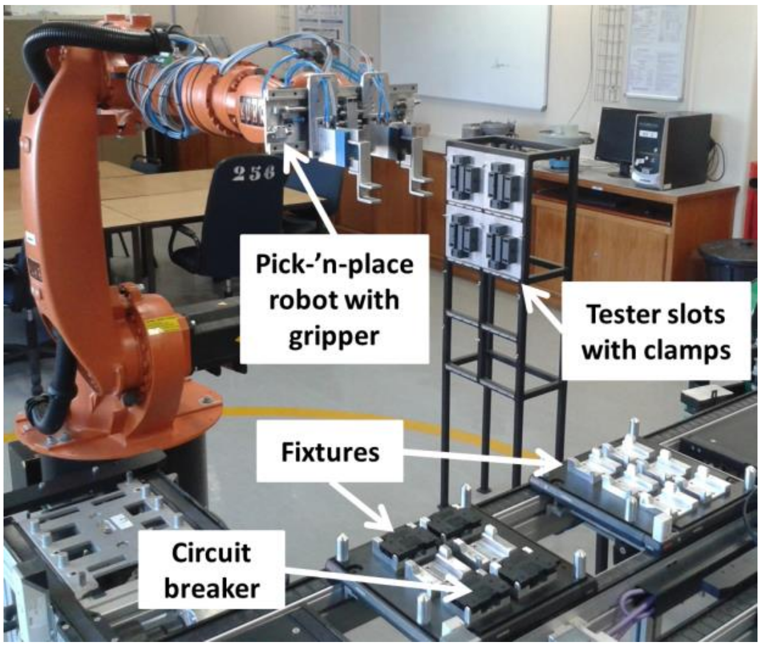


Figure 12: Circuit breaker test station.

4.1.4. Erlang-based Control Implementation

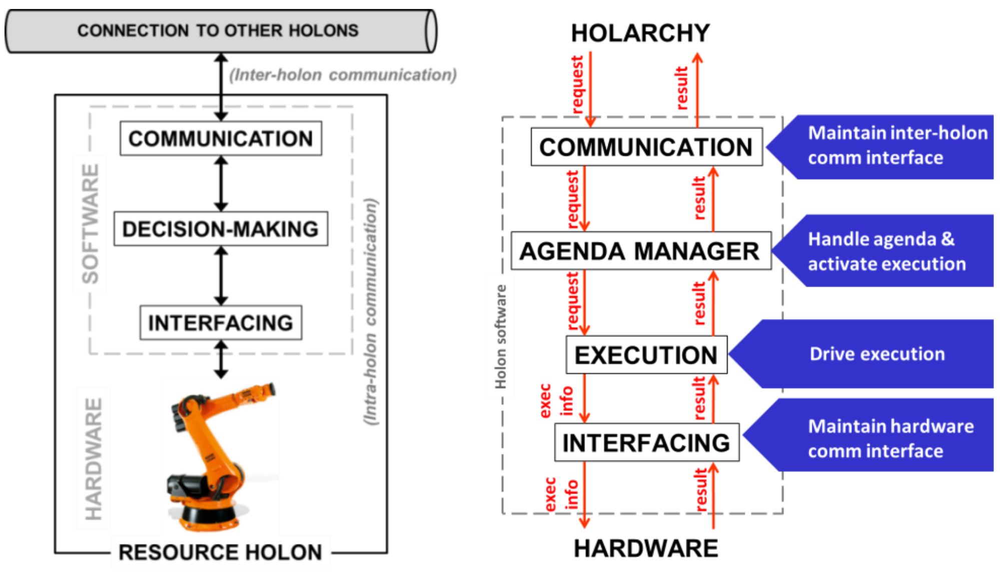
4.1.4.1. Product, Order and Staff Holon Implementation

Though not the focus of this paper, Product, Order and Staff holons are included in the holonic control implementation. A Product holon for each type of circuit breaker is included – this holon contains the information for testing parameters and sequence. For each breaker on the incoming fixture an Order holon is launched to drive production. These holons acquire the resource services necessary to complete the testing process. A Staff holon is included to facilitate the allocation of resource services to requesting Order holons.

4.1.4.2. Resource Holon Implementation

For the presented implementation a Resource holon was created for the robot and each of the tester slots. While the implementation of the Robot holon is complete, the service of the tester slot holons are simulated by replacing their hardware components with timer processes.

For the Robot holon, the software components are implemented on a separate PC which interfaces with the hardware via the robot's dedicated controller. The internal holon architecture, inter- and intra-holon communication and the holon functional components are briefly discussed in this section (a detailed discussion is presented in [15]).



(a) (b)

Figure 13: (a) A generic (adapted from [14]) and (b) the adapted Resource holon model.

4.1.4.2.1. Internal Architecture and Communication.
For the Erlang/OTP implementation, the internal architecture of a Resource holon (Figure 13 (a)) is adapted to that shown in Figure 13 (b). Though the *Communication* and *Interfacing* components are present in both models, the *Decision-making* component in Figure 13 (a) is split into two components, namely the *Agenda Manager* and *Execution* components.

The communication within the Erlang implementation can be classified as either inter- or intra-holon communication. Inter-holon communication is the exchange of messages between the different holons in the system, while intra-holon communication refers to the messages sent between the holon's software and hardware components.

Messages follow the tuple format `{Sender, Message}`. `Sender` holds the address of the process sending the message and `Message` holds the payload of the message. The payload, for messages received by a resource holon, is in the form of a record.

In addition to the inter-holon communication, Figure 13 (b) also shows intra-holon communication in terms of *requests*, *results* and *execution information*. As the *Communication* component receives messages from other holons requesting a service, *request* messages are formulated and forwarded to the *Agenda Manager* component. The *Agenda Manager* processes the request and responds to the *Communication* component, which in turn formulates and sends a reply to the requesting holon. The *Agenda Manager* can also send a message to the *Execution* component to activate execution. The *Execution* component parses the message to extract the *execution information* which is passed on to the hardware. The hardware, and subsequently the *Execution* component, gives feedback in the form of *result* messages.

4.1.4.2.2. Holon Functional Components.
The Resource holon model of Figure 13 (b) has four components: *Communication*, *Agenda Manager*, *Execution* and *Interfacing*.

The *Communication* component of the Resource holon is responsible for maintaining the inter-holon communication interface. It receives *request* messages from other holons in the system, evaluates the message type and content and forwards the message to the appropriate holon component. The holon component then returns a *result* message, which the *Communication* component then sends to the requesting holon.

The component is implemented as a single Erlang process running a *receive-evaluate* loop. This recursive process receives messages from other holons and, by means of pattern matching, identifies relevant messages and then forwards the necessary information to the appropriate holon component. The *Communication* component's process also receives intra-holon messages – by the same means the messages are forwarded to the corresponding holon.

The *Agenda Manager* component manages the service of the Resource holon. The component manages a list of service bookings by order holons and triggers the *Execution* component, with the necessary execution information, according to the agenda.

The *Agenda Manager* component is implemented through two processes - a *receive-evaluate* loop, for receiving messages, and a generic finite state machine (FSM) behaviour (using the OTP *gen_fsm* library). Through pattern-matching, received messages are related to events which cause state transitions in the FSM.

The *Execution* component of the holon drives the hardware actions which constitute the service(s) of the resource holon. It activates the sequential execution of hardware functions, with the necessary execution information.

The *Execution* component is also implemented using a *receive-evaluate* loop, for receiving messages, and a generic FSM behaviour. The required sequence of hardware actions is formulated into this FSM. With each execution state, the necessary activation and information messages are sent to the hardware via the *Interfacing* component. The process receives feedback regarding the execution status from the hardware – these messages are then used as events to trigger the transitions between the states. When execution is completed, the execution result is forwarded to the *Agenda Manager* and *Communication* components and ultimately replied to the Order holon.

Figure 14 (a) shows the execution state diagram for the Robot holon. This example shows three states: “ready”, “picking” and “placing” – each representing an execution state of the robot. The FSM switches between states in accordance with received messages from the *Agenda Manager* and the hardware.

The *Interfacing* component maintains the communication interface between the Erlang control processes and the program on the robot controller. This component isolates the hardware-specific communication structures from the *Execution* logic.

This component is implemented using a *receive-evaluate* loop for receiving messages and a process for TCP communication. For TCP communication, the process utilizes communication functions from the OTP *gen_tcp* and XML functions from the XMErL libraries [16].

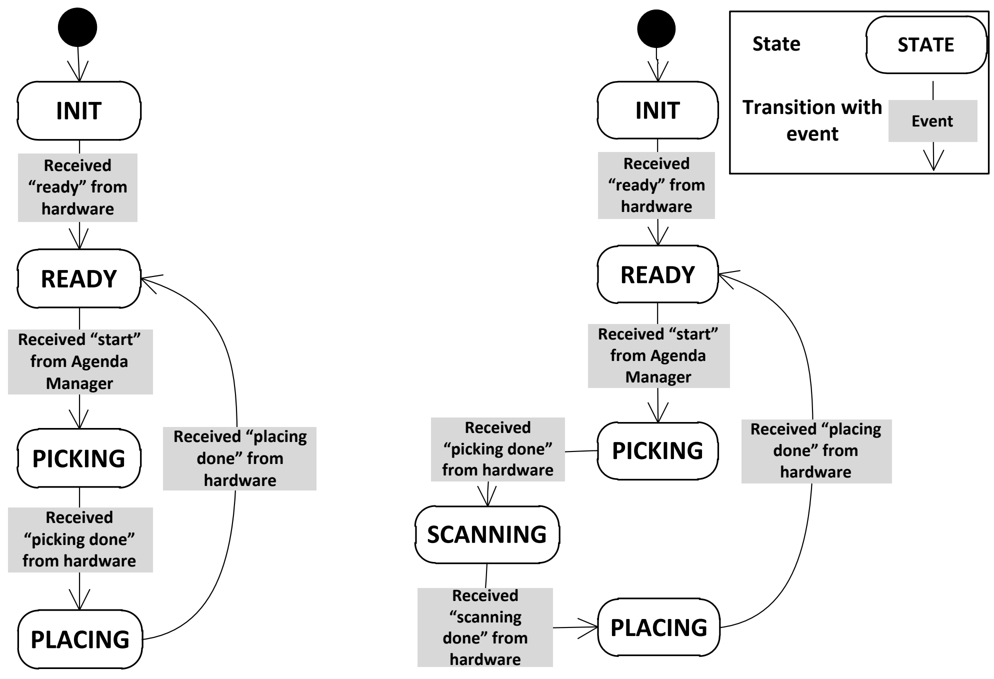
In addition to the OTP functionality used in the holon implementation described above, more tools offered by Erlang/OTP are available for enhancing the implementation. Two tools which can be very useful are the *Supervisor* and *Logging* modules. For this implementation, a *Supervisor* process for all the discussed components is included. The *Supervisor* process launches and shuts down the processes in a specified order and restarts the components if they fail. Erlang/OTP includes an *error_logger* module [17] which is used to output error, warning and information reports to the terminal or to file. The format of these reports can be customized according to the needs of the application.

4.1.5. Reconfiguration Experiment

A reconfiguration experiment was performed on the case study implementation to demonstrate the reconfigurability of the Erlang-based Resource holon. The experiment entailed a change to the service that is provided by the Robot holon – more specifically, the service was adjusted to include a scanning operation. The pick-‘n-place robot must then, prior to placing, bring the circuit breaker to the vicinity of a scanner.

The added scanning function is only intended for diagnostic purposes and does not entail a change to the product information. The addition then only affects the Robot holon, and not the Order or Product holons.

The scanning function must be included in the *Execution* component of the Robot holon. This means that an additional state must be added to the FSM. The state diagrams of the FSM before and after the addition of the scanning function are shown in Figure 14.


INIT
Received "ready" from hardware
READY
Received "start" from Agenda Manager
PICKING
Received "picking done" from hardware
PLACING
Received "placing done" from hardware
State
STATE
Transition with event
Event
INIT
Received "ready" from hardware
READY
Received "start" from Agenda Manager
PICKING
Received "picking done" from hardware
SCANNING
Received "scanning done" from hardware
PLACING
Received "placing done" from hardware

(a) (b)

Figure 14: *Execution* state diagrams (a) before and (b) after adding the scanning function.

The following code snippet shows the code for the FSM prior to the addition of the scanning operation:

```
init(_) -> {ok,ready,[]}.
%STATE: ready
ready(Msg=#service{message_type=start,info=#coords{}},_)
->
```

```
4)   robot_pi ! {robot_exec,
   #service{message_type=start,
   info=Msg#coords.pick_coords}},
5)   {next_state,picking,[Msg#service.info]}.
6) %STATE: picking
7) picking(Msg=#service{message_type=done,result=true},
8) Coords) ->
9)   robot_pi ! {robot_exec,
   #service{message_type=start,
   info=Coords#coords.place_coords}},
10)   {next_state,placing,[]}.
11) %STATE: placing
12) placing(Msg=#service{message_type=done,result=true}, _)
   ->
13)   robot_am ! {robot_exec,
   Msg#service{message_type=done,result=true}},
14)   {next_state,ready,[]}.
```

The states are defined as function heads (e.g. lines 3, 7 and 12) – the functions take two input arguments: a transition event and the state data. When the transition event occurs (e.g. a message is received), actions are performed and the new state is specified. Here the actions involve sending messages to other processes using the "!" operator (e.g. lines 4, 9 and 13). The new state to transition to is specified by `{next_state, StateName, StateData}`, as is shown in lines 5, 10 and 14. The following code snippet shows the inserted code for the additional scanning operation:

```
%STATE: picking
picking(Msg=#service{message_type=done,result=true},
Coords) ->
  robot_pi ! {robot_exec,
#service{message_type=start, info=?ScanCoords}},
  {next_state,scanning,[Coords]}.
 %STATE: scanning
 scanning(Msg=#service{message_type=done,result=true
},   Coords) ->
   robot_pi !
{robot_exec,#service{message_type=start,
info=Coords#coords.place_coords},
 {next_state,placing,Coords}.
 %STATE: placing
 placing(Msg=#service{message_type=done,result=true}
, _) -> …
```

The inserted code shows the definition of the new *scanning* state and, in lines 9 and 13, updates the transitions from and to the *picking* and *placing* states. The fixed coordinates of the scanner are defined in the module as the macro `?ScanCoords`.

The code shown above is added to the *Execution* FSM module and can, through hot code-loading, replace the old FSM code while the holon is operating.

4.1.6. Conclusion

RMSs commonly employ holonic control architectures to enable the rapid reconfiguration of hardware and control resources to adjust production capacity and functionality. This paper shows that Erlang/OTP is an attractive solution for implementing holonic control and presents an implementation of a Resource holon as example.

The implementation example uses a pick-‘n-place robot as Resource holon. The robot picks up circuit breakers from a fixture, places them in testers and ultimately removes them again. The paper describes the implementation of the functional holon components as Erlang processes, with specific use of the OTP generic finite state machine library. The reconfigurability of the holon is demonstrated through an experiment where an additional operation is added to the pick-‘n-place process. The experiment shows that reconfiguration is easy, as the FSM code offers good encapsulation of functionality and state transitions are clearly defined and easily changed. The reconfiguration could also have been done during holon operation.

Future work will entail the expansion of the Erlang/OTP implementation to the execution control system for an entire manufacturing cell, in which all of the PROSA holons will be incorporated.

4.1.7. References

1. 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
2. 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
3. Vyatkin, V., 2007. IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design. *North Carolina: Instrumentation, Systems and Automation Society, ISA*
4. Mehrabi, M.G., Ulsoy, A.G., Koren, Y., 2000. Reconfigurable Manufacturing Systems: Key to Future Manufacturing. *Journal of Intelligent Manufacturing*. Vol. 13: 135 - 146
5. ElMaraghy, H., 2006. Flexible and Reconfigurable Manufacturing System Paradigms. *International Journal of Flexible Manufacturing System*. Vol. 17:61-276
6. Bi, Z.M., Wang, L. and Lang, S.Y.T., 2007. Current Status of Reconfigurable Assembly Systems. *International Journal of Manufacturing Research*, Inderscience. Vol. 2, No. 3: 303 - 328
7. Paolucci, M. and Sacile, R., 2005. *Agent-Based Manufacturing and Control Systems*. London: CRC Press

8. Leitao, P. and Restivo, F.J., 2006. ADACOR: A Holonic Architecture for Agile and Adaptive Manufacturing Control. *Computers in Industry*. Vol. 57, No. 2: 121-130
9. Armstrong, J., 2003. *Making Reliable Distributed Systems in the Presence of Software Errors*. Doctor’s Dissertation. Royal Institute of Technology, Stockholm, Sweden
10. *Get Started with OTP*. [S.a.]. [Online]. Available: http://www.erlang.org (18 July 2013)
11. Chirn, J.L. and McFarlane, D., 2000. A Holonic Component-based Approach to Reconfigurable Manufacturing Control Architecture. *Proceedings of the International Workshop on Industrial Applications of Holonic and Multi-Agent Systems*. pp. 219–223.
12. 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
13. 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
14. Leitao, P. and Restivo, F.J., 2002. A Holonic Control Approach for Distributed Manufacturing. Knowledge and Technology Integration in Production and Services: Balancing Knowledge and Technology in Product and Service Life Cycle. pp. 263–270. Kluwer Academic Publishers.
15. Kruger, K. and Basson, A.H., 2014. An Erlang-based Holon for Reconfigurable Manufacturing Systems. Submitted To: *Computers for Industry*.
16. *XMErL Reference manual*. [S.a.]. [Online]. Available: http://www.erlang.org/doc/apps/xmerl (28 March 2014)
17. *Erlang Kernel Reference Manual*. [S.a.]. [Online]. Available: http://www.erlang.org/doc/apps/kernel (28 March 2014)

4.2. Erlang-based Control Implementation for a Holonic Manufacturing Cell

Karel Kruger [a] and Anton Basson [a,*]
[a] Dept of Mechanical and Mechatronic Eng, Stellenbosch Univ, Stellenbosch 7600, South Africa
[*] Corresponding author. Tel: +27 21 808 4250; Email: ahb@sun.ac.za

Abstract

Holonic control is generally used in reconfigurable manufacturing systems since the modularity of holonic control holds the promise of easier reconfiguration, reduction in complexity and cost, along with increased maintainability and reliability. As an alternative to the commonly used agent-based approach, this paper presents an Erlang-based holon internal architecture and implementation methodology that exploits Erlang's capabilities. The paper shows that Erlang is well suited to the requirements of holonic and reconfigurable systems - due to strong modularity, scalability, customizability, maintainability and robustness characteristics.

Keywords: Erlang/OTP; Holonic manufacturing system (HMS); Reconfigurable manufacturing system (RMS)

4.2.1. Introduction

The concept of Reconfigurable Manufacturing Systems (RMSs) is aimed at addressing the needs of modern manufacturing, as have been shaped by aggressive global competition and uncertainty resulting from dynamic changes in economical, technological and customer trends (Leitao and Restivo, 2006). The critical requirements for modern manufacturing systems include (Bi *et al*., 2008) short lead times for the introduction of new products into the system, the ability to produce a larger number of product variants and the ability to handle fluctuating production volumes.

RMSs aim to switch between members of a particular family of products, by adding or removing functional elements (hardware or software), with minimal delay and effort (Martinsen *et al.*, 2007; Vyatkin, 2007). RMSs are also designed to be able to rapidly adjust the production capacity and functionality in response to sudden changes, by reconfiguring hardware and control resources (Bi *et al.*, 2008; Bi, Wang, and Lang, 2007). RMSs therefore should be characterised by (Mehrabi, Ulsoy, and Koren, 2000; ElMaraghy, 2006): modularity of system components, integratability with other technologies, convertibility to other products, diagnosibility of system errors, customizability for specific applications and scalability of system capacity..

A popular approach for enabling control reconfiguration in RMSs is holonic control architectures. The term holon (first introduced by Koestler in 1967) 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 common goal) which integrates the entire range of manufacturing activities" (Paolucci and Sacile, 2005).

The application of the holonic concept to manufacturing control systems has been a popular field of research since the early 1990's. Even though several experimental implementations have been reported, predominantly based on agent based programming (such as Leitao and Restivo (2006)), we believe there is room for improvement in terms of reduced complexity, greater potential for industry acceptance, better robustness/fault-tolerance and better inherent scalability.

This paper evaluates a new alternative to agent-based approaches: the implementation of holonic control using the Erlang programming language. Erlang is a concurrent, functional programming language which was developed for programming concurrent, scalable and distributed systems. In Erlang, many lightweight processes can be employed to work concurrently while distributed over many devices. Processes are strongly isolated, having no shared memory, and can only interact through the asynchronous sending and receiving of messages (Armstrong 2003). The Erlang programming environment is supplemented by the Open Telecommunications Platform (OTP) - a set of robust Erlang libraries and design principles providing middle-ware to develop Erlang systems (Anonymous, s.a. (a); Logan, Merrit, and Carlsson, 2011).

The objective of this paper is to present an Erlang-based internal architecture for holons and an implementation methodology, targeting a reconfigurable manufacturing system. A resource holon in the PROSA holonic control architecture (discussed in section 4.2.2.2) is used as a prototype since it contains all the architectural elements required for the other holon types, as well as hardware interfacing.

4.2.2. Holonic Control

This section motivates the use of the holonic control approach and gives some background regarding reference architectures. The generic resource holon model, used for the Erlang implementation, is also discussed.

4.2.2.1. Advantages of Holonic Control

The use of holonic control for RMSs holds many advantages: Holonic systems are resilient to disturbances and adaptable in response to faults (Vyatkin 2007); have the ability to organise production activities in a way that they meet the requirements of scalability, robustness and fault-tolerance (Kotak *et al.*, 2003); and lead to reduced system complexity, reduced software development costs and improved maintainability and reliability (Scholz-Reiter and Freitag, 2007).

4.2.2.2. Holonic Architecture

The full utilization of the above-mentioned advantages relies on the holonic system's architecture. Several reference architectures, which specify the mapping of manufacturing resources to holons and to structure the holarchy, have been proposed (e.g. Chirn and McFarlane (2000); Leitao and Restivo (2006)), but the most prominent is PROSA (Product-Resource-Order-Staff Architecture) (Van Brussel *et al*., 1998).

PROSA defines four holon classes: product, resource, order and staff. The first three classes of holons can be classified as basic holons, because, respectively, they represent three independent manufacturing concerns: product-related technological aspects (product holons), resource aspects (resource holons) and logistical aspects (order holons).

The basic holons can interact with each other by means of knowledge exchange, as is shown in Figure 15. 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.

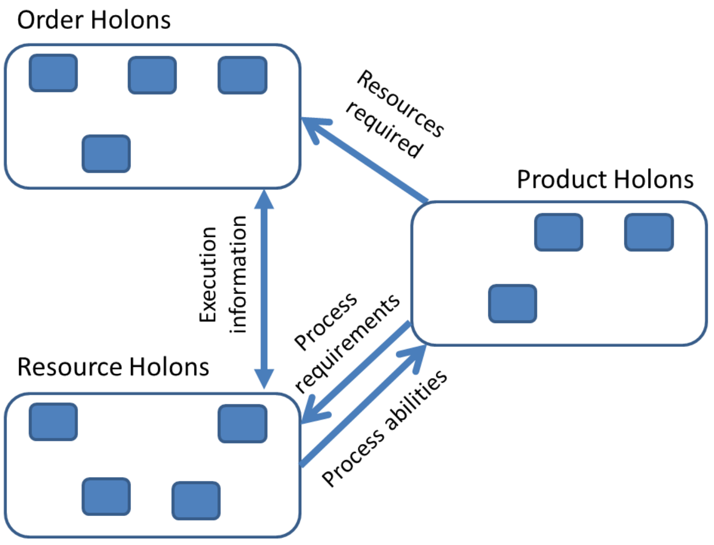


Figure 15: Knowledge exchange between the PROSA holons.

Staff holons are considered to be special holons as they 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 holonic characteristics of PROSA contribute to the different aspects of reconfigurability mentioned in section 4.2.1. The ability to decouple the control algorithm from the system structure, and the logistical aspects from the technical aspects, aids integrability and modularity. Modularity is also provided by the similarity that is shared by holons of the same type.

4.2.2.3. Resource Holon Model

The paper uses the resource holon as case study because of the range of capabilities that is required, such as communication, execution control and hardware interfacing. The resource holon model used as starting point is described in this section – an adapted model for implementation with Erlang follows in section 4.2.4.1.

The internal architecture of a resource holon is illustrated in Figure 16. Individual holons have at least two basic parts (Kotak *et al.*, 2003; Leitao and Restivo, 2002): a functional component and a communication and cooperation component. The functional component can be represented by a purely software entity or, as in resource holons, it could be a hardware interface represented by a software entity. The communication and cooperation component of a holon is implemented by software.

The communication component is responsible for the inter-holon information exchange. The decision-making component is responsible for the manufacturing control functions, regulating the behaviour and activities of the holon. The interfacing component handles the intra-holon interaction, providing mechanisms to access the manufacturing resources, monitor resource data and execute commands in the resource.

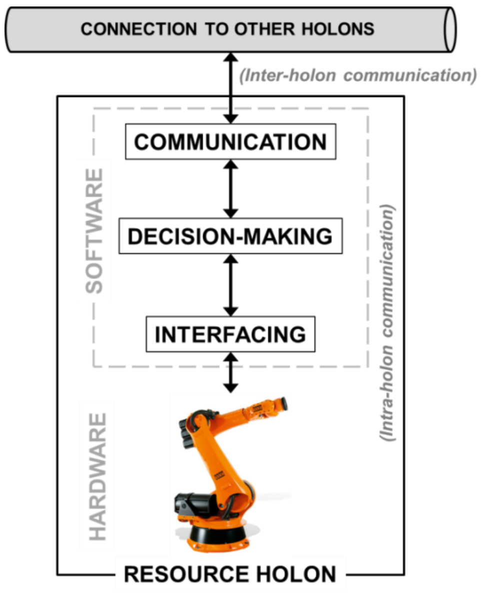

CONNECTION TO OTHER HOLONS
(Inter-holon communication)
SOFTWARE
COMMUNICATION
DECISION-MAKING
INTERFACING
(Intra-holon communication)
HARDWARE
RESOURCE HOLON

Figure 16: Internal architecture of a resource holon (adapted from Leitao and Restivo (2002)).

4.2.3. Advantages of using Erlang for Holonic Control Implementation

There are several inherent characteristics of Erlang which prove to be advantageous for the implementation of holonic control. The most prominent advantages relate to fault-tolerance, service availability and scalability.

The Erlang process model – whereby system functionality is distributed across a number of cooperating and communicating processes – ensures that Erlang is built on an inherently fault-isolating architecture. The processes act as abstraction boundaries, limiting the propagation of error through the system (Armstrong, 2003). This strong fault-tolerant nature of Erlang is further supplemented by the OTP libraries for supervisory structures, which can be utilized to detect and trap system errors and implement strategies to rectify the system behaviour (Armstrong, 2003).

Erlang allows for the updating of code without having to disturb the operation of a running program since it has primitives which allow code to be replaced in a running system (Däcker, 2000). Bug fixes and upgrades can be uploaded to a running system without disturbing the current operation. This capability, along with the previously mentioned fault-tolerance, enables Erlang systems to offer excellent service availability (Armstrong, 2007).

Finally, Erlang provides the infrastructure for massive scalability and concurrency. The lightweight nature of Erlang processes means that millions of processes can be supported on a single processor (Armstrong, 2007). Furthermore, since Erlang processes share no memory and all interaction is done through message passing, processes can easily be distributed over a network of processors (Armstrong, 2003).

A comprehensive comparison of Erlang with other implementation options is beyond the scope of this paper. However, from the authors' experience, the following comments are offered:

Multi-agent systems (MASs) have been often been used to implement holonic control architectures for manufacturing systems and cells. Interestingly, the advantageous characteristics of Erlang can be directly related to what has been identified as the shortcomings of commonly used agent based implementations. Almeida *et al*. (2010) identified that two of the main issues regarding agent-based implementations are that of scalability and fault-tolerance. Due to the high resource requirements of MAS threads (when implemented in Java or C (Vinoski, 2007)), the number of threads that can run on a processor limits scalability – this limitation is emphasized when the implementation is to be done on resource-constrained industrial controllers. In terms of fault-tolerance, there is still work to be done on the implementation of supervisory structures which can identify, diagnose and recover from disturbances or errors.

When considering the Java Agent DEvelopment (JADE) framework specifically, which is often used for holonic control implementations, JADE agent threads suffer drawbacks concerning scalability, as mentioned above, since they Java based. Furthermore, JADE is aimed at providing infrastructure for a wider range of implementations (i.e. beyond that of control applications for manufacturing systems), but this infrastructure is mostly underutilized in the type of implementations presented in this paper. In some cases, this additional functionality adds complexity and coding overhead – a scenario where the sense of "scalable complexity" (the idea that a system can be constructed through the inclusion of only the functions and interfaces for the necessary functionality, and thus complexity, of the system) of Erlang implementations could be beneficial. Lastly, it has been found that programming MASs, even with Java programming experience, involves a significant learning curve.

IEC 61131-3 languages are commonly used for control implementation in manufacturing. While they work well for low level control, attempts to use these languages for implementations of higher level control have achieved limited success. The reason for this, in the experience of the authors, is that the features of these languages that contribute to their reliability on the other hand restrict the flexibility and extensibility of the code that are valuable for the implementation of the high level control of holonic systems. Examples of these restrictions are that the programmes nominally operate in a single thread and that dynamic instantiation of objects, variables or data containers is not possible.

Object orientated programming (OOP) languages offer features between MASs and IEC 61131-3 languages, and can therefore also be considered for developing holonic control systems (Graefe and Basson, 2013). C# and Java appears to have a wide user base in the software world, but their popularity in manufacturing control is uncertain. The authors' research group have found C# to be a productive tool to develop holonic control systems, utilising the classical OOP features. C# has the advantage above Java that drivers for I/O devices are more readily available for C#. However, the resource implications of multiple threads in C# are similar to that for Java. Also, neither of these languages include the "built-in" fault-tolerance and fault-management of Erlang.

4.2.4. Erlang-based Resource Holon

The internal holon architecture, inter- and intra-holon communication and the holon functional components are discussed in this section. Furthermore, a general implementation methodology is described and an implementation case study for the Erlang-based resource holon is presented.

4.2.4.1. Internal Architecture

For the Erlang/OTP implementation, the internal architecture described in section 4.2.2.3 has been adapted to that shown in Figure 17. Though the *Communication* and *Interfacing* components are present in both models, the *Decision-making* component in Figure 2 is split into two components, namely the *Agenda Manager* and *Execution* components.

The division of the *Decision-making* component into the *Agenda Manager* and *Execution* components (discussed in section 4.2.4.2.2) is motivated by two factors: Firstly, for a separation of functionality. By separating the functionality of handling service bookings and that directly concerning execution, reconfigurability is improved – the way in which bookings are handled and how a process must be executed can be changed independently and with minimal effect on the other component. Secondly, for software reusability: while the execution control may differ from holon to holon, the way in which their services are managed is similar. The *Agenda Manager* component can thus be used as a generic inclusion for every service-rendering holon in the system.

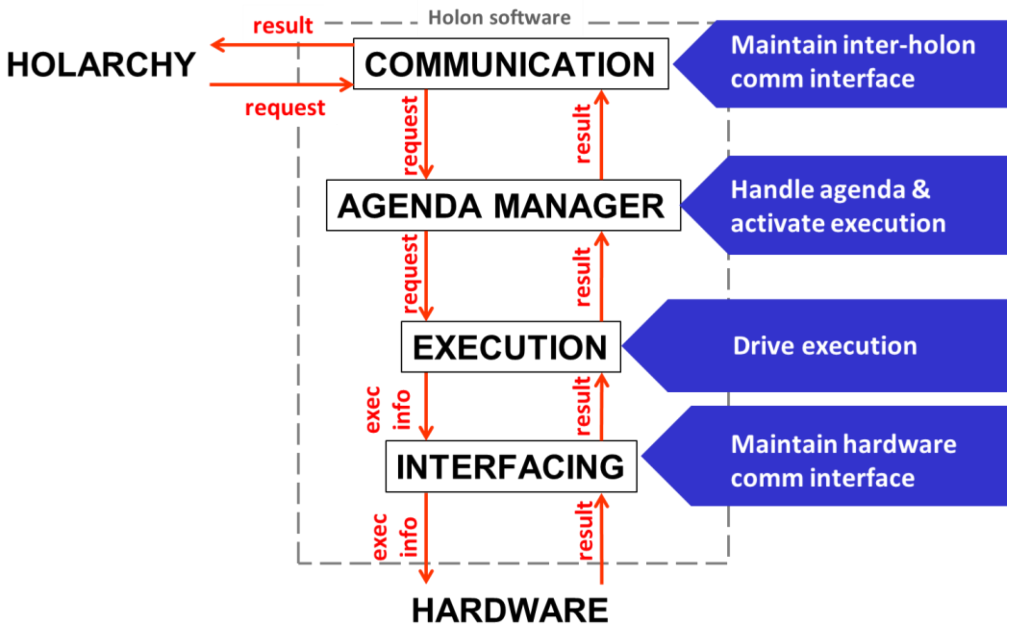


Figure 17: Resource holon model for the Erlang/OTP implementation.

4.2.4.2. Implementation Methodology

This section presents a general implementation methodology for a holonic control system with Erlang/OTP processes. A generic approach to facilitating communication and implementing the holon functional components is described.

4.2.4.2.1. Facilitating Communication

Inter- and Intra-Holon Communication

In holonic systems, communication between system entities can be classified as either inter- or intra-holon communication. Inter-holon communication refers to communication between different holons in the system, while intra-holon communication occurs between the internal components of a holon.

A typical example of inter-holon communication is the request of a resource holon service by an order holon – the order holon sends a request to the resource holon to which the resource holon replies with a request result. These *request* and *result* messages are shown in Figure 3 as interchanged by the *Holarchy* and the resource holon's *Communication* component. In addition to the inter-holon communication, Figure 17 also shows intra-holon communication - indicated as the exchange of *requests*, *results* and *execution information* between the functional components of the resource holon.

Messaging in Erlang

The Erlang process model dictates that information can only be shared amongst processes through messages. Messages are sent using the message operator "`!`" in the following format: `Receiver ! Message`. `Receiver` is a variable[4] that stores the process ID or registered name of the receiving process and the received message is stored in the `Message` variable. Messages can be received by using the `receive` statement with pattern matching, usually implemented in a loop (shown in section 4.2.4.3.1).

For increased traceability, the format by which messages are sent can be implemented as `Receiver ! {Sender,Message}`. In this case, the message payload is placed within a tuple together with the process ID or registered name of the process sending the message. This format offers more options on the receiving side, as pattern matching can then be performed on both the type and content of the message, and from where the message originated.

To further facilitate communication, an ontology can be incorporated in the implementation. The ontology definition can be done in one or many separate header files, and included in the necessary modules. Using *records*, an Erlang data type similar to *structs* in C, sets of information can be defined and used in creating messages and matching messages to patterns. *Records* allow for data fields to be accessed by name instead of order, and multiple *records* can be nested to accommodate complex sets of information. An example of a *record* used to define service messages is shown in section 4.2.4.3.1.

Communication in Functional Components

Taking advantage of the lightweight nature of processes, leading to cheap and easily-managed concurrency, each functional component of the resource holon will be implemented as one or more Erlang processes. For the components to cooperate, information must be exchanged by means of messages. For this reason, each functional component must employ a process which handles this communication.

A simple way to facilitate the communication is to spawn a concurrent process running a *receive-evaluate* loop. The process calls a recursive function which implements a `receive` statement, followed by a set of patterns which will be matched against incoming messages. Upon successfully matching to a pattern, some action can be taken (usually the sending of another message). After each matching case, the function calls itself, resulting in a continuous loop.

The communication process described above separates the communication functionality, within a functional component, from the execution logic. This separation increases the reconfigurability and maintainability of the implementation, as changes can be made to one process without influencing the functionality of the other.

[4] Variables in Erlang start with a capital letter.

4.2.4.2.2. Implementing the Holon Functional Components

Communication Component

The *Communication* component of the resource holon is responsible for maintaining the communication interface with the rest of the holarchy – i.e. all messages to and from other holons are handled by this component.

This component can be implemented using only the communication process discussed in section 4.2.4.2.1. This process then allows for concurrency in the communication and execution functionality of the holon – i.e. the *Communication* component can operate uninterrupted and independent of the other functional components.

