ARTI-based Holonic Control Implementation for a Manufacturing System Using the BASE Architecture

by

Alexander Wasserman

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

Supervisor: Dr. Karel Kruger

Co-supervisor: Prof. Anton Basson

April 2022

Declaration

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

April 2022

Abstract

ARTI-based Holonic Control Implementation for a Manufacturing System Using the BASE Architecture

A. Wasserman

Department of Mechanical and Mechatronic Engineering
Stellenbosch University
Private Bag X1, 7602 Matieland, South Africa
Thesis: M.Eng. (Mechatronic Engineering)

April 2022

With industry's drive to adopt Industry 4.0 technologies, and their enabling technologies, in manufacturing processes, intelligent automated manufacturing has become largely sought after. With defining features such as robustness, reconfigurability and scalability, the Holonic Manufacturing Execution System (HMES) approach shows great potential to satisfy Industry 4.0 requirements. Implementations of these systems have been historically known to require great development effort and time. These implementations are however being aided by the development of holonic reference architectures, such as the Product-Resource-Order-Staff-Architecture (PROSA) and its recent revision the Activity-Resource-Type-Instance (ARTI) architecture.

This thesis presents an ARTI-based HMES implementation. The implementation of this system is aided through the use of the Biography-Attributes-Schedule-Execution (BASE) architecture for digital administration shells. The BASE architecture was initially developed as a framework for the development of a digital administration shell for a human worker, in order to elevate the human worker to the level of a Cyber-Physical System. It was however proposed that the BASE architecture also had the potential to be used in a manufacturing context. The possibility of implementing the ARTI-based HMES using the BASE architecture for the respective ARTI holons is confirmed through a mapping of the ARTI architecture to the BASE architecture.

The HMES is implemented on a Fischertechnik Industry 4.0 Training Factory, a small-scale manufacturing system, as a case study system. The complexity of the case study, which comprises several interacting subsystems, provides a good basis for evaluating the ARTI and BASE architectures for HMES development. The

thesis concludes that the ARTI architecture provides a well-defined structure for the conceptual design of HMESs, while the BASE architecture effectively supports the implementation of ARTI-based HMESs with little additional development required.

Uittreksel

ARTI-gebaseerde Holoniese Beheerimplementasie vir 'n Vervaardigingstelsel met die Gebruik van die BASE-Argitektuur

A. Wasserman

Departement van Meganiese en Megatroniese Ingenieurswese
Universiteit Stellenbosch
Privaatsak X1, 7602 Matieland, Suid-Afrika
Tesis: M.Ing. (Megatroniese Ingenieurswese)

April 2022

Met die industrie se strewe om Industrie 4.0-tegnologieë, en hul bemagtigende tegnologieë, in vervaardigingsprosesse aan te neem, het intelligente geoutomatiseerde vervaardiging grootliks gesog geword. Met kenmerke soos robuustheid, herkonfigureerbaarheid en skaleerbaarheid, toon die *Holonic Manufacturing Execution System* (HMES) benadering groot potensiaal om aan Industrie 4.0 vereistes te voldoen. Dit is histories bekend dat die implementering van hierdie stelsels groot ontwikkelingspogings en tyd verg. Hierdie implementerings word egter aangehelp deur die ontwikkeling van holoniese verwysingsargitekture, soos die *Product-Resource-Order-Staff-Architecture* (PROSA) en die onlangse hersiening daarvan die *Activity-Resource-Type-Instance* (ARTI) argitektuur.

Hierdie tesis bied 'n ARTI-gebaseerde HMES-implementering aan. Die implementering van hierdie stelsel is aangehelp deur die gebruik van die *Biography-Attributes-Schedule-Execution* (BASE) argitektuur vir digitale administrasiedoppe. Die BASE argitektuur is aanvanklik ontwikkel as 'n raamwerk vir die ontwikkeling van 'n digitale administrasiedop vir 'n menslike werker, om die menslike werker tot die vlak van 'n *Cyber-Physical System* te verhef. Daar is egter voorgestel dat die BASE argitektuur ook die potensiaal het om in 'n vervaardigingskonteks gebruik te word. Die moontlikheid om die ARTI-gebaseerde HMES te implementeer deur gebruik te maak van die BASE argitektuur vir die onderskeie ARTI-holone, is bevestig deur 'n kartering van die ARTI argitektuur na die BASE argitektuur.

Die HMES is geïmplementeer op 'n *Fischertechnik Industry 4.0 Training Factory*, 'n kleinskaalse vervaardigingstelsel, as 'n gevallestudiestelsel. Die kompleksiteit van die gevallestudie, wat verskeie interaktiewe substelsels bevat, bied 'n goeie basis vir die evaluering van die ARTI en BASE argitekture vir HMES-ontwikkeling. Die tesis bevind dat die ARTI argitektuur 'n goed-gedefinieerde struktuur vir die konseptuele ontwerp van ‘n HMES verskaf, terwyl die BASE argitektuur die implementering van ‘n ARTI-gebaseerde HMES effektief ondersteun, met min bykomende ontwikkeling wat nodig is.

Acknowledgements

Firstly, I have to thank Dr. Karel Kruger and Prof. Anton Basson for the many hours of sitting and conceptually discussing the details of the implementation in this thesis and attempting to piece it all together. Thank you for asking the questions that challenged my way of thinking about the problem at hand, and for guiding me to find my own answers to questions that I couldn't think of answering on my own. I appreciate all of your time and effort that you have dedicated to ensuring that my thesis is of the best possible quality.

Aan my Pa, baie dankie vir al jou ondersteuning deur die jare, en dat jy altyd in my geglo het, veral wanneer ek moeg en moedeloos was en nie meer in myself geglo het nie. Baie dankie vir al die finansiele ondersteuning deur my studies; ek weet dit was lank, maar ek waardeur dit meer as wat jy weet. Aan my Ma, baie dankie vir jou ondersteuning, liefde, en lang gesprekke waar ek probeer verduidelik wat ek doen, en toevallig my werk beter leer ken in die process. Ek waardeur al die kospakkies wat jy oor die jare by my kom aflewer het, dit het my deur 'n hele paar lang nagte gebring. Aan my broer Marco, dankie vir al die lang gesprekke waar ek kla oor my werk, en dan na 'n ruk gee jy my die beste raad: "Dis eenvoudig. Doen dit net…" Ek waardeur al jou ondersteuning Marco.

Holly, thank you so much for your endless support. Thank you for listening when I need to let it out, and for always trying to help me solve my problems. There was more than once where your unique outlook solved a complex problem with a simple solution. I appreciate everything that you have done for me, and I couldn't have done it without you.

Table of Contents

List of Figures

Page

List of Tables

Page

List of Abbreviations

3SAL	Three-Stage Activity Lifecycle
AAI	Aggregate Activity Instance
AAT	Aggregate Activity Type
ADACOR	ADAptive holonic Control aRchitecture
AI	Activity Instance
AP	Analysis Plugin
API	Application Programming Interface
ARI	Aggregate Resource Instance
ART	Aggregate Resource Type
ARTI	Activity-Resource-Type-Instance
AT	Activity Type
BASE	Biography-Attributes-Schedule-Execution
BC	Business Card
CM	Communication Manager
CNP	Contract Net Protocol
CPS	Cyber Physical System
DMAS	Delegate Multi-Agent System
DOHA	Department of Holon Affairs
DSI	Input Station
DSO	Output Station
DT	Digital Twin
EP	Execution Plugin
FSM	Finite State Machine

GPIO	General Purpose Input Output
HBW	High-Bay Warehouse
HMES	Holonic Manufacturing Execution System
HMS	Holonic Manufacturing System
HRH	Human Resource Holon
IA	Intelligent Agent
IB	Intelligent Being
IoT	Internet of Things
I4.0	Industry 4.0
MPO	Multi-Processing Station
NEU	Next-Execute-Update
NFC	Near-Field Communication
PID	Process Identifier
PKS	Packaging Station
PLA	Production Line Aggregate
PROSA	Product-Resource-Order-Staff-Architecture
QoS	Quality of Service
RAMI4.0	Reference Architecture Model for Industry 4.0
RFP	Request for Proposal
RI	Resource Instance
RP	Reflection Plugin
RT	Resource Type
SLD	Sorting Line Station
SP	Scheduling Plugin
UID	Unique ID

VGR	Vacuum Gripper Robot
WOI	World of Interest
WP	Workpiece

1 Introduction

The background to this thesis is given in this introductory chapter, along with the context wherein the research took place. The objectives are then discussed, followed by the motivation for carrying out the research, and then finally the methodology of how the research was conducted.

1.1 Background

With the Digital Revolution having reached its goal of the digitization and automation of industry, the Fourth Industrial Revolution, known as Industry 4.0 (I4.0), has commenced with the goal of advancing these digital systems to become 'smart' systems. Since I4.0 intends to improve many industries by using a variety of digital means, it relies on the rapid development and implementation of digital systems. This is typically done by either refitting existing manufacturing systems or by designing new manufacturing systems from the ground up to integrate I4.0 principles and technologies. For this to be done, two things need to be known first: what exactly an Industry 4.0 system should be capable of, and how would these systems be deployed on a large scale. The first question has been comprehensively detailed by the Deutsche Kommission Elektrotechnik in their paper titled*: German Standardization Roadmap for Industry 4.0* (Adolph, Anlahr, Bedenbender, *et al.*, 2016). The second question is currently being worked on by multiple researchers around the world; creating frameworks and architectures for the standardization of I4.0 systems.

I4.0 is being implemented through the use of various new and older enabling technologies, one of which are Cyber Physical Systems (CPS). "CPSs are automated systems that enable connection of the operations of the physical reality with computing and communication infrastructures" (Bigliardi, et al. 2020). In essence, these are systems that can link the physical world with the digital world, in order for the physical system to benefit from the enhanced information that it can receive from the digital system, and vice-versa. Ideally, a CPS would be created in such a way that the digital and physical systems are in a symbiotic relationship, where each one benefits the other.

An example of a CPS is the notion of a Digital Twin (DT). A DT is a digital representation of a physical entity in cyber space, with which it has bidirectional communication. The physical twin would typically contain sufficient sensors to fully represent its state in the digital world. Bidirectional communication allows the DT to create detailed models and/or simulations of the physical twin using the data from these sensors. The DT could then analyse this modelled/simulated data for use in an advanced control system for the physical twin. With the use of various computational techniques on the collected data, these DT systems could become invaluable in industry (Jones, Snider, Nassehi, *et al.*, 2020).

Alongside I4.0 and its enabling technologies, the development of Holonic Manufacturing Systems (HMSs) has been rapidly advancing. They have also been seeing increasing deployment since the pinnacle of the Digital Revolution. A HMS is a "highly distributed control paradigm" (Van Brussel, Wyns, Valckenaers, *et al.*, 1998) that consists of a system of holons that work together to accomplish tasks within a manufacturing system. In the context of this research, a *holon* refers to a self-contained component of a system that is autonomous in itself, but cooperates with other holons in order to execute tasks to achieve a common goal (Kruger & Basson, 2017). Together this system of holons is called a holarchy, with each holon cooperating to form a network of functionalities and behaviours that can carry out complex manufacturing and control tasks (Kruger & Basson, 2018).