Agenda Manager Component

The agenda, in the context of this paper, refers to a list of service commitments (bookings) made by a resource holon to requesting order holons. The construction and management of such a list provides opportunity for the implementation of strategies to improve the performance of holonic systems by planning ahead through forecasting and tentatively committing future availability of resources. Two possible strategies that can be implemented are delegate multi-agent systems (D-MAS) (Holvoet and Valckenaers, 2006) and a facilitating supervisor as found in ADACOR (Leitao and Restivo, 2006). With D-MAS, holons delegate the responsibility of populating and consulting the agendas of resource holons to a swarm of lightweight agents. In ADACOR, a supervisor holon facilitates the booking of resource services by task holons, according to forecasts and optimized plans based on the inspection of agendas. Since the implementation of the mentioned strategies predominantly influence the order (or task) holons, the presented *Agenda Manager* component for resource holons will function similarly for both strategies.

The *Agenda Manager* component is responsible for managing the service provided by the resource holon. The component manages a list of service bookings by order holons and triggers the *Execution* component, with the necessary execution information, according to the agenda.

The *Agenda Manager* component requires two functions – one to receive and evaluate messages from the other holon components, and one to manage the resource's service bookings and execution. For handling the messages, a process running a *receive-evaluate* loop, similar to that of the *Communication* component, can be used. The messages are passed on to the process which manages the service.

The logic for the service management could be implemented in different ways. The logic can be implemented in a normal Erlang process or OTP behaviours can be used. OTP provides two useful behaviours – a generic server (*gen_server*) and a generic finite state machine (*gen_fsm*). The logic can thus be implemented in any of the mentioned ways, with the selection based on the approach which best matches the requirements of the service management model. A general summary of the *gen_fsm* behaviour library is provided in section 4.2.7.

Execution Component

The *Execution* component of the holon is responsible for driving the hardware actions related to the service of the resource holon. This component activates the execution of hardware functions, with the necessary execution information and in a specified sequence, to perform the service of the holon.

The *Execution* component is implemented similarly to the *Agenda Manager* component, i.e. a *receive-evaluate* loop process, for receiving messages, and a process for managing the service execution. The service execution can again be done in different ways, but using the finite state machine (FSM) behaviour is an attractive solution as the execution of resource holon services can usually be easily modelled as FSMs.

When using the FSM approach, the required sequence of execution actions is formulated into the *gen_fsm* behaviour. With each execution state, the necessary activation and information messages are sent to the hardware via the *Interfacing* component. The process receives feedback regarding the execution status from the hardware, which trigger the transitions between the states. When execution is completed, the execution result is replied to the *Agenda Manager* component, from where it is forwarded to the *Communication* component and ultimately replied to the order holon.

Interfacing Component

The *Interfacing* component maintains the communication interface between the Erlang control programs and the hardware. This component isolates the hardware specific communication structures from the execution logic.

The *Interfacing* component can be done in two ways, i.e. using OTP functions or using ports (or linked-in port drivers). When using the first approach, the component is implemented by a *receive-evaluate* loop process and a process implementing the OTP libraries for interfacing, such as *gen_tcp* or *gen_udp* (for TCP/IP or UDP communication). With the linked-in port driver approach, a program can be developed in another language (C, Java, etc.) and be wrapped with Erlang. The program can then be used as if it is just a pure Erlang module. This allows for the creation of communication structures which are not incorporated in OTP (such as Profibus or CANbus) or the use of a device specific driver or application programming interface (API). The use of ports and other Erlang/OTP integration tools is discussed in detail by Logan, Merrit, and Carlsson (2011).

Erlang also supports the use of eXtensible Markup Language (XML), which is frequently used with TCP/IP communication. Two popular libraries for XML functionality are XMErL (Anonymous s.a. (b)) and ErlSom (De Jong, 2007). These libraries can be used, in conjunction with *gen_tcp*, to build and parse XML strings and files for use in socket communication.

4.2.4.2.3. Applicability to other PROSA holons

The presented methodology can be extended to the other PROSA holons. As all holons (and holon functional components) communicate through an exchange of messages, the communication process presented in section 4.2.4.2.1 can be applied. The process can be adapted for each specific holon component, according to the messages that may be received.

The *gen_server* and *gen_fsm* OTP behaviours are equally useful in representing the logic of the other holon types. These behaviours are especially applicable to the functionality of the order holon where service bookings must be managed along with task executions.

4.2.4.3. Case Study

As a case study, a resource holon for a pick-‘n-place robot was implemented using Erlang/OTP. This section describes the implementation of the functional components.

4.2.4.3.1. Communication Component

The *Communication* component is implemented as a single *receive-evaluate* loop process. Messages are received and forwarded according to a successful pattern match. To facilitate the communication, a record was created for service-related messages. This record is constructed as follows:

```
#service{message_type, service_type, reply_to, conversation_ID, requester_pid, provider_pid, result, info}
```

- `message_type` - specification of service message, e.g. request, cancel, start.
- `service_type` - service specification, e.g. pick-‘n-place, inspect, transport.
- `reply_to` – holon process ID to which reply must be sent (for inter-holon communication)
- `conversation_ID` - unique reference to the sequence of messages
- `requester_pid` – process ID of the requesting process linked to the service message
- `provider_pid` – process ID of the providing process linked to the service message
- `result` - Boolean result of action linked to service message
- `info` - information linked to the service message

The following code snippet shows the working of the *receive-evaluate* process of the *Communication* component (in this example named `robot_comm`), as pattern matching is used to distinguish between an intra-holon message (from the *Agenda Manager* component) and an inter-holon message (from another holon):

```
rec_messages() ->
	receive
		%message from agenda_manager in reply to service request
		{agenda_manager_fsm, Message=#service{}} ->
			%extract the corresponding process ID
			Pid = Message#service.reply_to,
			%send response to holon
			Pid ! {robot_comm, Message},
			%loop again
			rec_messages();

		%SERVICE message from other holon requesting a service
		{From, Message=#service{}} ->
			%forward message to agenda_manager
			agenda_manager_fsm ! {robot_comm, Message},
			%loop again
			rec_messages()
	end.
```

4.2.4.3.2. Agenda Manager Component

Two processes are used to implement the Agenda Manager component – one for handling communication and one for managing the holon service. The communication is handled by a process similar to that described for the *Communication* component. To manage the service, a process using the OTP behaviour for a generic finite state machine was chosen.

The state diagram used in the *Agenda Manager* FSM is shown in Figure 18. The states of the FSM each constitute two elements: execution status and a list of bookings (combined as a tuple in Figure 18). The execution status reflects whether the holon hardware is currently in operation ("busy") or idle ("free"), while the booking list keeps record of commitments made to requesting holons. The state transitions are driven by messages received from either the *Execution* or *Communication* components.

Code snippets from the *Agenda Manager* FSM are shown below. The code shows how events (which in these cases are the arrival of messages) are handled according to the specific state and how state transitions are specified. The presented code implements the states, events and transitions highlighted in Figure 18. The handling of two different messages is shown when the *Agenda Manager* FSM is in the "free" state – the messages are of types "booking request" and "start", received from order holons. The code also shows the handling of a "done" message from the *Execution* component of the robot holon, in the "busy" state.

```
%STATE: free_booked --> resource is idle, but is booked
free_booked(Message=#service{message_type=booking_req},[Job_list]) ->
	%add request to bookings list
	NewJob_list=lists:append(Job_list,
[Message#service.requester_pid]),
	%reply request result to Order holon via robot_comm
	robot_comm ! {agenda_manager_fsm,Message#service{result=true}},
	%specify the next state and state information
```

```
        {next_state, free_booked, [NewJob_list]};

%STATE: free_booked --> resource is idle, but is booked
free_booked(Message=#service{message_type=start},[Job_list]) ->
        %forward "start" message to resource_exec
        robot_exec ! {agenda_manager_fsm,Message},
        %specify the next state and state information
        {next_state,busy_booked,[Message#service.requester_pid,
        lists:delete(Message#service.requester_pid, Job_list)]}.


%STATE: busy_booked --> resource is busy and is booked
busy_booked(Message=#service{message_type=done},[CurrJob,Job_list]) ->
        %forward result message to Order holon via robot_comm
        robot_comm ! {agenda_manager_fsm,Message},
        %specify the next state and state information
        {next_state,free_booked,[Job_list]}.
```

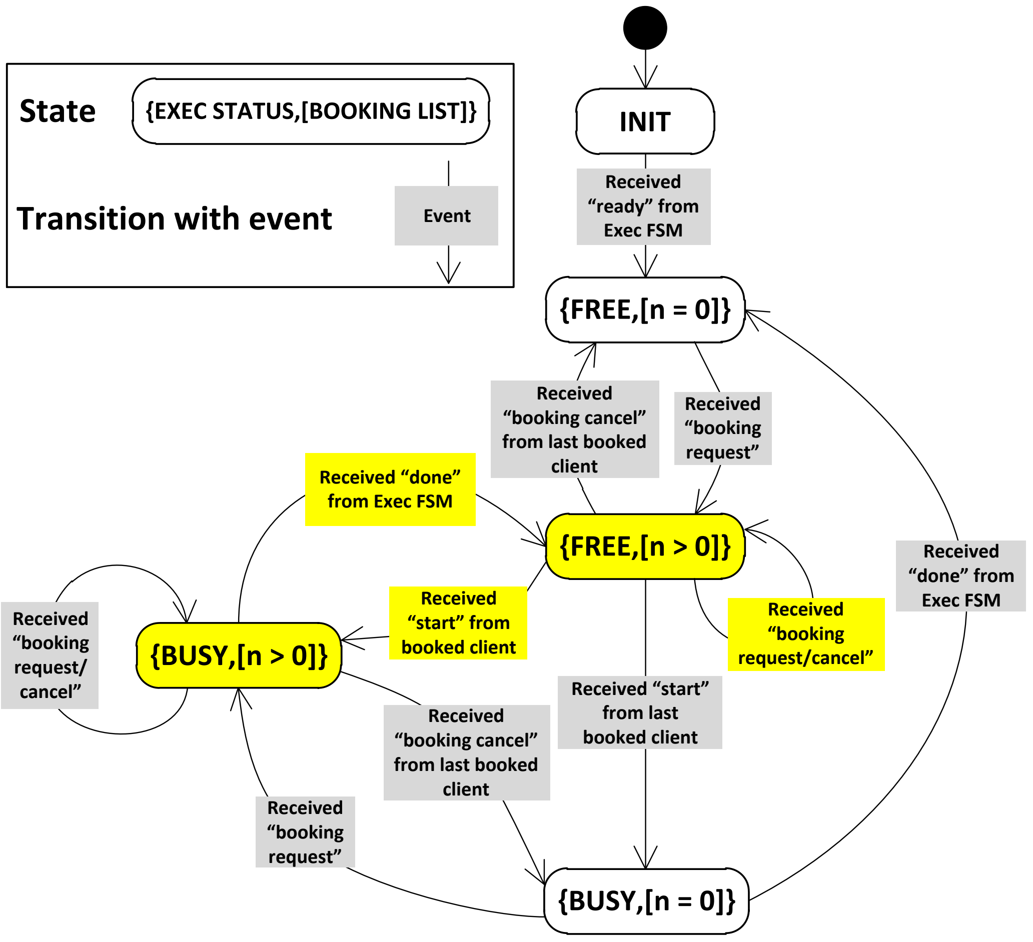

State
{EXEC STATUS,[BOOKING LIST]}
Transition with event
Event
INIT
Received "ready" from Exec FSM
{FREE,[n = 0]}
Received "booking cancel" from last booked client
Received "booking request"
Received "done" from Exec FSM
{FREE,[n > 0]}
Received "done" from Exec FSM
Received "booking request/ cancel"
{BUSY,[n > 0]}
Received "start" from booked client
Received "booking request/cancel"
Received "start" from last booked client
Received "booking cancel" from last booked client
Received "booking request"
{BUSY,[n = 0]}

Figure 18: State diagram of the *Agenda Manager* FSM.

4.2.4.3.3. Execution Component

The *Execution* component is implemented similar to the *Agenda Manager* component – one process for handling communication and a *gen_fsm* process for managing the execution.

Figure 19 shows a simple example of an execution state diagram for the pick-‘n-place robot holon. This example shows three states: “ready”, “picking” and “placing” – each representing an execution state of the robot. The FSM switches between states in accordance with received messages from the *Agenda Manager* and the hardware.

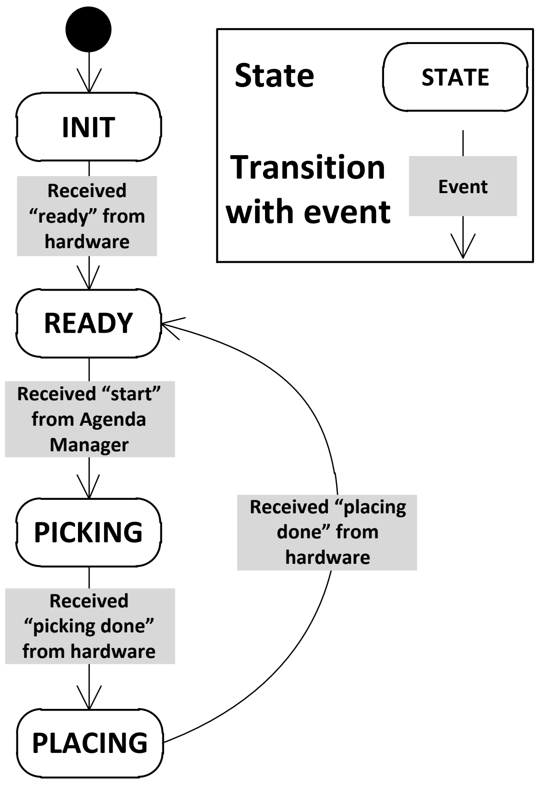

State
STATE
Transition with event
Event
INIT
Received “ready” from hardware
READY
Received “start” from Agenda Manager
PICKING
Received “picking done” from hardware
PLACING
Received “placing done” from hardware

Figure 19: Example state diagram of the *Execution* FSM.

The implementation of the state diagram of Figure 19 using the *gen_fsm* OTP behaviour is shown by the following code snippet:

```
%STATE: ready --> ready to perform operation
ready(Message=#service{message_type=start},_) ->
       %send picking coordinates to interfacing component
       robot_pi ! {robot_exec, Message#service.info.coords.pick_coords},
       %specify the next state and state information
       {next_state, picking, Message}.
```

```
%STATE: picking --> executing picking operation
picking(picking_done, Message) ->
        %send placing coordinates to interfacing component
        robot_pi ! {robot_exec, Message#service.info.coords.place_coords
},
        %specify the next state and state information
        {next_state, placing, {CurrJob, Message}}.

%STATE: placing --> executing placing operation
placing(placing_done, Message) ->
        %send result to agenda manager component
        agenda_manager ! {robot_exec, Message=#service{result=true}},
        %specify the next state and state information
        {next_state, ready, []}.
```

4.2.4.3.4. Interfacing Component

For the case study implementation, the control software of the resource holon interfaced with the controller of the robot through TCP/IP communication. The XMErL library is used to build and parse XML strings. The following code snippet shows how the *gen_tcp* OTP library (briefly summarized in section 4.2.7) is used to communicate to the robot controller:

```
socket_client(Info) ->
        %connect to TCP server
        {ok,Socket} = socket_connect(),
        %build XML string
        XML_string = build_XML(Info),
        %send string
        ok = gen_tcp:send(Socket, XML_string),
        %receive result of operation
        {ok,XML_data} = do_receive(Socket,[]),
        %close socket connection
        ok = gen_tcp:close(Socket),
        %extract result from string
        {XML_doc,_} = xmerl_scan:string(XML_data,[{encoding,latin1}]),
        Msg = extract_content('RESULT',[XML_doc]),
        Message=list_to_atom(Msg),
        Message.

socket_connect() ->
        %connect to socket
        case gen_tcp:connect(?address, ?port,
[list,{packet,0},{active,false}]) of
                %success - return socket reference
                {ok, Socket} -> {ok, Socket};
                %failure - try again
                _ -> timer:sleep(1000),
                           socket_connect()
        end.
```

4.2.4.3.5. Typical operation scenario

To illustrate the sequence of functionality of the presented Erlang based robot holon, the operations involved in a typical scenario will be explained. The scenario entails the receiving of a *start* message from some order holon, i.e. a request from an order holon for the robot holon to start a previously booked service. This scenario was selected as it involves functions from all of the robot holon components.

For the explanation of the of the scenario it is necessary to describe the state of the holon FSM components. Assume that the Agenda Manager FSM is in the "free_booked" state – i.e. the robot holon is currently idle, but its service has been booked for use in the near future by order holons. The *Execution* FSM is in the initial "ready" state, awaiting a *start* message from the *Agenda Manager* to execute a pick-'n-place service.

When the physical part associated with the order holon is in the position for the pick-'n-place service (which was previously booked by the order holon) to be executed, the order holon will request the execution to be started by sending a *start* message to the *Communication* component of the robot holon. As is presented in section 4.2.4.3.1, the *Communication* component continuously awaits the arrival of a message through the *receive* function. When the order holon sends the *start* message, the message is received by the *Communication* components and is compared against the defined message patterns. The start message will match the following pattern:

```
%SERVICE message from other holon requesting a service
{From, Message=#service{}} ->
        %forward message to agenda_manager
        agenda_manager_fsm ! {robot_comm, Message},
        %loop again
        rec_messages()
```

Upon matching the pattern, the *Communication* component will forward the message to the *Agenda Manager* FSM component. The *Agenda Manager* FSM is in the "free_booked" state, thus the *start* message forwarded from the *Communication* component will be compared to the defined state transition patterns. The message will match the event specified by the following transition pattern:

```
%STATE: free_booked --> resource is idle, but is booked
free_booked(Message=#service{message_type=start},[Job_list]) ->
        %forward "start" message to resource_exec
        robot_exec ! {agenda_manager_fsm,Message},
        %specify the next state and state information
        {next_state,busy_booked,[Message#service.requester_pid,
        lists:delete(Message#service.requester_pid, Job_list)]}.
```

The *Agenda Manager* FSM will trigger execution of the service by forwarding the message to the *Execution* component, then transition to the next state "busy_booked". The internal state data of the FSM is also changed – the process ID of the order holon is removed from the list of received bookings and rather stored as an additional variable `CurrJob` (indicating the PID of the order holon involved in the current service execution) in the state data tuple.

The *Execution* component receives the *start* message as an event in the "ready" state (as shown in the code snippet of section 4.2.4.3.3) and proceeds to execute the pickup action of the pick-'n-place service by sending a message – containing the pickup coordinates as stored in the `info` field of the message from the order holon – to the *Interface* component. The *Execution* FSM then transitions to the "placing" state.

The *Interface* component extracts the coordinate information from the message received from the *Execution* component, builds an XML string and sends it to the physical robot controller using the *gen_tcp* library functions. As the robot completes the pickup action, an XML message is sent to the *Interface* component where the message is parsed and sent to the *Execution* component as the Erlang atom `picking_done`.

The interaction between the *Execution* and *Interfacing* components continue as described above until all the actions of the service have been completed – in this scenario, when the *Interfacing* component sends the atom `placing_done` to the *Execution* component. Before the *Execution* component then transitions back to the "ready" state (awaiting a *start* message for the next service execution), it sends a *done* message to the *Agenda Manager* FSM.

The *Agenda Manager* FSM will receive the *done* message from the *Execution* component in the "busy_booked" state. With the *done* message event, the *done* message is forwarded to the *Communication* component (which will use the associated PID field of the message to forward the message to the correct Order holon), before transitioning to the "free_booked" state.

4.2.4.4. Additional Erlang/OTP functionality

In addition to the OTP functionality used in the holon implementation described above, two further tools offered by Erlang/OTP can be very useful, i.e. the *Supervisor* and *Logging* modules.

Through the *Supervisor* module, Erlang allows the implementation of supervision trees, in the form of a process structuring model in terms of workers and supervisors. Worker processes do the computational work, while supervisor processes monitor worker processes. This hierarchical structure allows for the development of fault-tolerant programs, since supervisor processes can start and stop worker processes, and restart them if they should fail (Anonymous, s.a. (a)).

As fault-tolerance is an important requirement for the modern manufacturing environment, supervision trees can be very advantageous. For the implementation of a resource holon, all the components discussed in the previous sections will be worker processes and can be supervised by a supervisor process. Upon starting, the supervisor process launches the processes in a specified order. The order to which they are terminated during shut down is also specified. A restart strategy can be specified for the supervisor process, i.e. the way in which processes are restarted in event of a process failing. Three options are available (Anonymous, s.a. (a)):

- "one-for-one" – only the process that fails is restarted.
- "one-for-all" – if a worker process fails, all of the supervised processes are terminated and restarted.
- "rest-for-one" – if a worker process fails, it and the subsequent processes (in the start order) are terminated and restarted.

A supervisor process can thus be a very useful addition to the holon implementation. At the very least, it provides a neat and simple way to start and stop all the holon processes. With the selection of an appropriate restart strategy, a supervisor process can add great robustness to the holon implementation.

Logging modules offer useful functionality related to diagnosibility, an important requirement for reconfigurable systems. In terms of software diagnosibility, logging is an important tool. Erlang/OTP includes an *error_logger* module (Anonymous, s.a. (c)) which can be used to output error, warning and information reports to the terminal or to file. The format of these reports can be customized according to the needs of the application. The *error_logger* module can be used by all holon processes to log events, errors and general process information to file, e.g. received and sent message information, state transitions and process failures. This information can be helpful for debugging or problem identification, or just for monitoring.

4.2.5. Conclusion

Reconfigurable manufacturing systems (RMSs) are intended for situations characterised by short product life cycles, large product variety and fluctuating product demand, since RMSs have the ability to reconfigure hardware and control resources to rapidly adjust the production capacity and functionality. RMSs commonly employ holonic control architectures, because they share many characteristics.

This paper motivates why the functional programming language Erlang and the Erlang-based OTP (Open Telecom Platform) present an attractive solution for implementing holonic control. It is shown that the requirements for which Erlang was developed are highly relevant to holonic and reconfigurable control. The paper then presents an implementation methodology and case study using Erlang/OTP.

The presented case study for the Erlang/OTP implementation focusses on the resource holon, as defined by PROSA (Product-Resource-Order-Staff Architecture). A generic model for a resource holon to suit an Erlang implementation is presented, with four functional holon components, i.e. communication, agenda manager, execution and interfacing. The implementation of these components, using Erlang/OTP processes, is described.

Future work will entail the expansion of the Erlang/OTP implementation to the control system for an entire manufacturing cell, in which all of the PROSA holons will be incorporated. The Erlang/OTP manufacturing cell will then be subjected to a series of experiments – the results of which will be used to perform a quantitative and qualitative comparison with an equivalent MAS implementation.

4.2.6. References

Almeida, F.L., Terra, B.M., Dias, P.A., and Gonçales, G.M., 2010. Adoption Issues of Multi-Agent Systems in Manufacturing Industry. *Fifth International Multi-conference on Computing in the Global Information Technology*. pp. 238-244.

Anonymous, s.a. (a) *Get Started with OTP*. [Online]. Available: http://www.erlang.org (18 July 2013).

Anonymous, s.a. (b). *XMErL Reference manual.*. [Online]. Available: http://www.erlang.org/doc/apps/xmerl (28 March 2014).

Anonymous s.a. (c). *Erlang Kernel Reference Manual*. [S.a.]. [Online]. Available: http://www.erlang.org/doc/apps/kernel (28 March 2014).

Anonymous s.a. (d). *Erlang/OTP System Documentation.* [S.a.]. [Online]. Available: http://www.erlang.org/doc/pdf/otp-system-documentation.pdf (28 March 2014).

Armstrong, J., 2003. *Making Reliable Distributed Systems in the Presence of Software Errors*. Doctor's Dissertation. Royal Institute of Technology, Stockholm, Sweden.

Armstrong, J., 2007. *Programming Erlang: Software for a Concurrent World*. Raleigh, North Carolina: The Pragmatic Bookshelf.

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

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.

Chirn, J.L. and McFarlane, D., 2000. A Holonic Component-based Approach to Reconfigurable Manufacturing Control Architecture. *Proceedings of the International Workshop on Industrial Applications of Holonic and Multi-Agent Systems*. pp. 219–223.

Däcker, B., 2000. *Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction*. Master's Thesis. Royal Institute of Technology, Stockholm, Sweden.

De Jong, W., 2007. *Erlsom*. [Online]. Available: http://erlsom.sourceforge.net (28 March 2014).

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

Graefe, R. and Basson, A.H., 2013. Control of Reconfigurable Manufacturing Systems using Object-Oriented Programming, *Proceedings of the 5th International Conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV2013)*. pp. 231-236.

Hebert, F., 2014. *Learn Some Erlang For Great Good*. No Starch Press.

Holvoet, T. and Valckenaers, P., 2006. Exploiting the Environment for Coordinating Agent Intentions. *AAMAS Conference*. Hakodate, Japan (8–12 May).

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

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.

Leitao, P. and Restivo, F.J., 2002. A Holonic Control Approach for Distributed Manufacturing. *Knowledge and Technology Integration in Production and Services: Balancing Knowledge and Technology in Product and Service Life Cycle*. pp. 263–270. Kluwer Academic Publishers.

Logan, M., Merrit, E., and Carlsson, R., 2011. *Erlang and OTP in Action*. Stamford: Manning Publications Co.

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.

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

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

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.

Vinoski, S., 2007. Concurrency with Erlang. *IEEE Internet Computing*. Vol. 11, No. 5: 90-93.

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

4.2.7. Appendix: OTP Libraries

This appendix provides a summary of the functionality and programmatic implementation of the OTP libraries mentioned in this paper. The presented work made use of two OTP libraries, namely the generic finite state machine (gen_fsm) behaviour and generic Transmission Control Protocol (gen_tcp) libraries. The description of the gen_fsm library is adapted from Anonymous (s.a. (d)) and, for the gen_tcp library, from Anonymous (s.a. (c)) and Hebert (2014).

4.2.7.1. Generic finite state machine behaviour library

A finite state machine can be described as a set of relations between states, events and actions. These relations can be expressed in the following form:

```
State x Event → Action(s), NextState
```

This expression states that when the FSM is in some `State` and some `Event` occurs, some `Action(s)` will be performed and the FSM will transition to `NextState`. Using the Erlang gen_fsm behaviour, these state transitions can be implemented by:

```
StateName(Event, StateData) ->
     %code for actions here
     {next_state, NextStateName, NewStateData}.
```

The name of the state the FSM is in when Event occurs is programmed as `StateName`. `StateData` represents internal information regarding the current state. When `Event` occurs, specific actions that must be performed can be programmed. After all the required actions are completed, the statement ends with a description of the state transition that follows. The transition description is represented as a tuple with three elements: the first element is the atom `next_state`, designating the transition description; the second element specifies the name of the state to which the FSM will transition to and the last element specifies the internal information associated with the next state.

The following code starts a gen_fsm behaviour in a new process:

```
gen_fsm:start_link({local, FsmName}, ModuleName, InitData,
Options)
```

- `FsmName` – the name by which the FSM process will be registered.
- `ModuleName` – the name of the module where the callback functions of the FSM (i.e. the functions defining the state transitions) are located.
- `InitData` – information passed to the FSM during initialization.
- `Options` – a list of possible options for the gen_fsm process – e.g. timeouts, debugging functions, etc.

When the gen_fsm behaviour is started, it enters the initialization function of the FSM, programmed as:

```
init(InitData) ->
    %code for initialization actions here
    {ok, InitialStateName, StateData}.
```

The function performs the necessary initialization functions and concludes with the definition of the initial state of the FSM. The FSM will consequently transition to `InitialStateName` with the accompanying `StateData`.

With the FSM now occupying a specific state, it can receive notifications regarding the occurrence of events. Processes can notify a specific gen_fsm process of an event using the following function:

```
gen_fsm:send_event(FsmName, Event)
```

This function constructs a message of the `Event` and sends it to the gen_fsm process. The event is handled in the current state of the FSM and will result in some corresponding state transition, as was discussed earlier in this section.

4.2.7.2. Generic Transmission Control Protocol library

The gen_tcp library included in OTP provides functions to communicate with sockets using Transmission Control protocol (TCP). Functions are included for both server and client implementations – the simplest forms of which are briefly presented in this section.

An Erlang process can act as a server for a designated TCP port, using:

```
{ok, Socket} = gen_tcp:listen(Port, Options)
```

- `Port` – the port number for the socket.
- `Options` – a list of socket configuration options.
- `Socket` – data type representing the TCP socket.

As the function name suggests, the server process will listen for incoming connection requests at the specified port. When such a request is received, the connection can be accepted:

```
gen_tcp:accept(Socket)
```

Also, a process can connect to a TCP socket as a client – this functionality is provided through the function:

```
gen_tcp:connect(Address, Port, Options)
```

- `Address` – the IP address or host name for the socket.

When the connection is accepted by the corresponding server process, TCP communication over the connected socket can be achieved. Both the server and client processes use the same functions for the sending and receiving of messages over the socket:

```
gen_tcp:send(Socket, DataPacket)
```

- `DataPacket` – information to be sent over socket.

```
gen_tcp:recv(Socket, Length)
```

- `Length` – the number of bytes to read from the socket.

4.3. Erlang-based Holonic Controller for a Modular Conveyor System

Karel Kruger [a,*] and Anton Basson [a]
[a] Dept of Mechanical & Mechatronic Eng, Stellenbosch Univ, Stellenbosch 7600, South Africa
[*] Corresponding author. Tel: +27 21 808 4258; Fax: +27 21 866 155 206;
kkruger@sun.ac.za

Abstract

Holonic systems have been a popular approach to face the challenges of the modern manufacturing environment and should continue to play a vital role in the fourth industrial revolution. Holonic control implementations have predominantly made use of the Java Agent Development (JADE) framework – this paper presents, as an alternative, a case study implementation based on Erlang. Erlang is a functional programming language with strong scalability, concurrency and fault-tolerance characteristics, which prove to be beneficial when applied to the manufacturing control context. The case study used in this paper is the holonic control of a modular conveyor system – this implementation was chosen to demonstrate the advantages that Erlang can offer as implementation language for holonic systems.

Keywords: Erlang/OTP; Holonic Manufacturing Systems; Reconfigurable Manufacturing Systems; Manufacturing Execution Systems; Automation

4.3.1. Introduction

The modern manufacturing environment is governed by a new set of requirements, driven by unpredictability in market and technology trends. Modern manufacturing systems must adhere to short lead times and enhanced adaptability for the individualization of products and services, all while remaining competitive in a global market. To address these challenges, recent movements towards the fourth industrial revolution (often referred to as Industry 4.0) aim to enhance the connectedness of the real and virtual worlds.

In [1], Monostori *et al.* consider Industry 4.0 to be characterized by the individualization of products and services, new organization and control of the entire value chain and the formulation of new business models. These characteristics can be facilitated through the connection of humans, objects and systems, and the generation and use of information in real-time.

CPS are systems of communicating computational entities, which are connected to the physical world, that simultaneously use and provide data and services using the Internet. These entities can monitor, control, coordinate and integrate the operations of physical or engineered systems. Cyber-Physical Systems (CPS) will play a key role in the connection of people, components/systems, information and services. Cyber-Physical Production Systems (CPPS) can facilitate these enhancements in manufacturing environments.

CPPS build on and utilize several developments in the field of manufacturing science and technology, including that of Holonic Manufacturing Systems (HMSs).

Holonic systems – which is based on the theories of Koestler [2] – have often been used in the design and control of manufacturing systems to reduce complexity and cost, and improve scalability, maintainability and robustness [3][4]. Holonic systems are based on the idea of dividing a complex system into smaller functional entities – autonomous components that, through cooperation, constitute the system functionality [4].

Holonic control architectures have predominantly been implemented in manufacturing systems using multi-agent systems. Of the platforms that are available for developing multi-agent systems, the Java Agent Development (JADE) framework (see [5]) has been the most popular. This paper proposes an alternative implementation, using Erlang/OTP.

Erlang is a functional programming language that was developed for programming large-scale, distributed control applications [6]. Erlang was developed specifically for the control of telecommunications switching systems [7], but the inherent characteristics of Erlang – namely concurrency, scalability and fault-tolerance – could prove greatly beneficial for the implementation of holonic control in modern manufacturing systems. The Erlang programming environment is supplemented by the Open Telecommunications Platform (OTP) [8][9] - a set of robust Erlang libraries and design principles providing middle-ware to develop Erlang systems.

This paper presents an implementation of Erlang-based holonic control for a modular conveyor system. The case study was selected for two reasons:

- The control of conveyor systems involve some of the key challenges that the holonic systems approach aims to address. To reduce the complexity of the system, the control implementation must exhibit good modularity characteristics and perform numerous concurrent, distributed actions – requirements that will clearly illustrate the advantages that Erlang has to offer.
- The frequent use of conveyor systems in manufacturing systems has inspired several research studies on the implementation of control. This allows for qualitative and quantitative comparisons to be performed in future work, which may be helpful in the formulation of benchmarks for the performance of holonic control implementations.

This paper starts by providing details of the case study (section 4.3.2) and a short overview of the important aspects of Erlang/OTP (section 4.3.3). Thereafter, section 4.3.4 presents a discussion of the holonic control architecture and the Erlang-based holonic control implementation is described in section 4.3.5. The paper concludes with a discussion of the presented research and future work in section 4.3.6.

4.3.2. Modular Conveyor Case Study

Modular palletized conveyor systems, as the system shown in Figure 20, are frequently used for material handling in manufacturing systems. These conveyor systems typically use motor-driven belts, along with stop gates and lifting/transverse mechanisms, to move pallets (from here on referred to as *carriers*) between the workstations of manufacturing systems. These conveyor systems are often also equipped with RFID read/write modules (installed on several locations on the conveyor), while the carriers are fitted with RFID tags. The RFID readers provide feedback when a specific carrier arrives at a RFID reader location. The RFID readers can be installed at the various stop gate locations.

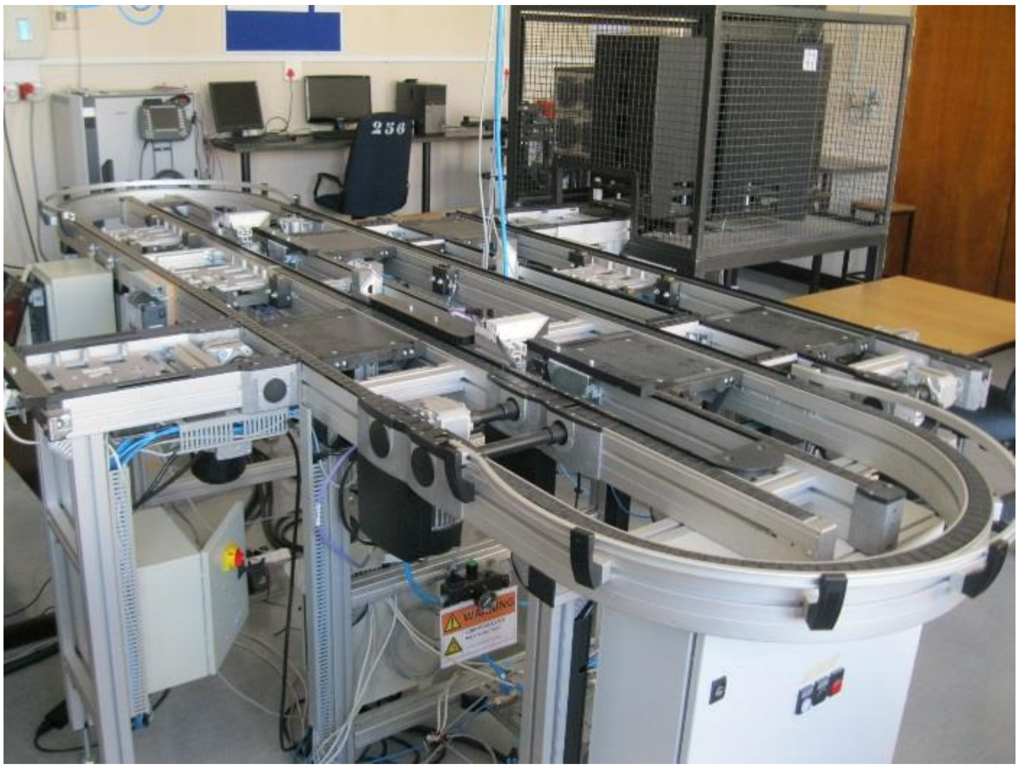


Figure 20: Conveyor system at the Automation Lab of Stellenbosch University.