Closely related to HMSs are Holonic Manufacturing Execution Systems (HMESs). HMSs are regularly confused with HMESs and often the terms are used interchangeably. "The wording *manufacturing control* commonly denotes the task performed by a manufacturing execution system" (Valckenaers & Van Brussel, 2005). HMESs are essentially the high-level control systems used for the control of process control systems. This can include tasks such as product route planning, process step execution, process scheduling and reactive response to system changes. In this thesis, the term HMES will be used for these types of systems.

HMESs offer many advantages, but the biggest advantages of such systems are: they are relatively easily reconfigurable in order to adapt to changing manufacturing cycles; they are robust in design, still functioning if one or multiple holons fail; and they offer reduced software development costs as system complexity is reduced.

Research in this field has made the development and implementation of HMESs more accessible due to the creation of holonic reference frameworks for manufacturing systems. Two of the most influential frameworks are the Product-Resource-Order-Staff-Architecture (PROSA) (Van Brussel *et al.*, 1998), and the Activity-Resource-Type-Instance (ARTI) architecture (Valckenaers, 2020). Essentially, PROSA splits the HMES into four holons with differing functionalities: Resource, Product, Order, and Staff holons. ARTI is an improvement on PROSA, which also splits the HMES into four holons: Activity Type and Activity Instance holons, and Resource Type and Resource Instance holons. The largest changes, besides the holon types, are that the ARTI architecture has improved upon the terminology of PROSA in order to make the architecture more accessible and applicable within different industries.

Closely related to the ARTI framework is the Biography-Attributes-Schedule-Execution (BASE) architecture (Sparrow, Kruger & Basson, 2021). In Sparrow's paper (Sparrow *et al.*, 2021) an architecture for a digital administration shell is developed, for use in a Human Resource Holon (HRH). The BASE architecture splits the holon into four core components: Biography, Attributes, Schedule, and Execution components. Upon inspection, close relations can be seen between the BASE architecture and the ARTI reference framework, indicating that the BASE

architecture could potentially be used in an implementation of the ARTI reference architecture for an HMES. In literature there are very few implementations of the ARTI reference framework, especially for the development of an HMES, making research in this field worthwhile. This thesis thus aims to develop an ARTI-based HMES implementation for a case study system, using the BASE architecture for the development of the respective holons.

1.2 Objectives

The objectives of this thesis are:

- The development of an ARTI-based HMES implementation for a case study system and the evaluation thereof.
- The demonstration and evaluation of the BASE architecture's suitability for use in a manufacturing environment through the implementation of an ARTI-based HMES using the BASE architecture as a development building block.
- To evaluate to what extent the ARTI and BASE architectures assist developers with the development of HMESs.

The scope of the thesis is limited to the development of the HMES for the case study system and the evaluation thereof and does not include further development of the BASE architecture, ARTI architecture or development of case study system features that are not essential for evaluation.

1.3 Motivation

With a variety of well-established manufacturing control paradigms available for use, such as hierarchical manufacturing execution systems, heterarchical manufacturing systems and HMESs, it was decided to research holonic systems due to their wide range of functionalities and advantages.

Due to the nature of a holonic control system having distributed control, the system is very resilient to disturbances, as well as adaptable in response to issues within the system (Kruger & Basson, 2017). Another advantage of the distributed nature of holonic systems is that they are easily scalable as the communication network between holons could easily allow for more holons to be added to the system. Holonic control systems, such as HMESs, also have a reduced system complexity when compared to other types of control, as their functionalities are carefully split into different holons with strict communication protocols between them. Complex systems can thus be broken down into many simpler parts, which reduces the development complexity. This reduced complexity leads to reduced software development costs, improved maintainability, and improved reliability.

Additionally, another advantage of holonic control systems is that holons are by definition autonomous, meaning that it would be possible to create a holonic control system on several different platforms and networks, so long as the communication protocol used between the holons is universal. This once again leads to lower development costs, improved compatibility, as well as allowing these holonic control systems to be implemented within pre-existing manufacturing systems that use different platforms for different devices.

Using an existing architecture for the implementation of an HMES is advantageous as it reduces the complexity and cost of software development. It also ensures that all required functionalities are present within the completed system by guiding the developer through the theoretical system design. For these reasons it was decided to use an architecture for the implementation of a HMES.

The ARTI reference architecture is a new architecture built on mature concepts, with great potential for large-scale market adoption. Having learnt lessons from the PROSA architecture, which was very influential, ARTI was developed with the intention of improving the language used in the terminology in order to make the architecture more universal for use in other industries, besides manufacturing. There also exist few real-world implementations of the ARTI architecture, which would make a case study implementation thereof valuable research. Considering that ARTI is a relatively new architecture, with few implementations, it would also be of value to determine to what extent the ARTI architecture assists with the development of a holonic system.

The BASE architecture was developed to be a modular, vendor neutral, adaptable and generic architecture which fulfils all of the roles of a resource holon (Sparrow *et al.*, 2021). The architecture is easily scalable as it is based on holonic systems principals, meaning that multiple digital administration shells, implemented as holons, could form an HMES. Its original intended use was as a digital administration shell for a Human Resource Holon (HRH) in a CPS, but it is believed that the BASE architecture could be used for other entities and processes in the manufacturing industry, and that it could form the basis for all holons in an ARTI HMES implementation. BASE is a generic architecture, which allows for customization and optimization plugin components. This also allows for the architecture to be further developed in any language and implemented on any, and multiple, platforms. The BASE architecture has many value-adding features and strong advantages, which motivates that it would be beneficial to investigate whether or not it would be a suitable architecture for the basis of the development of an HMES.

Overall, it would be of great value to obtain quantifiable metrics with which the effect on development of using the ARTI and BASE architectures for the implementation of HMESs can be measured. The case study in this thesis would provide a good testbed with which these metrics could be obtained.

1.4 Methodology

This section provides a discussion on the methodology used during this thesis, which covers key decisions made regarding approach, implementation, and evaluation. Before any system development could take place, it was necessary to first conduct a literature review to examine what reference architectures exist in literature and how they relate to each other, and then also to see what, if any, implementations of these architectures exist. This information was used to assist with the decision of choosing the ARTI architecture for implementation, as well as using the BASE architecture for the implementation thereof. These implementations from literature, as well as supplementary literature, were investigated to ascertain what criteria is commonly used in the testing and evaluation of HMESs.

The second step followed for this research was to investigate whether it would be feasible to use the BASE architecture for an ARTI implementation and whether BASE would be suitable for the manufacturing environment. This was done through mapping the BASE architecture to the ARTI reference architecture. This entailed comparing the structure and functionalities of the BASE architecture to those of the ARTI framework and determining if BASE was capable of performing these functions, or if BASE would need to be modified. This comparison served as a basis to make a well-informed decision on whether or not it would be feasible to use the BASE architecture for an ARTI-based HMES implementation.

The third step was making the decision, and motivating the decision, to use a case study approach, which would allow for meaningful results for the objectives of this thesis. It was then necessary to select and define a case study system. This involved selecting a case study system and investigating its system components and functionalities, in order to ensure that the system would provide an accurate reflection of what real-life systems might be like. A systems design approach was then followed to develop the system architecture for the case study, while following the ARTI architecture for holon design. This involved the development of system requirements and the development of the system holarchy with functions and features dedicated to their respective holons. After the system architecture was developed it was necessary to implement the case study system using BASE as a basis for the development of all the respective holons.

Once the system had been implemented it had to be thoroughly evaluated. In order for the evaluation to have value, suitable evaluation criteria were formulated. This evaluation criteria had to evaluate all aspects related to the system requirements in order to provide scientifically meaningful results. It was also of importance to select evaluation criteria that were not only qualitative, but also quantitative. This would allow for the degree to which the thesis objectives are met to be quantified in a meaningful manner. From these evaluation criteria, a set of experiments were designed to test the case study system to evaluate it. This methodology is based-on previous work from literature, which had similar objectives.

2 Literature Review

For the purpose of gaining insight into the process of developing an HMES implementation, the current state of industry, holonic systems, and reference architectures are investigated in this chapter through a literature-based review. In Section 2.1 an overview of the manufacturing industry's current state is given, including technologies that are currently in use in industry. Section 2.2 covers holonic manufacturing and its adoption in industry. This includes HMESs, HMES evaluations and holonic reference architectures. Section 2.3 looks at the ARTI reference architecture, and Section 2.4 looks at the BASE architecture. Finally, conclusions will be drawn from the literature review in Section 2.5.

2.1 State of Industry

2.1.1 Overview

The world has undergone three industrial revolutions and is currently undergoing the fourth industrial revolution. The first industrial revolution period was marked with the evolution of water and steam-powered mechanical production plants, and the second was the period of major technological innovations such as electricity and electronic mechanisms and was also a "period when industrial plants burgeoned both in volume and variety" (Yin, Stecke & Li, 2018). The third industrial revolution, which is still ongoing in some respects, has been about the use of electronics and Information Technologies in production. Automation and digitization within the Third Industrial Revolution paved the way for the kick-off of I4.0 (Bigliardi *et al.*, 2020).

The first concrete concept of I4.0 materialized in the paper *German Standardization Roadmap for Industry 4.0* (Adolph *et al.*, 2016). This paper introduced the notion of a "digitally integrated industry" (Galati & Bigliardi, 2019) where digital systems that were incorporated into physical systems from the third Industrial Revolution are superseded by more 'intelligent' systems which are capable of communicating with other systems as well as the outside world, connected to many different services such as data analytics and advanced control systems, in order to enable these systems to become 'smart' so as to increase their capabilities. This would be accomplished through the use of various enabling technologies, which will be discussed in section 2.1.3, but the main goals were: to improve efficiency in production; improve quality; increase output; reduce costs; and to optimize value chains (Galati & Bigliardi, 2019).

I4.0 has been a widely researched topic since its inception, which has led to the continuing development of I4.0 enabling technologies. Some of these enabling technologies include: the Internet of Things (IoT), Big Data Analytics, Cloud-Computing, Internet of Service, Augmented Reality, Cyber-Physical Systems (CPSs), Holonics and advanced Simulation and Modelling (Bigliardi *et al.*, 2020).

In many cases, multiple of these enabling technologies would need to be integrated together in order to make up the functionality needed to implement an I4.0 system, which has made the development of I4.0 systems challenging. This has led to an increase in research towards developing reference architectures for I4.0 systems, which would help with the implementation of mixed technology systems. One of the most notable such architectures is the Reference Architecture Model for Industry 4.0 (RAMI4.0) (Adolphs, Bebenbender, Dirzus, *et al.*, 2015).

2.1.2 Challenges

The development of I4.0 systems has been greatly hindered by high development difficulties (Leitão, Colombo & Karnouskos, 2016; McKinsey Digital, 2016; Raj, Dwivedi, Sharma, *et al.*, 2020; Schuh, Anderl, Gausemeier, *et al.*, 2017); however, architectures have greatly reduced this associated difficulty. There are however still many other significant barriers to entry. As noted in *Industrie 4.0 Maturity Index* (Schuh, Anderl, Gausemeier, *et al.*, 2017), one of the major obstacles to the introduction of I4.0 to many enterprises, especially SMEs, is that of companies not fully comprehending the benefits of I4.0 technologies. Schuh *et al.*, (2017) goes on to say that, in 2017, I4.0 "would appear to be a long way off in the industrial sector." They also mention that most of the I4.0 systems that were implemented at that time were technological feasibility studies, where enterprises were testing to see whether I4.0 technologies could indeed aid their processes. These studies often resulted in unfavourable results as vital parts of the implementation of I4.0 were overlooked, such as the enterprise's organisational structure and culture towards new technology (Schuh *et al.*, 2017).

Raj, Dwivedi, Sharma, *et al.* (2020) revealed through a survey that enterprise CEOs are still not confident in their firm's abilities to introduce I4.0 technologies. In a differing, but also somewhat similar view, Bigliardi *et al.* (2020) suggests that the research of I4.0 might be nearing the point of maturation. This, however, indicates that I4.0 has not yet reached the required maturity for the mass adoption of I4.0 technologies within industry. In terms of making actual progress in the implementation of I4.0 technologies, a survey from 2016 found that up to 40% of surveyed companies made very little progress in the previous year (McKinsey Digital, 2016). It is suggested that this is due to various development difficulties, such as development time and cost (McKinsey Digital, 2016; Raj *et al.*, 2020; Schuh *et al.*, 2017). This did, however, vary significantly by region, which might suggest that there are other economical and/or socio-political factors involved. Even though the adoption of I4.0 technologies has increased in the past few years, it can be seen that there are still significant barriers limiting enterprises from taking full advantage of I4.0.

I4.0 enabling technologies are generally well developed and matured technologies, but when they need to be used in conjunction with one another, and then integrated with people, problems and difficulties arise. Raj *et al.* (2020) mentioned that some of the major factors that limit I4.0 from being adopted broadly are high

implementation and development costs, fears of data security, low standardization between I4.0 technologies, poor understanding of I4.0 advantages and applications, a lack of a skilled workforce that can work with the I4.0 technologies and also fears of these technologies causing jobs to be lost. As previously discussed, Schuh *et al.*, (2017) also suggested that the organizational structure of an enterprise, as well as people's attitudes towards new technologies have been a large limiting factor. McKinsey Digital (2016) stated that, in addition to the previously mentioned barriers, there are large concerns about data ownership when using third-party hardware/software, there is an uncertainty about the opportunities available to enterprises to implement I4.0, and that there are large complications with the integration of data acquired from different sources to be used together for I4.0 applications.

2.1.3 Enabling Technologies

The concept of HMSs has been a popular field of study since the early 1990's for a variety of reasons, but primarily due to the abundance of advantages to the use of holonic systems for manufacturing control (Kruger & Basson, 2018). A HMS's holons work together to accomplish a common goal/task within a manufacturing system, much like machines on a manufacturing shop-floor. Holons can be seen as "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 & Sacile, 2005). Although the holons work together, they are individually autonomous and possess the ability to carry out multiple functions or behaviours. Due to the nature of holonic systems relying heavily on digital communication and cooperation, and their popularity in the manufacturing industry, they are considered a key I4.0 enabling technology. HMSs and their reference architectures are further investigated in section 2.2. Closely related to HMSs, and another key I4.0 enabling technology are CPSs.

Lee *et al.*, (2015) define a CPS as a set of technologies that manage communication and connection between a physical entity and its computational capabilities. It goes on to describe that CPSs are one of the leading enabling technologies of I4.0 as, if they are implemented correctly, they can provide immense value and information using existing networks, sensors, and computational capabilities to leverage Big Data to transform production industries into I4.0 facilities with great potential (Lee *et al.*, 2015).

A DT is very closely related to a CPS. First introduced in the paper *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems* (Grieves & Vickers, 2017), DTs are close relatives of CPSs in that they possess all attributes of CPSs and then also have a few added functionalities. Where a CPS would simply be the system of communication and interaction between a physical entity and computational capabilities, a DT takes that further in that a DT receives data from the physical entity (the physical twin) and then, using various computational means, it turns this data into valuable information. DTs typically

provide real-time modelling of the physical twin, as well as potentially provide data-driven simulations of future states of the physical twin. This information can then either be used for control purposes or used for further data analytics. Jones *et al.* (2020) corroborate this definition of a DT by going on to define a DT as "consisting of three components, a physical product, a virtual representation of that product, and the bi-directional data connections that feed data from the physical to the virtual representation, and information and processes from the virtual representation to the physical". The DT has been described as a multi-role, hugely versatile technology as it can be developed to carry out many functions. In Jones *et al.'s* (2020) paper a large number of these possible benefits that DTs bring to industry are listed. Some of these include: "reducing costs, risk and design time, complexity and reconfiguration time; improving after-sales service, efficiency, maintenance decision making, security, safety and reliability, manufacturing management, processes and tools; [and] enhancing flexibility and competitiveness of manufacturing system[s]".

The development of DTs is possibly one of the largest barriers to their widespread adoption. As described in both Zheng & Sivabalan (2020) and Redelinghuys *et al.*'s (2019) papers, the development of DTs typically relies on developers separating the required functionalities of the DT into separate 'layers' in which each layer characteristically addresses one critical function. These layers then communicate and interact with one another, passing data and information back and forth, in order to realise the full capabilities of the DT. This makes development easier as only one layer needs to be focused upon at a time.

2.2 Holonic Manufacturing Systems

Considering the large role that HMSs have played in manufacturing, and the large role they are expected to play in the future of manufacturing, this section will investigate them further.

2.2.1 Overview

HMSs have attempted to bridge the gap between automated manufacturing systems and smart manufacturing systems. Researchers have concentrated on addressing the problem of manufacturing control for many years, and HMSs partly address this problem. In the paper *Industry 4.0: contributions of holonic manufacturing control architectures and future challenges,* Derigent, *et al.* (2020) detail a range of key enablers for I4.0, namely: sustainability, secure communication, real-time capabilities, process virtualisation, service orientation, interoperability, adaptability, data analysis, autonomous and decentralized decision support systems and connectivity. HMSs satisfy many of these key enablers; "autonomous and decentralized decision support systems" is satisfied by a holon's ability for making its own decisions as well as its ability to be autonomous; "adaptability" is directly satisfied by a holon's need to be adaptable to its surrounding environment; "real-time capabilities" is satisfied by a holon being reactive and adapting to changes in

the system and its environment; and "connectivity" is satisfied by a holon's ability to cooperate with other holons in the system in order to complete a goal. The fact that these key enablers, amongst others, are satisfied by holonic systems shows that HMSs are a key I4.0 enabling technology and could play a large role in the future of the manufacturing industry. Due to this value of HMSs, researchers have developed multiple reference architectures for HMSs in order to reduce the difficulty of implementing these systems.

2.2.2 Holonic Manufacturing Execution Systems

HMESs differ from HMSs in that HMS refers to the entire holonic system, including software and hardware, whereas HMES refers to the software that manages the execution of the hardware. Valckenaers (2020) likened an HMES to a computer operating system, as the operating system manages and controls the execution of programmes on a computer's hardware, so does an HMES control the execution of tasks in a factory. The term HMS is however often used interchangeably with HMES in literature. HMESs ensure that certain process plans are executed correctly and timeously. This is done by managing manufacturing resources' workloads and schedules, which allows for the planning of product routing through production lines as well as the prediction of lead times. Execution control also comprises of the handling of process outcomes, whether those are successes or failures (Valckenaers & Van Brussel, 2005).

The concept of HMESs has been a popular field of research for a variety of reasons, but primarily due to the abundance of advantages to the use of holonic systems for manufacturing control. Some of these advantages are: HMESs organise production tasks in such a manner that they become scalable and robust, still functioning if holons fail (Kotak, Wu, Fleetwood, *et al.*, 2003); the organization of HMES holarchies promote lower system complexity, improved maintainability and reliability, which leads to reduced development costs (Scholz-Reiter & Freitag, 2007); and HMESs are resilient to system faults and errors, as they are highly adaptable to system disturbances (Vyatkin, 2015). The biggest advantage of HMESs is, however, that they are relatively easily reconfigurable to a changing manufacturing cycle, as they are modular in nature, allowing for only certain components to be changed while not affecting the others. One major disadvantage to using HMESs, however, is that they require a larger computational effort than traditional centralized manufacturing control systems (Valckenaers & Van Brussel, 2005).

Although there have only been a few successful implementations of HMESs in industry (Almeida, Terra, Dias, *et al.*, 2010), there have been many successful HMES implementations in research literature. Many of these successful implementations followed a common methodology for the development of their systems: first the facilitation of communications was developed and implemented, including inter and intra-holon communications; and then the individual holon functional components were identified and implemented. The majority of these

implementations were done using the Multi-Agent System approach using either the Java Agent Development framework, the object orientated approach using C#, and more recently an approach using Erlang (Kruger & Basson, 2017; Leitão & Restivo, 2008).

2.2.3 HMES Requirements

The requirements for a HMES are largely dependent on the specific use case of the system and are thus fairly broad. There are however certain requirements which are commonly found in literature, where HMES implementations were developed. These identified common requirements could be seen as the fundamental requirements of an HMES and are shown in Table 1.

Table 1: Commonly Identified HMES Requirements

Requirement	Description	References
Reconfigurability	Handle additions/removals/changes of hardware or software to/from the HMES. This would lead to the minimization of system down-time. This can also include maintenance-related system modifications.	(Bi, Lang, Shen, *et al.*, 2008; Kruger & Basson, 2019; Leitão & Restivo, 2008)
Robustness	Handle unplanned system disturbances, such as faults or errors. The goal is for the system to remain available for production.	(Kruger & Basson, 2019; Leitão & Restivo, 2008)
Reactive	React and respond timeously to system inputs, unexpected disruptions, faults, and errors.	(Giret & Botti, 2006; Kruger & Basson, 2019; Leitão & Restivo, 2008)
Intelligence	Acquire information from surroundings and convert this information into data which can be acted upon.	(Giret & Botti, 2006; Kruger & Basson, 2019; Leitão & Restivo, 2008)
Autonomous	Operate without user interventions, and capability for products to drive their own production.	(Giret & Botti, 2005; Kruger & Basson, 2017)
Collaborative	Communicate and collaborate with other elements of the system to perform certain tasks.	(Giret & Botti, 2006; Kruger & Basson, 2019)

2.2.4 HMES Evaluation

The evaluation of HMESs is a complex and highly subjective matter as each HMES differs from others, with different goals, functionalities, and emphasis on different performance metrics. It was however found that there were performance metrics that were common to many implementations. The most common performance metric that was measured during the evaluation of HMESs was that of *throughput* (Kruger & Basson, 2017; Leitão & Restivo, 2008; Valckenaers & Van Brussel, 2005). This metric measured the number of products that the HMES could produce per specified time period. This is a highly universal metric as most HMES implementations have the common goal of producing a certain product in the shortest duration possible, and it is also a highly valuable metric, as throughput is the primary goal of the factory. Other metrics that were found to be commonly measured, and that are closely related to throughput, are *lead times* and *resource utilization rates* (Kruger & Basson, 2017; Leitão & Restivo, 2008; Valckenaers & Van Brussel, 2005).

Kruger and Basson (2019) compiled an extensive list of evaluation criteria from literature for holonic control implementations in manufacturing systems, which identifies the most important characteristics, requirements, evaluation criteria and performance measures from literature. A summary of the relationship between these performance metrics and characteristics is shown in Table 2. These performance measures were selected as metrics that could be measured in the evaluation of an HMES, that could then be used to quantify the extent to which the HMES requirements are satisfied.