This paper uses a modular conveyor system as case study. The conveyor system that is installed in the Automation laboratory at Stellenbosch University (shown in Figure 20) is too small to sufficiently illustrate the complexity of the control that is encountered in industrial systems. Therefore, this research considered the control implementation for an up-scaled, simulated conveyor system. A discussion of the development of the simulation model can be found in [10].

The simulated model is based on the conveyor system that is used in an assembly and quality assurance cell for electrical circuit breakers – the system layout is illustrated in Figure 21. The conveyor moves carriers, which carry circuit breakers, between the workstations of the system. The circuit breakers are placed onto the carriers at the manual assembly station, from where they are moved to each of the workstations (sequentially and in a clockwise direction) and finally removed at the removal station.

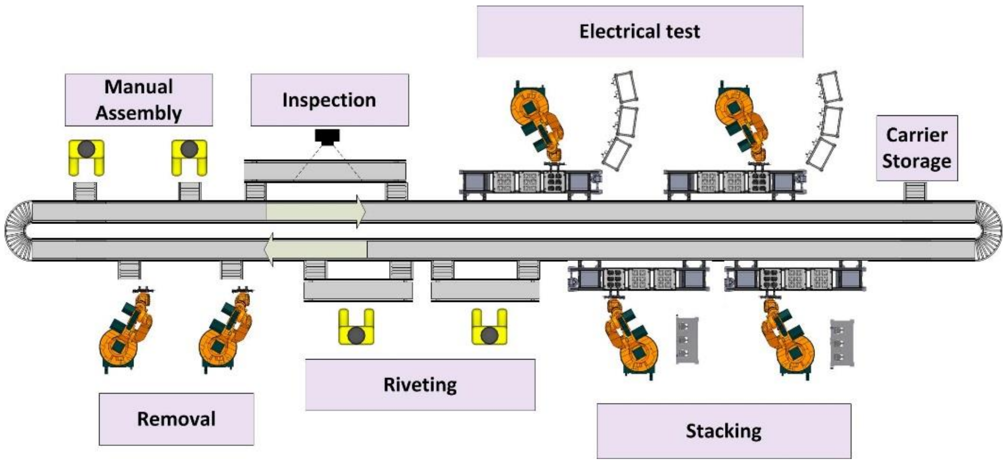


Figure 21: Layout of the simulated conveyor system.

As an extension to the modularity of the conveyor hardware, several small PLCs are used as opposed to using one centralized PLC – this is also incorporated in the simulated conveyor system model. Segments of the conveyor are allocated to a dedicated PLC, with all the interfacing between segments being handled at the higher level of control implementation – this modular architecture is presented in [11].

4.3.3. Erlang/OTP

While it would be over-ambitious to attempt a complete overview of Erlang/OTP, this section aims to explain the architectural provisions and language mechanisms that enable the suitability of Erlang/OTP for the implementation of holonic control architectures.

4.3.3.1. Erlang Process Model

Erlang owes its concurrency to the process model on which it is built. These processes, as the basic unit of abstraction, are extremely lightweight with memory requirements that can vary dynamically. Not only can many processes work concurrently, but they can be distributed across many devices (referred to as *nodes*). Process are strongly isolated, having no shared memory, and can only interact through the asynchronous sending and receiving of messages [7], as is discussed in the next section.

4.3.3.2. Process Communication

Since Erlang processes do not share any memory, all the data exchange occurs through message passing. Each process maintains its own mailbox to receive and handle messages, and Erlang provides a message operator "`!`" to simplify the sending of messages. A message can be sent from one process to another with the following code:

```
Receiver ! {Sender, Message}
```

The `Receiver` variable is used to specify the process to receive the message – the registered name or unique process identifier can be used. It is good practice for the message to be constructed as a tuple, containing both sender and message content information. The inclusion of the details of the sender process enables the receiver process to reply to the message if needed. `Message`, the variable containing the message content, can be of different data types (e.g. constant, tuple, list, etc.). The *record* data type (which is similar to *structs* in C) is used in the presented implementation to structure the content of messages. With records, sets of information can be defined and used in creating messages and matching messages to patterns. Records allow for data fields to be accessed by name instead of order, and multiple records can be nested to accommodate complex sets of information. An example of a record is shown below:

```
#service{message_type,   conversation_ID,   requester_pid,
provider_pid, result, info}
```

4.3.3.3. OTP Behaviours

OTP supplements Erlang development through the provision of robust libraries for commonly used functionality (referred to as *behaviours*). Behaviours are provided for the implementation of logic (as with the generic server and finite state machine behaviours) and for facilitating communication (behaviours are provided for serial, UDP and TCP communication). The generic finite state machine and TCP communication behaviours are used in the presented implementation and therefore a brief overview of the behaviours is given.

A Finite State Machine (FSM) can be described as a relation of *states*, *events* and *actions*. When a FSM is in a state and an event occurs, some action(s) will be performed and the FSM will transition to the next state. Using the Erlang `gen_fsm` behaviour, these state transitions can be implemented by:

```
StateName(Event, StateData) ->

      %code for actions to be performed

      {next_state, NextStateName, NewStateData}.
```

The name of the state the FSM is in when `Event` occurs is programmed as the function header `StateName`. `StateData` represents internal information regarding the current state. When `Event` occurs (which in this implementation is usually the arrival of a specific message in the process mailbox), specific actions that must be performed can be programmed. After all the required actions are completed, the statement ends with a description of the state transition that follows. The transition description is represented as a tuple with three elements: the first element is the atom `next_state`, designating the transition description; the second element specifies the name of the state to which the FSM will transition to and the last element specifies the internal information associated with the next state.

The `gen_tcp` behaviour included in OTP provides functions to communicate through network sockets using the Transmission Control protocol (TCP). Functions are included for both server and client implementations.

An Erlang process can act as a server for a designated TCP port, using:

```
{ok, Socket} = gen_tcp:listen(Port, Options)
```

The Port and Options variables specify the socket and configuration details, and the Socket variable stores the instance of the created TCP network socket. As the function name suggests, the server process will listen for incoming connection requests at the specified port. When such a request is received, the connection can be accepted with the function `gen_tcp:accept(Socket)`.

Also, a process can connect to a TCP socket as a client – this functionality is provided through the function:

```
gen_tcp:connect(Address, Port, Options)
```

The function requires the IP address or host name of the device where the socket resides, as well as the port and configuration details as input parameters.When the connection is accepted by the corresponding server process, TCP communication over the connected socket can be achieved. Both the server and client processes use the same functions for the sending and receiving of messages over the socket:

```
gen_tcp:send(Socket, DataPacket)

gen_tcp:recv(Socket, Length)
```

`DataPacket` contains the information to be sent, and `Length` specifies the number of bytes that must be read from the socket.

4.3.4. Holonic Control Architecture

The Conveyor holon presented in this paper forms part of a holonic cell control implementation. The cell control architecture is based on PROSA [12] – a simplified schematic representation of the architecture is presented in Figure 22. Detailed discussions of similar implementations are given in [13] and [14].

The architecture of the cell control implementation consists of three levels: High Level Control (HLC), Low Level (station) Control (LLC) and hardware control. The communication and coordination of the system holons occur within the HLC. The HLC purely exists in the virtual environment, as the Product, Order and Staff holons are all software entities. Resource holons, which consist of both hardware and software entities, must also be represented in the HLC – it is therefore necessary that these resource holons incorporate a component to handle the HLC functions.

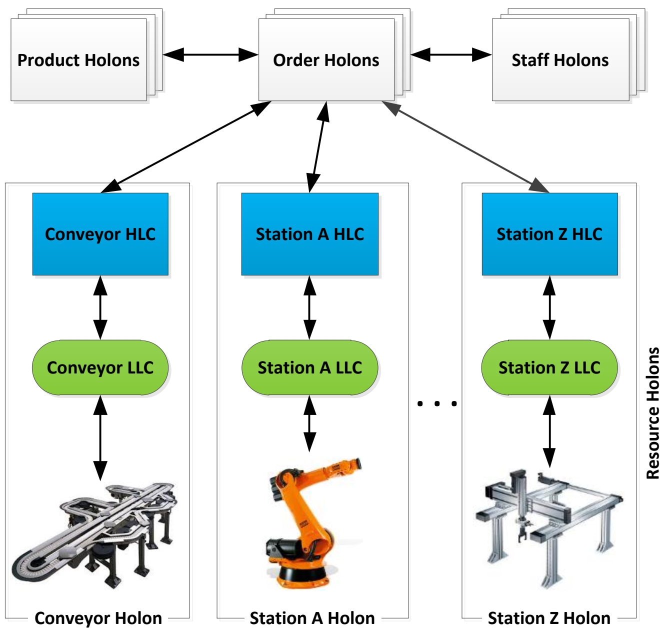


Figure 22: Schematic of the holonic control architecture for the manufacturing cell.

Where Resource holons consist of physical hardware entities, the station and hardware levels of control are encountered. Station LLC enables the coordination of the hardware functions for a station, to perform the service that the Resource holon advertises in the HLC. Hardware control refers to the control of actuators and sensors to successfully perform the various tasks included in the Resource holon's service.

4.3.5. Erlang-based Conveyor Holon

4.3.5.1. Conveyor Holon Architecture

As mentioned in section 4.3.4, the Conveyor holon forms part of the implementation of the holonic control for a cell. The Conveyor holon is itself implemented using a holonic architecture, i.e. the functions of the holon are mapped to several autonomous and cooperating entities which work together to perform complex transportation tasks. This holonic implementation then constitutes the HLC component of the Conveyor holon (as can be seen from Figure 22) – the LLC implementation is distributed over the number of PLCs that control dedicated segments of the conveyor hardware.

The holons which comprise the Conveyor holon are shown in Figure 23 and the respective roles and functions are discussed the following sections. The Conveyor holon entails three main functions: inter-holon communication within the HLC and intra-holon coordination within the Conveyor holon, execution of transportation tasks and the interaction with and virtual representation of the conveyor hardware.

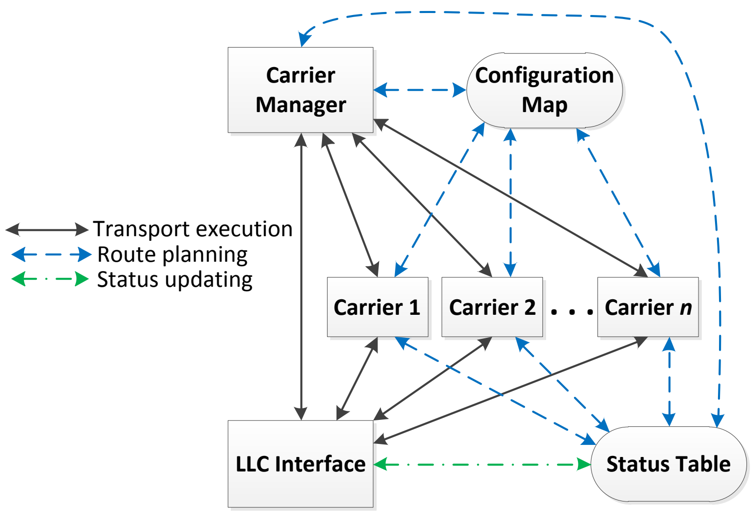


Figure 23: Intra-holon communication within the Conveyor holon.

4.3.5.2. Communication

4.3.5.2.1. Inter-holon Communication

The Carrier Manager holon is responsible for handling all communication with the other holons in the cell controller. The Conveyor holon interacts with three types of holons in the cell controller: the staff holon handling the service directory (a list of service-providing Resource holons), other Resource holons which have a physical interaction with the Conveyor holon, and Order holons.

As is usually encountered in the implementation of holonic systems, this presented implementation tries to mirror the physical system as far as possible. An example of this is when Resource holons must physically remove products from the conveyor or place products on it. This interaction is mirrored in the virtual environment - before removing or placing a product on the conveyor, Resource holons must first send a *release request* or *binding request* message to the Conveyor holon. This allows the Conveyor holon to ensure that a suitable carrier is present at the location of placing, or that the intended product is available to be removed by the resource holon. After the Conveyor holon replies to the request, the Resource holon can continue with the physical operation.

The approach of mirroring the physical interactions in the virtual system means that when an Order holon requires a transport service to the next booked service-providing station, the physical product instance for which it is responsible will already be present on a carrier on the conveyor. The Order holon can then proceed to send the Conveyor holon a *service start* message to perform the transportation service.

The Carrier Manager receives requests from Order holons to perform a transport service from some start position to a specified destination. The Carrier Manager then checks if a suitable Carrier holon is available at the requested starting position. If a Carrier is available, the Carrier manager sends a start message to the selected

Carrier – if no Carrier is currently available, the Carrier Manager will search for a compatible, idle Carrier holon and direct it to the designated starting location.

4.3.5.2.2. Intra-holon Communication

Three types of communication occur between the holons of the Conveyor holon: transportation service execution, route planning and status update communication – these interactions are shown in Figure 23.

Transportation service execution communication requires interaction between holons in order to coordinate and execute the transportation tasks that the Conveyor holon must perform for Order holons. As the Carrier Manager receives requests from Order holon, the requests are allocated to suitable Carrier holons. The Carrier Manager sends *service start* messages to the relevant Carrier holons - these messages specify the end destination to where the Carrier holons must navigate. To execute the movement between conveyor nodes along the selected route, the Carrier holons request actuation from the specific PLCs by sending *request* messages to the LLC Interface holon. The LLC Interface holon then in turn replies with a confirmation that the requested actuation has been performed by the conveyor hardware. When a Carrier holon has completed its assigned transport task, it sends this confirmation to the Carrier Manager and awaits a new task.

Route planning communication entails the gathering of information by holons to aid the route finding process. Predominantly, this communication is performed by Carrier holons – when Carrier holons are assigned a transportation task, they are responsible for planning their own route. The Carrier holons request information of the physical conveyor configuration from the Configuration Map holon and status information of the conveyor nodes and transitions from the Status Table holon – this process is discussed in more detail in section 4.3.5.6. The Carrier Manager holon will also occasionally initiate route planning communication – this occurs when the Carrier Manager must control Carrier holon movement for coordination purposes.

Finally, the status updating communication involves the LLC Interface holon passing status information, received from the PLCs, to the Status Table holon.

4.3.5.3. Virtual Conveyor Representation

The physical configuration and run-time status of the conveyor nodes and transitions are represented in the virtual environment by two holons: the Configuration Map holon and the Status Table Holon.

The Configuration Map holon contains the functions to read the configuration information, from an operator-defined description, into an accessible data structure (in this case, an Erlang Term Storage (ETS) table). The ETS table entries follow the format:

```
{Node_name, LLC_port_number, [Transition1, Transition2,…]}
```

The name of the node, the port number for communication to the controlling PLC and the transitions that are available from the node are specified. The transitions are described by the following information:

```
{Connected_node, Transition_time, Transition_capacity}
```

- `Connected_node` – name of the node which constitutes the end point of the transition.
- `Transition_time` – the time it takes for a carrier to travel the transition (based on the speed of the conveyor).
- `Transition_capacity` – the number of carriers that can travel along the transition at any given time.

The Status Table holon maintains an ETS table of the conveyor node and transition status based on messages received from the LLC Interface holon – i.e. the status information is dynamically updated as carriers move along the conveyor. The format of the ETS table entries is as follows:

```
{{Node,Connected_node},{Status, Queue_list, Capacity}}
```

- `{Node,Connected_node}` – the two nodes that constitute the start and end nodes of the transition.
- `Status` – indicates whether the transition can take another carrier or not, based on its capacity and current queue.
- `Queue_list` – a list of all the carriers currently travelling along the transition.
- `Capacity` – the number of carriers that can travel along the transition at any given time.

The Configuration Map and Status Table holons handle all request messages from other holons, searches for and replies with the desired configuration and status information.

4.3.5.4. Carrier Manager

The Carrier Manager holon maintains the interface for inter-holon communication with the other PROSA holons (as discussed in section 4.3.5.2). The Carrier Manager also handles intra-holon communication – i.e. messages from Carrier holons or the LLC Interface holon. The Carrier Manager thus functions as a server – messages are received and, according to message type and content, the appropriate functions are executed. Examples of such functions are `handleStartRequest()` or `handleCarrierDone()`.

An important function of the Carrier Manager is to allocate transportation tasks received from Order holons to the most suitable Carrier holon. A *start* message is then sent to the selected Carrier holon, upon which the transport service will be performed. Once the service is completed, the Carrier holon notifies the Carrier Manager, which in turn notifies the relevant Order holon.

Usually, the Carrier holons perform movements according to the Order holon request allocated to them by the Carrier Manager. However, the Carrier Manager also has the functionality to make decisions regarding the movement of carriers directly. This functionality is needed to ensure flow on the conveyor (i.e. not having carriers block certain segments) and to store carriers when they are no longer required.

4.3.5.5. Conveyor Low Level Control Interface

The LLC Interface holon is responsible for maintaining the interface between the Erlang control programs and the low level control PLCs – this is depicted in Figure 24.

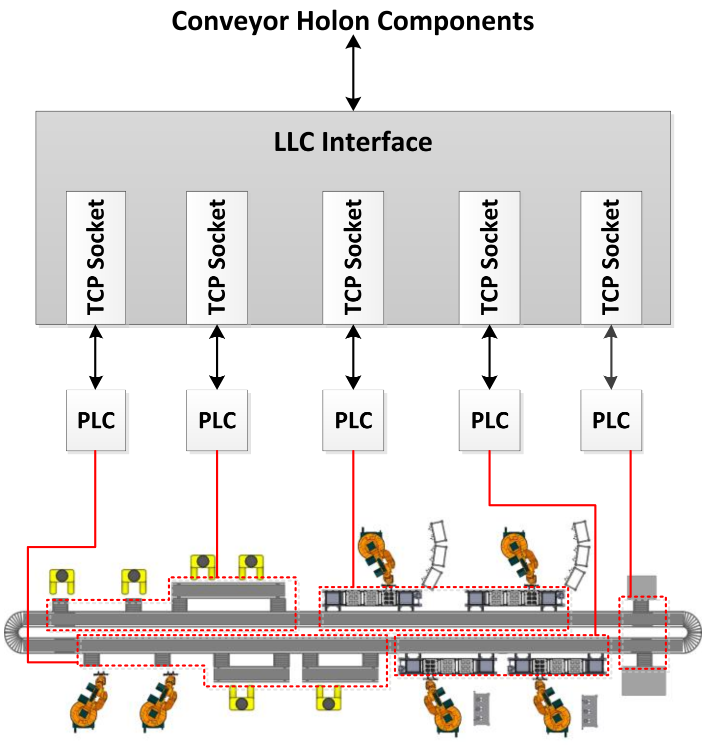


Figure 24: LLC interface of the Conveyor holon.

The communication to the PLCs is done over Ethernet, with messages encoded as XML strings. The PLCs can parse the XML strings to extract the necessary information pertaining to the actuation that must be performed. The LLC Interface holon employs a concurrent Erlang process for every TCP socket connection that must be maintained – i.e. a connection to each of the PLCs is maintained by a dedicated process.

The LLC Interface holon receives messages from both Carrier holons and the Carrier Manager holon. As the Carrier holons execute their delegated transport services, they must send request messages to the relevant PLCs via the LLC Interface. This occurs every time a Carrier holon arrives at a node – the message will request the actuation at the given node to direct the carrier towards the next desired node (according to the planned route). The LLC Interface interprets this message to determine which PLC the message is intended for (according to the segment of the conveyor where the node is located). The message is then compiled into an XML string and is send over the correct TCP socket to the PLC. Messages from the Carrier Manager holon are handled in the same way.

To maintain a representation of the conveyor status during operation, the LLC Interface holon sends messages to the Status Table holon when it receives notifications from or sends actuation commands to the PLCs.

4.3.5.6. Carrier Holon

Each carrier that is unloaded onto the conveyor is represented in the holonic system by a Carrier holon. Every time a carrier is unloaded, the Carrier Manager spawns a new instance of the Carrier holon Erlang process. The Carrier holon encapsulates the functionality to perform transportation services by controlling the movement of the physical carrier on the conveyor system. Although the physical carrier has no actuators or sensors, control of the movement is performed through communication between the Carrier holons and the controlling PLCs, via the LLC Interface holon.

4.3.5.6.1. Behaviour

The control logic of the Carrier holon is implemented using the standard OTP finite state machine behaviour. The Carrier holon transitions between states based on the occurrence of events (in this case, the arrival of messages).

The Carrier holon behaviour is described by two states: *stopped* and *moving*. The *stopped* state is entered when the holon awaits its next transportation task and when it reaches a node while travelling towards its destination. The behaviour enters the *moving* state once the LLC Interface holon confirms that the carrier has been physically routed towards the next node on the route. Once the LLC Interface holon notifies the Carrier holon of arrival at the next node, the state transitions to *stopped*.

4.3.5.6.2. Communication

As is shown in Figure 23, the Carrier holon engages in communication with other holons during transport service execution and route planning. In the transport execution activity, Carrier holons receive messages from the Carrier Manager holon to initiate a new transport service that must be performed by the carrier. The Carrier holons then send a notification message back to the Carrier Manager when the service is done and awaits the next service to be awarded. When the Carrier holons travel along their route, they send requests to the LLC Interface - which interprets the messages and forwards it to the correct PLC – to perform the necessary actuations to direct the carrier along its desired route. The Carrier holons also receive notification messages from the LLC Interface when the carriers arrive at conveyor nodes.

For route planning, Carrier holons must exchange messages with the Configuration Map and Status Table holons. When a Carrier holon is awarded a transportation task, it first determines which route to follow from its current location to its desired location. The Carrier holon can obtain the conveyor configuration and status information, which allows for the implementation of route finding algorithms and strategies.

4.3.6. Conclusion and Future Work

The paper presents an Erlang-based holonic control implementation for a modular conveyor system. The controller is successfully implemented for a simulated, medium-sized manufacturing cell (incorporating ten different workstations). The conveyor is incorporated as a Resource holon in the PROSA holonic architecture, upon which the control of the manufacturing cell is based.

The Conveyor holon is responsible for the movement of carriers (which transport products or work pieces around the cell) by controlling the actions of the conveyor hardware via low level control PLCs. The holon performs several functions – communication with other cell level holons, route planning and route execution through hardware coordination. The Conveyor holon is itself implemented as a holarchy, with the involved functions performed through the cooperation of the collection of holons.

The described implementation aims to exploit the advantages that are offered by Erlang, namely modularity, scalability and concurrency. From the presented research, the following remarks can be offered:

- The inherent modularity and concurrency of Erlang programming provides a natural facilitation for the implementation of holonic principles.
- The holonic controller exhibits good scalability and reconfigurability with very little effort.
- The compact, readable code, along with the modularity of Erlang programs, allow for a reduction in programming complexity.
- The standard libraries offered by OTP contribute greatly to the simplicity and robustness of the control implementation, with potential for further improvement.
- In future work, the research will focus on establishing benchmarks for a formal evaluation of this implementation and an equivalent multi-agent system for comparison.

4.3.7. References

1. L. Monostori, B. Kadar, T. Bauernhansl, S. Kondoh, S. Kumara, G. Reinhart, O. Sauer, G. Schuh, W. Sihn and K. Ueda. "Cyber-Physical Systems in Manufacturing", *CIRP Annals – Manufacturing Technology*, Vol. 65: 621-641, 2016.
2. Koestler, *The Ghost in the Machine,* London: Arkana Books, 1967.
3. Scholz-Reiter and M. Freitag. "Autonomous Processes in Assembly Systems", *Annals of the CIRP*, Vol. 56: 712-730, 2007.
4. Kotak, S. Wu, M. Fleetwood and H. Tamoto, "Agent-Based Holonic Design and Operations Environment for Distributed Manufacturing", Computers in Industry, Vol. 52: 95–108, 2003.
5. F. Bellifemine, G. Caire and D. Greenwood, *Developing Multi-Agent Systems with JADE*, John Wiley & Sons, Ltd., 2007.

6. J. Armstrong, “Erlang”, *Communciations of the ACM*, Vol. 53, No. 9:68-75, 2010.
7. J. Armstrong, *Making Reliable Distributed Systems in the Presence of Software Errors*, Doctor’s Dissertation, Royal Institute of Technology, Stockholm, Sweden, 2003.
8. M. Logan, E. Merrit and R. Carlsson, *Erlang and OTP in Action*, Stamford: Manning Publications Co., 2011.
9. *Get Started with OTP*. [S.a.]. [Online]. Available: http://www.erlang.org (18 July 2013)
10. K. Kruger and A.H. Basson. “Validation of a Holonic Controller for a Modular Conveyor System using an Object-Oriented Simulation Framework”, *Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing 2016*, Lisbon, Portugal (October 2016).
11. M.J. Kotze and A.H. Basson, “Control of a Modular Conveyor System Using Object Oriented Programming”, submitted for review.
12. H. Van Brussel, J. Wyns, P. Valckenaers, L. Bongaerts and P. Peeters, “Reference Architecture for Holonic Manufacturing Systems: PROSA”, *Computers in Industry*, Vol. 37: 255 – 274, 1998.
13. K. Kruger and A.H. Basson, “Implementation of an Erlang-based Resource Holon for a Holonic Manufacturing Cell”, *Service Orientation in Holonic and Multi-Agent Manufacturing*, Studies in Computational Intelligence, Springer International Publishing, 2015.
14. K. Kruger and A.H. Basson, “Erlang-based Control Implementation for a Holonic Manufacturing Cell”, *International Journal of Computer Integrated Manufacturing*, Vol. 30, No. 6:641-652, Taylor & Francis, 2017.

5. Multi-Agent System Holonic Control Implementation

To evaluate the Erlang holonic control implementation, a comparison will be performed with the *status quo* in holonic control implementation – multi-agent systems. This section provides an overview of the MAS holonic control implementation for the testbed system described in section 3.2.

The MAS implementation is presented in the form of a paper, titled "JADE Multi-Agent System Holonic Control Implementation for a Manufacturing Cell". The paper presents the implementation of the PROSA holonic reference architecture using the JADE platform for MAS development. The use of JADE behaviours to implement agent functionality and facilitate agent communication is described.

JADE Multi-Agent System Holonic Control Implementation for a Manufacturing Cell

Karel Kruger [a,*] and Anton Basson [a]
[a] Dept of Mechanical and Mechatronic Eng, Stellenbosch Univ, Stellenbosch 7600, South Africa
[*] Corresponding author. Tel: +27 21 808 4258; Email: kkruger@sun.ac.za

Abstract

Multi-Agents Systems (MASs) is a popular approach for the implementation of holonic control architectures in manufacturing systems. Software agents and holons share several similarities, allowing for the exploitation of the advantages that are offered by holonic systems. The Java Agent Development (JADE) framework is the tool most often used in implementations of holonic control. This paper describes a JADE MAS implementation of the Product-Resource-Order-Staff Architecture (PROSA) for holonic control of a manufacturing cell. The mapping of the holonic and MAS architectures is explained and the communication and functionality of the individual agents in the MAS is detailed.

Keywords: Multi-Agent System (MAS); Java Agent Development framework (JADE); Holonic manufacturing system (HMS); Reconfigurable manufacturing system (RMS)

5.1. Introduction

Modern manufacturing systems require short lead times for the introduction of new products into the system, the ability to produce a larger number of product variants and the ability to handle fluctuating production volumes (Bi *et al.*, 2008). The concept of Reconfigurable Manufacturing Systems (RMSs) is aimed at addressing these requirements.

RMSs aim to switch between members of a family of products, through the addition or removal of functional elements (hardware or software), with minimal delay and effort (Martinsen *et al.*, 2007; Vyatkin, 2007). RMSs can rapidly adjust the production capacity and functionality in response to sudden changes, by reconfiguring hardware and control resources (Bi *et al.*, 2008; Bi, Wang, and Lang, 2007). RMSs are characterised by (Mehrabi, Ulsoy, and Koren, 2000; ElMaraghy, 2006): modularity of system components, integrability with other technologies, convertibility to other products, diagnosability of system errors, customizability for specific applications and scalability of system capacity.

Holonic control architectures is a popular approach for enabling control reconfiguration in RMSs. The term holon (first introduced by Koestler in 1967) comes from the Greek words “holos” (meaning “the whole”) and “on” (meaning “the particle”). Holons are “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 systems, holons are autonomous and cooperative building blocks for transforming, transporting, storing or validating the information of physical objects. A Holonic Manufacturing System (HMS) is a system of holons that can cooperate to integrate the entire range of manufacturing activities (Paolucci and Sacile, 2005).

The use of holonic control for RMSs holds many advantages: holonic systems are resilient to disturbances and adaptable in response to faults (Vyatkin, 2007); have the ability to organise production activities in a way that they meet the requirements of scalability, robustness and fault tolerance (Kotak *et al.*, 2003); and lead to reduced system complexity, reduced software development costs and improved maintainability and reliability (Scholz-Reiter and Freitag, 2007).

The application of the holonic concept to manufacturing control systems has been a popular field of research since the early 1990's. The most popular approach to implementing holonic control architectures has been Multi-Agent Systems (MASs). The main motivation for this approach is the similarities between holons and software agents – both must exhibit autonomy and provide interfaces to facilitate cooperation. Several experimental implementations have been reported, such as Leitao and Restivo (2006) and Giret and Botti (2009).

Several tools exist for the development of MASs – of these tools, the Java Agent Development (JADE) framework is most commonly used in the control of manufacturing systems. JADE was developed by Telecom Italia and has been distributed under an open source license since 2000. The JADE framework provides the middleware to facilitate distributed applications that exploit the software agent abstraction (Bellifemine *et al.*, 2007). JADE provides tools that simplify the development, testing and operation of MASs, such as the Agent Management System (AMS) and the Directory Facilitator (DF). The AMS includes all the functionality to manage the agents in the MAS, from the creation of agents, to the migration and termination of agents. The DF provides a mechanism for the registration and discovery of resources by agents in the MAS. JADE also provides special Java classes, called *behaviours*, for implementing common functionality of agents – this includes behaviours for communication protocols that comply with the Foundation for Intelligent, Physical Agents (FIPA) specifications for agent communication.

This paper presents a JADE MAS implementation of a holonic reference architecture for a manufacturing cell. The implemented PROSA holonic architecture is discussed in section 5.2 and the case study, on which the implementation is based, is presented in section 5.3. The MAS holonic control implementation is described in section 5.4 and the paper concludes with a discussion of related and future work.

5.2. Holonic Reference Architecture

The exploitation of the advantages of holonic control, as mentioned in section 5.1, relies on the holonic system's architecture. Several reference architectures, which specify the mapping of manufacturing resources and information to holons and to structure the holarchy, have been proposed (e.g. Chirn and McFarlane (2000) and

Leitao and Restivo (2006)), but the most prominent is the Product-Resource-Order-Staff Architecture (PROSA), as developed by Van Brussel *et al.* (1998).

PROSA defines four holon classes: Product, Resource, Order and Staff. The first three classes of holons can be classified as basic holons, because, respectively, they represent three independent manufacturing concerns: product-related technological aspects (Product holons), resource aspects (Resource holons) and logistical aspects (Order holons).

The basic holons can interact with each other by means of knowledge exchange, as is shown in Figure 25. 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.

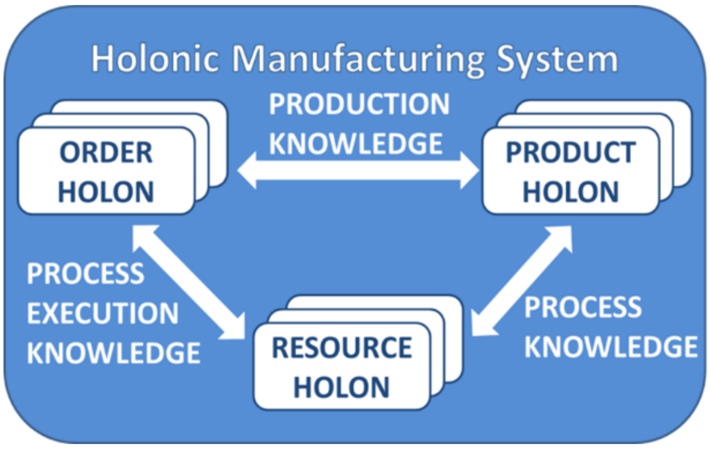


Figure 25: Knowledge exchange between the PROSA holons.

Staff holons are considered to be special holons as they 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 holonic characteristics of PROSA contribute to the different aspects of reconfigurability mentioned in section 5.1. The ability to decouple the control algorithm from the system structure, and the logistical aspects from the technical aspects, aids integrability and modularity. Modularity is also provided by the similarity that is shared by holons of the same type.

5.3. Case Study

The case study used for the presented implementation is a manufacturing cell for the assembly and quality assurance of electrical circuit breakers. The layout of the cell is shown in Figure 26. The cell consists of the following workstations:

- Manual assembly station – the sub-components of circuit breakers are assembled and placed on empty carriers on the conveyor.
- Inspection station – a machine vision inspection is performed on the circuit breakers as the carriers are moved by the conveyor.
- Electrical test station – circuit breakers are picked up by a robot and placed into testing machines. The testing machines perform the necessary performance and safety tests on every breaker. When the testing is completed for a breaker, it is removed from the testing machine by the robot and placed on an empty carrier on the conveyor.
- Stacking station – multiple circuit breakers are stacked to produce multi-pole circuit breakers. The breakers are removed, stacked and placed on empty carriers by a robot.
- Riveting station – the casings of the circuit breakers are manually riveted shut.
- Removal station – the completed circuit breakers are removed from carriers. The breakers are then moved to the next cell for packaging.

The conveyor moves product carriers between the various workstations. The conveyor is equipped with stop gates and lifting stations at every workstation. The carriers are fitted with Radio Frequency Identification (RFID) tags and RFID readers are placed at multiple positions along the conveyor, to provide feedback of carrier location.

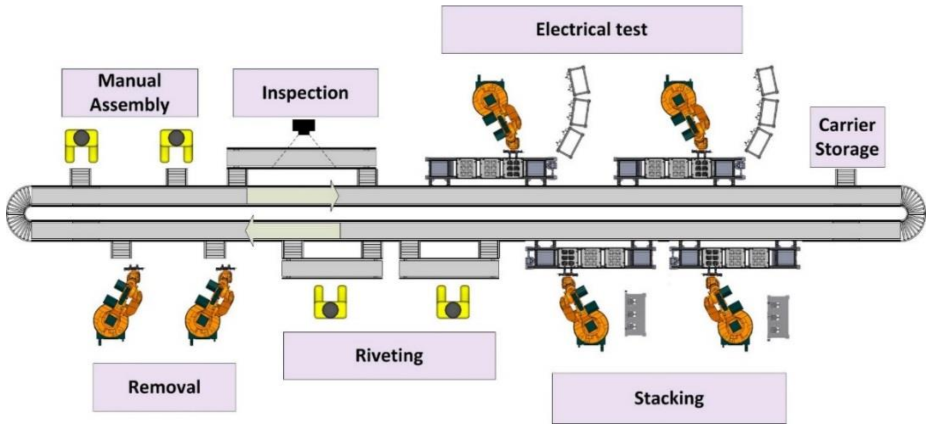


Figure 26: Layout of the electrical circuit breaker assembly and quality assurance cell.

5.4. Holonic Control Implementation

This section presents the JADE MAS implementation of holonic control, based on PROSA, for a manufacturing cell. The embodiment of the holonic architecture through a MAS is explained and the communication between system agents is discussed. Finally, the functionality and implementation of the individual agent types are described.

5.4.1. Holonic Architecture

In accordance with PROSA, the various functional components of the manufacturing cell are embodied as Product, Resource, Order or Staff holons. All of the holons are represented in the high level control implementation by software agents. The cooperation of the agents within the MAS implementation provide all the necessary functionality to drive the production of the manufacturing cell.

The information that pertains to the production of every product that is to be manufactured by the cell is contained within Product holons. Since these holons exist purely as information within the control implementation, the holons are wholly represented as Product agents within the MAS.

The Order holons should exhibit the functionality to utilize the product information to produce a product of a specific type. Order holons encapsulate the logic and information needed for production, and thus only exist within the high level control implementation, where Order holons are represented as Order agents.