Kruger and Basson (2019) also specify that for these performance measures to provide the greatest value, it is necessary that the measures be used on similar HMES implementations. These could be the same system at different stages of the implementation, or with slight changes in configuration, or it could be two similar systems that were implemented in a different manner. The quantitative results that can be obtained from these performance measures can offer greater value than qualitative and subjective measures, as they are verifiable, and often repeatable (Kruger & Basson, 2019).

Table 2: Relationships between requirements and performance measures (Kruger & Basson, 2017)

			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			*		*	*	

2.2.5 HMES Reference Architectures

2.2.5.1 Overview

An HMES reference architecture is a basis from which HMESs can be developed. Using an architecture for development negates the process of deciding which technologies should be used, which standards should be followed and how the backbone of the HMES should be implemented. These choices for technologies and protocols have already been researched and validated for use in the architecture implementations, which leads to the biggest benefit of these architectures; that of potentially greatly reducing the development time of the systems. Another benefit is that of reusability; one architecture could be used to develop multiple systems for varying use cases and applications. These reference architectures refer

predominantly to software architectures; however, these architectures also encompass all necessary systems for the function of holonic systems. This includes, but is not limited to, communication, data, and information architectural elements.

Since HMESs can be applied to a wide variety of use cases and scenarios, reference architectures are forced to either be highly generic, or highly context specific. A highly generic reference architecture requires more development for an implementation than a context-specific one but is also applicable to more implementations. HMES reference architectures often also define certain protocols that need to be used with the implementation, which for some implementations, might not be feasible. Consequently, it is of great importance that, if a reference architecture is used for the implementation of a HMES, the best possible reference architecture for that implementation be chosen. Some of the most influential HMES reference architectures are discussed in the next section.

2.2.5.2 ADACOR

The ADAptive holonic COntrol aRchitecture (ADACOR) for distributed manufacturing systems (Leitão & Restivo, 2006) was developed out of a need for manufacturing control systems to be designed and implemented with the capability for adaption and evolution. ADACOR was designed to improve performance of manufacturing systems in terms of agile reactions to system disturbances and change, and it was also designed to be a flexible and adaptive manufacturing execution system. ADACOR shares many of the previously discussed generic HMES characteristics, such as being autonomous, collaborative, robust, and scalable.

The ADACOR architecture is comprised of four manufacturing holon classes: product, task, operational and supervisor holon classes; with the product, task and operational holons being similar to the PROSA (Van Brussel *et al.*, 1998) product, order and resource holons, respectively. The supervisor holons are responsible for the creation and management of the hierarchy of the system, in order to coordinate and optimise the HMES. The product holons exist for each product that the HMES is capable of producing, and each product holon is responsible for the management and process planning for that product. Task holons represent individual production orders, in likeness to an instance of a product holon. This holon contains all information relevant to that specific order and is responsible for the management of the order's execution. Operational holons represent the physical resource hardware, and manage the resources' operations in order to fulfil the goals of the task holons.

ADACOR's main advantages are that of being well suited to dealing with manufacturing problems in a distributed manner, and that of being comprised of holons that make it possible to "add a new element without the need to re-initialize and re-programme the system", leading to good system reconfigurability (Leitão & Restivo, 2006).

2.2.5.3 PROSA

As briefly mentioned previously, the PROSA architecture consists out of four types of holons: Product, Resource, Order, and Staff holons. The Product holons represent all the different classes of products that the HMES is capable of producing. They are responsible for holding the process plans and product knowledge necessary for producing the specific product and offer this knowledge to surrounding holons as a service. Order holons are then essentially single instances of the production of one of these products, as they represent an order placed within the system. They can also represent any other task in the manufacturing system, such as maintenance related activities or other production related activities. Resource holons represent the physical hardware of the resource in the manufacturing system. They are responsible for managing and processing all information regarding their specific resource and are responsible for offering the resource's service to surrounding holons in the HMES. Staff holons assist other holons in completing their work and operate in an advisory role. The staff holon can provide advice on a certain matter, by providing information or functionality, but the other holon is still responsible for making the decisions.

PROSA enables easy reconfiguration through having a "high degree of self-similarity," meaning that many holons in the system are highly similar to each other, and can thus easily be reproduced or changed in reconfiguration (Van Brussel *et al.*, 1998). PROSA handles aggregation by structuring the holons in a dynamic hierarchy, which in turn leads to dynamic aggregation. Thus, aggregated holons are clustered together to form an aggregate holon with its own identity, which in turn creates an aggregate hierarchy, which is also open to aggregation within itself. This way, highly complex systems with interactions between a large number of low-level holons that can be difficult to control can be simplified into less complex, and more manageable systems. PROSA does not dictate the number of hierarchical levels and are thus dependant on the specific implementation.

A revision was made to the PROSA reference architecture which mainly addressed the terminology used. PROSA was created with the intention of the architecture being used in the manufacturing industry, but it was soon found that there was a high interest in using the architecture in other domains. The revision avoids using manufacturing-specific wordings and has more symmetry in its structure. This revision was named the Activity Resource Type Instance (ARTI) architecture (Valckenaers & Van Brussel, 2020).

2.3 ARTI Holonic Reference Architecture

2.3.1 Overview

The ARTI reference architecture was developed as update to the PROSA reference architecture, addressing some required refinements. The major refinement was the terminology used, as PROSA's terminology was manufacturing-specific, and

difficult to apply to neighbouring fields (Valckenaers & Van Brussel, 2020). The ARTI reference architecture is intended to be used in many different holonic use-cases, not just in manufacturing systems, so it was designed to have universal terminology and functionality. ARTI does however preserve the aggregation and specialization from PROSA.

2.3.2 Holon Types

As mentioned before, the ARTI reference architecture consists of four different types of holons: Resource Type and Instance holons, as well as Activity Type and Instance holons. These four holons are then each sub-divided into decision making and reality reflection components: Intelligent Agents (IAs) and Intelligent Beings (IBs), respectively.

Activity Type (AT) holons refer to holons that represent a class of an activity. The holon has the activity-related knowledge that would be common amongst all instances of that particular activity such as process plans, material requirements, expected activity time etc., but it does not contain instance specific information, such as the instance's current state values or its activity progress. An example given by Valckenaers & Van Brussel (2015) is that an AT would be similar to a person who is an expert in his field. Each different type of activity within a holonic system, including process activities, maintenance, transport, worker activities etc. would have its own AT holon, and this holon would communicate with the Activity Instance holons in order to complete an activity. The AT holons would be likened to Staff holons in PROSA.

Activity Instance (AI) holons are then holons that represent the real-world activity taking place and are likened to Order holons in the PROSA architecture. These AIs contain the current state information specific to that instance of the activity's execution only, as well as the activity's history, and rely on their ATs for the information that is generic to the activity. AIs also manage the execution of their real-world activity, which includes resource selection and management of schedule bookings with the resource. An example used is that AIs are similar to managers in how they operate. These AIs would exist for every instance of a real-world activity, whether it is production related, maintenance related, or worker related. AIs are known to be decision-intensive and have significant responsibilities.

Resource Type (RT) holons are similar to AT holons in that they represent a class of a certain type of resource. The holon has the resource-related knowledge that would be common amongst all instances of that particular resource, such as capabilities, dimensions, maintenance needs etc. but it does not contain instance specific information, such as the resource instance's current state values or schedule. In practice, all different types of physical resources, such as transport, materials, machinery, workers, infrastructure etc. would need RT holons. RT holons communicate with AT holons in order to see whether or not the real-world resource

would be able to complete the given activity. They would also communicate as to how this activity would take place.

Resource Instance (RI) holons represent real-world instances of resources. As with AI holons, they contain the current state information specific to that instance of the resource's execution only, as well as the resource's history, and rely on their RTs for the information that is generic to the resource. RIs also manage the operation of their real-world resource, which includes, but is not limited to, topological information collection, state tracking, management of schedule bookings with the activity etc. Similar to AIs, the RIs are also decision-intensive components with large responsibilities.

2.3.3 Intelligent Agents and Intelligent Beings

IBs are seen as the reality-reflecting components of a holon, using either sensors, models, or other types of inputs to mirror the reality of their World of Interest (WOI), and can be seen as DTs of the hardware resource. IAs, on the other hand, are seen as the components of a holon that are responsible for decision making using a variety of methods, including computationally simple (such as first-come, first-served) principles, or more computationally complex methods (such as complex scheduling algorithms or even machine learning algorithms in some cases).

2.3.4 Delegate Multi-Agent Systems

Within ARTI there is the option of using Delegate Multi-Agent Systems (DMASs) for assistance in decision making. With a DMAS, the previously mentioned ARTI holons delegate the responsibility of scheduling tasks with the resource holons to a swarm of light-weight agents. These lightweight agents, called *ants*, are spawned dynamically (usually at a fixed rate, depending on the computing resources available) by a holon that performs activities, such as an AI holon, in order to assist the holon to perform its required scheduling functions. The holons which created these ants are responsible for them during their lifetime and these ants are not aware of what is going on outside of their given responsibilities. There exist three types of ants: Intention, Feasibility and Exploring Ants. Exploring ants can be seen as processes that virtually travel around within the holarchy looking for possible production routes that a product can take and relay this information back to the AI. The AI then chooses the best option out of the presented routes, and using the Intention ants, propagates this decision (the AI's intentions) to the respective holons, where the task is added to the resource's schedule. Leading up to, and during the execution of this planned task, Feasibility ants virtually travel around the holarchy looking for possible disturbances to the system, and also constantly look for alternate production paths. If a better suited production path is found, the Feasibility ant reports back to the AI, who can then change its schedule booking so that the product takes the optimal path.

This system creates a network of advanced decision-making capabilities that AI holons can use to optimally schedule tasks with RIs and allows for the optimisation of production within the HMES.

2.3.5 Implementations

Implementations of the ARTI reference architecture are scarce in literature and virtually non-existent in industry. One implementation in literature is that of Rossouw (2021), where they implemented an ARTI architecture based system to aid table grape production management. Their implemented system was then evaluated against an existing legacy management system, showing improvements in communications, information management and decision support. The ARTI architecture was also used in the development and implementation of an "embedded aggregate digital twin for the hybrid supervised control of semi-continuous production process" (Borangiu, Oltean, Răileanu, *et al.*, 2019).

2.4 BASE Architecture for Digital Administration Shells

The BASE architecture (Sparrow, 2021; Sparrow *et al.*, 2021) was developed as an administration shell for an HRH in an HMS, and was developed with holonic systems and ARTI principles in mind. It uses centralized core components, with additional plugin components that add extra functionalities which would be specific to its use-case. "BASE stands for Biography, Attributes, Schedule, and Execution, and is a time-based separation of concerns for key augmentations provided to the human worker" (Sparrow, 2021). An outline of this structure is provided in Figure 1. A major advantage of this architecture, even though it has been designed for humans, is that with the use of plugins it can be extremely versatile.