In the presented architecture, it is only the Resource holons that include both physical and software functional components. A Resource holon contains the resource hardware (as present on the factory floor), the low level control component (that control the actuators of the hardware and receives feedback from sensors) and the high level control component. The high level control components is implemented as a Resource agent in the MAS control implementation. Resource agents must provide the functionality to communicate with the other agents in the MAS, manage the agenda of the resource (i.e. the schedule of the execution of the resource’s services), control the service execution tasks and sequences and maintain a communication interface with the low level control components of the Resource holon. The internal architecture of the Resource agent is presented in Figure 27.

The implementation includes one special Resource agent – the Transport agent. The Transport agent is responsible for the high level control of the conveyor system, which moves the product carriers between the different workstations. The implementation makes use of conveyor controller that was previously developed using Erlang (see Kruger and Basson (2016) for details). The Transport agent included in this implementation acts as a wrapper, i.e. to provide an agent interface to the Erlang controller. This interface allows the agents in the MAS to communicate with the Erlang controller as if it was just another Resource agent.

The Staff holons for the manufacturing cell are implemented as different agents in the MAS. Some of the Staff holons functionality are provided by JADE, such as the AMS and DF. Two other Staff agents are included: the Order Manager agent (to manage the creation and monitoring of Order agents within the MAS) and the Performance Logger agent (to record the performance of Resource and Order agents for diagnostic purposes).

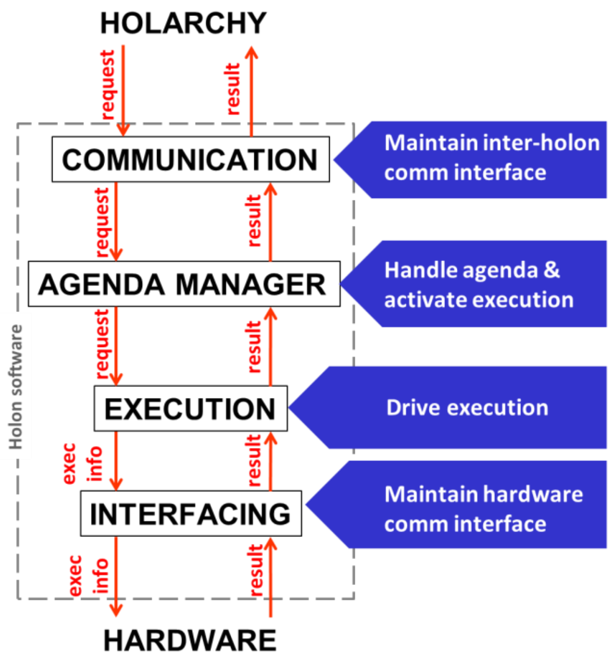


Figure 27: Internal architecture for the Resource agent.

5.4.2. Agent Communication

The cooperation of agents within the MAS is achieved through communication – information is passed as messages between agents. The implementation aimed to make use of the communication protocols and accompanying functionality provided by JADE – specifically, the FIPA Rational-Effect protocol and the contract net protocol. To supplement the communication a messaging ontology is defined and is applied to the construction of the content information that is added to the various message instances. This ontology and the formation of customized communication protocols using the JADE protocols are described in the following sections.

5.4.2.1. Messaging ontology

The implementation makes use of eXtensible Markup Language (XML) ontology for structuring the information exchanged during communication. The XML ontology specifies the information that must accompany a specific message type, as is determined by the elements that comprise the XML document.

The templates of the XML documents for the various message types are included in every agent. When a message is composed, the template is used and the required information is added to the elements. The constructed XML document is then converted to a *string* data type, so that the data can be added to the content slot of a normal JADE ACL message. On the receiver side, the template is used to determine the elements of the received message from where data must be extracted. The string obtained from the content slot is converted back to an XML document, from which point it can be parsed and the required information can be extracted. An example of the content of a *start* request message, as would be sent from an Order agent to a Resource agent, is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
     <initiator>OrderAgent_O24</initiator>
     <responder>ResourceAgent_R06</responder>
     <msg>
          <message_type>start</message_type>
          <service_type>test</service_type>
          <conversation_ID>C021</conversation_ID>
          <result>undefined</result>
          <info>
               <product_ID>P02</product_ID>
          </info>
     </msg>
</message>
```

5.4.2.2. Service booking, confirmation and execution

Order agents, as embodiments of Order holons, are responsible for driving production – each Order holon exhibits the functionality coordinate the resources necessary to produce their specific product. The Order agents then follow a protocol for the booking of resource services, according to the tasks specified by the product information. When the part is ready for the next service to be performed on it, the Order agent must first confirm the service booking and then start the execution of the service. This interaction is illustrated in Figure 28.

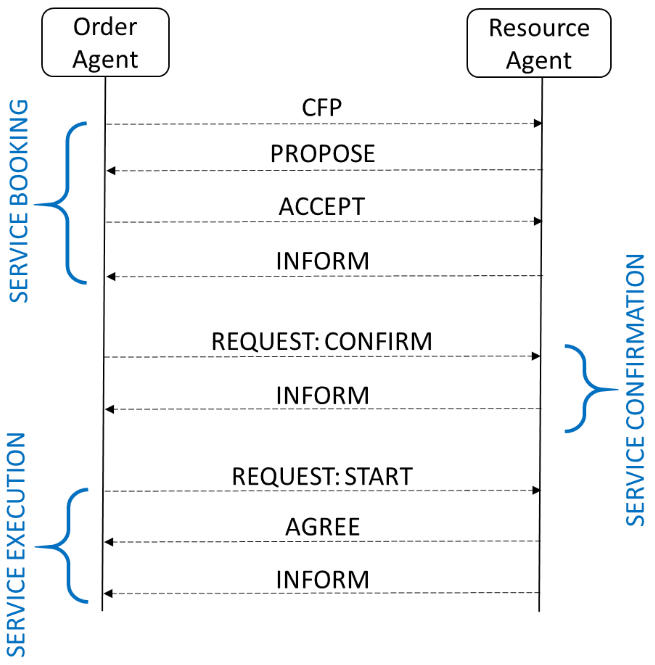


Figure 28: Communication between an Order agent and a Resource agent.

The booking of services is accomplished through a contract net protocol between the Order agent and the Resource agents. After the Order agent obtains the agent identifiers of all the Resource agents capable of performing a specific service, it initiates the communication protocol with each Resource agent. The protocol commences with the sending of *Call For Proposal* (CFP) messages to each Resource agent. The Resource agents reply to this CFP message with a proposal. The received proposals are compared and the Resource agent that sent the best proposal is sent an *Accept Proposal* message. Once the selected Resource agent replies with an *inform* message, the booking is completed.

When the product that is controlled by the Order agent is ready for the next booked service to be performed on it, the Order agent must first confirm that the service booking is still valid – this is done by using the simple FIPA Rational-Effect (RE) protocol. The Order agent initiates the protocol by sending a *request* message to the booked Resource agent. This *request* message contains a XML string in its content slot, which contains the "confirm" string in the element holding data for the message type. The Resource agent parses the XML string content of the request message and identifies it as a confirmation message. If the details of the Order agent are present in the bookings list of the resource agent, it replies with an *inform* message (if not, a *failure* message is sent – this is an indication of a fault in the execution of the Order agent). The confirmation step is included in the communication protocol as an additional check.

Upon receiving confirmation, the Order agent again initiates a simple RE protocol – in this case, the content slot of the request message is similar to the string version of the XML document presented in section 5.4.2.1. The Resource agent identifies the request as a *start* message and immediately replies to the Order agent with an Agree message and start the execution of the service. The *agree* message provides an indication to the Order agent that execution of the service has started on the product – this indication can be used to start a timer, which can indicate when an error has occurred in the Resource agent. Upon completion of the service, the Resource agent sends an *inform* message to the Order agent.

5.4.2.3. Interaction with the Transport Agent

Most of the services performed at the workstations involves physical interaction with the carriers of the conveyor – e.g. at the input of the Electrical Test Station (ETS) products are removed from carriers for testing and, upon completion, are placed back on empty carriers available at the output of the station. This physical interaction between resources and the conveyor at the workstations is replicated in the virtual interaction, i.e. in the communication between the various Resource agents and the Transport agent.

The MAS architecture dictates that the coordination of services is done by Order agents – e.g. an Order agent will trigger the execution of a transportation service and, once completed, will thereafter trigger the execution of a testing service. The Order agent is blind to the interaction between the ETS and Transport agent necessary for the testing service to be executed – this interaction is completed through Resource-to-Resource communication.

From the physical system, two types of interaction between resources and the conveyor are identified: the placing of products on empty carriers and the removal of products from carriers. These physical interactions are represented by two AchieveRE protocols – one performing a *binding_request* and the other a *release_request*. The *binding_request* is used to initiate the placement of a product on a carrier, i.e. the binding of a product to a carrier. Alternatively, the *release_request* initiates the removal of a product from a carrier, so that a previously bound product is released from a carrier. The sequence of communication between an Order, ETS and Transport agent for the execution of a testing service is illustrated in Figure 29.

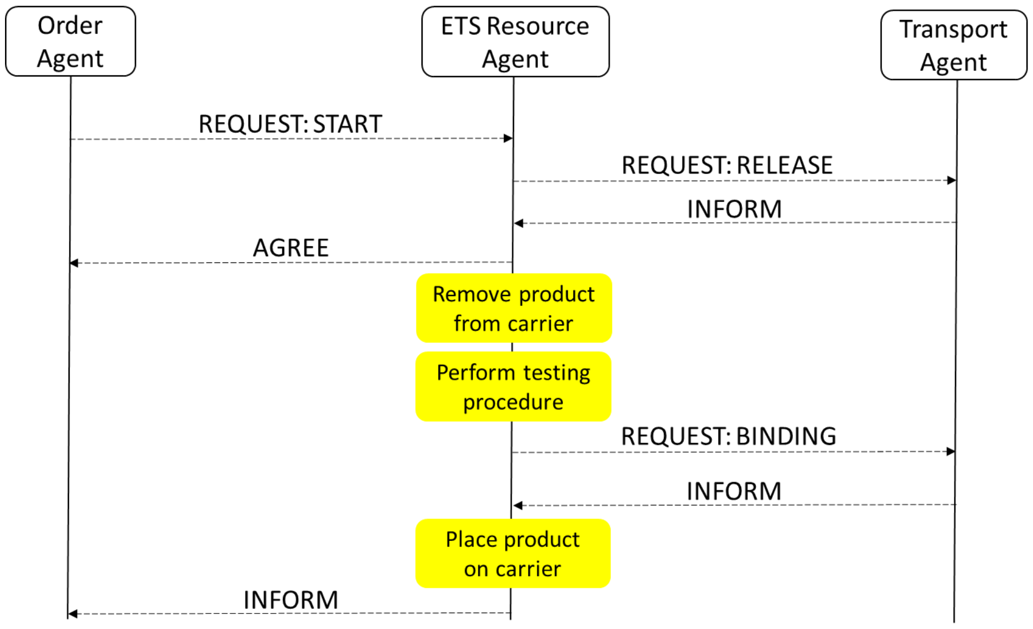


Figure 29: Communication sequence between the Order, ETS Resource and Transport agents.

Each type of request is accompanied by the exchange of important information. With a *binding_request* message the Resource agent must include information regarding the type of product that it wants to place on a carrier – since carriers might be fitted with fixtures that are specifically designed for certain product types, this information is used by the Transport agent to determine if a suitable carrier is available at the workstation. The message content is structured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
      <initiator>ResourceAgent_R06</initiator>
      <responder>TransportAgent</responder>
      <msg>
```

```
            <message_type>binding_request</message_type>
            <conversation_ID>C021</conversation_ID>
            <result>undefined</result>
            <info>
                  <order_ID>OA43</order_ID>
                  <product_ID>P02</product_ID>
            </info>
      </msg>
</message>
```

The Transport agent will reply with the result of the request – either an *inform* or *failure*. If the result is true, the specific position on the carrier may be specified, as carriers can be fitted with multiple fixtures and are thus capable of carrying more than one product at a time. The message content has the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
      <initiator>ResourceAgent_R06</initiator>
      <responder>TransportAgent</responder>
      <msg>
            <message_type>binding_request</message_type>
            <conversation_ID>C021</conversation_ID>
            <result>true</result>
            <info>
                  <place_coords>
                        <x>0.0</x>
                        <y>200.0</y>
                        <z>10.0</z>
                        <ang>0.0</ang>
                  </place_coords>
            </info>
      </msg>
</message>
```

For a *release_request*, the Resource agent must specify the product to be released, based on the Order agent that governs it. For a *release_request* message, the content slot of the FIPA RE Request message will contain the following XML string:

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
      <initiator>ResourceAgent_R06</initiator>
      <responder>TransportAgent</responder>
      <msg>
            <message_type>release_request</message_type>
            <conversation_ID>C027</conversation_ID>
            <result>undefined</result>
            <info>
                  <order_ID>OA43</order_ID>
            </info>
      </msg>
</message>
```

Should the requested product be available on the carrier at the workstation, the Transport agent will reply with an *inform* message. If multiple products are present on the carrier, the Transport agent must also specify the position of the product on the carrier – the information is structured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
      <initiator>ResourceAgent_R06</initiator>
      <responder>TransportAgent</responder>
      <msg>
            <message_type>release_request</message_type>
            <conversation_ID>C027</conversation_ID>
            <result>true</result>
            <info>
                  <pick_coords>
                        <x>0.0</x>
                        <y>200.0</y>
                        <z>10.0</z>
                        <ang>0.0</ang>
                  </pick_coords>
            </info>
      </msg>
</message>
```

5.4.3. Agents

The MAS implementation contains agents of four types, as prescribed by PROSA, namely Product, Resource, Order and Staff agents. The functionality of each agent type is described in this section.

5.4.3.1. Product Agent

The Product agent exhibits the behaviour of a simple server, only replying to received messages requesting the information for a specified product. The agent employs an `AchieveREResponder` behaviour to receive and handle request messages from Order agents. These request messages specify the product type in the content of the request message. The product information is then retrieved from the product information XML file and is converted to an XML string. The product information string is then added to the content slot of the inform ACL message that is replied to the requesting agent.

An extract from the product information XML file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<product_information>
      <product id="p01">
            <task_info>
                  <task type="feed">
                              <place_coords>
                                    <x>0.0</x>
                                    <y>0.0</y>
                                    <z>0.0</z>
                                    <ang>0.0</ang>
                              </place_coords>
                  </task>
                  <task type="transport">
                        <origin>"undefined"</origin>
                        <destination>"undefined"</destination>
                  </task>
                  •
                  •
                  •
            </task_info>
      </product>
```

5.4.3.2. Resource Agent

The Resource agents employ behaviours to negotiate service bookings via the contract net protocol, handle confirmation and start requests and the execution of the resource's service.

To negotiate service bookings, Resource agents employ the `ContractNetResponder` behaviour. In the `handleCFP()` method, the agent creates a proposal – this proposal contains a value that indicates the length of the resource's booking list. If a proposal is successful and an *accept_proposal* message is received, the information of the booking Order agent is added to the bookings list.

The confirmation of service bookings by an Order agent, as discussed in section 5.4.2.2, is handled with an `AchieveREResponder` behaviour added to the execution of Resource agents. The behaviour matches every incoming message to a message template – the message template uses a regular expression to evaluate the XML string content of the received message.

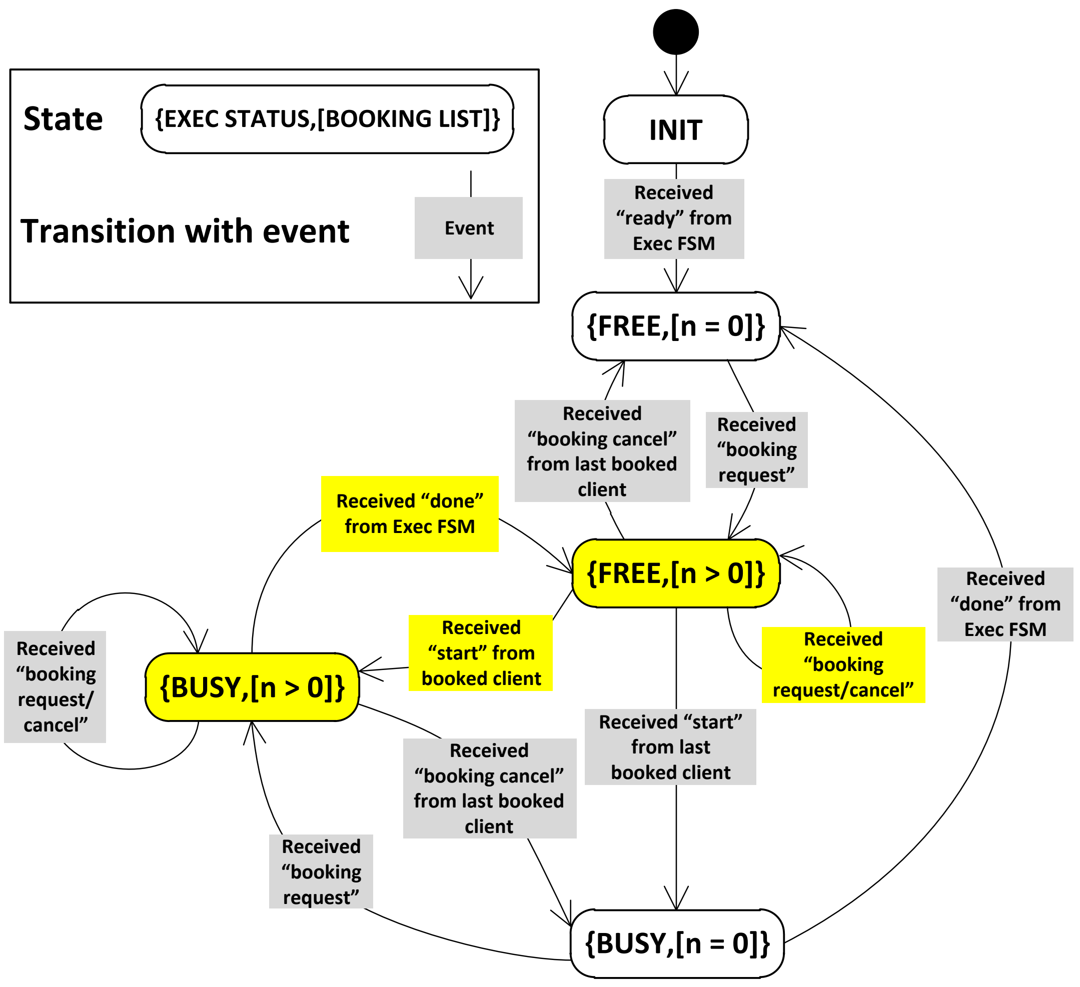

State
{EXEC STATUS,[BOOKING LIST]}
Transition with event
Event
INIT
Received "ready" from Exec FSM
{FREE,[n = 0]}
Received "booking cancel" from last booked client
Received "booking request"
Received "done" from Exec FSM
{FREE,[n > 0]}
Received "done" from Exec FSM
Received "booking request/ cancel"
{BUSY,[n > 0]}
Received "start" from booked client
Received "booking request/cancel"
Received "start" from last booked client
Received "booking cancel" from last booked client
Received "booking request"
{BUSY,[n = 0]}

Figure 30: State diagram for the Agenda Management component of the Resource agent.

The agent behaviour, as described by the state diagram in Figure 30, is constructed through three concurrently active JADE behaviours. Concurrency within the execution of the agent is needed to ensure that the Resource agent remains available for communication even when it is performing its designated service. One behaviour is responsible for handle service booking requests using the contract net protocol and another behaviour handles the confirmation protocol for service bookings. The third behaviour is responsible for the execution of the service as initiated by an Order agent that previously booked the Resource agent's service.

To handle service bookings from Order agents a `ContractNetResponder` behaviour is added to the execution of the Resource agent. The `ContractNetResponder` behaviour is built on the JADE finite state machine behaviour – the behaviour is constructed with the necessary states to participate in a CNP negotiation. The states provide the necessary methods to handle the communication with the CNP initiator agent.

An `AchieveREResponder` is added to the agent execution to handle the confirmation of service bookings by Order agents. Similar to the `ContractNetResponder` behaviour described above, the `AchieveREResponder` behaviour embodies a finite state machine that is configured to handle the communication of the FIPA RE protocol. The behaviour compares a received message with a defined message template – in this case, the content of the message is matched to a template specifying a confirmation message.

To handle the communication intended to start the execution of a booked service, a `SSResponderDispatcher` behaviour is added. The `SSResponderDispatcher` behaviour launches a behaviour that is dedicated to handle the communication with one specific agent, for a single communication session only. The `createResponder()` method of this behaviour allows the developer to specify which behaviour to handle the session. Here, a `SSIteratedAchieveREResponder` is utilised to handle the communication involved with the execution of the Resource's service. The `SSIteratedAchieveREResponder` is similar to the `AchieveREResponder` behaviour discussed earlier, but is different in the sense that the behaviour terminates after a single communication session.

For the `SSIteratedAchieveREResponder` behaviour that is launched to handle the start message, the standard `handleRequest()` method is overwritten. Instead, by using the `registerHandleRequest()` method, the actions that occur when a start request is received can be specified by the developer. This method is then used to add an `FSMBehaviour` that describes the execution of the Resource's service.

The behaviours described above, up to the service execution `FSMBehviour`, are generic for all Resource agents. Each Resource agent adds a `FSMBehaviour` that is specific to the service(s) that it can perform. The behaviour executes all the actions that are necessary to perform the booked service. Upon completion, the `FSMBehaviour` returns the result of the execution to the `SSIteratedAchieveREResponder` behaviour, which in turn replies to the Order agent with an *inform* or *failure* message.

5.4.3.3. Order Agent

Order agents must book and trigger the execution of the services, provided by Resource agents, to complete all the tasks specified in the product information of a certain product type. The finite state machine behaviour to implement this functionality and facilitate the necessary communication (as explained in section 5.4.2.2) is described in this section.

The Order agent firstly adds a behaviour to request and receive the product information from the Product agent – this is done by an `AchieveREInitiator` behaviour. Thereafter, the Order agent executes a `FSMBehaviour` until the product that it is responsible for is completed. The `FSMBehaviour` embodies the state diagram shown in Figure 31.

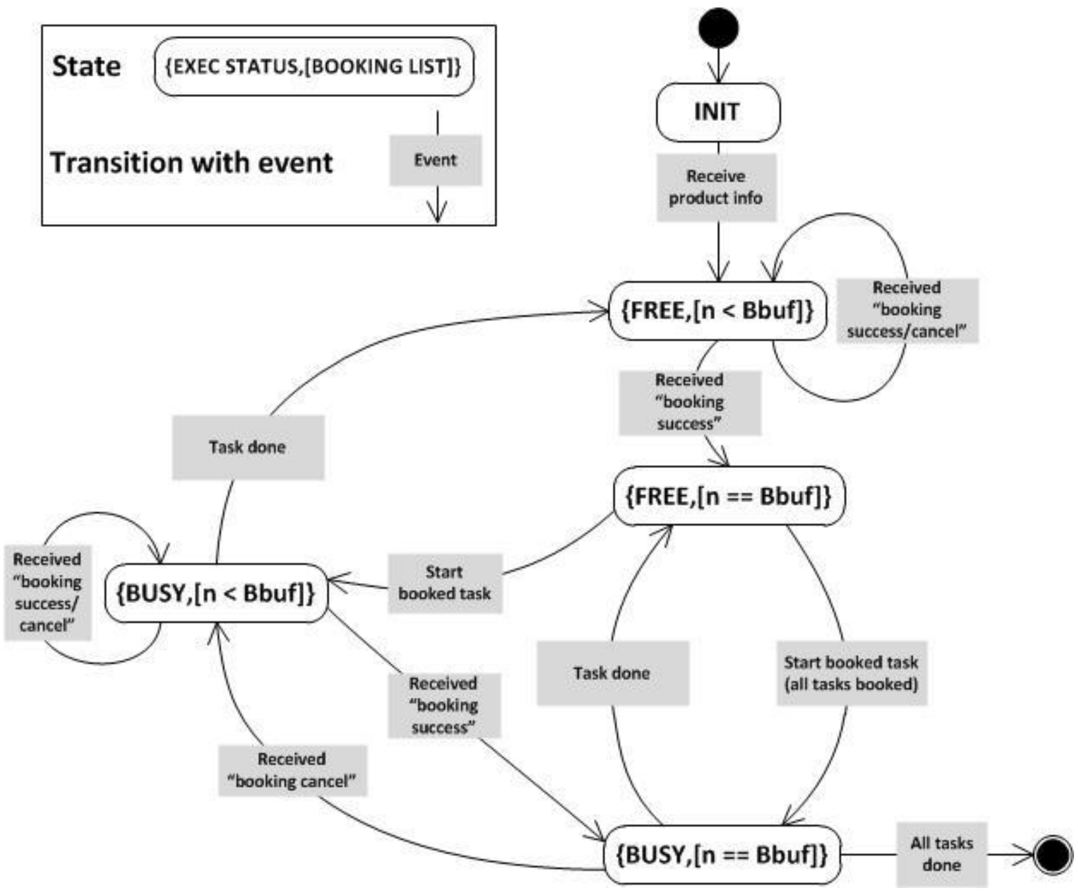

State
{EXEC STATUS,[BOOKING LIST]}
Transition with event
Event
INIT
Receive product info
{FREE,[n < Bbuf]}
Received "booking success/cancel"
Received "booking success"
{FREE,[n == Bbuf]}
Task done
Received "booking success/ cancel"
{BUSY,[n < Bbuf]}
Start booked task
Received "booking success"
Task done
Start booked task (all tasks booked)
Received "booking cancel"
{BUSY,[n == Bbuf]}
All tasks done

Figure 31: State diagram for the behaviour of an Order agent.

The execution in the "free-booking" (`{FREE, [n < Bbuf]}`) state is implemented using a `TickerBehaviour`. The function of this behaviour is to perform service bookings sequentially for the services specified in the product task list. This behaviour is executed periodically, adding a new `ContractNetInitiator` behaviour for service booking every time. The number of service bookings to be made in advance is determined by the user-defined booking buffer variable (`Bbuf` in Figure 31) – when the number of bookings made (`n` in Figure 31) is equal to the booking buffer, the `FSMBehaviour` transitions to the next state.

In the "free-booked" (`{FREE, [n == Bbuf]}`) state a `OneShotBehaviour` is added that triggers the execution of the first booked service in the bookings list of the Order agent. The execution, which includes the confirmation and starting of the service via communication with the booked Resource agent, is performed by a `SequentialBehaviour` (discussed at the end of this section). The `SequentialBehaviour` is started in a separate thread to simplify concurrency of the Order agent behaviours. The service that is being executed is removed from the bookings list, meaning that the number of entries is less than the specified booking buffer – the `FSMBehaviour` now transitions to the "busy-booking" state.

The behaviour of the "busy-booking" (`{BUSY, [n < Bbuf]}`) state is similar to the "free-booking" state. A `TickerBehaviour` adds `ContractNetInitiator` behaviours until the booking buffer is reached. When all required bookings have been made, a state transition to the "busy-booked" state occurs.

The "busy-booked" (`{BUSY, [n == Bbuf]}`) state also implements a `TickerBehaviour`, but here the behaviour just periodically checks the status of the booking list and service execution. During the first execution of the `TickerBehaviour` an `AchieveREResponder` behaviour is added. This behaviour will receive any booking cancellations from booked Resource agents (which can occur when the Resource agent either fails or is manually shut down) – in which case the cancelled booking is removed from the bookings list and a state transition is triggered back to the "busy-booking" state. Also, at every execution, the variable indicating the status of service completion is checked. If the service is completed, the state transitions to the "free-booked" state again so that the execution of the next booked service can be started. If the service is completed and it was the last service required for the product, the `FSMBehaviour` transitions to a "done" state to terminate execution of the Order agent.

When an Order agent has made enough service bookings to fill the booking buffer, the first service in the bookings list (which corresponds to the next service to be performed according to the product information) can be started. The confirmation of the service bookings and the starting of the service execution, through the protocols discussed in section 5.4.2.2, are done in a separate behaviour to the `FSMBehaviour` described above, and in a dedicated thread. The use of a dedicated thread, instead of adding concurrency through behaviours, was selected due to the simplicity of implementation. The thread is again terminated once the service is completed by the Resource agent.

The thread implements a JADE `SequentialBehaviour` to sequentially execute two `AchieveREInitaitor` behaviours – one for confirming the service booking and the other to start the execution. Once the Resource agent indicates the successful completion of the service execution through an *inform* message, the necessary updates are made to the agent variables and the behaviour, and thereafter the thread, terminates.

5.4.3.4. Staff Agents

Staff agents are included in the MAS implementation to provide the functionality that is not exhibited by the Product, Order and Resource agents. Apart from the Staff agents included in JADE (such as the Directory Facilitator), two Staff agents were added to the implementation: an Order Management agent and a Performance Logger agent.

5.4.3.4.1. Order Management Agent

As the name suggests, the Order Manager (OM) agent is responsible for the management of the Order agents within the MAS. The OM agent maintains a Graphical User Interface (GUI) to receive input from the user concerning the creation of Order agents. The user can specify the number of Order agents and the type of products they must produce – this information can be entered manually in the GUI fields, or a XML production schedule filename can be specified.

From the input information, the OM agent launches Order agents by sending requests to the Agent Management System. The OM agent displays the number of active Order agent in the MAS in the GUI – this number is incremented with each launched agent. Once Order agents have completed all required tasks, a *done* message is sent to the OM agent before the agent terminates – the number of Order agents is decremented when a *done* message is received.

5.4.3.4.2. Performance Logger Agent
To gather diagnostic information on the performance of Resource and Order agents, a Performance Logger (PL) agent is added to the MAS. The PL agent records the number of times a Resource agent performs its service, the duration of each service execution and the total time that the Resource agent spends in service execution – all data required to calculate the utilization of the resource during a period of production. The agent also records the start and end times of the execution of Order agent, in order to provide data for the calculation of the time-in-system of each product and the overall production throughput.

Resource agents send a *start* message to the PL agent every time a service execution is started and a *done* message when the execution is completed. The PL agent starts a timer for every *start* message received from a Resource agent and stops the timer when the *done* message is received. The information is stored in an `ArrayList` data structure – the entries of the `ArrayList` are of a custom class type, with fields for the Resource agent's name, its activity status, the total number of services performed and the total time that the Resource has been active. Similarly, Order agents send corresponding messages upon their instantiation and termination.

5.5. Conclusion
Holonic control architectures have been frequently used in manufacturing systems to reduce the complexity of the control system, simplify reconfiguration and improve robustness. Multi-Agent Systems (MASs) have often been used to facilitate the implementation of holonic control due to the similarities between holons and software agents. Of the MAS development tools used to implement holonic control in manufacturing systems, the Java Agent Development (JADE) framework is the most popular choice.

This paper presented a JADE MAS implementation of a reference holonic control architecture for a manufacturing cell. The implementation used an electric circuit breaker assembly and testing cell as case study, of which the various functional components were mapped to the holon types as prescribed by the PROSA reference architecture. The high level control components of the holons are implemented as agents in the MAS. The communication between the agents is described and the implementation of the functionality for each agent type is discussed.

The implementation of holonic control using JADE MASs has become the *status quo* within the field of holonic and reconfigurable manufacturing systems. For this reason, the presented MAS implementation is used as a baseline for a comparison with an equivalent holonic control implementation that is based on the Erlang programming language (details on this implementation can be found in Kruger and Basson (2015; 2017 (a)). The evaluation criteria and the comparison are presented in Kruger and Basson (2017 (b); 2017 (c)).

5.6. References

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

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.

Bellifemine, F., Caire, G. and Greenwood, G., 2007. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, Ltd.

Chirn, J.L. and McFarlane, D., 2000. A Holonic Component-based Approach to Reconfigurable Manufacturing Control Architecture. *Proceedings of the International Workshop on Industrial Applications of Holonic and Multi-Agent Systems*. pp. 219–223.

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

Giret, A. and Botti, V., 2009. Engineering Holonic Manufacturing Systems. *Computers in Industry*. Vol. 60:428-440.

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.

Kruger, K. and Basson, A.H., 2015. Implementation of an Erlang-Based resource Holon for a Holonic Manufacturing Cell. *Service Orientation in Holonic and Multi-Agent Manufacturing*, Studies in Computational Intelligence, Springer International Publishing.

Kruger, K. and Basson, A.H., 2016. Erlang-based Holonic Controller for a Modular Conveyor System. *6th Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*. Lisbon, Portugal (October 2016).

Kruger, K. and Basson, A.H., 2017 (a). Erlang-Based Control Implementation for a Holonic Manufacturing Cell. *International Journal of Computer Integrated Manufacturing*. Vol. 30, No. 6:641-652.

Kruger, K. and Basson, A.H., 2017 (b). Evaluation Criteria for Holonic Control Implementations in Manufacturing Systems. *Submitted to the International Journal of Computer Integrated Manufacturing, September 2017.*

Kruger, K. and Basson, A.H., 2017 (c). Comparison of Multi-Agent System and Erlang Holonic Control Implementations for a Manufacturing Cell. *Submitted to the International Journal of Computer Integrated Manufacturing, September 2017.*

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

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., and Koren, Y., 2000. Reconfigurable Manufacturing Systems: Key to Future Manufacturing. *Journal of Intelligent Manufacturing*. Vol. 13: 135-146.

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

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

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.

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

6. Evaluation

This section describes the evaluation of the Erlang holonic control implementation. The evaluation is aimed at determining the suitability of the Erlang programming language, supplemented by OTP, for the implementation of holonic control in manufacturing systems. As is discussed in section 1.4, the evaluation uses a comparison with an equivalent control implementation using a JADE MAS.

The section is comprised of two papers. In section 6.1, the first paper, "Evaluation Criteria for Holonic Control Implementations for Manufacturing Systems", presents the evaluation criteria that is used for the proposed evaluation. The paper formulates a set of requirements for holonic control implementations and relates these requirements to quantitative and qualitative performance measures. The second paper, which performs an evaluation and comparison of the Erlang and MAS holonic control implementations, is presented in section 6.2 and is titled "Comparison of Multi-Agent System and Erlang Holonic Control Implementations for a Manufacturing Cell". The comparison is performed according to the evaluation criteria presented in the first paper.

The two papers presented in this section have not been published, but have been submitted to an appropriate international journal for review.

6.1. Evaluation Criteria for Holonic Control Implementations in Manufacturing Systems

Karel Kruger [a,*] and Anton H Basson [a]
[a] Dept of Mechanical and Mechatronic Eng, Stellenbosch Univ, Stellenbosch 7600, South Africa
[*] Corresponding author. Tel: +27 21 808 4258; Email: kkruger@sun.ac.za

Abstract

Holonic control architectures have often been implemented in research on modern manufacturing systems. Although many holonic control systems were implemented using agent-based programming platforms, other platforms hold potential advantages. A comparison of alternative implementations of holonic control requires a set of evaluation criteria that can be used to compare the alternative. This paper presents such a set of evaluation criteria that are focussed on the implementation of holonic control in manufacturing systems. The evaluation criteria are formulated from a review of literature, combined with experience in developing holonic control implementations. The most important characteristics, requirements and performance measures are identified and discussed.

Keywords: Holonic Manufacturing System (HMS); Reconfigurable Manufacturing System (RMS); evaluation criteria

6.1.1. Introduction

Modern manufacturing has been shaped by aggressive global competition and uncertainty, characterised by dynamic changes in economical, technological and customer trends (Leitao and Restivo, 2006). Bi *et al.* (2008) identifies the critical requirements for modern manufacturing systems to be the shortening of lead times for the introduction of new products into the system, the ability to produce a larger number of product variants and the ability to handle fluctuating production volumes.

The requirements for modern manufacturing necessitate paradigm shifts, such as Reconfigurable Manufacturing Systems (RMSs) and, more recently, Cyber-Physical Production Systems (CPPSs) and Industry 4.0. RMSs are designed to facilitate fast and easy switching between members of a particular family of products, by adding or removing functional elements (hardware or software) (Martinsen, 2007; Vyatkin, 2007). Bi *et al.* (2007; 2008) argue that RMSs should also be able to rapidly adjust the production capacity and functionality in response to sudden changes, by reconfiguring hardware and control resources. Sharing some of RMSs' properties, CPPSs have recently become a major focus. The three main characteristics of CPPSs are (Monostori *et al.*, 2016):

- Intelligence – the elements are able to acquire information from their surroundings and act autonomously;
- Connectedness – the ability to set up and use connections to the other elements of the system – including human beings – for cooperation and

collaboration, and to the knowledge and services available on the Internet; and

- Responsiveness towards internal and external changes.

Holonic control architectures are well suited to enable control reconfiguration in RMSs and CPPSs. Koestler (1967) first introduced the term *holon* – a combination of 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). In manufacturing systems, a holon can be generally defined as 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 common goal) which integrates the entire range of manufacturing activities" (Paolucci and Sacile, 2005).

In HMSs, holons can comprise software alone or a combination of software and hardware. In this paper, the focus is on the software parts of HMSs and the computer hardware on which the software runs, which usually correspond to the control parts of the HMS. For brevity, the term HMS will further be used in this paper, but readers should keep in mind that it only refers to the software and associated computers. However, low level controllers tightly coupled with specific hardware, such as variable speed drives, are also excluded from consideration in this paper.

There have been several implementations of HMSs. The most common approach is Multi-Agent Systems (MASs). The two most prominent examples of this approach can be found in the implementation of the PROSA (Van Brussel *et al.*, 1998) and ADACOR (Leitao and Restivo, 2006). MASs are also commonly applied in CPPS (Monostori *et al.*, 2016).

It is important here to distinguish between the holonic architecture and its implementation. Often, in manufacturing systems context, the terms MAS and HMS are used nearly interchangeably, presumably since they share so many characteristics. However, a HMS need not be a MAS and can be implemented using other approaches. For example, the IEC 61499 standard has been used to implement holonic control on industrial PLCs (Vyatkin, 2007). Other implementations are the Holonic Component Based Architecture (Chirn and McFarlane, 2000) and an Erlang/OTP implementation (Kruger and Basson, 2017 (a)). Reasons for considering alternatives to an MAS when implementing a HMS include that MASs have found little acceptance in manufacturing industries and that MAS place high demands on the computer systems when systems become complex.

The evaluation of alternative HMS implementations has proven to be a challenging task. Several studies have included evaluation criteria, with the formulations varying in focus and perspective. These variations inhibit the comparison of different researchers' work. The HMS Consortium initially identified a set of critical factors for holonic systems to facilitate agile manufacturing systems for the 21st century (Christensen, 1994). As HMSs have often been closely linked with research on RMSs, the six key characteristics for reconfigurable systems (as defined by Koren and Shpitalni (2010)) have frequently been used as a starting point for the formulation of evaluation criteria – naturally, these formulations are focussed on reconfigurability. In the validation of ADACOR, the evaluation was focussed on agility in the event of unexpected disturbances (Leitao and Restivo, 2008).

This paper presents evaluation criteria that are focussed on providing a base for comparing the implementation of holonic control in manufacturing systems. It must be emphasised that the focus here is not on the evaluation of a holonic architecture, but on alternative implementations of a given architecture. The formulation of the evaluation criteria is based on considerations from literature and from the authors' experience in researching alternative implementations of holonic control in manufacturing systems.

The paper starts with identifying the desired characteristics for holonic control implementations, from which a set of requirements are derived. The requirements are then used to formulate a set of quantitative and qualitative performance measures that can be used to evaluate and compare holonic control implementations.

6.1.2. Characteristics and Requirements for Holonic Control Implementation

There have been many attempts to formulate a set of desired characteristics and resulting requirements for modern manufacturing systems – several have focussed specifically on HMSs. The formulations presented in Christensen (1994), Bussman (1998) and Bussman and McFarlane (1999) during early holonic systems research are still relevant. These formulations are used here as the basis for the formulation of the characteristics and requirements that are used as base for the proposed evaluation criteria.

The set of desired characteristics to be exhibited by holonic control implementations is derived from the following objective: developing holonic control systems that are tailored to satisfy the needs of industry, in order to achieve greater industry adoption.

Despite a great deal of research and laboratory implementations of holonic systems – predominantly using MASs – there are only few cases of effective adoption by industry (Almeida *et al.*, 2010). Considering the needs of industry, two over-arching characteristics that hold great value for industry, are identified – *availability* and *supportability*. To meet these needs, a third characteristic, *development productivity,* focusses on the development of reliable, customized holonic control implementations with short lead times. Elaborations on the proposed characteristics, and the requirements necessary to achieve them, are presented in the following sections.

6.1.2.1. Availability and Supportability

The HMS Consortium identified availability as a critical factor for a successful modern enterprise (Christensen, 1994). In the manufacturing industry, the importance of availability is seen in the conservative approach used when selecting controllers, with a strong preference for well-established brands of automation controllers, in spite of the availability of lower cost alternatives. In this paper, the availability of the system is defined as the percentage of time that a system is capable of production, even in some sub-optimal capacity. To achieve high availability, a manufacturing system must meet three important requirements: reconfigurability, robustness and maintainability.

For industry adoption, it is then also important for systems to be easily supported. Supportability, in this context, refers to the presence of infrastructure and mechanisms to facilitate the adoption, customization and maintenance of a HMS. The supportability of an implementation is dependent on maintainability, portability and controller requirements.

The importance of reconfigurability have been emphasised in the literature on RMSs (e.g. Koren (2010)), but has also found renewed interest in research on CPPSs and Industry 4.0 (Monostori *et al.*, 2016). Reconfiguration refers to the process wherein functional entities (hardware or software) are added, removed or rearranged in a manufacturing system. Reconfiguration activities may involve changes to the products, machinery, production capacity or control system. All of these changes can halt production, but a system with good reconfigurability can achieve short down- and ramp-up times – thus improving system availability. However, to reduce the complexity and level of effort associated with such changes, the system must inherently support reconfigurability.

The availability and supportability of a system can be adversely affected by the occurrence of faults, e.g. machine breakdowns or communication failures. The ability of the system to remain available for production (be it in some sub-optimal state) amidst the occurrence of such events is characterised as robustness. A fault is defined here, as in Leitao (2004), as a disturbance that causes an unexpected disruption to production. The HMS Consortium (Christensen, 1994) identified fault tolerance as a critical factor for holonic system architectures and it has been an area of focus in studies on HMSs (see Leitao (2004), Leitao *et al.* (2006) and Leitao and Restivo (2008)).

All systems require periodic maintenance to ensure continued production over an extended period. The maintenance of software involves the modification of the system or its components to correct faults, improve performance or adapt to the changes in the environment (Coleman *et al.*, 1994). Maintainability then refers to the complexity and effort involved in maintenance activities. Maintenance is generally considered to incur a significant fraction of the life cycle costs of software. In a manufacturing control context, good maintainability characteristics reduce the production downtime and costs due to maintenance and thus improve the availability and supportability of the system.

The requirements for the controllers to be used for the holonic control implementation are an important consideration for availability and supportability. Two types of controller requirements are considered: the computational capacity of controllers and the ability to use controllers in distributed networks. The computational requirements focus specifically on processing power and memory usage, as improvements in microprocessor technology have opened the way for highly distributed control networks in the manufacturing industry – as driven by the CPPS and Industry 4.0 movements. These distributed networks of connected controllers are essential for the implementation of decentralized control architectures. Controllers might be obtained from different vendors, varying in software and hardware platforms and interfaces. It is thus vital for holonic control implementations to utilize a variety of limited-resource controllers effectively in distributed networks, with minimal changes to architecture and execution.

6.1.2.2. Development Productivity

From the perspective of the developers of HMSs, productivity is an important consideration. Trendowicz and Münch (2009) agree with Kennedy *et al.* (2004) that development productivity is highly dependent on the attributes of the selected programming language. Productivity can be dependent on numerous attributes, but the following are selected as the most relevant to this evaluation: the *complexity* of the software, the *reusability* of software artefacts and the *verification* of the developed software.

Trendowicz and Münch (2009) report that software complexity is commonly used as a productivity factor in different domains of software development. This is not surprising, since complexity can be indicative of the difficulty in implementing, understanding, modifying and maintaining software programs (Weyuker, 1988).

The reuse of software involves using existing software artefacts in the construction of a new software system (Krueger, 1992). The reusability that is offered by a programming language can reduce the amount of code generation and thus increase productivity (Prieto-Diaz and Freeman, 1987).

Finally, every piece of software that is developed must be tested to verify that the desired functionality and reliability is exhibited. The programs constituting a holonic control implementation will typically be subjected to dynamic software testing (Vaos and Miller, 1995) to ensure probable correctness. It is important that the chosen programming language offer mechanisms to facilitate the efficient verification of code.

6.1.3. Relationships between Requirements and Performance Measures

The characteristics and requirements identified in the previous section can be related to a set of quantitative and qualitative performance measures. The performance measures are indicative of one or more of the requirements for holonic control implementations – this relationship is shown in Table 3. The performance measures are discussed in the following sections.

Table 3: Relationships between requirements and performance measures.

			Characteristics						
			Availability Supportability Development Productivity						
			Requirements						
			Reconfigurability	Robustness	Maintainability	Controller requirements	Complexity	Verification	Reusability
Performance measures	Quantitative	Reconfiguration time	*				*	*	*
		Development time					*	*	*
		Code complexity			*		*		
		Code extension rate	*		*		*		
		Code re-use rate	*		*		*		*
		Computational resource requirements				*			
	Qualitative	Modularity	*		*			*	*
		Integrability	*						*
		Diagnosability	*	*	*			*	
		Convertibility	*		*				
		Fault tolerance		*					
		Distributability				*			
		Developer training requirements			*		*	*	

6.1.4. Performance Measures

6.1.4.1. Quantitative Measures

This section introduces each of the quantitative performance measures that form part of the evaluation criteria. For each measure, the relevance concerning holonic control implementations is discussed, the underlying concept or philosophy is described and the method of measurement is explained.

6.1.4.1.1. Reconfiguration Time

Section 6.1.2.1 explains the importance of reconfigurability in improving system availability and supportability. Reconfigurability is determined by the complexity and amount of work involved in performing a reconfiguration. A time measurement is used to indicate the ease and required effort by which reconfiguration of a holonic control implementation can be performed. The time it takes to perform a reconfiguration activity is referred to as the reconfiguration time.

Reconfiguration time can be measured by conducting a reconfiguration experiment, e.g. by introducing a new holon into an existing HMS, where the new holon differs from the types of holons already in the HMS. Reconfiguration time would then be measured as the time required by the developer to adapt the system to effectively utilize the new holon, including the time needed to implement the required changes to the code of the holons and the verification that the system performs as required. The development time for the new holon itself is excluded from this measurement, since it is considered in the next performance measure.

6.1.4.1.2. Development Time

As mentioned in section 6.1.1, short lead times is a key requirement for holonic manufacturing systems. The lead time for the introduction of new production capability or system functionality is strongly influenced by the development process. Development time here therefore refers to the time required to develop new control software. The development may include the reuse of code or be based on existing functional software components, but the end product exhibits functionality different to that of any existing component.

As with reconfiguration time, development time can be measured through a reconfiguration experiment. In such an experiment, a new type of holon can be developed and added to an existing holonic system. The source code of the holon software can consist of both new and reused code and software artefacts. The verification of the developed software is included in the development time measurement.

Development time is then measured in terms of developer work hours. To ensure that the focus remains purely on the part of the implementation process affected by the implementation platform, the measurement excludes an initial planning period, i.e. the time required for the developer to fully understand the problem at hand and orientate himself within the source code library of the holonic control implementation.

6.1.4.1.3. Code Complexity

In section 6.1.2.2 it is argued that software complexity has a significant influence on development productivity. Moreover, the complexity of implementing reconfigurable manufacturing and control systems is considered as a barrier to industry adoption (Almeida, 2010). It is thus necessary to include a performance measure with focus on the perceived complexity of the source code in the control implementation.

Many studies have focussed on the development and use of measures to quantify the complexity of program source code. Some commonly used measures are aimed at the complexity of the algorithm that is implemented, such as the cyclomatic number (McCabe, 1976). However, many holonic control implementations follow similar reference architectures and algorithms – the complexity measure used here is thus aimed at the complexity of the resulting source code.

Any evaluation based on complexity measures should be approached with care. It is a challenging task to formulate objective metrics. The proposed evaluation is similar to that shown in Cesarini *et al.* (2008), using a code complexity measure that is based on a simple "source lines of code" (SLOC) measurement. This measure is based on a simple philosophy: more lines of code mean more work, and more errors. Hubbard (1999) argues that a SLOC measurement is dependent on at least three factors:

- Program functionality
- Programmer skill
- Programming language

Assuming that similar architectures are used for holonic control implementations, which allow for similar functionality and performance (which can be explicitly verified), a bias in SLOC due to differences in program functionality is avoided. It is inevitable that a bias would exist due to programmer skill, but the impact thereof can be diminished by considering multiple programmers with varied programming experience. Finally, the influence of programming language on SLOC is in line with the aim of this evaluation.

The SLOC measure is intuitive to understand and attractive due to the ease by which the counting of SLOC can be automated. The SLOC count includes lines of code that:

- Are non-blank
- Are not comments
- Are not delimiters for code elements
- Are not declarations for the inclusion of software artefacts to the inspected module or class (*import* in Java and *include* in Erlang)
- Have been produced by the developer (i.e. not automatically generated).

In the event of software reconfiguration, the SLOC measure is also used as the basis for two other quantitative measures – *code extension rate* and *code reuse rate* – as described in the following sections.

6.1.4.1.4. Code Extension Rate

Any HMS reconfiguration usually involves the reconfiguration of the controller source code for one or more holons. The time and effort of such a reconfiguration is dependent on the ease by which the source code can be adapted. *Code extension rate* is an index that represents the growth rate of the scale, and thus complexity, as an existing implementation is reconfigured to meet new functional requirements (Chirn and McFarlane, 2005).

Code extension rate (E_{i+1}) is calculated as the ratio of the code complexity measure in SLOC in the final configuration $(SLOC)_{i+1}$ to that of the initial configuration $(SLOC)_i$ – a visual explanation is illustrated in Figure 32. Code extension rate is calculated as follows:

$$E_{i+1} = \frac{(SLOC)_{i+1}}{(SLOC)_i}$$

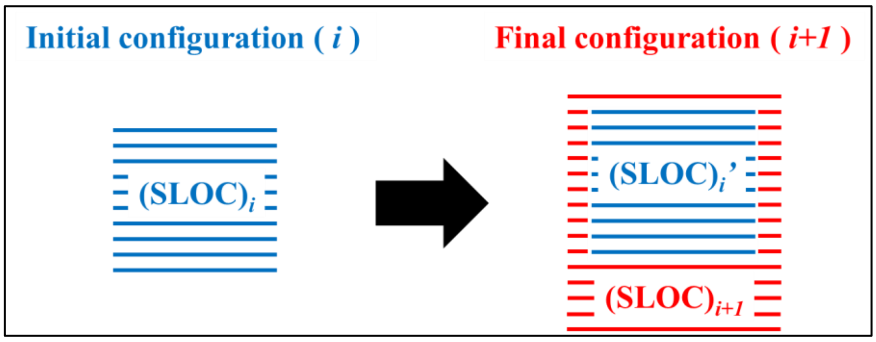


Figure 32: Illustration for the calculation of code extension and reuse rates.

6.1.4.1.5. Code Reuse Rate

The importance of software reusability in achieving high productivity is explained in section 6.1.2.2. Similar to the code extension rate, *code reuse rate* can be calculated in the event of reconfiguration. Code reuse rate provides a measure of the percentage of source code in a new configuration that is reused from an initial configuration.

Again considering Figure 32, code reuse rate (R_{i+1}) is calculated as the ratio of SLOC in the new configuration that was reused from the initial configuration $(SLOC)'_i$, to the total SLOC of the final configuration $(SLOC)_{i+1}$:

$$R_{i+1} = \frac{(SLOC)'_i}{(SLOC)_{i+1}}$$

6.1.4.1.6. Computational Resource Requirements

Advances in microprocessor technology in recent years have led to enhanced functionality at lower cost. This evolution of controllers holds advantages for HMSs, where control implementations are often distributed over multiple communicating controllers. These controllers are, however, still limited in their computational and memory capacity. It is then important that the programming language used for holonic control implementations allow for the exploitation of these controllers, within their limits of use.

Considering holonic control architectures, it might be desired that a significant portion of the control processes be distributed to dedicated, resource-limited controllers. The measure of the computational resource requirements of the holonic control implementation is an important indicator of the extent to which the functionality can be supported by resource-limited controllers.

To assess the computational resource requirements, two measures are used:

- CPU time – as an indicator of CPU usage, CPU time is the measurement of the combined time, over all available cores, that the CPU executes instructions for the holonic control implementation (Microsoft TechNet, s.a.). CPU time is measured by the operating system and in Windows is available in the processes window of the task manager application. The CPU time is recorded at the start of production (thus the CPU time involved with system startup is excluded) and again when production of the last product is completed.
- Memory usage – the Random Access Memory (RAM) consumed by each implementation is monitored during production. Windows includes the Performance Monitor application, which allows the user to record many of the counters exposed by the operating system. There exist counters for every active process on the PC. The *Private Working Set* counter measures the RAM (in bytes) that is consumed by a single process (Microsoft TechNet, s.a.) – this counter is recorded for the duration of a production run.

6.1.4.2. Qualitative Measures

The qualitative performance measures included in this set of evaluation criteria are discussed in this section. The relevance and importance of each measure is discussed and the specific qualities that are compared are identified.

6.1.4.2.1. Modularity

Modularity is considered a critical characteristic in most aspects of modern manufacturing. Wiendahl *et al.* (2007) list modularity as a key enabler for changeable manufacturing, affecting both physical and software elements. Koren (2006) includes modularity as a key characteristic for RMSs, classifying it as a supporting characteristic that reduces reconfiguration time and effort. Baldwin and Clark (2006) argue that modularity, in general, has three purposes:

- Managing complexity, as it provides an effective division of cognitive labour;
- Enabling parallel work, as it allows work on modules to be performed simultaneously and independently; and
- Accommodating future uncertainty, as it facilitates changes or improvements to the system without affecting the system as a whole.

In computer science and industry, modularity has been considered and implemented since as early as the 1950s. Software modularity refers to the architectural provisions of a software framework to facilitate the encapsulation and compartmentalization of functionality. Baldwin and Clark (2006) indicate that software modularity depends on three specifications:

- Architecture – identifying the modules;
- Interfaces – defining how modules interact; and
- Tests – verifying the performance of individual and interacting modules.

The proposed criteria, based on the above-mentioned specifications, in this context are:

1. The architectural considerations in the platform that enable the modular implementation of a holonic reference architecture.
2. The provisions in the programming language to facilitate the interaction of software modules.
3. The mechanisms for verifying the performance of individual and interacting modules provided in the programming platform/language.

6.1.4.2.2. Integrability

Koren (2006) also lists integrability as a key, supporting characteristic for RMSs. The connectedness expected in CPPSs also relies on integrability (Monostori *et al.*, 2016). Integrability refers to the ability to quickly and effectively integrate mechanical, informational and control components with an existing system. In a dynamic manufacturing environment where new technologies are rapidly developed, the extent to which such technologies can be integrated and effectively exploited is critically important. Considering the adoption of holonic control systems by industry, it is also crucial that legacy systems can be integrated with new control implementations.

In the context of control implementation, integrability depends on the interfaces provided by the programming language to facilitate integration with software and communication technologies. The integrability of the implementations is thus evaluated in terms of the following aspects:

- The interfaces that each programming language provides to integrate with foreign code, i.e. software components developed in different programming languages. Common examples are the integration of Dynamic Linked Libraries (DLLs) (to incorporate specific functionality) or a device driver to utilize hardware (e.g. network cards and cameras). New holonic control implementations might be required to utilize legacy software systems written in a different programming language. The use of these systems might be desired as they could already be optimized for performance, were specifically developed for some context or it would be too time consuming to rewrite the code.
- The provision of libraries or functions to implement communication protocols. This is important for the interface between the high level and low level control, where the communication protocol may be prescribed by the low level controller or machine specifications.

6.1.4.2.3. Diagnosability

Diagnosability is the last of the supporting characteristics for RMSs, as identified by Koren (2006), and is also a key characteristic of CPPSs and Industry 4.0 (Monostori *et al.*, 2016). Diagnosability here refers to the ease and speed by which the source of quality and reliability problems can be identified in a system. The diagnosability of a system also affects the amount of time required to determine whether a system is performing correctly and reliably. It is then intuitive that good system diagnosability reduces ramp-up time after reconfiguration and downtime during maintenance.

Diagnosability is also important in the context of software systems. Le Traon *et al.* (2003) reason that an important part of the software validation effort is spent on testing and diagnosis. While testing is concerned with uncovering and detecting errors, diagnosis aims to locate the components of the system where the error originated. Diagnosability then refers to the effort and speed by which the source of errors can be precisely located in a software system. The diagnosis of errors, and hence diagnosability of a system, is dependent on the capacity of the testing strategy to isolate components in the system.

The modular nature of holonic control implementations allow for the classification of two types of errors: errors that occur within the execution of a holon, or errors arising from the interaction between holons or holon components. The following provisions of the implementations influence the diagnosis of these errors:

- The functionality for constructing tests to identify the cause and location of errors.
- The built-in functionality or mechanisms for monitoring communication and execution.

6.1.4.2.4. Convertibility

Koren (2006) lists convertibility as a key, critical characteristic for reconfigurable systems, enabling a reduction of reconfiguration time and system life-cycle cost and increasing system productivity. At the control level, convertibility refers to the transformation of the functionality of the existing system to meet new production requirements.

It is often necessary for the operator (or some external process) to make changes to the manufacturing system during operation. Examples of such changes include alterations to the production schedule (e.g. rush orders) or the manual shut down and restarting of workstations (e.g. for unscheduled maintenance). The provisions in the implementation to facilitate such changes, with minimal disruption, are therefore evaluated.

6.1.4.2.5. Fault Tolerance

It is inevitable that faults will occur within a manufacturing system. These faults might be the result of programming errors, machine or controller breakdowns, or communication failures. Fault tolerance refers to the ability to remain operational with a useful degree of system stability, and is a critical indicator of system robustness. The evaluation criteria for fault tolerance is based on the following:

- Fault isolation – it is critical for control implementations to limit the propagation of errors, i.e. to minimize the effects of an error on other components of the system. The isolation of the fault minimizes the impact of the disturbance on the system.

- Fault detection – for a system to be tolerant of faults, it is essential that faults are identified when they occur. Only when a fault is detected is it possible for the system to react.
- Fault handling – it is desired for the system to react in the event of a detected fault in order to ensure system stability and reduce the effect on the overall performance.

6.1.4.2.6. Distributability

An attractive characteristic of the holonic systems approach is that it inherently enables distributed control. Different definitions exist for distributed control in manufacturing systems (see Bousbia and Trentesaux (2002) for a summary) – in this paper, it refers to the implementation of a decentralized control architecture of which the control components run on multiple independent controllers, connected on a network.

The distribution of control promise advantages in robustness and portability. Having the control implemented on multiple controllers ensures a greater tolerance for faults and simplifies the application of unplanned changes in the system. The capacity for distribution allows the control implementation to be extended to utilize additional controllers (possibly added to the cell/system during a reconfiguration) and support the physical distribution of manufacturing. The evaluation criteria are thus based on the following:

- The architectural provisions to facilitate distribution.
- The facilitation of communication between distributed control components.
- The availability of tools for developing, testing and commissioning distributed systems.
- The portability properties – i.e. the provisions for the implementation to be installed on different platforms.

6.1.4.2.7. Developer Training Requirements

It is natural to expect the developers of holonic control implementations to be trained in holonic systems theory and in the development of software. However, since the holonic systems community is still relatively small, developers are scarce – new developers, regardless of their prior background in software development, usually require training in the specifics of holonic control principles and implementation practices. This training regimen can be costly and time consuming.

It is considered here that a developer must understand the holonic architecture, be able to implement the execution and communication functionality in a specified programming language and verify the functionality of the system. With these capabilities, the developer is able to commission the system and perform reconfiguration and maintenance activities as facilitated by the implementation.

From these actions it is clear that the developer training requirements are indicative of the reconfigurability and maintainability of the control implementation, and also the complexity and verification effort involved in the software development. The evaluation criteria examine the expertise and experience required of the developer to perform the following tasks:

- Implement holon behaviour.
- Implement concurrency in the holonic control implementation.
- Implement mechanisms for inter-holon communication.
- Implement mechanisms for external communication.
- Verify the functionality of the control implementation.

6.1.5. Conclusion

This paper presents a set of evaluation criteria for comparing alternative implementations of the software of HMSs. The criteria build on criteria used in literature, such as the key requirements for RMSs and the critical factors for HMSs, but is specifically formulated to emphasise the implementation of holonic control in manufacturing systems.

Three characteristics of the control implementation that will promote the development of holonic systems tailored to the needs of industry were identified: availability, supportability and development productivity. From these characteristics, several requirements for control implementations are derived. To enable an evaluation and comparison based on the requirements, the paper proposes a set of quantitative and qualitative performance measures.

It should be noted that a comparison of alternative implementations would only be possible on a case study basis for the quantitative performance measures, since the values attributed to the performance measures are case-dependent. To achieve a more generic comparison of implementation alternatives, further research is required to identify a standardised set of test cases.

6.1.6. References

Almeida, F.L., Terra, B.M., Dias, P.A., and Gonçales, G.M., 2010. Adoption Issues of Multi-Agent Systems in Manufacturing Industry. *Fifth International Multi-conference on Computing in the Global Information Technology*. pp. 238-244.

Baldwin, C. and Clark, K., 2006. Modularity in the Design of Complex Engineering Systems. *Complex Engineered Systems*. pp.175-205.

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

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.

Bousbia, S. and Trentesaux, D., 2002. Self-Organization in Distributed Manufacturing Control: State-of-the-Art and Future Trends. *2002 IEEE International Conference on Systems, Man and Cybernetics*. Vol. 5, pp. 6-12.

Bussman, S., 1998. An Agent-Oriented Architecture for Holonic Manufacturing Control. *1st Workshop on Intelligent Manufacturing Systems*. Lausanne, Switzerland.

Bussman, S. and McFarlane, D., 1999. Rationales for Holonic manufacturing Control. *Proceedings of the 2nd International Workshop on Intelligent Manufacturing Systems*. Leuven, Belgium (September 1999). pp. 177-184.

Cesarini, F., Pappalardo, V. and Santoro, C., 2008. A Comparative Evaluation of Imperative and Functional Implementations of the IMAP Protocol. *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*. pp. 29-40. ACM.

Chirn, J.L. and McFarlane, D., 2000. A Holonic Component-based Approach to Reconfigurable Manufacturing Control Architecture. *Proceedings of the International Workshop on Industrial Applications of Holonic and Multi-Agent Systems*. pp. 219–223.

Christensen, J.H., 1994. Holonic Manufacturing Systems: Initial Architecture and Standards Directions. *First European Conference on Holonic Manufacturing Systems*. Hannover, Germany (December, 1994).

Coleman, D., Ash, D., Lowther, B. and Oman, P., 1994. Using Metrics to Evaluate Software System Maintainability. *Computer*. Vol. 27, No. 8:44-49.

Prieto-Diaz, R. and Freeman, P., 1987. Classifying Software for Reusability. *IEEE Software*. Vol. 4, No. 1:6-16.

Hubbard, D., 1999. The IT Measurement Inversion. *CIO Enterprize Magazine*.

Kennedy, K., Koelbel, C. and Schreiber, R., 2004. Defining and Measuring the Productivity of Programming Languages. *International Journal of High Performance Computing Applications*. Vol. 18, No. 4:441-448.

Koestler, A., 1967. *The Ghost in the Machine*. London: Arkana Books.

Koren, Y., 2006. General RMS Characteristics: Comparison with Dedicated and Flexible systems. *Reconfigurable Manufacturing Systems and Transformable Factories*. pp. 27-45. Springer, Berlin Heidelberg.

Koren, Y. and Shpitalni, M., 2010. Design of Reconfigurable Manufacturing Systems. *Journal of Manufacturing Systems*. Vol. 29, pp. 130-141.

Krueger, C.W., 1992. Software Reuse. *ACM Computing Surveys (CSUR)*. Vol. 24, No. 2:131-183.

Kruger, K. and Basson, A.H., 2017 (a). Erlang-Based Control Implementation for a Holonic Manufacturing Cell. *International Journal of Computer Integrated Manufacturing*. Vol. 30, No. 6:641-652.

Kruger, K. and Basson, A.H., 2017 (b). Comparison of Multi-Agent System and Erlang based Control Implementations for a Holonic Manufacturing Cell. *To be submitted to the International Journal of Computer Integrated Manufacturing, January 2018.*

Leitao, P., 2004. *An Agile and Adaptive Holonic Architecture for Manufacturing Control*. Ph.D. Dissertation. University of Porto, Portugal.

Leitao, P. and Restivo, F.J., 2006. ADACOR: A Holonic Architecture for Agile and Adaptive Manufacturing Control. *Computers in 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 – Part C: Applications And Reviews*. Vol. 38, No. 5:699-709.

Le Traon, Y., Ouabdesselam, F., Robach, C. and Baudry, B., 2003. From Diagnosis to Diagnosability: Axiomatization, Measurement and Application. *Journal of Systems and Software*. Vol. 65, No. 1:31-50.

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.

McCabe, T.J., 1976. A Complexity Measure. *IEEE Transactions on Software Engineering*. Vol. 4, pp. 308-320.

Microsoft TechNet, s.a. *Task Manager*. Available: https://technet.microsoft.com/en-us/library (30 August 2017)

Microsoft TechNet, s.a. *Overview of Performance Monitor*. Available: https://technet.microsoft.com/en-us/library/cc749154 (30 August 2017)

Monostori, L., Kádár, B., Bauernhansl, T., Kondoh, S., Kumara, S., Reinhart, G., Sauer, O., Schuh, G., Sihn, W. and Ueda, K., 2016. Cyber-Physical Systems in Manufacturing. *CIRP Annals - Manufacturing Technology*. Vol 65, pp 621–641.

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

Trendowicz, A. and Münch, J., 2009. Factors Influencing Software Development Productivity: State-of-the-Art and Industrial Experiences. *Advances in Computers*. Vol. 77, pp.185-241.

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, pp. 255–274.

Voas, J.M. and Miller, K.W., 1995. Software Testability: The New Verification. *IEEE Software*. Vol. 12, No. 3:17-28.

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

Weyuker, E.J., 1988. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*. Vol. 14, No. 9:1357-1365.

Wiendahl, H.P., ElMaraghy, H.A., Nyhuis, P., Zah, M.F., Wiendahl, H.H., Duffie, N. and Brieke, M., 2007. Changeable Manufacturing: Classification, Design and Operation. *Annals of CIRP*. Vol. 56, No. 2: 783 – 809.

6.2. Comparison of Multi-Agent System and Erlang Holonic Control Implementations for a Manufacturing Cell

Karel Kruger [a,*] and Anton H Basson [a]
[a] Dept of Mechanical and Mechatronic Eng, Stellenbosch Univ, Stellenbosch 7600, South Africa
[*] Corresponding author. Tel: +27 21 808 4258; Email: kkruger@sun.ac.za

Abstract

This paper presents a comparison between two holonic control implementations, using the Erlang programming language and Java Agent Development (JADE) framework multi-agent system, respectively. Erlang exhibits several characteristics that make it suitable – and even potentially beneficial – for holonic control, while JADE multi-agent systems have become the *status quo* in holonic systems research. The comparison is done using both quantitative and qualitative performance measures, which are specifically formulated to focus on the implementation of holonic control. The results show that the Erlang implementation is inherently capable of most of the functionality offered by the JADE implementation, while even exhibiting some advantages over its counterpart. The comparison indicates that Erlang is very well suited for implementing holonic control and warrants further exploration and development.

Keywords: Erlang/OTP; Multi-agent systems; Holonic manufacturing system (HMS); Reconfigurable manufacturing system (RMS)

6.2.1. Introduction

Holonic systems have become a popular approach for addressing the challenges of modern manufacturing systems: short lead times for the introduction of new products into the system; producing a larger number of product variants and handling fluctuating production volumes (Bi *et al.*, 2008). The term *holon* comes from the Greek words “holos” (meaning “the whole”) and “on” (meaning “the particle”) (Koestler, 1967). Holons are “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 systems, holons are autonomous and cooperative building blocks for transforming, transporting, storing or validating the information of physical objects. A Holonic Manufacturing System (HMS) is a system of holons that can cooperate to integrate the entire range of manufacturing activities (Paolucci and Sacile, 2005).

The holonic systems approach to manufacturing holds many advantages: holonic systems are resilient to disturbances and adaptable in response to faults (Vyatkin, 2007); have the ability to organise production activities in a way that they meet the requirements of scalability, robustness and fault tolerance (Kotak *et al.*, 2003); and lead to reduced system complexity, reduced software development costs and improved maintainability and reliability (Scholz-Reiter and Freitag, 2007).

The application of the holonic concept to manufacturing control systems has been a popular field of research since the early 1990's – often utilised to enable control reconfiguration in Reconfigurable Manufacturing Systems (RMSs). The RMS concept aims to produce manufacturing systems that can switch between members of a family of products, through the addition or removal of functional elements (hardware or software), with minimal delay and effort (Martinsen, 2007; Vyatkin, 2007). RMSs can rapidly adjust the production capacity and functionality in response to sudden changes, by reconfiguring hardware and control resources (Bi *et al.*, 2008; Bi *et al.*, 2007). Koren and Shpitalni (2010) characterise RMSs by: modularity of system components, integrability with other technologies, convertibility to other products, diagnosability of system errors, customizability for specific applications and scalability of system capacity.

Sharing some of RMSs' properties, Cyber-Physical Production Systems (CPPSs) have recently become a major focus. The three main characteristics of CPPSs are (Monostori *et al.*, 2016): "Intelligence (smartness), i.e. the elements are able to acquiring information from their surroundings and act autonomously; Connectedness, i.e. the ability to set up and use connections to the other elements of the system – including human beings – for cooperation and collaboration, and to the knowledge and services available on the Internet; and Responsiveness towards internal and external changes. Holonic systems share many of these characteristics with CPSSs.

Several experimental implementations of holonic control have been reported, such as Leitao and Restivo (2006) and Giret and Botti (2009). The most popular approach has been Multi-Agent Systems (MASs), which has become the *status quo* in holonic control implementation. The main motivation for this approach is the similarities between holons and software agents – both must exhibit autonomy and provide interfaces to facilitate cooperation.

This paper aims to evaluate, through a comparison, an alternative to MASs. The implementation is based on the Erlang programming language – a concurrent, functional programming language that was developed for programming concurrent, scalable and distributed systems. Erlang employs many lightweight processes to work concurrently, while distributed over many devices. The process model of Erlang facilitates processes that are strongly isolated, do not share memory and only interact through the exchange of messages (Armstrong, 2003). The Erlang programming environment is supplemented by the Open Telecommunications Platform (OTP) - a set of robust Erlang libraries and design principles providing middle-ware to develop Erlang systems (Anonymous, s.a. (a); Logan *et al.*, 2011).

The evaluation of holonic systems, including holonic control of RMSs, has proven to be a challenging task. Several studies and developments have generated evaluation criteria, with the formulations varying in focus and perspective (e.g. Christensen (1994), Koren and Shpitalni (2010) and Leitao and Restivo (2008)). In alignment with the objective of the presented research, this paper will make use of the evaluation criteria formulated by Kruger and Basson (2017 (c)), which is focussed on the implementation phase of holonic control.

This paper presents the methodology that was followed for the proposed comparison (section 6.2.2), briefly describes the implementations (section 6.2.3) and introduces the case study that is used as context (section 6.2.4). Section 6.2.5 provides an overview of the evaluation criteria and sections 6.2.6 and 6.2.7 perform the comparison according to the set of performance measures. Finally, the results of the comparison are discussed and the findings are presented in section 6.2.8.

6.2.2. Methodology

This paper presents a comparison of two different holonic control implementations based on the same reference architecture. One implementation is done using Erlang/OTP and the other is done using a MAS. The MAS is developed using the Java Agent Development (JADE) framework, which is middleware that facilitates the development of agent-based systems (Bellifemine *et al.*, 2007). From here on, the *programming language* of the implementations will refer either to Erlang with OTP, or MAS developed with JADE.