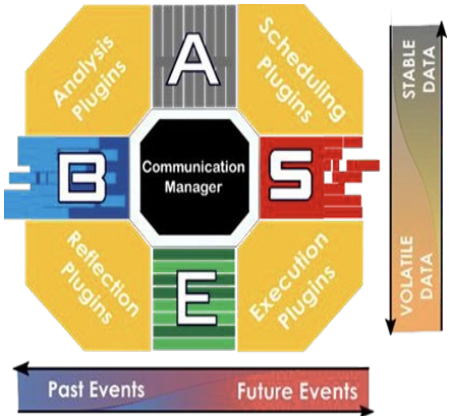

Figure 1: BASE architecture showing the core and plugin components (Sparrow, 2021)

The core components of BASE are depicted in Figure 1 as the blocks labelled 'B', 'A', 'S', 'E' and 'Communication Manager'. The Schedule deals with future events that are scheduled for execution by the holon using the Three-Stage Activity

Lifecycle (3SAL) description (Sparrow, Kruger & Basson, 2020). The Schedule can include activities, maintenance, database validation, communication etc that has been scheduled for execution. Execution deals with the holon's present. It interfaces with sensors, calculates the system's current state and interfaces with the Execution plugin to execute activities. The Biography deals with past events that have transpired within the holon and its surroundings. These could include logs from completed activities, maintenance logs, errors etc. Attributes deals with the properties of the holon that are considered stable and that do not change (or, in some cases, only change steadily). This would include properties such as a holon's specifications, capabilities, age, make/type etc. Lastly, there is the Communication Manager which facilitates the communication between internal components, and between the holon and other external systems.

In Figure 1, the yellow-coloured corner blocks are called the BASE plugins. These plugins consist of the Analysis, Scheduling, Reflection and Execution plugins (AP, SP, RP, and EP, respectively). These plugins interface mainly with the core components that are depicted on either side of the plugin.

SPs represent a set of tools, algorithms, software systems and decision-making interfaces, which create, manage, and optimise the scheduled activities of the HRH. The activities that are to be scheduled can originate from external logistics holons as service requests or internal (BASE component) requests. EPs are responsible for managing the execution of scheduled activities. The EPs take scheduled activities and instantiates their execution by monitoring and communicating with the resource through the Execution component. RPs create and maintain biographic entries of completed activities or events. When an activity has been completed it is entered into the Biography. Data about the events of an activity can still be gathered post execution such as reviews, quality checks etc. APs generate value from the data recorded in the Biography with the aim of updating and amending the Attributes. The APs close the information flow loop of the BASE architecture by updating Attributes from Biography, which enables the self-improving, self-analysis and self-optimisation of the HRH. The cycle repeats with the SPs utilising the updated Attributes to make better scheduling decisions and the EPs better execution decisions (Sparrow, 2021).

Data flow through the BASE architecture follows that of the activities as they flow through the architecture. The flow is time-based and follows the 3SAL structure as mentioned before. The 3SAL structure defines that activities have a three-stage lifecycle: the first being the scheduled stage, the second the executing stage, and the third the completed stage.

Sparrow (2021) implemented the BASE architecture in Erlang software, which included all concepts from the architecture. This implementation can be used as a tool for the development of holonic systems as the implementation provides an administration shell for holons to communicate with one another. This allows for holons providing services to form a network of functionalities and behaviours. The

BASE implementation caters for service provision and finding through a directory facilitator of sorts: the Department of Holon Affairs (DOHA). Holons can register services with DOHA and clients can then find these service providers through DOHA. Service provision is based on contracts between the client and the service provider, detailing the expected service, the parties to the contract, service request arguments and other information that is used in contract processing. Holons in the BASE implementation are identified via their Business Cards (BCs), which contain all relevant information pertaining to the holon. A revision has also been made to the BASE architecture implementation (Van Niekerk, 2021). This revision improved multiple features of the BASE implementation, but the most noteworthy was that of increasing the scalability of the implementation, allowing thousands of instances of the BASE core to be spawned while using minimal computational resources.

The BASE architecture is envisioned as a I4.0 enabling architecture, and as such, it was developed with other I4.0 architectures in mind. The most notable being the RAMI4.0 reference architecture (Adolphs *et al.*, 2015), which BASE draws close similarities with. RAMI4.0 is a reference architecture for digital systems in the I4.0 age. It specifically focuses on I4.0 systems and was described as "a uniform architecture model [used] as a reference, serving as a basis for the discussion of its interrelationships and details" (Adolphs *et al.*, 2015). RAMI4.0 is a six-layered, three-dimensional architecture represented by a layered cube as shown in Figure 2. It was developed with meeting I4.0 requirements as a priority, and it is currently largely accepted as doing so. Each layer of the architecture represents a different component of an I4.0 system that is required for the entire system to operate.

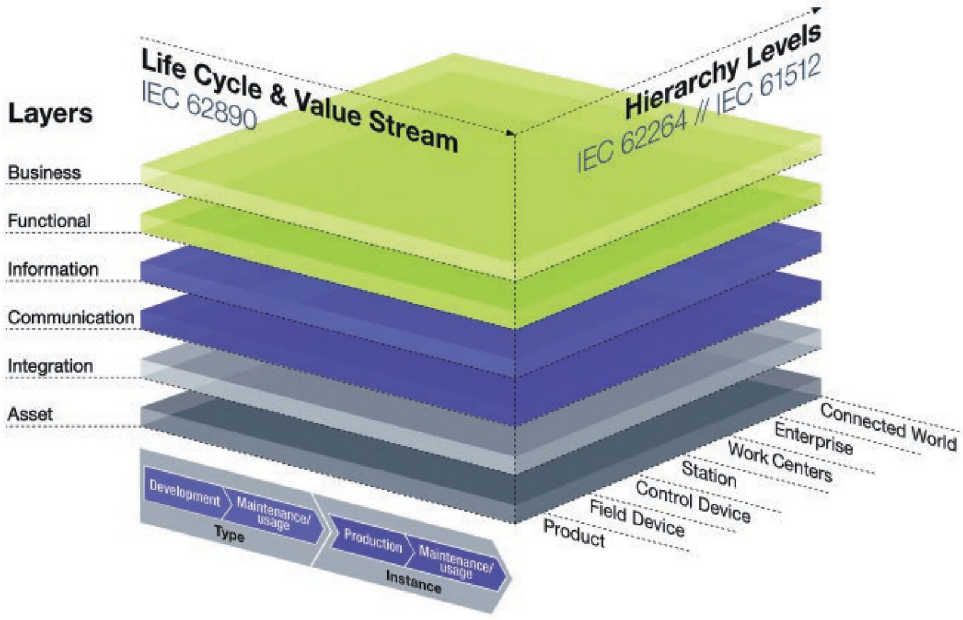

Figure 2: Reference architecture model for Industrie 4.0 (RAMI4.0) "Umsetzungsstrategie Industrie 4.0 (Adolphs et al., 2015)

RAMI4.0 was designed as a digital administration shell that encapsulates a physical entity, which would then allow it to communicate with other digital components, systems and CPSs. Similar to RAMI4.0, the BASE architecture was also developed as a digital administration shell, however it was designed for human administration shells as opposed to other physical entities. Similar to BASE, RAMI4.0 also separates concerns of a digital system, however BASE separates these concerns based on time (as seen by the 3SAL structure) and RAMI4.0 separates concerns based on physical interfacing layers.

2.5 Conclusion

It was found in this literature review that systems comprising of I4.0 enabling technologies are being implemented more often. It is, however, evident that even though the adoption of I4.0 enabling technologies is increasing, there are still large barriers in place that prevent enterprises, especially SMEs, from transitioning their systems to I4.0 platforms. It can be seen, arguably, that the largest barrier to the widespread introduction of I4.0 systems is the development difficulty of these systems. This barrier is hoped to be overcome with the aid of reference architectures for the development of I4.0 systems and their enabling technologies.

It was found that CPSs and HMESs were major drivers of I4.0 systems' implementation, as these systems inherently incorporate many I4.0 enabling technologies. Existing infrastructure can be repurposed or augmented with these technologies for an implementation, instead of having to construct an entirely new system, which is seen as highly beneficial. These systems, HMESs in particular, are also widely seen to provide many advantages to manufacturing systems over traditional manufacturing execution systems. Benefits such as reconfigurability and robustness are highly desirable in industry. It was also found, however, that the development of these HMESs is also a barrier to adoption in itself, due to the high development time, cost and difficulty associated with them.

Through examining some of the most influential reference architectures for holonic systems, it was seen that the optimal selection of reference architectures for use in an implementation of a HMES is very important. During the examination of these reference architectures, it was also found that very few implementations exist in literature, and even less in industry, of the ARTI reference architecture, and thus it would be of worth to research ARTI architecture implementations further.

Key HMES requirements and evaluation criteria were also identified from literature and will be used in Chapter 6 of this thesis for the evaluation of the case study implementation.

3 Mapping the ARTI Architecture to the BASE Architecture

This chapter details the mapping of the ARTI architecture to the BASE architecture. This involves investigating what features and functionalities are required by the ARTI architecture and then to evaluate whether the BASE architecture satisfies these requirements. Mapping these ARTI features onto the BASE architecture provides an indication of the BASE architecture's suitability to be used in the implementation of an ARTI-based HMES.

3.1 System Features and Functions

The ARTI architecture defines that IBs encompass reality reflection components, and that IAs encompass decision making components and other functionalities that carry great responsibilities. From a high-level, individual BASE holon components can thus be split up between being IA and IB components. This can be seen in Figure 3 where BASE components are mapped to their respective ARTI components. The BASE core serves as an IB component, as the core is dedicated to communication, execution, and storage of the state of the holon and its World of Interest (WOI), which all contribute to reality-reflection of the holon's current state. BASE core components also do not implement any crucial decision making; this is rather left to the plugins, which distinctly splits the IB components from the IA components. The four different types of BASE plugins together represent the IA component of the BASE holon as these plugins are dedicated to making context-specific decisions, which influence the behaviour and execution of the holon.

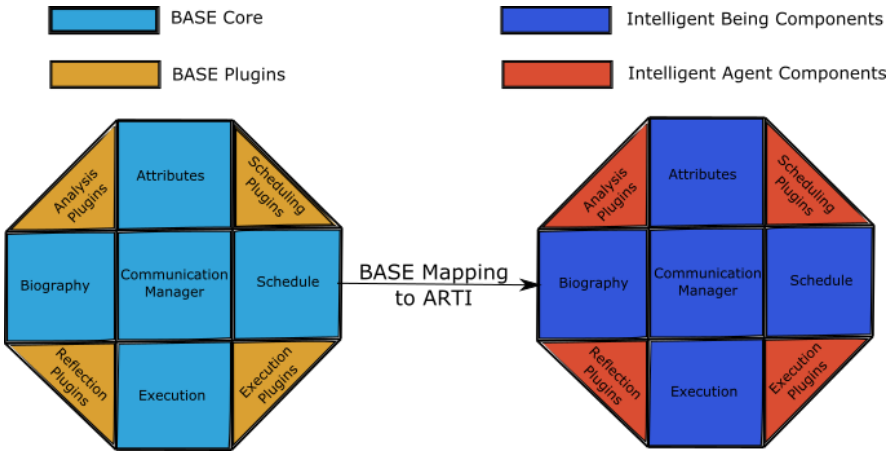

Figure 3: BASE to ARTI mapping of core components

It must, however, be noted that for certain implementations, it might seem that the plugin is not making any decisions (such as a Reflection Plugin that simply reflects data to the biography), but it was seen in this thesis that the developer/programmer

that developed the algorithm/program made important decisions when writing the algorithm that converts the data into information, and it is believed that that is sufficient reasoning for all plugins to be classified as IAs.