To perform a comparison of the two implementations is a challenging task – the implementations are different in not just the programming language, but also in programming paradigm (imperative and functional). While several studies have attempted such comparisons (e.g. Harrison *et al.* (1996), Prechelt (2000) and Cesarini *et al.* (2008)), assessments based on generic, objective and quantitative measures are hard to come by. Aiming to avoid this treacherous terrain, the comparison presented in this paper has a specific focus: the suitability of the Erlang programming language as a *tool* for implementing holonic control. The comparison thus pays less attention to the philosophical and semantic differences between the programming languages, and rather compares the provisions of each programming language to facilitate the implementation of holonic control. This methodology is similar to that adopted by Chirn and McFarlane (2005) in evaluating the effectiveness of a holonic system design.

The implementation of the same architecture in the two programming languages forms the basis for the comparison. The PROSA reference architecture (described in section 6.2.3.1) was used as the foundation for the development of both the Erlang and MAS holonic control implementations (described in section 6.2.3.2 and section 6.2.3.3 respectively). The use of a common reference architecture allowed for comparable functionality in the two implementations – the similarity was verified through a series of verification experiments, as presented in section 6.2.5.1.

For the implementations, the software was developed according to common practices – i.e. libraries provided with the software were used as far as possible, and the development followed the principles outlined in literature (Logan *et al.* (2011), Armstrong (2007) and Anonymous (s.a. (b)) for Erlang and Bellifemine *et al.* (2007) for JADE).

To improve the reliability and validity of the proposed comparison, the same developer performed the two implementations. The premise of implementing a common reference architecture relies on consistency in the developer's understanding and interpretation of the architecture. Additionally, even though the code is significantly different, the developer follows a similar approach in each implementation. The developer possesses the following relevant expertise and experience:

- Undergraduate degree in mechatronic engineering.
- Master's degree in mechatronic engineering, of which the thesis focussed on the development and evaluation of two holonic control implementations – one being a MAS developed in JADE and the other a IEC61499 application using Function Block Development Kit (Vyatkin, 2007) – further details can be found in Kruger and Basson (2013).
- Online course on Erlang programming (prior to which the developer was unfamiliar with Erlang programming).

In line with the objective of this research, the evaluation criteria is set up from the perspective of the developers and consumers of holonic control implementations, as opposed to that of computer scientists. The performance measures are thus derived from the requirements for holonic manufacturing systems and the evaluation aims to emphasise the extent to which each implementation satisfy these requirements. The evaluation criteria are discussed in section 6.2.5.3.

Several aspects of the comparison involve impressions, experiences and philosophies, which are not suited to quantification, leading to criteria comprised of both quantitative and qualitative performance measures (sections 6.2.6 and 6.2.7). Even though the evaluation is inherently subjective, the comparison strives to provide an unbiased reflection of the suitability of Erlang for holonic control implementation – this is enforced through reference to experimental data, examples from code and findings from literature, as far as possible.

6.2.3. Holonic Control Implementations

6.2.3.1. Holonic Architecture

The advantages of holonic control are largely provided by the holonic system's architecture. Several reference architectures, which specify the mapping of manufacturing resources and information to holons and to structure the holarchy, have been proposed (e.g. Chirn and McFarlane (2000) and Leitao and Restivo (2006)), but the most prominent is the Product-Resource-Order-Staff Architecture (PROSA) – developed by Van Brussel *et al.* (1998).

PROSA defines four holon classes: Product, Resource, Order and Staff. The first three classes of holons can be classified as basic holons, because, respectively, they represent three independent manufacturing concerns: product-related technological aspects (Product holons), resource aspects (Resource holons) and logistical aspects (Order holons).

The basic holons interact with each other by means of knowledge exchange, as is shown in Figure 33. Resource holons query the Product holons for the process requirements, of a given product type, pertaining to the specific tasks that the Resource holons can perform. The Order and Product holons exchange production knowledge related to the Resource holon services that are required for producing a product. The Order and Resource holons exchange process execution knowledge, which is the information regarding the progress of executing processes on resources.

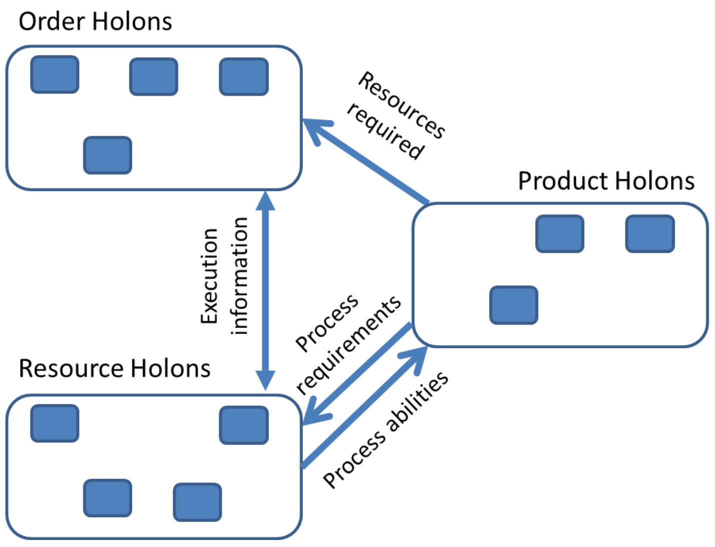


Figure 33: Knowledge exchange between the PROSA holons

Staff holons are considered to be special holons as they 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 holonic characteristics of PROSA contribute to the different aspects of reconfigurability mentioned in section 6.2.1. The ability to decouple the control algorithm from the system structure, and the logistical aspects from the technical aspects, aids integrability and modularity. Modularity is also provided by the similarity that is shared by holons of the same type.

6.2.3.2. Erlang Implementation

Details of the Erlang-based holonic control implementation are given by Kruger and Basson (2017 (a)). Each PROSA holon comprises a number of Erlang processes in the control implementation. The implementation makes use of the generic OTP behaviours – notably those for supervision, finite state machines and Transmission Control Protocol (TCP) communication. The process model of Erlang is used to incorporate a high degree of concurrency in the control implementation, with message passing between processes to share data. The *record* data type in Erlang was used to develop a custom communication ontology and protocol for the implementation.

6.2.3.3. MAS Implementation

The MAS control implementation is described in Kruger and Basson (2017 (b)). The holons of PROSA are represented as software agents in the control level of the manufacturing system. The implementation is constructed using JADE, with standard behaviours for functionality and communication being used as far as possible. A combination of standard communication protocols are used, along with an eXtensible Markup Language (XML) ontology, to achieve the desired interaction between the agents of the MAS.

6.2.4. Case Study

The case study used for the comparison of the implementations is a manufacturing cell for the assembly and quality assurance of electrical circuit breakers. The layout of the cell is shown in Figure 34. The cell consists of the following workstations:

- Manual assembly station – the sub-components of circuit breakers are assembled and placed on empty carriers on the conveyor.
- Inspection station – a machine vision inspection is performed on the circuit breakers as the carriers are moved by the conveyor.
- Electrical test station – circuit breakers are picked up by a robot and placed into testing machines. The testing machines perform the necessary performance and safety tests on every breaker. When the testing is completed for a breaker, it is removed from the testing machine by the robot and placed on an empty carrier on the conveyor.
- Riveting station – the casings of the circuit breakers are manually riveted shut.
- Removal station – the completed circuit breakers are removed from carriers. The breakers are then moved to the next cell for packaging.

As part of a reconfiguration experiment (presented in section 6.2.5.2), an additional workstation is added to the manufacturing cell – the stacking station. At this station, multiple circuit breakers are stacked to produce multi-pole circuit breakers. The breakers are removed, stacked and placed on empty carriers by a robot.

The conveyor moves product carriers between the various workstations. The conveyor is equipped with stop gates and lifting stations at every workstation. The carriers are fitted with Radio Frequency Identification (RFID) tags and RFID readers are placed at multiple positions along the conveyor, to provide feedback of carrier location.

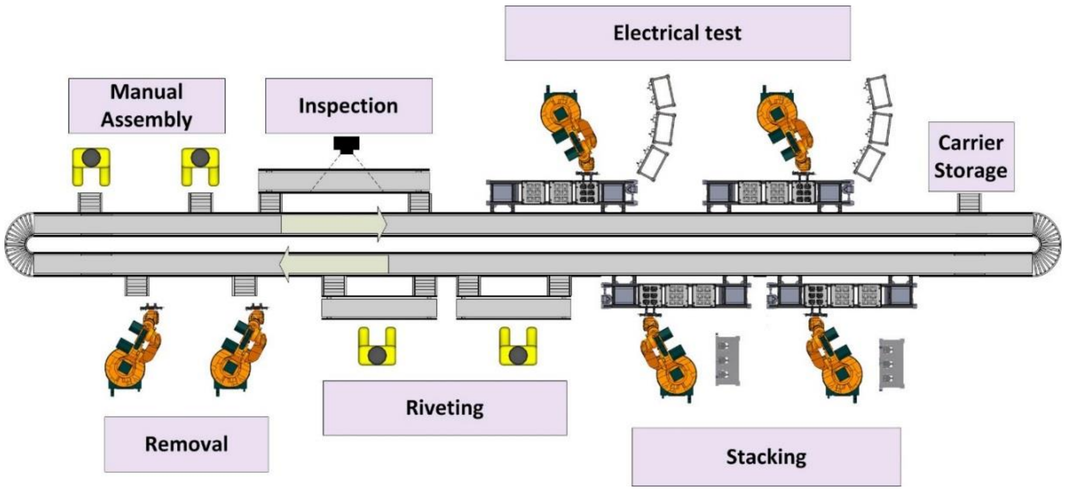


Figure 34: Layout of the electrical circuit breaker assembly and quality assurance cell.

6.2.5. Evaluation Overview

An overview of the evaluation of Erlang for the implementation of holonic control is presented in this section. As discussed in section 6.2.2, the suitability of Erlang for the implementation of holonic control is evaluated in comparison with a MAS implementation. Firstly, this section presents the verification that both implementations exhibit similar functionality and performance. Thereafter, the design of a reconfiguration experiment is described and an overview of the evaluation criteria is presented.

6.2.5.1. Verification Experiments

The verification experiments aimed to prove that the implementations, as embodiments of the same reference architecture, exhibit similar functionality and performance. This verification was done through experiments that emphasise the functionality of the holonic architecture, negating the influence of the respective programming languages. Performance measures, as computed from the results of the experiments, are compared to verify the intended similarity.

The experiments were performed using simulations of the manufacturing cell described in section 6.2.4. The experiments involved the simulation of two production scenarios for each implementation - the simulated production of ten orders (of the same product type) with a cell configuration that:

1. does not include redundant workstations (i.e. only one workstation was active for the electrical testing, riveting and removal tasks, respectively).
2. includes active redundant workstations for the electrical testing, riveting and removal tasks.

The first experiment aimed to exhibit the basic functionality as defined by the implemented holonic architecture – for each order, the required service-providing resource holons must be identified and booked, and the execution of each service must be triggered. With no redundant resources, the production sequence is fixed.

The second experiment required the additional functionality of selecting the best service-providing resource holon according to received proposals. The additional redundant resources introduce emergent behaviour in the holonic implementations and thus unpredictability in the production sequence.

The most obvious method of verifying the similarity in functionality was to simply observe each control implementation during the experimental production scenarios. While it can be confirmed that the implementations performed similar functions and successfully executed the simulated production, quantitative measures are more convincing. Therefore, two quantitative performance measures were extracted from the simulated production scenarios:

- *Production throughput* refers to the rate by which orders are completed, calculated as total number of completed orders over the total production time.
- *Resource utilization* is measured as the percentage of the total production time that a resource is active (i.e. performing a specific task/operation on an order).

Both production throughput and resource utilization have been used as quantitative performance measures in previous studies on manufacturing system control (e.g. Leitao (2004) and Bussman and Sieverding (2001)). Production throughput gives an indication of the performance of the overall basic functionality, while resource utilization provides an indication of the performance of scheduling and executing services. It is expected that two implementations of the same reference architecture should achieve similar results for the two performance measures.

Table 4 summarizes the results obtained from the verification experiments. For both experiments, the results show a close correlation in the performance of the two implementations. The results serve as proof that the Erlang and MAS control implementations exhibit similar functionality, allowing for a fair comparison.

Table 4: Results of verification experiments.

	Experiment 1		**Experiment 2**	
	Resource utilization (%)		**Resource utilization (%)**	
Resources	**Erlang**	**MAS**	**Erlang**	**MAS**
Feeder station	19.3	20.1	24.6	23.8
Inspection station	11.6	12.0	14.7	14.2
Electrical test station	59.8	60.9	38.7	36.7
Electrical test station (#2)	n/a	n/a	36.7	35.7
Riveting station	38.5	40.1	25.4	24.9
Riveting station (#2)	n/a	n/a	26.2	24.9
Removal station	15.4	16.1	9.8	10.6
Removal station (#2)	n/a	n/a	9.8	9.5
	Throughput (parts/min)		**Throughput (parts/min)**	
	2.3	2.4	2.9	2.8

6.2.5.2. Reconfiguration Experiment

A reconfiguration experiment was performed to obtain a number of the quantitative performance parameters for the reconfiguration and development of the two implementations. The experiment involved the addition of a new Resource holon, capable of stacking multiple circuit breakers, to the system described in section 6.2.4.

The reconfiguration experiment required changes to the Product and Order holons to incorporate the newly added Stacking holon. Two Product holons were added – one for producing single-pole circuit breakers that can be stacked to form multiple-pole breakers, and one for the production of stacked three-pole breakers. The general messaging functionality for Order holons needed to be updated to incorporate the exchange of task-specific information with the Stacking holon. After the required alterations, the performances of the implementations were verified.

6.2.5.3. Evaluation Criteria

The comparison makes use of the evaluation criteria formulated by Kruger and Basson (2017 (c)) as shown in Table 5.

The criteria are based on the desirable characteristics of manufacturing systems. Availability, as a measure of reliability, is widely considered to be an important characteristic in manufacturing contexts. Since alternative implementations of the same holonic architecture is considered here, development productivity is also included as a desirable characteristic. Supportability is related to both availability and development productivity, but is listed explicitly because the choice of implementation for the controller can have a significant influence in the supportability.

Seven requirements were derived from the desirable characteristics, as shown in Table 5. The requirements often affect more than one characteristic and therefore no explicit linkages are attempted in Table 5. A set of quantitative and qualitative performance measures are also presented in Table 5. The performance measures are indicative of one or more of the requirements for holonic control implementations.

Table 5: Relationships between requirements and performance measures.

			Characteristics						
			Availability Supportability Development Productivity						
			Requirements						
			Reconfigurability	Robustness	Maintainability	Controller requirements	Complexity	Verification	Reusability
Performance measures	Quantitative	Reconfiguration time	*				*	*	*
		Development time					*	*	*
		Code complexity			*		*		
		Code extension rate	*		*		*		
		Code re-use rate	*		*		*		*
		Computational resource requirements				*			
	Qualitative	Modularity	*		*			*	*
		Integrability	*						*
		Diagnosability	*	*	*			*	
		Convertibility	*		*				
		Fault tolerance		*					
		Distributability				*			
		Developer training requirements			*		*	*	

6.2.6. Quantitative Performance Measures

This section presents the results of a comparison of MAS and Erlang implementations in terms of the quantitative performance measures given in Table 5. The discussion of the implications of the results of each comparison is deferred to section 6.2.8 since the performance measures should be considered together to draw sensible conclusions.

6.2.6.1. Reconfiguration Time

The reconfiguration times measured in the reconfiguration experiment described in section 6.2.5.2 are presented in Table 6. The reconfiguration time measurements do not include the development time for the Stacking holon, which is compared in section 6.2.6.2.

The results show that to complete the reconfiguration Erlang required 67% of the time required by the MAS implementation. The difference in reconfiguration time occurred during the verification activity. This can attributed to the ease by which functions can be tested individually and reliably, since Erlang functions are not affected by program state. For the MAS implementation, it is difficult to test functionality at a functional granularity finer than that of the agent encapsulation. Even in that case, the functionality is still subject to the state of the program.

Table 6: Reconfiguration and development time measurements

	MAS	Erlang
Reconfiguration time (hours)	2.7	1.8
Development time (hours)	3	2.5

6.2.6.2. Development Time

During the reconfiguration experiment (section 6.2.5.2), the time was measured for the development of a new Stacking holon. In both implementations, the generic components (in this case, the encapsulations for holon communication and agenda management) could be used as is, with development only required for the execution components. The Stacking holon performs two services:

1. *Remove_to_stack* – single-pole circuit breakers are removed from the conveyor and placed in a buffer, from where they will be used to produce stacked three-pole breakers. This service resembles that of the Removal Resource holon – the code of the execution component of this holon could be reused and modified to perform the *remove_to_stack* service.
2. *Stacking* – three-pole breakers are assembled through the stacking of single-pole breakers that are stored in the buffer. With this service, the code from the execution component of the Feeding holon could be reused and modified to constitute the *stacking* service.

The time required to develop the Stacking holon in both implementations was measured – the measurements are shown in Table 6. The measurements indicate that the Erlang implementation required 83% of the development time required for the MAS implementation. As in the reconfiguration time evaluation, the difference in development time can be attributed to different mechanisms available for software verification in the two implementations.

6.2.6.3. Code Complexity

As motivated by Kruger and Basson (2017 (c)), *source lines of code* (SLOC) is used as measure of code complexity. Table 7 presents the SLOC count for the Order holon and Resource holon associated with the electrical test station (ETS Resource holon) in each control implementation. The Order holon was chosen since it is typical of a holon that requires considerable interaction with a variety of other holons, while the ETS Resource holon was chosen since a holonic controller will typically contain several Resource holons.

Table 7: Code complexity measurements.

	MAS	Erlang
Order holon	441	318
ETS Resource holon	352	175

The data shows that implementation in Erlang resulted in a SLOC reduction of 28% for the Order holon and 50% for the ETS resource holon. Considering the results reported in Nyström *et al.* (2007), where SLOC comparisons are made between Erlang and C++ implementations, this reduction in code is expected.

6.2.6.4. Code Extension Rate

Code extension rate is an index that represents the growth rate of the scale, and thus complexity, as an existing implementation is reconfigured to meet new functional requirements (Chirn and McFarlane, 2005). The development of the Stacking holon (as described for the reconfiguration experiment in section 6.2.5.2) was used for the measurement of code extension rates for the two implementations. The initial configuration was based on the execution components from the Feeding and Removal Resource holons (since these components were combined and modified to construct the execution behaviour of the Stacking holon), as well as the generic Resource holon communication and agenda management components.

Table 8 shows the calculated code extension rates for the implementations. The closer the code extension rate is to unity, the less growth in complexity occurred in converting from the initial to the final configuration – this indicates better system reconfigurability. The extension rates for the languages are similar, but the MAS implementation produced a marginally better result.

Table 8: Code extension rate measurements.

	MAS	Erlang
Initial configuration SLOC	429	228
Final configuration SLOC	480	275
Code extension rate	1.1	1.2

6.2.6.5. Code Reuse Rate

Software reusability is important for achieving high productivity (Kruger and Basson, 2017 (c)). *Code reuse rate* can be calculated in the event of reconfiguration and provides a measure of the percentage of source code in a new configuration that is reused from an initial configuration.

The development of the Stacking holon in the reconfiguration experiment (section 6.2.5.2) was based on the reuse and modification of existing software artefacts. Code was reused from the generic communication and agenda management components of Resource holons, and the execution components of the Feeding and

Removal Resource holons. Table 9 presents, for each implementation, the SLOC count for the final configuration, the count of SLOC that have been reused from the initial configuration and the calculated reuse rates.

The data shows that for the MAS implementation, 88% of the final configuration source code was reused (i.e. code from the previous configuration retained without modification). The Erlang implementation also showed good reusability, but achieved 11% lower reuse of code in the reconfiguration process.

Table 9: Code reuse rate measurements.

	MAS	**Erlang**
Final configuration SLOC	480	275
Final configuration SLOC (reused)	424	211
Code reuse rate	0.88	0.77

6.2.6.6. Computational Resource Requirements

Manufacturing automation controllers are usually limited in their computational and memory capacity. It is therefore important that the implementation allows for the efficient exploitation of these controllers, within their limits of use.

The data for the computational resource requirements was obtained by performing a simulated production experiment. For both control implementations, the production of ten orders was simulated. The measurements were started as the production was triggered, thus excluding the start-up processes of the implementations from the measurement. The data was obtained during the experiment by using the Performance Monitor and Task Manager applications of the Windows operating system.

Table 10 presents the data obtained from the Performance Monitor application – the total operating system (OS) thread count for each implementation and the RAM used in each case. The results show that the Erlang implementation consumes far less memory resources than the MAS. On average, the Erlang implementation utilizes almost three times fewer OS threads and consumes about five times less memory than the MAS implementation. It should be noted that in Erlang, the use of OS threads are not typically controlled by the developer and the scheduling of Erlang processes to OS threads is done automatically by the Erlang virtual machine (Logan *et al.*, 2011). The number of concurrent Erlang processes far exceed the number of OS threads utilized.

Table 10: Thread count, memory usage and CPU time consumption measurements.

		MAS	Erlang
Thread count	**Min**	56	28
	Max	90	28
	Avg	77	28
RAM usage (MB)	**Min**	29.4	24
	Max	152.9	24.3
	Avg	123.3	24.1
CPU time consumption	**Total (s)**	7	16
	% of total CPU time	3.3	7.8

The Task Manager application was used to measure the CPU time used by each implementation. The results in Table 10 show that the MAS implementation used 7 seconds of CPU time (3.3% of the total CPU time) for the duration of the experiment, while 16 seconds (7.8% of the total CPU time) were used by the Erlang implementation. The better performance by the MAS implementation is because the implementation has fewer concurrent processes than the Erlang implementation. Moreover, the use of behaviours with execution blocking means that agent threads operate in an idle state for a large fraction of the time.

6.2.7. Qualitative Performance Measures

This section uses the qualitative performance measures given in Table 5 to compare MAS and Erlang implementations. As with the qualitative measures, the discussion of the implications of the results of each comparison is deferred to section 6.2.8, to allow the performance measures to be considered together.

6.2.7.1. Modularity

Following from Kruger and Basson (2017 (c)), the modularity of software is determined through three specifications – that of architecture, module interaction and testing. These three specifications form the basis for comparison of the Erlang and MAS implementations.

Architecture

The JADE MAS is built using the Java programming language. Java incorporates the JADE framework as a Java archive (JAR) file. Java program code is contained in classes that may contain definitions and methods, and collections of classes can be encapsulated in packages. JADE provides the *Agent* class, which encapsulates all the basic functionality needed to construct the MAS. The Agent class utilizes another special class – *behaviours* – to encapsulate functionality that can be combined to constitute the behaviour of agents.

In Erlang, *modules* are used as the containers of program code. All program code in Erlang is structured as *functions*. Erlang provides standard libraries that include many modules containing useful functions. Similarly, OTP is a set of robust libraries that can be used in the structuring and execution of processes. Erlang also provides a mechanism for encapsulating definitions and functions to be utilized by several modules – *header files*.

Module Interaction
Java classes and methods can be used by other classes in the system – this exposure is defined by access modifiers. In Erlang, the functions that are contained in modules can be exported – this allows for the use of these functions in other modules. As mentioned above, the definitions and functions contained in header files can be included in modules.

Testing
Individual Java methods can be tested using unit tests. JADE offers several tools for verifying the behaviour of the MAS or individual agents, such as the *Sniffer*, *Introspector* and *Dummy* agents (details are presented in Bellifemine *et al.* (2007)).

Every function in Erlang that is exported from a module (i.e. the function can be called from other modules) can be individually tested. The function can be called, with a set of input arguments, from a testing process (often a *shell* process, where the developer gives the inputs). Erlang also provides a mechanism for verifying the behaviour of processes – the *observer* application can, among other things (Anonymous, s.a. (c)), trace messages received and sent by processes, and provide information on the function that is executed by a process at any given time. The functionality and use of the Observer application will be highlighted in following sections.

The functional programming of Erlang means that processes have no state or side effects – this ensures that the output of the function, to a set of input arguments, is reliable and repeatable. This is, however, not true for Java programming, where the output returned by methods can be affected by the state of the class exposing the method.

6.2.7.2. Integrability
The first point of comparison focusses on the interfaces provided by the implementations to incorporate software components developed in other programming languages. Thereafter the support for communication protocols is compared.

Integration of foreign code
With JADE, Java provides the Java Native Interface (JNI) (Anonymous, s.a. (d)) – a native programming interface that allows Java code to interoperate with applications written in other programming languages, such as C, C++ and Assembler. This interface is useful for integrating legacy systems, supplementing the functionality offered by Java or improving performance.

Logan *et al.* (2011) explain how the Erlang message-passing paradigm is extended to interface with code written in other languages. Foreign code can be represented in an Erlang application as a process-like object, called a *port*. The Erlang processes can then pass messages to the foreign code via the port. For *plain ports*, the foreign code (of any programming language) runs in a separate OS process and communicates to the port using the standard inputs/outputs, with all data passed as a byte stream. Alternatively, with *linked-in port drivers*, the foreign code runs in the same OS process space as the Erlang virtual machine. When working with distributed Erlang nodes, C and Java programs can be made to masquerade as Erlang nodes. This functionality is contained in two libraries – *Erl_Interface* for C and *Jinterface* for Java.

The above-mentioned mechanisms to integrate foreign in each application code can also be used to access industrial communication protocols, e.g. through industrial Ethernet. The support in each implementation for common PC based communication is discussed in the following section.

Support for communication protocols
The Java platform includes the *net* package, which provides classes for implementing networking applications. The classes provide the functionality to facilitate socket communication over networks and supports both TCP/IP and UDP (Anonymous, s.a. (e)). Libraries are available for TCP/IP and UDP (Anonymous, s.a. (f)) network communication in Erlang.

For basic text interface implementations, it is often appealing to use XML for structuring the text information. XMErL (Anonymous, s.a. (g)) is an Erlang library for XML functions. Several libraries for building and parsing XML are available for Java.

For serial communication, Java provides the *JavaComm* serial communication API – however, it is not available for all Java platforms and support has been withdrawn for use with Windows OS (Anonymous, s.a. (h)). Alternatively, the free-software libraries *RXTX* (Anonymous, s.a. (i)) and *jSerialComm* (Anonymous, s.a. (j)) can be used. For serial communication in Erlang the *gen_serial* library (Anonymous, s.a. (k)) allows for the use of standard serial ports, on both Windows and UNIX platforms.

6.2.7.3. Diagnosability
Errors in a holonic control implementation can occur within the execution of a holon, or in the interaction between holons or holon components. The time and effort required to diagnose an error depends on the availability of information. It is therefore necessary for the developer to have access to information regarding the execution of each holon and the interaction between holons. The provisions in the Erlang and MAS implementations for diagnosing such errors are considered in this section.

Both implementations provide mechanisms to gather information from and test the interaction of holons – JADE includes the *Sniffer* agent and Erlang provides the *Observer* application. These tools provide the functionality to trace the communication between holons. The trace can provide information on the senders and receivers of messages, the message type and message content. To test the interactions, JADE includes the *Dummy* agent tool and in Erlang, *shell* processes can be used to this effect. Dummy agents can be used to construct messages to be sent to other agents and to receive messages, as these agents react to the dummy messages. Dummy agents provide a graphical user interface for easy construction of messages to be sent and viewing of received messages. Shell processes in Erlang are “interactive” processes that allow the developer to input expressions. The developer can construct messages in the shell process and send them to any other active process. Every process in Erlang has a mailbox – the shell process can thus also receive messages that can be viewed by the developer.

To diagnose errors occurring within a holon's execution, both implementations also provide tools to obtain information on the execution of a holon. JADE provides the *Introspector* agent, which can be used to debug the behaviour of a single agent. The Introspector agent allows the developer to monitor the queue of scheduled behaviours and control their execution (e.g. a behaviour can be executed step by step) (Bellifemine *et al.*, 2007). Similarly, the Observer application in Erlang can be used to monitor Erlang processes by tracing the execution of functions by a process.

The Sniffer tool in JADE is good for verifying the interaction between agents. However, the verification of holon execution is not as simple. The Introspector agent provides some detail of the execution of an agent, but often that is not enough to identify the reasons for or sources of detected bugs. Occasionally an agent receives a message, but does not react as expected (or does not react at all). In such cases, it might be required to use a tool such as the Java debugger, which is powerful but less user friendly.

Erlang’s Observer tool is not as user friendly as the JADE Sniffer, but includes the additional functionality to trace the execution of processes at different levels. Along with this tracing, the easy construction of test code to verify the behaviour of a process affords the developer freedom in the verification process. The increased modularity of the Erlang implementation further simplifies the verification process.

Finally, it is important to consider the capacity for error isolation – or, alternatively, the minimization of error propagation – in each implementation. Errors in software systems can propagate, resulting in failures in components other than where the error originated. This propagation can complicate the process of locating the source of errors and has a detrimental effect on the diagnosability of the system. It is in this respect that the process model of Erlang offers significant advantages, as is further discussed in section 6.2.7.5.

6.2.7.4. Convertibility

At the control level, convertibility refers to the transformation of the functionality of the existing system to meet new production requirements. The mechanisms provided in each application to make changes to the controller functionality are discussed in this section.

Making changes to the MAS implementation is simplified by a set of effective tools. JADE includes the Remote Monitoring Agent (RMA) – a graphical tool for the monitoring and manipulation of the running agent platform. The RMA facilitates interaction with the Agent Management System (AMS), which allows for control over the execution of agents. Among other functions, this allows the developer to stop a running agent and remove it from the system – the agent can then be launched again or a different agent can then be launched in its place (possibly with the same name).

Erlang modules can be constructed to allow for similar control over the execution of processes. Functions can be exported from modules to allow the developer (through a shell process) or a supervisor process to stop and start an executing process. As Erlang does not readily include a tool like the RMA of JADE, so this functionality must be implemented by the developer.

An interesting capability of Erlang is that it allows for hot code loading – i.e. the code to be executed can be changed while the system is running. A second, newer version of a module can be loaded and the transition to the new code will be made automatically (Armstrong, 2007). This functionality means that bug fixes, updates and upgrades to code can be introduced with no system downtime.

6.2.7.5. Fault Tolerance

The isolation, detection and handling of faults are critical for achieving fault tolerance in control implementations. The evaluation here concentrates on the functionality provided in the two implementations for each of these aspects.

Armstrong (2003) identifies the inability to isolate components as the main limitation of developing fault tolerant systems in many popular programming languages. Specifically considering Java, Czajkowski and Daynés (2001) argue that to run multiple Java applications safely on the same computer, each application should be run in its own Java Virtual Machine and in its own OS process – a scenario detrimental to efficiency, performance and scalability. In contrast, the provisions for error isolation in Erlang are present at the architectural level. The process model allows for the isolation of errors - processes, as the basic unit of abstraction, act as abstraction boundaries that limit the propagation of errors (Armstrong, 2003).

In the MAS implementation, exceptions are thrown when errors are detected. The onus lies on the developer to catch exceptions where necessary and handle them accordingly. Erlang provides similar functionality for the detection of errors. The problem with this method of detecting errors, and subsequently handling them, is that it provides only one opportunity for reaction – should the exception not be handled correctly, the process will fail.

Erlang thus provides additional functionality to improve fault tolerance, employing supervision hierarchies to detect and handle faults. The supervision behaviour is an important provision of OTP. Where *worker* processes execute specific tasks as required by the application, *supervisor* processes can be used to monitor the execution of workers. Supervisor processes can also monitor other supervisors, thus constructing supervision trees. Supervisor processes act as an error trapping "layer", which can monitor applications and restore it to a safe state in the event of an error (Armstrong, 2010). The supervisor behaviour allows for the implementation of strategies to handle the occurrence of errors in processes (details are presented in Logan *et al.* (2011)). This functionality is not readily available in JADE and would require implementation by the developer.

6.2.7.6. Distributability

Distributability is important for implementing decentralized control architectures. Some important provisions in the implementations for distributability are considered in this section.

Distributable architecture

A JADE platform is composed of agent *containers*. Containers are Java processes that maintain the execution space in which agents can exist - providing the JADE run-time and all the services needed for hosting and executing agents. Containers can be distributed over a network of controllers. JADE provides the infrastructure for communication between agents residing in different containers and also for agent mobility, allowing agents to move between containers.

In Erlang, nodes are the architectural provision for distribution. Nodes are instances of the Erlang VM that are configured for networking and a set of connected nodes are referred to as a *cluster*. Similar to the agents in JADE, the processes in Erlang can communicate and migrate between nodes in a cluster.

Communication in distribution

Achieving communication in distributed systems firstly involves the discovery of components that are distributed on a network. In order for control components to communicate, they must identify and locate the other components in the distributed system. Thereafter, the communication between these distributed components must be facilitated.

Every JADE platform has a *main* container, which acts as the bootstrap point for the platform. The main container provides functions to allow for the dynamic discovery of control components in a distributed MAS:

- Managing the container table, which holds the object references and addresses of all the containers in the platform.
- Managing the global agent descriptor table, which is the registry of all agents in the platform.
- Hosting the Agent Management System (AMS) and Directory Facilitator (DF) providing services to the entire platform.

In Erlang, the discovery of distributed nodes is done by the Erlang Port Mapper Daemon (EPMD) process. An EPMD process is automatically started on the machine when a node is started. When a local node wants to communicate with a remote node, the EPMD process on the local machine queries the EPMD process on the remote machine for the specified communication port. Erlang does not come with a standard service for dynamic resource discovery over a cluster, but the basic functionality can be easily developed or, for more advanced functions, an OTP application (see Logan (2010)) is available as an add-on.

In both implementations, the communication mechanism is unaffected by the distribution of the control components. The JADE platform provides a unique location-independent interface that abstracts the underlying communication infrastructure, allowing for transparent communication between agents that exist on different remote machines. Similar location transparency exists in Erlang. The processes on connected nodes can exchange messages by using the Process Identifiers (PIDs) – the node on which a process resides is embedded in its PID. The registered names of processes can also be used to address messages – these names are only registered on the residing node, thus messages must be sent as:

```
{RegisteredName, Node} ! Message.
```

Tools for distribution
For both MAS and Erlang, the tools provided for debugging and monitoring are equally useful for non-distributed and distributed implementations. The location transparency of the distributed control components mean that the functionality of the tools remains unaffected. In JADE, the Dummy and Sniffer agents can be used to test and monitor distributed control components communicating over connected machines – the same applies to the Observer application in Erlang. The AMS in JADE allows for the easy migration of agents between containers – a useful tool that is not included in the standard Erlang tools. Erlang, on the other hand, allows for the creation of remote shells on the local machine. These shell processes can provide an interface to the processes of remote nodes, simplifying and adding functionality to the testing of distributed implementations.

Portability
Both Java and Erlang run on virtual machines, which makes applications in these languages platform independent. The MAS and Erlang implementations are supported on the most prominent PC operating systems – Windows, Unix/Linux, and Mac OS X. There have also been efforts in both languages for enabling embedded applications on resource-limited microcontrollers (Brouwers *et al.* (2008) and Anderson and Bergström (2011)).

Standardization and guidelines
In both implementations, the development of code is guided by behaviours. The behaviours provided by JADE and OTP define a broad structure for implementing the functionality that holons must exhibit. This structure promotes uniformity in the software.

For communication, JADE adheres to the standards of FIPA. While adherence to these standards might limit the freedom of the developer, it allows for interoperability between MASs created by different developers. In standard Erlang, there is no guidelines concerning communication – developers have total freedom to implement the communication to fit their application. The lack of uniformity decreases the capacity for interoperability, but allow for increased customization to meet a given set of requirements.

6.2.7.7. Developer Training Requirements

The comparison between the MAS and Erlang implementations is here based on five tasks that must be performed by the developer: implement holon behaviour; implement inter-holon communication; implement external communication; implement concurrency and verify the functionality of the developed system.

Holon behaviour

Both implementations provide structures for constructing complex functionality. The JADE *behaviour* class provides several options for construction, like finite state machines, sequential or parallel execution or timer-based behaviours. Erlang/OTP provides two generic behaviours, i.e. servers and finite state machines, which can be customized to exhibit a desired functionality.

JADE behaviours are not pre-emptive and the control of their execution is left to the developer. Implementing complex functionality using behaviours can thus be difficult – especially when multiple behaviours are concurrently active in an agent. Developers often need to consult the source code of the JADE classes to understand the intended use of behaviours.

In Erlang/OTP, the behaviour classes are executed sequentially by processes – the logical flow of implemented behaviours is easier to predict and control. As with all Erlang functions, the functions within the OTP behaviours only have access to the data received as inputs. Considering a finite state machine implementation, the information describing the state of the behaviour must be passed from one function to another – this can become complicated and tedious when implementing complex behaviour.

It is natural to encounter challenges in both implementations, but the implications of such challenges must be compared. It is the experience of the authors that, due to the complexity of using behaviours, the construction of complex functionality is more challenging in the MAS implementation and requires more training and experience from the developer.

Inter-holon communication

The facilitation of inter-holon communication involves four main tasks: message construction; message sending; receiving messages and implementing communication protocols. Achieving these tasks in the two implementations is compared here.

Message construction. Messages in the MAS implementation are based on the Agent Content Language (ACL) and JADE provides the `ACLMessage` class with methods for message construction. The construction is done by assigning data to the defined data fields for a specific message instance. In the presented Erlang implementation, messages are constructed as *records*. Records have defined data fields to which values can be assigned.

Message sending. Both implementations provide a simple mechanism for sending messages. A constructed message can be sent to a recipient by using the `send(ACLMessage message)` method in JADE (the `ACLMessage` class requires that the receiver information be added to the recipient field). In Erlang, the message operator (`!`) provides this functionality – e.g. `process_id ! {sender, message}`.

Receiving messages. Both implementations use a `receive()` method/function to receive a message in a running agent/process. JADE requires the construction of message templates to handle messages, i.e. received messages are compared to predefined templates. Erlang uses pattern matching – a received message is compared to predefined patterns describing message structure and content.

Implementing communication protocols. JADE provides behaviour classes to facilitate communication protocols as defined by the FIPA standards. These behaviours are based on the finite state machine behaviour, with the state transitions determined by the exchange of messages. In the Erlang implementation, such protocols were implemented through customized OTP finite state machine behaviours.

From the above, and considering that both implementations are based on the exchange of messages, it is evident that Erlang and JADE strive to simplify the construction, exchange and handling of messages. The communication is easier to facilitate in the Erlang implementation – this is expected, since message passing is a critical aspect of the Erlang language. The formalization of communication is simplified by using the FIPA standards in the MAS implementation. It is the responsibility of the developer to construct such formalizations in the Erlang implementation – for this reason the developer of the Erlang implementation requires more experience and a clear definition of communication protocols.

External communication

The implementations used TCP communication to interact with the lower level controllers in the case study. In both cases, the utilized libraries provided functions for server and client functionality, maintaining socket connections and exchanging data over connected sockets. This communication was achieved with similar ease in the Erlang and MAS implementations.

Concurrency

Holonic control implementations assume concurrency at the holonic system level, but often some concurrency is desired within a holon. A common example, from the implementations presented in this paper, is for a holon to participate in synchronous network socket communication with a lower level controller, while remaining available to handle booking requests from other holons. A comparison shows significant differences in achieving concurrency in the Erlang and MAS implementations.

For the MAS, concurrency at system level is facilitated by the JADE AMS. The AMS ensures that every agent starts in a dedicated OS thread. To achieve concurrency within the execution of an agent, JADE provides the `ThreadedBehaviourFactory` class. This class provides a method to wrap a normal JADE behaviour into a *threaded* behaviour, allowing the behaviour to be executed in its own thread (Bellifemine *et al.*, 2007). Threaded behaviours should be used with care, as possible issues may arise concerning agent termination and synchronization when accessing resources.

As mentioned in section 6.2.3.2, each holon comprises a number of Erlang processes. In Erlang, all processes run concurrently and can be created either using the OTP Supervisor behaviour (wherein concurrent child processes are automatically created) or explicitly using the `spawn(Module, Function, Arguments)` function. Since processes have no shared memory and information can only be shared through message passing, synchronization issues are negated.

Erlang was designed with concurrency as a key requirement – it is thus much easier and safer to achieve concurrency in Erlang than in a JADE MAS. Of course, this may tempt developers to overuse concurrency in the software design, but the lightweight processes of Erlang negate the potential pitfalls (especially concerning performance and computational requirements).

Verification of functionality
A comparison of the strategies and tools for verification of the two implementations is presented in sections 6.2.7.1 and 6.2.7.3. The comparison here will consider the previous discussions, focussing on the required capabilities of the developer to verify the execution and interaction of holons in the control implementations.

The verification of holon interaction is simplified for the developer by the Sniffer tool, available for the MAS implementation. For verification of the holon execution, the freedom to adjust the level of detail and easily creating test code (or supplying specific inputs) aid the developer in the Erlang implementation. Furthermore, the functional nature of Erlang allows the developer to perform verification with a higher level of granularity – i.e. smaller components of the software can be verified, and with much greater ease, than in the MAS implementation.

6.2.8. Comparison
This section discusses of the implications of the performance measures, as presented in sections 6.2.6 and 6.2.7, on the requirements for holonic manufacturing systems as presented in Table 5. Thereafter the discussion is extended to the desired characteristics to be exhibited by holonic manufacturing systems.

6.2.8.1. Reconfigurability
The performance of a reconfiguration experiment (section 6.2.5.2) provided data for reconfiguration time, code extension rate and code reuse rate performance measures. The experiment produced interesting results – while the Erlang implementation required less time to perform the reconfiguration, it showed a greater growth in complexity with less reuse.

Two reasons for these contrasting results are:

- The impact of the difficulty and effort involved in verifying the reconfigured code is ignored in the code extension and reuse rate measures. As mentioned in section 6.2.7.3, the verification of code was experienced to be more challenging for the MAS implementation than its Erlang counterpart.
- The difference in the number of SLOC in each implementation prior to the reconfiguration should also be taken into account. It is shown in section 6.2.6.3 that the Erlang implementation initially had significantly fewer SLOC – additional SLOC added during the experiment will thus have a greater impact on the calculation of the code extension and reuse rates.

Koren and Shpitalni (2010) lists modularity, integrability, convertibility and diagnosability as key characteristics for reconfigurability. Evaluating and comparing these qualitative performance measures shows that the Erlang implementation provides similar integration and conversion mechanisms to the MAS implementation. The Erlang process model, however, affords some advantages over the MAS implementation concerning modularity and diagnosability.

The comparison shows that the Erlang implementation has very good reconfigurability properties – arguably even more so than the MAS implementation.

6.2.8.2. Robustness

The evaluation of diagnosability and fault tolerance – two qualitative performance measures – are used to illustrate the robustness of the control implementations. Sections 6.2.7.3 and 6.2.7.5 indicate that the implementations provide similar tools for obtaining execution and communication information and supplying test inputs to the system, but the *shell* process in the Erlang implementation provides extra freedom and flexibility to the developer.

Where the Erlang implementation poses the greatest advantage is with its inherent fault tolerance. The process model of Erlang decreases the propagation of errors through the system. As Erlang was designed with robustness as a key requirement, it is not surprising that it out performs a standard MAS implementation.

6.2.8.3. Maintainability

Code complexity, code extension and reuse rates are considered to be indicative of the maintainability property of the control implementations. The effect of the code extension and reuse rate results, as obtained from the reconfiguration experiment, have been discussed earlier in this section. The measure for code complexity, however, is obtained from the initial configuration of the implementation source code. The results indicate that the Erlang implementation is less complex than the MAS implementation.

Considering qualitative measures, Table 5 indicates that modularity, convertibility, diagnosability and the developer requirements have a significant influence on the maintainability of a control implementation. Modularity allows for maintenance to specific system components without having to consider the remainder of the system, while convertibility facilitates the adaption of the functionality of the control implementation to meet new requirements. Diagnosability makes it easier to identify and locate components in need of maintenance. The maintenance must be performed by staff – the expertise and experience required by these staff members are indicated by the developer requirements for the implementation. The convertibility of the implementations are similar. The modularity and diagnosability advantages of the Erlang implementation have been highlighted earlier in the discussion. The comparison of developer requirements also showed the benefits of the Erlang implementation, specifically to the verification process that usually accompanies any maintenance activity.

The comparisons therefore show that an Erlang implementation holds significant advantages above a MAS implementation regarding maintainability.

6.2.8.4. Controller requirements

The controller requirements, as imposed by the needs of holonic control implementations, are evaluated through two performance measures: computational resource requirements and distributability. The results obtained for memory usage and processor time for each implementation are presented in section 6.2.6.6. The results indicated that the Erlang implementation required significantly less memory, but was more processor intensive. The increased processor time can be attributed to the nature of the Erlang implementation, where a focus on concurrency leads to a high number of active processes consuming processor time. The qualitative evaluation of the distributability of each implementation (in section 6.2.7.6) showed similar functionality.

6.2.8.5. Complexity

The complexity of the control implementations is reflected by several quantitative performance measures: development time, reconfiguration time, code complexity, code extension rate and code reuse rate. The additional time required for development and reconfiguration of the MAS implementation is indicative of an increased perceived complexity – especially concerning the verification of the software functionality. Along with the time measurements, this complexity is also reflected in the higher code complexity calculated for the MAS implementation. The code extension and reuse rates are in favour of the MAS implementation, but the reasons for this have already been highlighted in the discussion above about reconfigurability. Considering the requirements for the developer, the use of behaviours for specifying holon functionality, using threaded behaviours for achieving concurrency and verifying the behaviour of the system requires a higher level of expertise and experience in development of MAS implementations than with Erlang. However, the functionality of the AMS and DF of JADE makes it easier to implement and manage distribution in the MAS implementation.

6.2.8.6. Verification

As mentioned, the measurements for development and reconfiguration time indicate the advantages offered by the Erlang implementation. In Erlang, greater freedom is afforded to the developer for the construction of specific tests and the testing of individual functions allows for high granularity in the verification process. The process model also provides high modularity, simplifying the testing strategy.

The complexity and effort involved in the verification of the functionality exhibited by a control implementation can be deduced from three qualitative performance measures: modularity, diagnosability and developer requirements. Modularity allows for the verification of smaller, individual system components that together constitute complex functionality. Diagnosability points the developer to the source of errors. The requirements of the developer, concerning expertise and experience, indicate the difficulty and amount of work required to verify the system functionality. The advantages of the Erlang implementation regarding modularity and diagnosability have been referred to in earlier discussion. The advantages of using Erlang, concerning the verification of developed software, are discussed in the comparison of developer requirements (section 6.2.7.7).

The opinion of the authors is that the Erlang implementation inherently provides better support for the developer in the verification process, but that the ease of use can be improved through the inclusion of tools resembling those offered by the MAS implementation.

6.2.8.7. Reusability

The development time, reconfiguration time and code reuse rate measurements are used to evaluate the software artefact reusability in each control implementation. The reconfiguration experiment showed that the main differences in the development and reconfiguration times are due to the verification process – it was observed that the implementations allowed for similar levels of code reuse. The MAS implementation showed better code reuse rate results, but further testing (on a larger scale) is required for confirmation.

Modularity and integrability properties are considered to be indicative of the reusability in each implementation. The comparison shows that the Erlang implementation exhibits better modularity and provides integration mechanisms that are similar to the MAS implementation.

6.2.9. Findings, Considerations and Future Work

The comparison of the MAS and Erlang holonic control implementations yielded interesting results. The evaluation indicates that Erlang matches the functionality of the MAS implementation, and even offers advantages regarding the desired characteristics for the holonic control of manufacturing systems.

The Erlang process model exhibits enhanced modularity and robustness properties, leading to improved system availability. It is easier to support Erlang implementations, due to good maintainability and distributability properties. The development productivity that can be achieved using Erlang is also a significant benefit, due to the resulting reduction in software complexity and simplification of the verification process.

The premise of the comparison makes this result even more significant. The MAS was developed using JADE – a framework specific for the development of agents, which has been considered as a very suitable medium for implementing the software components of a holonic system. This specific tool is then compared to an implementation using standard Erlang with generic OTP libraries – perhaps a fairer comparison would have been between implementations in standard Java and Erlang. Still, the comparison with a JADE MAS confirms the inherent suitability and potential of the Erlang programming language for the implementation of holonic control.

However, some challenges were identified with the Erlang implementation. The first is a lack of standardization – the JADE compliance with FIPA standards, along with the suggested use of standard behaviours, offer advantages pertaining to uniformity and interoperability. Erlang also lacks some tools to simplify the verification and distribution of holonic systems – e.g. a graphical tool for tracing communication (like the JADE Sniffer) and a service for discovering resources within a distributed system (such as the Directory Facilitator of JADE). Furthermore, while the Erlang implementation used notably less memory than its MAS counterpart did, it required more processor time – this could have a detrimental effect on the performance of large, highly concurrent software systems. Further testing and evaluation of Erlang holonic control implementations are required to address this issue.

The following topics have been identified for future work:

- An Erlang framework for holonic control – the creation of functions, modules, libraries and tools to provide a framework for the development of holonic control implementations in Erlang.
- MAS in Erlang – to introduce standardization in Erlang applications, it would be useful to integrate the existing FIPA standards. An implementation and evaluation of an Erlang-based MAS for holonic control should be investigated. The Erlang experimental agent tool, eXAT (Di Stefano and Santoro, 2003), is an existing framework that can be used for such an implementation.

6.2.10. References

Andersson, F. and Bergström, F., 2011. *Development of an Erlang System Adapted to Embedded Devices*. Department of Information Technology, Uppsala University, Sweden.

Anonymous, s.a. (a). *Get Started with OTP*. [Online]. Available: http://www.erlang.org (18 July 2013).

Anonymous, s.a. (b). *Erlang/OTP System Documentation*. [Online]. Available: http://www.erlang.org/doc (1 September 2017).

Anonymous, s.a. (c). *Erlang Observer*. [Online]. Available: http://www.erlang.org/doc (1 September 2017).

Anonymous, s.a. (d). *Java Native Interface Specification*. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html (1 September 2017).

Anonymous, s.a. (e). *Java Networking*. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/net/index.html (1 September 2017).

Anonymous, s.a. (f). *Erlang Kernel Reference Manual*. [Online]. Available: http://www.erlang.org/doc (1 September 2017).

Anonymous, s.a. (g). *XMErl Reference Manual*. [Online]. Available: http://www.erlang.org (18 July 2013).

Anonymous, s.a. (h). *Serial Programming/Serial Java*. [Online]. Available: https://en.wikibooks.org/wiki/Serial_Programming/Serial_Java (1 September 2017).

Anonymous, s.a. (i). *RXTX*. [Online]. Available: http://rxtx.qbang.org/wiki/index.php/Main_Page (1 September 2017).

Anonymous, s.a. (j). *jSerialComm*. [Online]. Available: http://fazecast.github.io/jSerialComm (1 September 2017).

Anonymous, s.a. (k). *Gen_serial Documentation*. [Online]. Available: http://tomszilagyi.github.io/gen_serial/api/gen_serial.html (1 September 2017).

Armstrong, J., 2003. *Making Reliable Distributed Systems in the Presence of Software Errors*. Doctor's Dissertation. Royal Institute of Technology, Stockholm, Sweden.

Armstrong, J., 2007. *Programming Erlang: Software for a Concurrent World*. Raleigh, North Carolina: The Pragmatic Bookshelf.

Armstrong, J., 2010. Erlang. *Communications of the ACM*. Vol. 53, No. 9:68-75.

Bellifemine, F., Caire, G. and Greenwood, G., 2007. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, Ltd.

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

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.

Brouwers, N., Corke, P. and Langendoen, K., 2008. Darjeeling, a Java Compatible Virtual Machine for Microcontrollers. *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*. pp. 18-23.

Bussmann, S. and Sieverding, J., 2001. Holonic Control of an Engine Assembly Plant: An Industrial Evaluation. *2001 IEEE International Conference on Systems, Man, and Cybernetics*. Vol. 5, pp. 3151-3156.

Cesarini, F., Pappalardo, V. and Santoro, C., 2008. A Comparative Evaluation of Imperative and Functional Implementations of the IMAP Protocol. *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*. pp. 29-40.

Chirn, J.L. and McFarlane, D., 2000. A Holonic Component-based Approach to Reconfigurable Manufacturing Control Architecture. *Proceedings of the International Workshop on Industrial Applications of Holonic and Multi-Agent Systems*. pp. 219–223.

Chirn, J.L. and McFarlane, D., 2005. Evaluating Holonic Control Systems: A Case Study. *Proceedings of the 16th IFAC World Conference*. Prague, Czech Republic.

Christensen, J.H., 1994. Holonic Manufacturing Systems: Initial Architecture and Standards Directions. *First European Conference on Holonic Manufacturing Systems*. Hannover, Germany (December, 1994).

Czajkowski, G. and Daynés, L., 2001. Multitasking without Compromise: A Virtual Machine Evolution. In *ACM SIGPLAN Notices*. Vol. 36, No. 11:125-138.

Di Stefano, A. and Santoro, C., 2003. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. *Workshop on Objects and Agents*. Villasimius, Italy (September 2003).

Giret, A. and Botti, V., 2009. Engineering Holonic Manufacturing Systems. *Computers in Industry*. Vol. 60:428-440.

Harrison, R., Samaraweera, L.G., Dobie, M.R. and Lewis, P.H., 1996. Comparing Programming Paradigms: An Evaluation of Functional and Object-Oriented Programs. *Software Engineering Journal*. Vol. 11, No. 4:247-254.

Nyström, J., Trinder, P. and King, D., 2007. Evaluating High-Level Distributed Language Constructs. *ACM SIGPLAN Notices*. Vol. 42, No. 9:203-212.

Koestler, A., 1967. *The Ghost in the Machine*. London: Arkana Books.

Koren, Y. and Shpitalni, M., 2010. Design of Reconfigurable Manufacturing Systems. *Journal of Manufacturing Systems*. Vol. 29, pp. 130-141.

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, pp. 95–108.

Kruger, K. and Basson, A.H., 2013. Multi-agent Systems vs IEC 61499 for Holonic Resource Control in Reconfigurable Systems. *46th CIRP Conference on Manufacturing Systems*. Vol. 7, pp. 503-508.

Kruger, K. and Basson, A.H., 2017a. Erlang-Based Control Implementation for a Holonic Manufacturing Cell. *International Journal of Computer Integrated Manufacturing*. Vol. 30, No. 6:641-652.

Kruger, K. and Basson, A.H., 2017b. Holonic Control Implementation Using a JADE Multi-Agent System. *Internal Technical Report*. Department of Mechanical and Mechatronic Engineering, Stellenbosch University, South Africa.

Kruger, K. and Basson, A.H., 2017c. Evaluation Criteria for Holonic Control Implementations in Manufacturing Systems. *Submitted to the International Journal of Computer Integrated Manufacturing, September 2017.*

Leitao, P., 2004. *An Agile and Adaptive Holonic Architecture for Manufacturing Control.* Ph.D. Dissertation. University of Porto, Portugal.

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

Logan, M., 2010. *The Resource Discovery Application*. [Online]. Available: https://libraries.io/github/erlware/resource_discovery (1 September 2017).

Logan, M., Merrit, E., and Carlsson, R., 2011. *Erlang and OTP in Action*. Stamford: Manning Publications Co.

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.

Monostori, L., Kádár, B., Bauernhansl, T., Kondoh, S., Kumara, S., Reinhart, G., Sauer, O., Schuh, G., Sihn, W. and Ueda, K., 2016. Cyber-Physical Systems in Manufacturing. *CIRP Annals - Manufacturing Technology*. Vol. 65, pp. 621–641.

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

Prechelt, L., 2000. An Emperical Comparison of Seven Programming Languages. *Computer*. Vol. 33, No. 10:23-29.

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

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, pp. 255–274.

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

7. Conclusion

The dynamic and highly competitive nature of the modern manufacturing environment has introduced a new set of challenges, urging researchers and industry to formulate new and innovative solutions. The concepts of holonic and reconfigurable manufacturing systems showed great promise to address the challenges. Unfortunately, these concepts could not achieve significant adoption by industry and hence were predominantly restricted to academic, experimental implementations.

The latest emerging paradigm in manufacturing science and technology, Industry 4.0, can potentially have a significant impact on the manufacturing industry. The difference in impact can be attributed to the support from the German government and several big players in the manufacturing automation industry. Industry 4.0 considers all aspects of the manufacturing industry, aiming to enhance individualization of products through highly flexible production, extensively integrate customers and businesses in value-added processes and link production and high-quality services to deliver hybrid products. To achieve these goals, Industry 4.0 relies on Cyber-Physical Production Systems (CPPs) to enhance the connectedness throughout all levels of the manufacturing enterprise.

CPPSs aim to enhance the intelligence, connectedness and responsiveness of manufacturing systems. These goals closely resemble those of holonic and reconfigurable manufacturing systems, indicating the relevance of research on these topics to the development and implementation of CPPSs.

The objective of the presented research is to evaluate the suitability of the Erlang programming language as an alternative for the implementation of holonic control in manufacturing systems. The dissertation presents an Erlang-based holonic control implementation for a manufacturing cell. The Erlang implementation is evaluated through a comparison with an equivalent implementation using Multi-Agent Systems (MASs), which is considered as the *status quo* for holonic control implementation in manufacturing systems research.

To accomplish the evaluation of the holonic control implementations, a case study was selected and evaluation criteria were formulated. The case study involves the execution control of an assembly and quality assurance cell for electrical circuit breakers. The evaluation criteria focusses on both the development of control implementations and the adoption of the implementations by industry. The criteria are related to a set of quantitative and qualitative performance measures that are indicative of seven critical requirements for holonic control implementations. The Erlang and MAS implementations are evaluated and compared according to these performance measures and requirements.

The comparison of the MAS and Erlang holonic control implementations yielded interesting results. The evaluation indicated that the Erlang implementation matches the functionality of the MAS implementation and even offers some advantages for the desired characteristics for the holonic control of manufacturing systems. The advantages in availability and supportability can be attributed to the enhanced modularity and fault tolerance of the Erlang implementation. The Erlang implementation is also found to allow for increased development productivity through a reduction in software complexity and simplification of software verification.

The premise of the comparison makes this result even more significant. The MAS was developed using JADE – a framework specific for the development of agents, which is well established as a suitable medium for implementing the software components of holonic systems. This implementation is then compared to an implementation using standard Erlang, with generic OTP libraries – perhaps a fairer comparison would have been between implementations in standard Java and Erlang. Still, the comparison with a JADE MAS confirms the inherent suitability of the Erlang programming language for the implementation of holonic control, which warrants further research on the topic.

However, some challenges were identified with the Erlang implementation that requires further investigation and development:

- Standardization – the JADE compliance with FIPA standards, along with the suggested use of standard behaviours, offer advantages pertaining to uniformity and interoperability.
- Tools – Erlang lacks some important tools to simplify the verification and distribution of holonic systems – e.g. a graphical tool for tracing communication (like the JADE Sniffer) and a service for discovering resources within a distributed system (such as the Directory Facilitator of JADE).
- Computational resource requirements – the evaluation showed that Erlang implementation used significantly less memory than its MAS counterpart did, but that it consumed more processor time. While not necessarily a problem in all applications, high processor usage could have a detrimental effect on the performance of large, highly concurrent software systems. Further testing and evaluation of the architecture for Erlang holonic control implementations are required to address this issue.

Furthermore, there is great potential for further research on the use of Erlang for control implementation in manufacturing systems. The following topics have been identified for future work:

- Further refinement of the architecture for Erlang holonic control implementations – the architecture presented in this dissertation should serve as a starting point for the development of more complete and advanced architectures. These architectures should consider the inclusion and exploitation of the other interesting features of Erlang – specifically, the use of supervision trees to increase fault tolerance and the increased availability, supportability and maintainability that can be achieved through hot code loading.

- An Erlang framework for holonic control – the creation of functions, modules, libraries and tools to provide a framework specifically for the development of holonic control implementations.
- MAS in Erlang – to introduce standardization in Erlang applications, it would be useful to integrate the existing FIPA standards. An implementation and evaluation of an Erlang-based MAS for holonic control should be investigated. The Erlang experimental agent tool, eXAT (Di Stefano and Santoro, 2003), is an existing framework that can be used for such an implementation.
- Standardized test cases and benchmarks for the evaluation of holonic control implementations – the dissertation presents criteria and a methodology for the evaluation of holonic control implementations. The applicability of this framework to other cases must be evaluated and it is recommended that further research be conducted into the formulation of standardized test cases and benchmarks.

8. References

Andersson, F. and Bergström, F., 2011. *Development of an Erlang System Adapted to Embedded Devices*. Department of Information Technology, Uppsala University, Sweden.

Anonymous, s.a. (a). *Get Started with OTP*. [Online]. Available: http://www.erlang.org/doc (1 September 2017).

Anonymous, s.a. (b). *Erlang/OTP System Documentation*. [Online]. Available: http://www.erlang.org/doc (1 September 2017).

Anonymous, s.a. (c). *XMErL Reference Manual*. [Online]. Available: http://www.erlang.org/doc/apps/xmerl (1 September 2017).

Anonymous, s.a. (d). *Erlang Kernel Reference Manual*. [Online]. Available: http://www.erlang.org/doc/apps/kernel (1 September 2017).

Anonymous, s.a. (e). *Erlang Observer*. [Online]. Available: http://www.erlang.org/doc (1 September 2017).

Anonymous, s.a. (f). *Java Native Interface Specification*. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html (1 September 2017).

Anonymous, s.a. (g). *Java Networking*. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/net/index.html (1 September 2017).

Anonymous, s.a. (h). *Serial Programming/Serial Java*. [Online]. Available: https://en.wikibooks.org/wiki/Serial_Programming/Serial_Java (1 September 2017).

Anonymous, s.a. (i). *RXTX*. [Online]. Available: http://rxtx.qbang.org/wiki/index.php/Main_Page (1 September 2017).

Anonymous, s.a. (j). *jSerialComm*. [Online]. Available: http://fazecast.github.io/jSerialComm (1 September 2017).

Anonymous, s.a. (k). *Gen_serial Documentation*. [Online]. Available: http://tomszilagyi.github.io/gen_serial/api/gen_serial.html (1 September 2017).

Almeida, F.L., Terra, B.M., Dias, P.A., and Gonçales, G.M., 2010. Adoption Issues of Multi-Agent Systems in Manufacturing Industry. *Fifth International Multi-conference on Computing in the Global Information Technology*. pp. 238-244.

Armstrong, J., 2003. *Making Reliable Distributed Systems in the Presence of Software Errors*. Doctor's Dissertation. Royal Institute of Technology, Stockholm, Sweden.

Armstrong, J., 2007. *Programming Erlang: Software for a Concurrent World*. Raleigh, North Carolina: The Pragmatic Bookshelf.

Armstrong, J., 2010. Erlang. *Communications of the ACM*. Vol. 53, No. 9:68-75.

Armstrong, J., Virding, R. Wikström, C. and Williams, M., 1996. *Concurrent Programming in Erlang*. Second Edition. New Jersey: Prentice Hall.

Baldwin, C. and Clark, K., 2006. Modularity in the Design of Complex Engineering Systems. *Complex Engineered Systems*. pp. 175-205.

Bellifemine, F., Caire, G. and Greenwood, G., 2007. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, Ltd.

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.

Bousbia, S. and Trentesaux, D., 2002. Self-Organization in Distributed Manufacturing Control: State-of-the-Art and Future Trends. *2002 IEEE International Conference on Systems, Man and Cybernetics*. Vol. 5:6-12.

Brouwers, N., Corke, P. and Langendoen, K., 2008. Darjeeling, a Java Compatible Virtual Machine for Microcontrollers. *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion.* pp. 18-23.

Bussman, S., 1998. An Agent-Oriented Architecture for Holonic Manufacturing Control. *1st Workshop on Intelligent Manufacturing Systems*. Lausanne, Switzerland.

Bussman, S. and McFarlane, D., 1999. Rationales for Holonic manufacturing Control. *Proceedings of the 2nd International Workshop on Intelligent Manufacturing Systems*. Leuven, Belgium (September 1999). pp. 177-184.

Bussmann, S. and Sieverding, J., 2001. Holonic Control of an Engine Assembly Plant: An Industrial Evaluation. *2001 IEEE International Conference on Systems, Man, and Cybernetics.* Vol. 5:3151-3156.

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.

Cesarini, F., Pappalardo, V. and Santoro, C., 2008. A Comparative Evaluation of Imperative and Functional Implementations of the IMAP Protocol. *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang.* pp. 29-40. ACM.

Chirn, J.L. and McFarlane, D., 2000. A Holonic Component-based Approach to Reconfigurable Manufacturing Control Architecture. *Proceedings of the International Workshop on Industrial Applications of Holonic and Multi-Agent Systems.* pp. 219–223.

Chirn, J.L. and McFarlane, D., 2005. Evaluating Holonic Control Systems: A Case Study. *Proceedings of the 16th IFAC World Conference.* Prague, Czech Republic.

Christensen, J.H., 1994. Holonic Manufacturing Systems: Initial Architecture and Standards Directions. *First European Conference on Holonic Manufacturing Systems.* Hannover, Germany (December, 1994).

Coleman, D., Ash, D., Lowther, B. and Oman, P., 1994. Using Metrics to Evaluate Software System Maintainability. *Computer*. Vol. 27, No. 8:44-49.

Czajkowski, G. and Daynés, L., 2001. Multitasking without Compromise: A Virtual Machine Evolution. In *ACM SIGPLAN Notices.* Vol. 36, No. 11:125-138.

Däcker, B., 2000. *Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction*. Master's Thesis. Royal Institute of Technology, Stockholm, Sweden.

De Jong, W., 2007. *Erlsom*. [Online]. Available: http://erlsom.sourceforge.net (28 March 2017).

Di Stefano, A. and Santoro, C., 2003. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. *Workshop on Objects and Agents.* Villasimius, Italy (September 2003).

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

FIPA (Foundation for Intelligent Physical Agents), 2010. [Online]. Available: http://www.fipa.org. (2017, July 20).

Giret, A. and Botti, V., 2009. Engineering Holonic Manufacturing Systems. *Computers in Industry*. Vol. 60:428-440.

Graefe, R. and Basson, A.H., 2013. Control of Reconfigurable Manufacturing Systems using Object-Oriented Programming. *Proceedings of the 5th International Conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV2013)*. pp. 231-236.

Harrison, R., Samaraweera, L.G., Dobie, M.R. and Lewis, P.H., 1996. Comparing Programming Paradigms: An Evaluation of Functional and Object-Oriented Programs. *Software Engineering Journal*. Vol. 11, No. 4:247-254.

Hebert, F., 2014. *Learn Some Erlang For Great Good*. No Starch Press.

Holvoet, T. and Valckenaers, P., 2006. Exploiting the Environment for Coordinating Agent Intentions. *AAMAS Conference*. Hakodate, Japan (8–12 May).

Hubbard, D., 1999. The IT Measurement Inversion. *CIO Enterprize Magazine*.

Kennedy, K., Koelbel, C. and Schreiber, R., 2004. Defining and Measuring the Productivity of Programming Languages. *International Journal of High Performance Computing Applications*. Vol. 18, No. 4:441-448.

Koestler, A., 1967. *The Ghost in the Machine*. London: Arkana Books.

Koren, Y., 2006. General RMS Characteristics: Comparison with Dedicated and Flexible systems. *Reconfigurable Manufacturing Systems and Transformable Factories*. pp. 27-45. Springer, Berlin Heidelberg.

Koren, Y., Heisel, U., Jovane, F., Moriwaki, T., Pritschow, G., Ulsoy, G and Van Brussel, H., 1999. Reconfigurable Manufacturing Systems. *Annals of the CIRP*. Vol. 48, No. 2:527-540.

Koren, Y. and Shpitalni, M., 2010. Design of Reconfigurable Manufacturing Systems. *Journal of Manufacturing Systems*. Vol. 29:130-141.

Koren, Y. and Ulsoy, A.G., 1997. Reconfigurable Manufacturing Systems. *ERC/RMS Report #1*. Ann Arbor, USA.

Koren, Y. and Ulsoy, A.G., 2002. Vision, Principles and Impact of Reconfigurable Manufacturing Systems. *Production International*. pp. 14-21.

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.

Kotze, M.J., 2016. *Modular Control of a Reconfigurable Conveyor System*. Thesis. Stellenbosch University, South Africa.

Krueger, C.W., 1992. Software Reuse. *ACM Computing Surveys (CSUR)*. Vol. 24, No. 2:131-183.

Kruger, K. and Basson, A.H., 2013. Multi-agent Systems vs IEC 61499 for Holonic Resource Control in Reconfigurable Systems. *46th CIRP Conference on Manufacturing Systems*. Vol. 7:503-508.

Kruger, K. and Basson, A.H., 2015. Implementation of an Erlang-based Resource Holon for a Holonic Manufacturing Cell. *Service Orientation in Holonic and Multi-agent Manufacturing*. Studies in Computational Intelligence, Vol. 594:49-58. Springer.

Kruger, K. and Basson, A.H., 2017 (a). Erlang-Based Control Implementation for a Holonic Manufacturing Cell. *International Journal of Computer Integrated Manufacturing*. Vol. 30, No. 6:641-652.

Kruger, K. and Basson, A.H., 2017 (b). Erlang-Based Holonic Controller for a Modular Conveyor System. *Service Orientation in Holonic and Multi-Agent Manufacturing*. Studies in Computational Intelligence, Vol. 694:191-200. Springer.

Kruger, K. and Basson, A.H., 2017 (c). Validation of a Holonic Controller for a Modular Conveyor System using an Object-Oriented Simulation Framework. *Service Orientation in Holonic and Multi-Agent Manufacturing*. Studies in Computational Intelligence, Vol. 694:427-435. Springer.

Lee, J., Bagheri, B. and Kao, H., 2015. A Cyber-Physical Systems Architecture for Industry 4.0-based Manufacturing Systems. *Manufacturing Letters*. Vol. 3:18-23.

Leitao, P., 2004. *An Agile and Adaptive Holonic Architecture for Manufacturing Control*. Ph.D. Dissertation. University of Porto, Portugal.

Leitao, P. and Restivo, F.J., 2002. A Holonic Control Approach for Distributed Manufacturing. *Knowledge and Technology Integration in Production and Services: Balancing Knowledge and Technology in Product and Service Life Cycle*. pp. 263–270. Kluwer Academic Publishers.

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.

Leitao, P., Colombo, A.W. and Karnouskos, S., 2016. Industrial Automation Based on Cyber-Physical Systems Technologies: Prototype Implementations and Challenges. *Computers in Industry*. Vol. 81:11-25.

Le Traon, Y., Ouabdesselam, F., Robach, C. and Baudry, B., 2003. From Diagnosis to Diagnosability: Axiomatization, Measurement and Application. *Journal of Systems and Software*. Vol. 65, No. 1:31-50.

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.

Logan, M., 2010. *The Resource Discovery Application*. [Online]. Available: https://libraries.io/github/erlware/resource_discovery (1 September 2017).
Logan, M., Merrit, E., and Carlsson, R., 2011. *Erlang and OTP in Action*. Stamford: Manning Publications Co.

Lowy, J. and Montgomery, M., 2015. *Programming WCF Services.* 4th edition. O'Reilly Media.

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.

McCabe, T.J., 1976. A Complexity Measure. *IEEE Transactions on Software Engineering*. Vol. 4:308-320.

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.

Microsoft TechNet, s.a. *Task Manager*. Available: https://technet.microsoft.com/en-us/library (30 August 2017).

Microsoft TechNet, s.a. *Overview of Performance Monitor*. Available: https://technet.microsoft.com/en-us/library/cc749154 (30 August 2017).

Monostori, L., Kádár, B., Bauernhansl, T., Kondoh, S., Kumara, S., Reinhart, G., Sauer, O., Schuh, G., Sihn, W. and Ueda, K., 2016. Cyber-Physical Systems in Manufacturing. *CIRP Annals - Manufacturing Technology*. Vol. 65:621–641.

Nyström, J., Trinder, P. and King, D., 2007. Evaluating High-Level Distributed Language Constructs. *ACM SIGPLAN Notices*. Vol. 42, No. 9:203-212.

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

Pegden, C.D., 2007. Simio: A New Simulation System Based on Intelligent Objects. *Proceedings of the 2007 Winter Simulation Conference*. pp. 2294-2300.

Pegden, C.D., 2008. Introduction to Simio. *Proceedings of the 2008 Winter Simulation Conference*. pp. 229-235.

Prechelt, L., 2000. An Emperical Comparison of Seven Programming Languages. *Computer*. Vol. 33, No. 10:23-29.

Prieto-Diaz, R. and Freeman, P., 1987. Classifying Software for Reusability. *IEEE Software*. Vol. 4, No. 1:6-16.

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.

Trendowicz, A. and Münch, J., 2009. Factors Influencing Software Development Productivity: State-of-the-Art and Industrial Experiences. *Advances in Computers*. Vol. 77:185-241.