The four types of holons that form part of the ARTI reference architecture need to each be implemented for the ARTI architecture. Each type of holon can be implemented as an instance of the BASE core along with various respective plugins. Each holon would receive its own set of all types of plugins, providing for the required functionality as dictated by ARTI. This requirement is followed by the fact that most of these types of holons will need multiple instances of them. BASE is fortunately highly scalable, especially in its recent version, BASE Factory (Van Niekerk, 2021), and this will sufficiently satisfy the scale requirements of the ARTI architecture. Instance holons will rely primarily on the Execution and Schedule core components along with SPs and EPs, whereas the Type holons will rely primarily on the Attributes and Execution core components as well as the APs, EPs, and SPs. Since the BASE architecture implementation runs in Erlang, processes are lightweight and spawned easily, thus it would not be resource intensive, and thus problematic, to have BASE instances running for each holon in the ARTI architecture, along with its plugins.

It is also necessary for the ARTI holons (specifically the Resource holons) to communicate with their hardware resources. This is accommodated for with BASE's open communication philosophy. BASE can support multiple communication protocols which can be used to communicate with hardware, such as TCP/IP connections, MQTT, Serial, etc.

The BASE architecture thus accommodates for the required features and functionalities of the ARTI architecture as an overall system. Communication will be considered next.

3.2 Communication

The BASE architecture allows for a great degree of freedom with communication. The Communication Manager (CM) core component, that is solely dedicated to communication between BASE holons, has multiple in-built functionalities such as to send and receive requests between holons, and to receive Requests for Proposals (RFPs) as part of the Contract Net Protocol (CNP). Additional functionalities and communication protocols can be added to the CM with relative ease. ARTI specifies that it is necessary for AT holons to communicate activity instructions with AI holons, and visa-versa, using the Next-Execute-Update (NEU) communication protocol, and BASE has the in-built functionality for communication between "an entity with process knowledge and an entity with the ability to execute process actions" (Sparrow, 2021) using the NEU protocol. BASE also takes this requirement a step further in that it accommodates for many different communication protocols through the CM.

Within ARTI there is the option of using DMASs for assistance in decision making. The implementation of each type of these lightweight agents within BASE would be in the form of an Erlang module within a plugin that spawns ant processes. Exactly which plugin the process would exist in would be context specific and therefore selected by the developer. The reason for these ants not being individual BASE instances is because these ants typically require very limited functionality, and their lifespan is generally very short. Also, a BASE instance normally consists of at least five Erlang processes, thus it would be much less computationally intensive to have an ant only be a single Erlang process.

3.3 Discussion

As seen by the above mapping, all inter and intra-holon features and functions that are required by the ARTI reference architecture are able to be fulfilled by the BASE architecture's core components and plugins. In some areas there would have to be more development for implementation than others, but overall, the BASE architecture is adaptable and versatile thanks to its use of plugins and open communication standards. The BASE architecture's scalability also plays a major role in its ability to behave as any one of the individual ARTI architecture holons.

In the mapping process, it was found that there are some ambiguities in the ARTI architecture's descriptions and classifications of what separates an IB from an IA. The ARTI architecture does not clarify precisely where the line is drawn when looking at decision making as the deciding factor between IAs and IBs. If only surface-level functionalities are inspected, it may seem in some cases that there is no decision-making taking place, but if the decisions of the developer/programmer are inspected, then it is evident that important decisions have indeed been made within that function. It is believed that this could be sufficient reasoning to classify such functions as IAs.

Concluding this chapter, it can be seen that the BASE architecture is a suitable tool for use in developing an accurate implementation of the ARTI reference architecture. The BASE architecture is adaptable and versatile, and able to be developed as any one of the individual holons required by the ARTI architecture. It is also expected that the BASE architecture could greatly decrease the development difficulty due to having many built-in features, such as task management, communication, and holon management. The BASE architecture also facilitates communication between holons, and it has features and functionalities that would support an accurate, reliable, robust, and relatively simple implementation of the ARTI reference architecture.

4 Case Study Development

This chapter details the development of the ARTI-based HMES case study. This involves an introduction to the case study system and motivation for the choice of the system. It also includes an overview of the case study system's hardware. The requirements for the development of the HMES are then detailed. The system structure and holarchy are then developed.

4.1 Case Study Selection

This case study aims to satisfy the objectives of this thesis as set out in Section 1.2 of developing an ARTI-based HMES implementation using the BASE architecture as a development building block. For this purpose, it is necessary to ensure that the selected case study system allows for these objectives to be met.

Requirements for the case study system are outlined in Table 3. These requirements will be used to evaluate whether the selected case study is suitable to achieve the research objectives.

The selected case study system is the Fischertechnik Industry 4.0 Training Factory (henceforth referred to as the Mini-Factory). It is a small-scale manufacturing system built as a training tool for I4.0 technologies. It can be seen from Table 3 that the selected case study system meets all requirements and will provide a platform of sufficient complexity similar to real-life production systems on which the ARTI-based HMES implementation can be developed and evaluated.

Table 3: Case Study Requirements

Req. No.	Description	Mini Factory Properties
R1	The system needs to be extendable by the user so that new resources could be added to the system. This will enable reconfigurability experiments to be performed.	Consists of six unique stations and is completely extendable as the physical system is constructed piecewise from a wide range of plastic components, similar to LEGO™, but also includes many types of interchangeable sensors and actuators. The user can add and remove parts or whole stations to/from the factory as desired, leading it to be easily scaled.
R2	The system needs to be able to be extended with more than one instance of a resource. This will enable scalability and reconfigurability experiments.	
R3	The system needs be able to be programmed by the user/developer so that the correct low-level functionalities can be implemented in order to interact with the high-level control system.	Controlled on the low-level by six user-programmable controllers, that are network connected for process communications. The six hardware stations need to interact with each other in order to move workpieces through the production line, but the resources are not physically connected to one another, which leads to sufficient complexity to satisfy the requirement.
R4	The system needs to have multiple resource/hardware stations.	
R5	The system needs to have complex interactions between hardware resources in order for the interactions between holons to be tested thoroughly.	
R6	The system needs to be able to process multiple activities in order to enable resource selection and scheduling experiments.	Capable of producing three different coloured workpieces, and it has two main modes of operation: receiving workpieces to store in the warehouse; and producing workpieces for dispatch.
R7	The system needs to have multiple paths which production can follow in order to enable scheduling and aggregation experiments.	The default setup only has one linear production path, but due to the system being easily extendable and scalable, it should be possible to add more stations to the production line in order to provide multiple production paths.

4.2 Case Study Description

4.2.1 Hardware

The Fischertechnik Mini-Factory hardware and firmware are detailed in this section in order to provide a better understanding of the case study system and its capabilities. The system is shown in Figure 4.

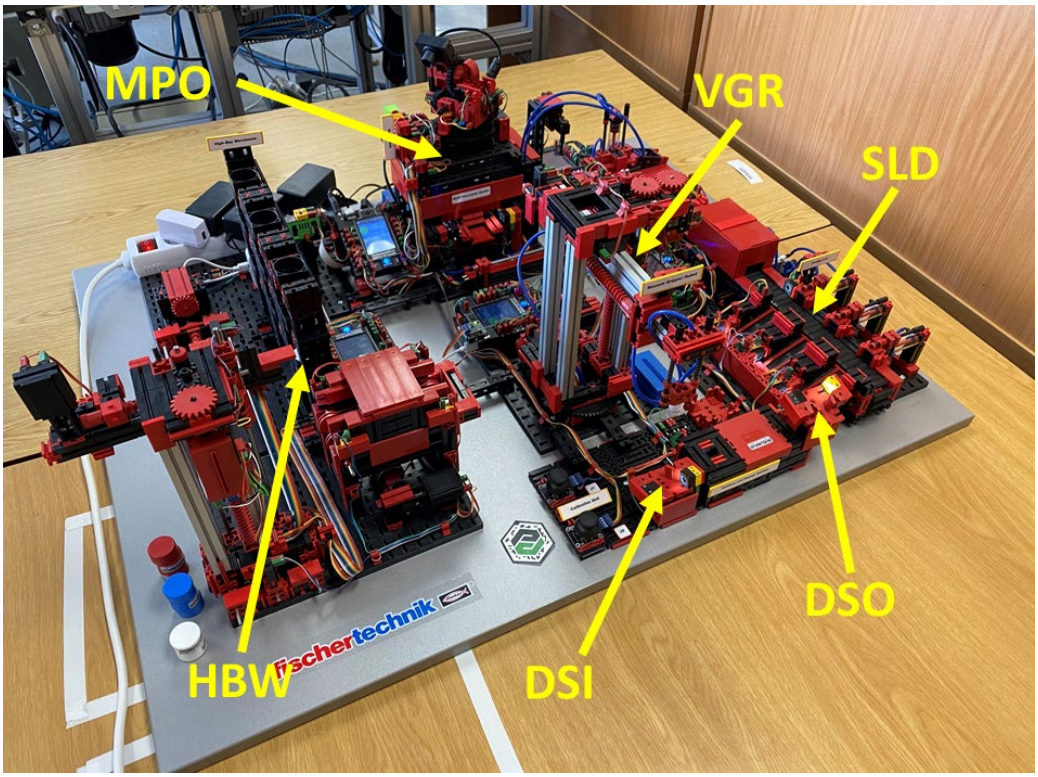

Figure 4: Fischertechnik Industry 4.0 Training Factory

The Mini-Factory consists of six different hardware stations, as identified in Figure 4: a High-Bay Warehouse (HBW) for storing unprocessed workpieces; a Multi-Processing station (MPO) where two simulated processes are executed on the Workpieces (WPs), being fired in a kiln and being worked on in a milling machine; a Sorting Line Station (SLD) that sorts WPs based on colour; a Vacuum Gripping Robot (VGR) that transports the workpieces between different stations; an Output Station (DSO) where the completed workpieces are taken to be dispatched from the factory; and a Input Station (DSI) where 'raw' workpieces are delivered to the factory before being stored in the HBW. Each of these six stations consist of a different combination of sensors and actuators, which are listed in Appendix A.1.

The system is equipped with a default low-level automated control system, using Fischertechnik TXT controllers. The system is setup to connect to a Fischertechnik Cloud Dashboard where the user/operator can view the system's real-time state, as

well as place orders for red, white, or blue workpieces. The system includes environmental sensors, as well as a camera which can be monitored from the Cloud Dashboard.

The TXT controllers communicate with one another using the MQTT protocol, with one of the controllers (which acts as a 'main' controller) being setup as a MQTT Broker and the rest as Clients. The Broker (and main controller) also behaves as the connection to the Cloud Dashboard via an MQTT Tunnel, as a part of the MQTT broker. The network for the system is provided by a wireless router that is bridged to an internet connection.

The default system operation is capable of performing orders and deliveries. These processes consist of various steps that the WP needs to follow through the Mini-Factory. This default order process is depicted in a simplified flowchart in Figure 5. The full flowcharts for the default order and delivery processes can be found in Appendix A.2 and A.3, respectively.