Valckenaers, P. and van Brussel, H., 2015. *Design for the Unexpected*. 1st edition. Butterworth-Heinemann, ISBN: 9780128036624.

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.

Vinoski, S., 2007. Concurrency with Erlang. *IEEE Internet Computing*. Vol. 11, No. 5:90-93.

Voas, J.M. and Miller, K.W., 1995. Software Testability: The New Verification. *IEEE Software*. Vol. 12, No. 3:17-28.

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.

Wang, L. and Haghighi, A., 2016. Combined Strength of Holons, Agent and Function Blocks in Cyber-Physical Systems. *Journal of Manufacturing Systems*. Vol. 40:25-34.

Weyuker, E.J., 1988. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*. Vol. 14, No. 9:1357-1365.

Wiendahl, H.P., ElMaraghy, H.A., Nyhuis, P., Zah, M.F., Wiendahl, H.H., Duffie, N. and Brieke, M., 2007. Changeable Manufacturing: Classification, Design and Operation. *Annals of CIRP*. Vol. 56, No. 2:783-809.

Wiger, U., 2001. Four-fold Increase in Productivity and Quality. *Workshop on Formal Design of Safety Critical Embedded Systems*. March 21-23, Munich, Germany.

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.

Appendix A: Erlang and JADE Source Code

This appendix presents the source code for the Electrical Test Station Resource holon, as implemented in Erlang and JADE. The source code for the communication, agenda management and execution components of the internal Resource holon architecture, as presented in Figure 27, is shown. The line numbers added to the code indicate the SLOC measurement, as used in section 6.2.

A.1. Erlang Resource Holon

A.1.1. Communication Component

```
-module(resource_comm).
-include("messaging.hrl").
%macro for function that adds the reference to the registered name
-define(MAKE_NAME(Name,Ref),list_to_atom(atom_to_list(Name)++Ref)).
-define(MAKE_NETWORK_NAME(Name,Ref),{list_to_atom(atom_to_list(Name)++Ref),'main@meg461398.stb.sun.ac.za'}).
%% ====================================================================
%% API functions
%% ====================================================================
-export([rec_messages/1,start/1]).
start(Ref) -> Pid = spawn_link(resource_comm,rec_messages,[Ref]),
  register(?MAKE_NAME(resource_comm,Ref), Pid),
  {ok,Pid}.
%% ====================================================================
%% Internal functions
%% ====================================================================
%function maintaining the inter-holon communication interface
rec_messages(Ref) ->
  Resource_comm = ?MAKE_NAME(resource_comm,Ref),
  Resource_am = ?MAKE_NAME(resource_am,Ref),
  Resource_exec = ?MAKE_NAME(resource_exec,Ref),
  receive
    %SERVICE message from resource_am process, in reply to some service request
    {Resource_am,Message=#service{message_type=register}} ->
       error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
    >  message_received}, {content, {Resource_am,Message}}]),
       service_directory ! {?MAKE_NAME(resource_comm,Ref),Message},
       rec_messages(Ref);
    {Resource_am,Message=#service{}} ->
       error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
       message_received}, {content, {Resource_am,Message}}]),
       Pid = Message#service.requester_pid,
       Pid ! {?MAKE_NETWORK_NAME(resource_comm,Ref),Message},
       rec_messages(Ref);
    {Resource_exec,Message=#service{}} ->
       error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
       message_received}, {content, {Resource_am,Message}}]),
       Pid = Message#service.provider_pid,
       Pid ! {?MAKE_NETWORK_NAME(resource_comm,Ref),Message},
       rec_messages(Ref);
    %SERVICE message from some holon providing a service
    {From,Message=#service{requester_pid=Resource_exec}} ->
       error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
    >  message_received}, {content, {From,Message}}]),
       Resource_exec ! {Resource_comm,Message},
       rec_messages(Ref);
    %SERVICE message from some holon requesting a service
    {From,Message=#service{}} ->
       error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
    >  message_received}, {content, {From,Message}}]),
       Resource_am ! {Resource_comm,Message},
       rec_messages(Ref);
    {From,Message} ->
       io:format("~p received unexpected message: {~p,~p}~n",[Resource_comm,From,Message])
  end.
```

A.1.2. Agenda Manager Component

```
-module(resource_am).
-include("messaging.hrl").
-behaviour(gen_fsm).
%macro for function that adds the reference to the registered name
-define(MAKE_NAME(Name,Ref),list_to_atom(atom_to_list(Name)++Ref)).
%% ====================================================================
%% API functions
%% ====================================================================
-export([start/1,rec_messages/1]).
-export([init/1,ready/2,free_free/2,free_alloc/2,busy_alloc/2]).
%start gen_fsm process
start(Ref) -> gen_fsm:start_link({local,?MAKE_NAME(resource_am_fsm,Ref)}, resource_am, [Ref], []),
   %start comm interface process
   Pid = spawn_link(resource_am,rec_messages,[Ref]),
   register(?MAKE_NAME(resource_am,Ref), Pid),
   {ok,Pid}.
%% ====================================================================
%% Internal functions
%% ====================================================================
%process for handling communication to the FSM
rec_messages(Ref) ->
   Resource_comm = ?MAKE_NAME(resource_comm,Ref),
   Resource_exec = ?MAKE_NAME(resource_exec,Ref),
   Resource_am_fsm = ?MAKE_NAME(resource_am_fsm,Ref),
   receive
     {Resource_exec,Message} -> gen_fsm:send_event(Resource_am_fsm, Message),
        error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
     >  message_received}, {content, {Resource_exec,Message}}]),
        rec_messages(Ref);
        {Resource_comm,Message=#service{}} -> gen_fsm:send_event(Resource_am_fsm, Message),
           error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
        >  message_received}, {content, {Resource_comm,Message}}]),
           rec_messages(Ref)
        end.
%FSM initialization
init([Ref]) -> {ok,ready,[Ref]}.
%STATE: ready --> fsm is initialized and awaits "ready" message from resource_exec
ready(Message=#service{message_type=status,info={ready,Service_type}},[Ref]) ->
   error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
>  state_transition}, {content, {ready,free_free}}]),
   ?MAKE_NAME(resource_comm,Ref) !
>  {?MAKE_NAME(resource_am,Ref),#service{message_type=register,service_type=Service_type,requester_pid=resource_comm},
   {next_state,free_free,[Ref]}.
%STATE: free_free --> the operational holon is idle, with no jobs allocated
free_free(Message=#service{message_type=propose},[Ref]) ->
   Proposal = create_proposal([]),
   ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message#service{result=true,info=Proposal}},
   {next_state, free_free,[Ref]};
free_free(Message=#service{message_type=allocate},[Ref])  ->
   ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message#service{result=true}},
   error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
>  state_transition}, {content, {free_free,free_alloc}}]),
   {next_state, free_alloc,[[Message#service.requester_pid],Ref]}.
%STATE: free_alloc --> the operational holon is idle, but jobs have been allocated
free_alloc(Message=#service{message_type=propose},[Job_list,Ref]) ->
   Proposal = create_proposal(Job_list),
   ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message#service{result=true,info=Proposal}},
   {next_state, free_alloc,[Job_list,Ref]};
free_alloc(Message=#service{message_type=allocate},[Job_list,Ref]) ->
   NewJob_list=lists:append(Job_list, [Message#service.requester_pid]),
   ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message#service{result=true}},
   {next_state, free_alloc, [NewJob_list,Ref]};
free_alloc(Message=#service{message_type=deallocate},[Job_list,Ref]) ->
   NewJob_list=lists:delete(Message#service.requester_pid, Job_list),
   io:format("Upon deallocate in free_alloc - new job list is ~p~n",[NewJob_list]),
   ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message#service{result=true}},
   case NewJob_list of
     [] ->    error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
     state_transition}, {content, {free_alloc,free_free}}]),
        %go to free_free state if no more jobs are allocated
        {next_state, free_free,[Ref]};
     NewJob_list -> {next_state, free_alloc,[NewJob_list,Ref]}
   end;
free_alloc(Message=#service{message_type=confirm},[Job_list,Ref]) ->
   %{confirm, true} is sent when Pid is an element of the Job_list
   ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message#service{result =
>  lists:member(Message#service.requester_pid, Job_list)}},
   {next_state, free_alloc,[Job_list,Ref]};
free_alloc(Message=#service{message_type=start},[Job_list,Ref]) ->
   %send "start" message to resource_exec - the process must reply to resource_am process
   ?MAKE_NAME(resource_exec,Ref) ! {?MAKE_NAME(resource_am,Ref),Message},
   {next_state,busy_alloc,[Message#service.requester_pid,lists:delete(Message#service.requester_pid, Job_list),Ref]}.
%STATE: busy_alloc --> the operational holon is busy performing a job and jobs are allocated
busy_alloc(Message=#service{message_type=start,requester_pid=CurrJob,result=true},[CurrJob,Job_list,Ref]) ->
   %forward confirmation of the "action start" to resource_comm
   ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message},
   error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
>  state_transition}, {content, {busy_alloc,busy_alloc}}]),
   %go to busy_alloc as next state
   {next_state,busy_alloc,[CurrJob,Job_list,Ref]};
busy_alloc(Message=#service{message_type=start,requester_pid=CurrJob,result=false},[CurrJob,Job_list,Ref]) ->
```

```
    %forward rejection of the "action start" to resource_comm
    ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message},
    error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
    state_transition}, {content, {busy_alloc,free_alloc}}]),
    %go to free_alloc as next state
    {next_state,free_alloc,[lists:append(Job_list,[CurrJob]),Ref]};
busy_alloc(Message=#service{message_type=start},[CurrJob,Job_list,Ref]) ->
    %already busy
    ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message#service{result=false}},
    error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
>   state_transition}, {content, {busy_alloc,busy_alloc}}]),
    %go to busy_alloc as next state
    {next_state,busy_alloc,[CurrJob,Job_list,Ref]};
busy_alloc(Message=#service{message_type=confirm},[CurrJob,Job_list,Ref]) ->
    %{confirm, true} is sent when Pid is an element of the Job_list
    ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message#service{result =
>   lists:member(Message#service.requester_pid, Job_list)}},
    {next_state, busy_alloc,[CurrJob,Job_list,Ref]};
busy_alloc(Message=#service{message_type=propose},[CurrJob,Job_list,Ref]) ->
    Proposal = create_proposal(Job_list),
    ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message#service{result=true,info=Proposal}},
    {next_state, busy_alloc,[CurrJob,Job_list,Ref]};
busy_alloc(Message=#service{message_type=allocate},[CurrJob,Job_list,Ref]) ->
    ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message#service{result=true}},
    {next_state, busy_alloc,[CurrJob, lists:append(Job_list, [Message#service.requester_pid]),Ref]};
busy_alloc(Message=#service{message_type=deallocate},[CurrJob,Job_list,Ref]) ->
    NewJob_list=lists:delete(Message#service.requester_pid, Job_list),
    io:format("Upon deallocate in busy_alloc - new job list is ~p~n",[NewJob_list]),
    ?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message#service{result=true}},
    {next_state, busy_alloc,[CurrJob,NewJob_list,Ref]};
busy_alloc(Message=#service{message_type=done},[CurrJob,Job_list,Ref]) ->
```

```
      case Message#service.requester_pid of
         CurrJob ->?MAKE_NAME(resource_comm,Ref) ! {?MAKE_NAME(resource_am,Ref),Message},
            case Job_list of
              [] -> error_logger:info_report([{reporter, {self(), erlang:process_info(self(),
           >  registered_name)}},{event, state_transition}, {content, {busy_alloc,free_free}}]),
                %go to free_free state if no more jobs are allocated
                 {next_state,free_free,[Ref]};
              Job_list -> error_logger:info_report([{reporter, {self(), erlang:process_info(self(),
           >  registered_name)}},{event, state_transition}, {content, {busy_alloc,free_alloc}}]),
                 {next_state,free_alloc,[Job_list,Ref]}
            end
      end.
%=================================================================================================
create_proposal(Bookings_list) ->
    (length(Bookings_list) + 1).
```

A.1.3. Execution Component

```
-module(resource_exec_ets).
-behaviour(gen_fsm).
-include("messaging.hrl").
%macro for function that adds the reference to the registered name
-define(MAKE_NAME(Name,Ref),list_to_atom(atom_to_list(Name)++Ref)).
%% ====================================================================
%% API functions
%% ====================================================================
-export([start/1]).
-export([rec_messages/1]).
-export([init/1,ready/2,ready_for_start/2,ready_to_test/2,testing_done/2]).
%start gen_fsm process
start(Ref) -> gen_fsm:start_link({local,?MAKE_NAME(resource_exec_fsm,Ref)}, resource_exec_ets, [Ref], []),
  %start comm interface process
  Pid = spawn_link(resource_exec_ets,rec_messages,[Ref]),
  register(?MAKE_NAME(resource_exec,Ref), Pid),
  {ok,Pid}.
%% ====================================================================
%% Internal functions
%% ====================================================================
%process for handling communication to the FSM
rec_messages(Ref) ->
  Resource_pi = ?MAKE_NAME(resource_pi,Ref),
  Resource_exec_fsm = ?MAKE_NAME(resource_exec_fsm,Ref),
  receive
    {Resource_pi,Message} ->
      error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
    > message_received}, {content, {Resource_pi,Message}}]),
      gen_fsm:send_event(Resource_exec_fsm, Message),
      rec_messages(Ref);
    {From,Message=#service{}} ->
      error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
    > message_received}, {content, {From,Message}}]),
      gen_fsm:send_event(Resource_exec_fsm,Message),
      rec_messages(Ref)
  end.
%FSM initialization
init([Ref]) -> {ok,ready,[Ref]}.
ready(Message=#service{message_type=status,info=ready},[Ref]) ->
  %status received from resource_pi - status sent to resource_am
  ?MAKE_NAME(resource_am,Ref) ! {?MAKE_NAME(resource_exec,Ref),#service{message_type=status,info={ready,test}}},
  error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
> state_transition}, {content, {ready,ready_for_start}}]),
  {next_state,ready_for_start,[Ref]}.
```

```
%STATE: ready_for_start --> the exec process is ready to start testing process
ready_for_start(Message=#service{message_type=start},[Ref]) ->
  %notify event logger
  event_logger ! {?MAKE_NAME(resource_exec,Ref),start},
  io:format("~p received start request from ~p~n",[?MAKE_NAME(resource_exec,Ref),Message#service.requester_pid]),
  %find name/pid of transport holon to which release_request must be sent
  ?MAKE_NAME(resource_comm,Ref) !
> {?MAKE_NAME(resource_exec,Ref),#service{message_type=request,service_type=transport,requester_pid=?MAKE_NAME(resource
> _exec,Ref),provider_pid=service_directory}},
  {next_state,ready_for_start,[Message#service.requester_pid,Message,Ref]};
ready_for_start(Msg=#service{message_type=request, service_type=transport},[CurrJob,Message,Ref]) ->
  [Transport_holon] = Msg#service.info, %assuming there will be only one transport holon
  %send release_request to transport holon
  Task_ref = CurrJob,
  ?MAKE_NAME(resource_comm,Ref) !
> {?MAKE_NAME(resource_exec,Ref),#service{message_type=release_request,requester_pid=?MAKE_NAME(resource_exec,Ref),prov
> ider_pid=Transport_holon,info=Task_ref}},
  {next_state,ready_to_test,[Message#service.requester_pid,Message,Ref]}.
%STATE: ready_to_test --> the exec process is ready to execute testing process
ready_to_test(Msg=#service{message_type=release_request,result=true},[CurrJob,Message,Ref]) ->
  io:format("Release_request successful!~n"),
  Pick_coords = Msg#service.info, %extract task info
  ?MAKE_NAME(resource_am,Ref) ! {?MAKE_NAME(resource_exec,Ref),Message#service{result=true}}, %send confirmation of
  service started
  %send placing coordinates to robot_pi
  ?MAKE_NAME(resource_pi,Ref) ! {?MAKE_NAME(resource_exec,Ref),Pick_coords},
  error_logger:info_report([{reporter, {self(), erlang:process_info(self(), registered_name)}},{event,
> state_transition}, {content, {ready_to_test,testing_done}}]),
  {next_state,testing_done,[CurrJob,Message,Ref]}.
%STATE: done --> the testing process is complete
testing_done(done,[CurrJob,Message,Ref]) ->
  io:format("Testing done~n"),
  ?MAKE_NAME(resource_comm,Ref) !
> {?MAKE_NAME(resource_exec,Ref),#service{message_type=request,service_type=transport,requester_pid=?MAKE_NAME(resource
> _exec,Ref),provider_pid=service_directory}},
  {next_state,testing_done,[CurrJob,Message,Ref]};
testing_done(Msg=#service{message_type=request, service_type=transport},[CurrJob,Message,Ref]) ->
  [Transport_holon] = Msg#service.info, %assuming there will be only one transport holon
  %send release_request to transport holon
  Task_ref = CurrJob,
  ?MAKE_NAME(resource_comm,Ref) !
> {?MAKE_NAME(resource_exec,Ref),#service{message_type=binding_request,requester_pid=?MAKE_NAME(resource_exec,Ref),prov
> ider_pid=Transport_holon,info={Task_ref,p01,?MAKE_NAME(?MAKE_NAME(resource_comm,Ref),"_output")}}},
  {next_state,testing_done,[CurrJob,Message,Ref]};
testing_done(Msg=#service{message_type=binding_request, result=true},[CurrJob,Message,Ref]) ->
  %notify event logger
  event_logger ! {?MAKE_NAME(resource_exec,Ref),done},
  io:format("~p placed task ~p on transport holon carrier at
> ~p~n",[?MAKE_NAME(resource_exec,Ref),CurrJob,Msg#service.info]),
  ?MAKE_NAME(resource_am,Ref) ! {?MAKE_NAME(resource_exec,Ref),Message#service{message_type=done,result=true}},
  {next_state,ready_for_start,[Ref]};
testing_done(Msg=#service{message_type=binding_request, result=false},[CurrJob,Message,Ref]) ->
  timer:sleep(1000),
  ?MAKE_NAME(resource_comm,Ref) !
> {?MAKE_NAME(resource_exec,Ref),Msg#service{result=undefined,info={CurrJob,p01,?MAKE_NAME(?MAKE_NAME(resource_comm,Ref
> ),"_output")}}},
  {next_state,testing_done,[CurrJob,Message,Ref]}.
```

A.2. JADE Resource Agent

A.2.1. Resource Agent

```java
package agents;
import java.io.StringReader;
...
import jade.util.leap.Set;

public class ResourceAgent extends Agent{
  private ArrayList<AID> booking_list = new ArrayList<AID>();
  private int booking_buff = 10;
  private AchieveREResponder started_task_responder;
  private Boolean task_started = false;
  private Boolean task_done = false;
  private String[] service_type;
  private String service_requested;
  public HashMap<Integer,additional.BufferEntryData> stack_buffer = new HashMap<Integer,additional.BufferEntryData>();
  protected void setup(){
      Object[] args = getArguments();
      service_type = (String[]) args;
      // register agent services with the Directory Facilitator
      DFAgentDescription dfd = new DFAgentDescription();
      dfd.setName(getAID());
      for(int index = 0; index < service_type.length; index++){
          ServiceDescription sd = new ServiceDescription();
          sd.setType(service_type[index]);
          sd.setName(getLocalName());
          dfd.addServices(sd);
      }
      try{
          DFService.register(this, dfd);
      }
      catch (FIPAException fe){
          fe.printStackTrace();
      }
      //add behaviour to respond to booking requests using CNP
      MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
      addBehaviour(new ContractNetResponder(this, mt){
          protected ACLMessage handleCfp(ACLMessage cfp){
             ACLMessage reply = cfp.createReply();
             //check bookings list to see if available for booking
             if(booking_list.size() < booking_buff){
               reply.setPerformative(ACLMessage.PROPOSE);
               int proposal = booking_list.size() + 1;
               reply.setContent(String.valueOf(proposal));
             }
             else{
               reply.setPerformative(ACLMessage.REFUSE);
             }
             return reply;
          }
          protected ACLMessage handleAcceptProposal(ACLMessage cfp, ACLMessage propose,ACLMessage accept){
             ACLMessage result = accept.createReply();
             //update bookings list
             booking_list.add(accept.getSender());
             result.setPerformative(ACLMessage.INFORM);
             return result;
          }
      });
```

```java
        //add behaviour to handle confirmation inquiries
        MessageTemplate confirm_req = MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.REQUEST),MessageTemplate.MatchContent("confirm"));
        addBehaviour(new AchieveREResponder(this,confirm_req){
            protected ACLMessage prepareResultNotification(ACLMessage request,ACLMessage response){
                //create reply to message
                ACLMessage result = request.createReply();
                //check if requesting agent has made a booking
                if(booking_list.indexOf(request.getSender()) != -1){
                  result.setPerformative(ACLMessage.INFORM);
                  result.setContent(request.getContent());
                }
                else{
                  result.setPerformative(ACLMessage.FAILURE);
                  result.setContent(request.getContent());
                }
                return result;
            }
        });
        MessageTemplate req_temp = MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.REQUEST),new MessageTemplate(new RegexMatchExpression("<message_type>start\\.*") {}));
        //add behaviour to launch a Responder behaviour for every incoming request
        addBehaviour(new SSResponderDispatcher(this,req_temp){
            public Behaviour createResponder(ACLMessage req_msg) {
                System.out.println(this.myAgent.getName() + " created a Responder for received request!");
                //get XML content of request message
                String req_msg_content = req_msg.getContent();
                System.out.println(myAgent.getName() + " request content: " + req_msg_content.toString());
                Document xml_content = XmlTools.buildXmlDoc(req_msg_content);
```

```
service_requested =
> xml_content.getDocumentElement().getElementsByTagName("service_type").item(0).getTextContent();
System.out.println(myAgent.getName() + " request to perform service: Execute_" + service_requested);
SSIteratedAchieveREResponder responder = null;
ThreadedBehaviourFactory tbf = new ThreadedBehaviourFactory();
Boolean resource_busy = (Boolean) this.getDataStore().get("resource_busy");
if(resource_busy == null){
  resource_busy = false;
}
if(resource_busy==false){
  System.out.println("Stack buffer at start: ");
  for(int i: stack_buffer.keySet() ){
    String order_id = stack_buffer.get(i).getID();
    String prod_id = stack_buffer.get(i).getProdType();
    System.out.println(i + " -> " + order_id + " / " + prod_id);
  }
  this.getDataStore().put("resource_busy", true);
  responder = new SSIteratedAchieveREResponder(this.myAgent,req_msg);
  //get keys for DataStore entries
  String req_key = responder.REQUEST_KEY;
  String reply_key = responder.REPLY_KEY;
  //put received request message in DataStore
  responder.getDataStore().put(responder.REQUEST_KEY, req_msg);
  //construct arguments object to pass to execute behaviour
  Object[] args1 = new Object[5];
  args1[0] = responder.getDataStore();
  args1[1] = req_key;
  args1[2] = reply_key;
  args1[3] = this.getDataStore();
  args1[4] = stack_buffer;
```

```
              //instantiate a new behaviour instance to handle the received request message;
              responder.registerHandleRequest(setBehavName("Resource_execution.Execute_" + service_requested,args1));
              //construct and send AGREE to requesting holon
              ACLMessage agree = req_msg.createReply();
              responder.sendAgree(agree);
            }
            else{
              //resource is busy and FAILURE must be replied to START request
              responder = new SSIteratedAchieveREResponder(this.myAgent,req_msg){
                protected ACLMessage handleRequest(ACLMessage req_msg){
                  ACLMessage refuse = req_msg.createReply();
                  refuse.setPerformative(ACLMessage.FAILURE);
                  return refuse;
                }
              };
            }
            //close/terminate behaviour when the current session ends
            responder.closeSessionOnNextReply();
            return tbf.wrap(responder);
        }
    });
}
public FSMBehaviour setBehavName(String className,Object args){
    FSMBehaviour b = new FSMBehaviour();
    try {
        Class[] carg = new Class[1];
        carg[0] = Object[].class;
        ExecuteBehaviourMethods instance = new ExecuteBehaviourMethods();
        Method meth = ExecuteBehaviourMethods.class.getDeclaredMethod("execute_" + service_requested, carg);
```

```
                b = (FSMBehaviour) meth.invoke(instance,args);
            } catch (Throwable e) {
                e.printStackTrace();
            }
            return b;
        }

        protected void takeDown(){
            //deregister from DF
            try { DFService.deregister(this); }
            catch (Exception e) {}
            //send cancellation messages to all booked agents
            ACLMessage cancel = new ACLMessage(ACLMessage.REQUEST);
            cancel.setContent("cancel");
            for(int i = 0; i < booking_list.size(); i++){
                cancel.addReceiver(booking_list.get(i));
            }
            send(cancel);
        }
    }
```

A.2.2. Execution Behaviour FSM

```
package Resource_execution;
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;

public class ExecuteBehaviourMethods extends Agent{
  //private AchieveREResponder start_responder;
  private  DataStore ds;
```

```
    private String req_key;
    private String result_key;
    private ACLMessage response;
    public FSMBehaviour execute_test(Object[] args){
      ds = (DataStore) args[0];
      req_key = (String) args[1];
      result_key = (String) args[2];
      Execute_test exec = new Execute_test();
      exec.var_init(args);
      FSMBehaviour b = new FSMBehaviour();
      b.setDataStore(ds);
      b.registerFirstState(exec.new ReleaseRequestor(exec, response, new DataStore()), "ReleaseRequestor");
      b.registerState(exec.new Execute(), "Execute");
      b.registerState(exec.new BindingRequestor(exec, response, new DataStore()), "BindingRequestor");
      b.registerLastState(exec.new Done(exec, response, ds), "Done");
      b.registerTransition("ReleaseRequestor", "Execute",1);
      b.registerTransition("Execute", "BindingRequestor",2);
      b.registerDefaultTransition("BindingRequestor", "Done");
      return b;
    }
  }
```

A.2.3. Execution Behaviour

```
package Resource_execution;
import java.util.ArrayList;
...
import jade.util.leap.Set;

public class Execute_test extends Agent{
  private DataStore ds;
  private DataStore disp_ds;
  private String req_key;
  private String result_key;
  public void var_init(Object[] args){
    ds = (DataStore) args[0];
    req_key = (String) args[1];
    result_key = (String) args[2];
    disp_ds = (DataStore) args[3];
  }
  public class ReleaseRequestor extends AchieveREInitiator{
    public ReleaseRequestor(Agent a, ACLMessage req_msg, DataStore ds1) {
       super(a, req_msg, ds1);
    }
    public Vector prepareRequests(ACLMessage msg){
       //indicate event with performance logger
       ACLMessage sm = new ACLMessage(ACLMessage.INFORM);
       sm.addReceiver(new AID("PerformanceLogger",AID.ISLOCALNAME));
       sm.setContent("start");
       myAgent.send(sm);
       printDS("prepReqs DS",parent.getDataStore());
       printDS("prepReqs DS1",getDataStore());
       AID[] service_providers = findServiceProviders("transport");
       ACLMessage req_msg = new ACLMessage(ACLMessage.REQUEST);
       //build XML message content
       XML message_type = XmlTools.buildXmlElement("message_type", "release_request");
       ACLMessage start_req = (ACLMessage)parent.getDataStore().get(req_key);
       XML task_ref = XmlTools.buildXmlElement("task_ref", start_req.getSender().getLocalName());
       ArrayList<XML> info_elements = new ArrayList<XML>();
       info_elements.add(task_ref);
```

```
        XML info = XmlTools.buildXmlElement("info", info_elements);
        ArrayList<XML> msg_elements = new ArrayList<XML>();
        msg_elements.add(message_type);
        msg_elements.add(info);
        XML msg_info = XmlTools.buildXmlElement("msg", msg_elements);
        XML initiator = XmlTools.buildXmlElement("initiator", this.getAgent().getLocalName());
        XML responder = XmlTools.buildXmlElement("responder", "TransportAgent");
        ArrayList<XML> message_elements = new ArrayList<XML>();
        message_elements.add(initiator);
        message_elements.add(responder);
        message_elements.add(msg_info);
        XML message = XmlTools.buildXmlElement("message", message_elements);
        XMLDocument xmlDoc = XmlTools.buildXmlDoc(message);
        String xmlMsg = xmlDoc.toString();
        req_msg.setContent(xmlMsg);
        req_msg.addReceiver(service_providers[0]);
        Vector messages = new Vector();
        messages.add(req_msg);
        return messages;
    }
    public void handleAgree(ACLMessage agree_msg){
        System.out.println(myAgent.getName() + " received AGREE from " + agree_msg.getSender().getName() + " during
    > AchieveRE: " + agree_msg);
    }
    public void handleFailure(ACLMessage fail_msg){
        System.out.println(myAgent.getName() + " received FAILURE from " + fail_msg.getSender().getName() + " during
    > AchieveRE: " + fail_msg);
        try {
           Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        //reset variables and AchieveREInitiator behaviour
        reset();
    }
    public void handleInform(ACLMessage inform_msg){
        printDS("inform DS1",getDataStore());
        System.out.println(myAgent.getName() + " received Inform from " + inform_msg.getSender().getName() + " with
    >   content: " + inform_msg.getContent());
    }
    public int onEnd(){
        return 1;
    }
    public AID[] findServiceProviders(String service_type){
        int i;
        AID[] service_providers = null;
        DFAgentDescription template = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType(service_type);
        template.addServices(sd);
        try {
          DFAgentDescription[] result = DFService.search(this.myAgent, template);
          service_providers = new AID[result.length];
          if(service_providers.length != 0){
             System.out.println("Found the Resource agents:");
             for (i=0;i < result.length;i++) {
                                             service_providers[i] = result[i].getName();
                                             System.out.println(result[i].getName());
             }
          }
          else{
             System.out.println("Did not find any Resource agents providing the service: " + service_type);
          }
       }
       catch (FIPAException fe) {
          fe.printStackTrace();
       }
       return service_providers;
    }
    public void printDS(String name, DataStore ds){
       Set keys = ds.keySet();
       Iterator it = keys.iterator();
       while(it.hasNext()){
          String key = it.next().toString();
          try{
             String value = ds.get(it.next()).toString();
          }
          catch(NoSuchElementException e){
             System.out.println("No element found");
          }
          catch(NullPointerException e){
             System.out.println(key + " = Element null");
          }
       }
    }
 }
 public class Execute extends Behaviour{
    Boolean started = false;
    Boolean done = false;
    public void action() {
       do_test();
       done = true;
    }
    public void do_test(){
       try {
          Thread.sleep(15000);
       } catch (InterruptedException e) {
          e.printStackTrace();
       }
    }
    public boolean done() {
       if(!done){
          return false;
       }
       else{
          return true;
       }
    }
    public int onEnd(){
       return 2;
    }
  }
  public class BindingRequestor extends AchieveREInitiator{
    public BindingRequestor(Agent a, ACLMessage req_msg, DataStore ds1) {
       super(a, req_msg, ds1);
    }
    public Vector prepareRequests(ACLMessage msg){
       printDS("prepReqs DS",parent.getDataStore());
```

```
        printDS("prepReqs DS1",getDataStore());
        AID[] service_providers = findServiceProviders("transport");
        ACLMessage req_msg = new ACLMessage(ACLMessage.REQUEST);
        //build XML message content
        XML message_type = XmlTools.buildXmlElement("message_type", "binding_request");
        //get original start request message from task holon
        ACLMessage start_req = (ACLMessage)parent.getDataStore().get(req_key);
        XML task_ref = XmlTools.buildXmlElement("task_ref", start_req.getSender().getLocalName());
        XML prod_id = XmlTools.buildXmlElement("prod_ID", "p01");
        XML binding_location = XmlTools.buildXmlElement("binding_location", myAgent.getLocalName()+"_output");
        ArrayList<XML> info_elements = new ArrayList<XML>();
        info_elements.add(task_ref);
        info_elements.add(prod_id);
        info_elements.add(binding_location);
        XML info = XmlTools.buildXmlElement("info", info_elements);
        ArrayList<XML> msg_elements = new ArrayList<XML>();
        msg_elements.add(message_type);
        msg_elements.add(info);
        XML msg_info = XmlTools.buildXmlElement("msg", msg_elements);
        XML initiator = XmlTools.buildXmlElement("initiator", this.getAgent().getLocalName());
        XML responder = XmlTools.buildXmlElement("responder", "TransportAgent");
        ArrayList<XML> message_elements = new ArrayList<XML>();
        message_elements.add(initiator);
        message_elements.add(responder);
        message_elements.add(msg_info);
        XML message = XmlTools.buildXmlElement("message", message_elements);
        XMLDocument xmlDoc = XmlTools.buildXmlDoc(message);
        String xmlMsg = xmlDoc.toString();
        req_msg.setContent(xmlMsg);
        req_msg.addReceiver(service_providers[0]);
        Vector messages = new Vector();
        messages.add(req_msg);
        return messages;
    }
    public void handleAgree(ACLMessage agree_msg){
        System.out.println(myAgent.getName() + " received AGREE from " + agree_msg.getSender().getName() + " during
    >   AchieveRE: " + agree_msg);
    }
    public void handleFailure(ACLMessage fail_msg){
        System.out.println(myAgent.getName() + " received FAILURE from " + fail_msg.getSender().getName() + " during
    >   AchieveRE: " + fail_msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //reset variables and AchieveREInitiator behaviour
        reset();
    }
    public void handleInform(ACLMessage inform_msg){
        printDS("inform DS1",getDataStore());
        System.out.println(myAgent.getName() + " received Inform from " + inform_msg.getSender().getName() + " with
    >   content: " + inform_msg.getContent());
    }
    public AID[] findServiceProviders(String service_type){
        int i;
        AID[] service_providers = null;
        DFAgentDescription template = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType(service_type);
        template.addServices(sd);
        try {
            DFAgentDescription[] result = DFService.search(this.myAgent, template);
            service_providers = new AID[result.length];
            if(service_providers.length != 0){
               System.out.println("Found the Resource agents:");
               for (i=0;i < result.length;i++) {
                                                      service_providers[i] = result[i].getName();
                                                      System.out.println(result[i].getName());
               }
            }
            else{
               System.out.println("Did not find any Resource agents providing the service: " + service_type);
            }
        }
        catch (FIPAException fe) {
            fe.printStackTrace();
        }
        return service_providers;
    }
    public void printDS(String name, DataStore ds){
        System.out.println(name + " info: ");
        Set keys = ds.keySet();
        Iterator it = keys.iterator();
        while(it.hasNext()){
            String key = it.next().toString();
            try{
               String value = ds.get(it.next()).toString();
               System.out.println(key + " = " + value);
```

```
            }
            catch(NoSuchElementException e){
               System.out.println("No element found");
            }
            catch(NullPointerException e){
               System.out.println(key + " = Element null");
            }
         }
      }
   }
   public class Done extends OneShotBehaviour{
      public ACLMessage result = null;
      public Done(Agent a, ACLMessage req_msg, DataStore ds) {
         super();
      }
      public void action(){
         //indicate event with performance logger
         ACLMessage sm = new ACLMessage(ACLMessage.INFORM);
         sm.addReceiver(new AID("PerformanceLogger",AID.ISLOCALNAME));
         sm.setContent("done");
         myAgent.send(sm);
         System.out.println("Execute_fsm done!");
         //printDS("Done DS", ds);
         printDS("Done DS1", parent.getDataStore());
         //obtain original "start" request message as received by AchieveREResponder behaviour
         ACLMessage start_req = (ACLMessage)parent.getDataStore().get(req_key);
         System.out.println("Got original start request from " + start_req.getSender().getName() + " with content: " +
      >  start_req.getContent());
         //create reply to original request message
         ACLMessage result = start_req.createReply();
```

```
            result.setPerformative(ACLMessage.INFORM);
            result.setContent(start_req.getContent());
            parent.getDataStore().put(result_key, result);
            disp_ds.put("resource_busy", false);
        }
        public int onEnd(){
            printDS("OnEnd DS1", parent.getDataStore());
            return 0;
        }
        public void printDS(String name, DataStore ds){
            System.out.println(name + " info: ");
            Set keys = ds.keySet();
            Iterator it = keys.iterator();
            while(it.hasNext()){
                String key = "no_key";
                try{
                    key = it.next().toString();
                    String value = ds.get(it.next()).toString();
                    System.out.println(key + " = " + value);
                }
                catch(NoSuchElementException e){
                    System.out.println("No element found");
                }
                catch(NullPointerException e){
                    System.out.println(key + " = Element null");
                }
            }
        }
    }
```