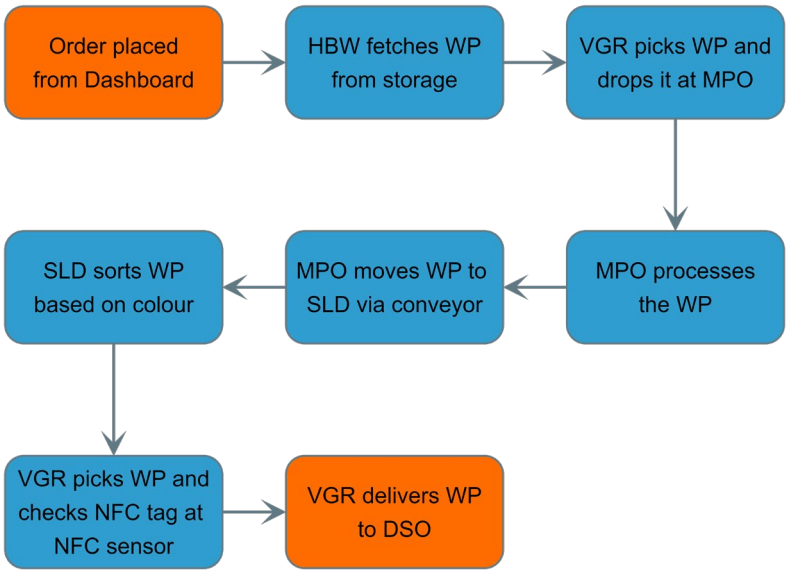

Figure 5: Simplified Default Order Process Steps

4.2.2 Mini-Factory Stations

4.2.2.1 High-Bay Warehouse

The HBW resource consists of three separate sub-stations that interact with one another: a high-bay storage facility which holds containers in a 3-by-3 vertical rack; a two-axis Cartesian robot that stores WPs in the high-bay storage; and an Input/Output station with conveyor belt which moves containers between the robot and the location where the VGR picks and drops WPs. The I/O station has photoelectric sensors which detect a container. The robot uses rotary encoders on its DC motors in order to save and return to different locations within the HBW.

4.2.2.2 Multi-Processing Station

The MPO resource consists of a simulated kiln, a vacuum gripper, a milling station, and a short conveyor belt. The simulated kiln has a DC Motor actuated platform which moves the WP into, and out of the kiln. The vacuum gripper consists of a two-axis cartesian robot with a vacuum gripping nozzle. The milling station consists of an actuated turntable and a simulated milling machine which uses a DC motor to spin a plastic cutting bit. And lastly, the conveyor consists of a conveyor belt driven by a DC motor, and a photoelectric barrier sensor at the end of the conveyor to signal to the system that the WP has reached the end of the conveyor.

4.2.2.3 Sorting Line

The SLD consists of a conveyor belt, a colour sensing booth, and a sorting station. The conveyor belt is similar to the MPO's conveyor belt and it also has a photoelectric barrier sensor at the start of the conveyor belt to signal to the system that the WP has moved onto the SLD's conveyor belt. The colour detection booth consists of a dark enclosure over the conveyor belt with a LED light and colour detection sensor in it. The colour detection booth also has a photoelectric barrier sensor at the exit of the booth, to trigger a timer for the sorting station. The sorting station uses this timer to determine when to eject a WP into its respective colour bin. The WPs are ejected into their bins by being pushed off of the conveyor belt by a pneumatic actuated piston. The bins also have photoelectric barrier sensors which signal to the system that the WP has been sorted.

4.2.2.4 Vacuum Gripper Robot

The VGR consists of a three-axis rotating robot with a vacuum gripping head which is used to transport WPs between stations. The VGR has a base which can rotate approximately 270 degrees, a boom arm which can move up and down using a linear screw-driven rail, and the boom arm can extend and retract using a linear screw-driven rail. All actuation is done via DC motors with rotary encoders. The vacuum gripping head is actuated by use of a combination of an air compressor, a solenoid valve and two pneumatic piston cylinders.

4.2.2.5 Input and Output Stations

The DSI and DSO both consist of a WP sized bay which has a photoelectric barrier sensor which is activated when a WP is placed in the bay. This signals to the system that a WP has either been delivered or dispatched.

4.2.3 TXT Controllers

The Mini-Factory uses six ARM Cortex-based Fischertechnik TXT controllers for the low-level control of the system. One of these acts as a main controller which hosts the MQTT broker and also acts as the MQTT tunnel to the Cloud Dashboard. This main controller is also responsible for publishing the data from the

environmental sensor, and the image stream from the camera. The second TXT controller is the HBW controller. The third and fourth controllers are used for the control of the MPO. One controller is the primary controller and the second acts as a slave. This is done to increase the number of available GPIO pins to interface with the sensors and actuators of the resource. The fifth controller is responsible for the control of the SLD. The last controller controls the VGR, the DSI, and the DSO. The TXT controllers' most noteworthy features can be found in Appendix A.4.

The controllers run a distribution of the Linux Operating System which allows them to incorporate many features. Some of these features include: a VNC server; an FTP server; a webserver; and a MQTT broker (server). The controllers can be programmed via the ROBO Pro software or by uploading compiled C files via either webserver or FTP.

4.2.4 Gateway

The local network for the Mini-Factory is provided by a TP-link nano router which is then bridged to an internet connection via either Wi-Fi or Ethernet. This allows the TXT controllers to connect to the router's wireless network and then communicate with the other controllers over the network (using MQTT). This gateway also allows the Mini-Factory to connect to the remote Fischertechnik Cloud MQTT broker, which enables Cloud Dashboard control and monitoring.

4.3 HMES Requirements

HMESs are diverse in application and implementation. This leads to system requirements needing to be specific to the implementation. It is possible however to identify common and reoccurring requirements from implementations and discussions in literature. The majority of these requirements are however qualitative requirements, which results in the evaluation of the system being somewhat subjective and open to interpretation. For the development of the ARTI-based HMES for this case study, the following primary requirements were identified from literature, as discussed in section 2.2.1.

The system needs to be reconfigurable, meaning that it should be possible to adapt the HMES to a changing manufacturing system with relative ease. If the manufacturing process changes, or if resources are added or removed, it should be within the system's capability to be able to simply add or remove resources or change the production steps/process in the control system. Scalability ties in with the requirement of reconfigurability, as the system needs to be able to be reconfigured for a larger scale, where the system needs to scale as new resources are added to the system, and as new instances of activities or resources are spawned.

The system also needs to be robust, adaptable, and reactive to changes within the system. This means that if the system experiences some unforeseen state, that it can handle the state and provide a suitable outcome. The system needs to be able to

react appropriately to knowledge that it gains from itself or the environment. This includes error handling, the handling of resources going offline and new resources being added, the handling of unforeseen schedule changes, and the handling of successfully completed activities.

The system must consist of autonomous, self-organising holons which can exist on their own with no reliance on other holons in order to perform their tasks, besides when the task requires other holons. The holons must organise themselves into a holarchy in order to form a network of functionalities and behaviours, with which complex manufacturing and control tasks can be executed. This leads to the requirement that the individual holons, and the system as a whole, needs to be collaborative. This means that holons need to be able to work together to achieve a common goal. An example of this might be for two resource holons to work together to produce a single product.

Intelligence is also a key requirement for a HMES, but it is a highly subjective requirement. In the most basic sense, the system needs to be capable of making decisions which generate the desired outcome. This could involve using various simulation or optimisation models to inform decision making. This leads to the requirement of being pro-active. The system needs to be capable of responding to system inputs and stimuli in a goal-orientated manner, in which the system chooses to react in a manner that favours its own goal.

The system also needs to be able to learn from the information that it has gained from its past actions and from the environment, by generating knowledge from collected data. This information could then be used for future process planning and scheduling, or for maintenance related activities.

Lastly, the implementation of the ARTI-based HMES should not negatively affect the throughput of the system, as measured against a baseline experiment.

4.4 Case Study HMES Architecture

4.4.1 Holon Identification

A crucial step in the development of the ARTI-based HMES is that of developing the holarchy structure of the system. This involves defining: the individual holons and their types; the hierarchical layout of the holons within the system; the inter-holon interactions; the individual holon internal architecture; and their intra-holon interactions.

As per the ARTI architecture, Type, and Instance holons form the foundation of the system, with each respective holon category comprising of Resource and Activity Types and Instances. This case study system is thus comprised of Resource Type, Resource Instance, Activity Type and Activity Instance holons. Each of these holons have their own respective sub-structure consisting of Intelligent Agent and Intelligent Being components. The system architecture is hierarchical with Type holons being at the top of the hierarchy, followed by Instance holons, and then followed by sub-Types that fall below RIs and then sub-Instances. This case study contains an aggregate RI holon called the Production Line Aggregate (PLA) which encompasses the MPO and SLD in a single RI. This hierarchy is shown in Figure 6. It must however be noted that the holons in this Holonic system have an ever-adapting structure, which at times could be classified as a heterarchy. This depends on which holon is currently in a contract with another, and where they thus have a client-service provider hierarchy. The tree structure shown in Figure 3 is derived from a certain situation in which the Activity Type holons request a service from the Resource Instance holons, and thus form a temporary hierarchy during the execution of that contract. This situation is the primary operating situation for the system. This architecture is expanded on in this section below by examining each of the holons that the system is comprised of.

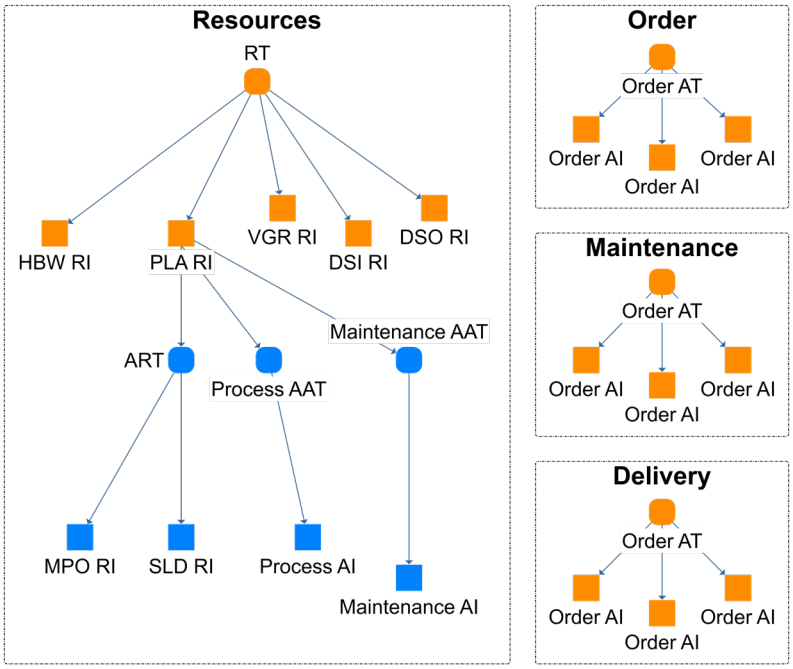

Figure 6: Case Study System Holarchy

4.4.1.1 Resource Types

The system has two RTs: one that represents the five different resource stations in the system (HBW, VGR, PLA, DSI and DSO); and another that represents the aggregated resources (MPO and SLD). This was done as all the resources in the system are similar in operation and execution with only minor differences, related

to communication (e.g., MQTT topics) and attributes. The services that these RTs would provide to the resources remains unchanged for each of the resources in this case study, which justifies using a single RT for multiple types of resources. This is also valid for the Aggregate RT (ART). Also, both the RT and ART would be very similar, besides for the slight changes that are required for the RT to operate as an ART.

An alternative approach would be to create an RT for each different station in the system and have a single instance of each RT for each respective station. This was decided against as this would introduce a large amount of code repetition, with no added advantages.

The RT is present in the system for as long as there is an instance of it in the system and has the role of being the major decision maker and manager for the RIs. RIs request the RT to perform more complex computations for them, as a service. This is done to keep the RIs free from computationally heavy tasks, to be available to handle and execute their tasks and communications with hardware, which may be sensitive to delays. An advantage of using Erlang for the implementation of the system is that processes can easily be spawned as needed, which is beneficial for the RT as a handler process can be spawned for each request that is received from RIs, alleviating any bottlenecks. RTs are also responsible for managing their RIs, which includes providing their attributes after being spawned, as a part of attribute inheritance, and also to monitor the state of their tasks and provide this information to ATs as a service. The following services are provided by the RT and ART:

- Resource status monitoring: used by the ATs to initiate the process of the RT/ART retrieving the status of the RIs/ARIs, and then forwarding this information to the AT. This includes collecting workpiece stock information and resource states. This is a service that can be requested by the AT holons at a fixed interval (e.g., 1 Hz) to continually keep up to date with the status of the system resources.

- Process Time Calculations: used by the RIs for scheduling purposes, to get the earliest estimated available schedule slot for a new task to take place. This service is requested by RI holons when the RI has received an RFP. The earliest estimated completion time is returned to the RI, along with the starting time of the respective schedule slot.

- Resource Attribute Inheritance: used by RIs when they first spawn in order to get their starting attributes. This method is used to get the most up-to-date starting attributes from the RT. The RIs do however spawn with default starting attributes. These include default task durations, hardware timeout limits etc.

4.4.1.2 Resource Instances

The system comprises of five RIs and two Aggregate RIs (ARIs): one for each hardware station in the system. An instance is spawned or terminated when a hardware resource enters or leaves the system. RIs are spawned with a set configuration for the type of resource hardware, which is specified in the start-up file of the instance and includes information such as configuration name, MQTT topics that the instance needs to subscribe to for communication with the hardware and default attributes for the instance.

Scheduling of internal tasks for the RI holons is carried out using the CNP. The resource advertises its services to other holons in the holarchy and then once an RFP has been received, the RI requests a service from its RT to calculate when the RI's first available timeslot in its schedule is, considering the RI's schedule and process time attributes. This information is then used by the RI to send a proposal back to its client with the earliest estimated completion time for the requested task. Upon receiving an 'accept' response from the client, the RI adds the requested task to its schedule, to start execution at the agreed upon time.

Execution within the RI is carried out by a finite state machine (FSM), which ensures that the correct sequence of tasks is followed for that specific resource, to execute the requested task (this is discussed further in section 5.6). The RI holon communicates to its respective low-level controller using JSON encoded messages sent via MQTT to that resource's hardware topic.

The RI stores data, such as scheduled start time, actual start time, completion time and completion state, from its executed tasks. This information can then later be used to update the attributes of the resource through various means of analysis. These updated attributes are then used in future for improved scheduling. The RIs provide the services in Table 4 to the holarchy. The ARIs provide the services in Table 5 to the aggregate holons of the holarchy.

Table 4: RI Services

Resource Instances:	**Services:**
High-Bay Warehouse	Fetch WP from storage and store container once fetched.
	Fetch Container from storage and store the WP once it has been dropped.
	Reset storage to default state (empty).
Vacuum Gripping Robot	Start an order by fetching a fetched WP from the HBW and take it to the MPO.
	Fetch a WP from a specific SLD sorting bin and take it to the DSO.
	Fetch a WP from the DSI and take it to the NFC reader and colour sensor.
	Take a WP from the NFC reader to the HBW.
Production Line Aggregate	Start the production of a WP.
Output Station	Await a dispatched WP.
Input Station	Await a delivered WP.

Table 5: ARI Services

Aggregate Resource Instances:	**Services:**
Multi-Processing Station	Start producing a WP.
	Inherit attributes from ART.
Sorting Line Station	Start sorting a WP.
	Inherit attributes from ART.

4.4.1.3 Activity Types

Four AT holons were used in the system, namely: Order, Delivery and Maintenance ATs, and then an Aggregate AT (AAT) for the PLA's Process activity. The Order Type manages orders that were placed by the user via the dashboard. It allows for the ordering of red, white, and blue workpieces. Once an order has been placed, the AT spawns an AI holon, as well as an FSM with which it handles the order. Once the AI has initialized, the AT sends the first process step in the order process to the AI. These process steps are read in from the AT's configuration file, which allows the user to change the process steps of the order production process in the case that reconfiguration is needed. The process steps are then provided to the AI in order, after the previous has been completed, until all process steps have been completed. This is done according to the NEU protocol (Valckenaers & De Mazière, 2015) as illustrated in Figure 7. In a similar manner the Delivery and Maintenance Type holons manage the execution of maintenance and delivery activities. A

Maintenance AI books-out the schedule of a selected RI for a specified duration, ensuring that no tasks are scheduled with that resource during that time, so that maintenance can take place. A Delivery AI starts the delivery process of a WP that has been delivered to the Mini-Factory.

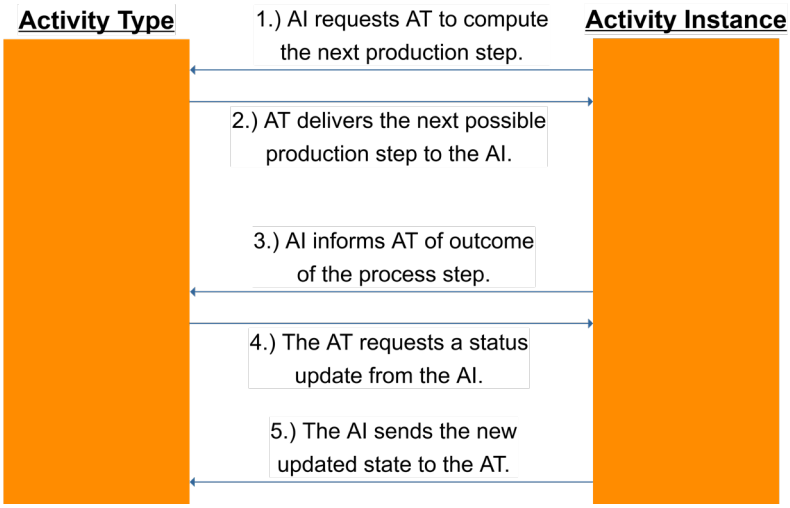

Figure 7: NEU Protocol

The AT offers services to its instances, just as the RT does, and it also has some additional managerial roles. The AT is responsible for monitoring the state of activities that the AIs are busy handling, attribute inheritance, and spawning and terminating AIs. The AT also communicates and interacts with the dashboard service to receive user input as well as to display information to the user. As a part of the dashboard interfacing, the AT also collects the state of all resources in the system to display this information to the user.

4.4.1.4 Activity Instances

An AI is spawned by the AT for each activity (Order, Delivery, etc.) that is requested from the AT. When spawned, the AI registers for service handling of its specific type, thereby notifying its AT that it is initialized and ready for activity handling. As mentioned previously, the AI receives process steps that it needs to carry out to complete the activity, one at a time, and notifies its AT once each process step is completed. Process steps are carried out by the RIs that advertise service handling for those specific process steps. As mentioned before, the coordination of which RI will execute which process step is handled via the CNP. Once all proposals are received from service providers, the best proposal is chosen based on which service provider estimates that it can complete the requested task the earliest (scheduling is discussed further in section 4.7). The AI is responsible for service handling to remove computational load from the AT, since the AT needs to handle multiple requests from numerous instances. The AI is only required to handle one process step at a time and is thus very unlikely to become over-encumbered with requests, or other computations.

Once the AI has completed its list of required process steps and the activity has been completed, it sends a message to its AT containing an information packet with all relevant information from the activity, such as activity state, start time, predicted completion time, and actual completion time. This information is then saved to the ATs non-volatile biography. The AI is then terminated by the AT. The collected activity information is then analysed by the AT to update attributes in the AT that relate to the specific type of activity. These updated attributes will then be inherited by future AIs.

4.4.1.5 Basic Resources

The system requires two additional software resources for its operation: a MQTT service provider, as well as an Application Programming Interface (API) service provider for the web dashboard. These are hosted in the 'BASE factory' as 'Basic Resources', which offer their services to other holons within the system. The MQTT service provider provides MQTT messaging to the HMES as a service. This could potentially create a bottleneck, but with the scale of the case study system not being very large, should not present a problem. This problem could however be alleviated by simply spawning another instance of the MQTT service provider. The MQTT Service uses the EMQTT Erlang library (Lee, 2012) to provide services to subscribe and unsubscribe to/from MQTT topics. The Dashboard Service provides services to send order requests from the dashboard to the AT, and to send data such as status updates to the web dashboard.

4.4.2 Holon Aggregation

Aggregation is the process of combining a group of resources that lie in a lower level of the hierarchy into a single Aggregate Resource, whose position in the hierarchy is a level higher. This Aggregate acts as all of the resources encompassed below it and offers the services of its resources/parts. Often though, an aggregated service could be provided instead.

An aggregation process is demonstrated in the PLA resource in this case study system. The PLA is an aggregate of the MPO and SLD resources and provides a single service to other holons; that service is "Produce Workpiece." However, this service consists of a combination of the services of the MPO "Produce" and SLD "Sort" services.

From the viewpoint of the AIs, this PLA resource will appear as only a single resource with its own service, but within this Aggregate holarchy, there will be multiple different holons. The aggregate holarchy is comprised of an AAT, ART, Aggregate AIs (AAIs) and ARIs. Within this aggregate holarchy, the functions of the different holons are the same as if they were on any other level of the holarchy, besides for a slight change in the AAT; instead of receiving its activity requests from the User Dashboard, it will receive its requests from the PLA RI, which acts as a dummy RI that advertises the aggregate service to the other holons, and just relays service requests on to the AAT for processing.

This aggregation process is useful in many situations. One such situation is when the resource holon's hardware system becomes too complex to be able to be handled by a single holon. The resource holon could then be converted into an aggregate, with its subsystems becoming their own resource holons within an aggregate. An example of this could be a manufacturing plant that consists of multiple different manufacturing cells, with each cell containing multiple machines or stations. In this case it would be beneficial for the cell to be represented by an aggregated resource holon, and each machine/station inside the cell to be represented by its own resource holon. The aggregate resource holon would then advertise the service that the entire cell provides, instead of all of the services of the individual machines/stations.

The second scenario where aggregation is useful is for parallel production paths, of the same types of resources, that do not interact with one another besides for a common starting point, such as shown in Figure 8. The challenge that this situation presents is with the scheduling of production steps; since the resources are of the same type, scheduling via a non-complex method such as the CNP could allow for consecutive process steps to be scheduled on different branches of the production line, which would not be physically feasible. A solution to this would be to create an aggregate of each parallel line. Scheduling would thus see this as a single resource and the scheduling of production steps would remain within that aggregate. The aggregate would then be responsible for the internal scheduling of the line. This would greatly simplify system scheduling and reduce possible scheduling errors.

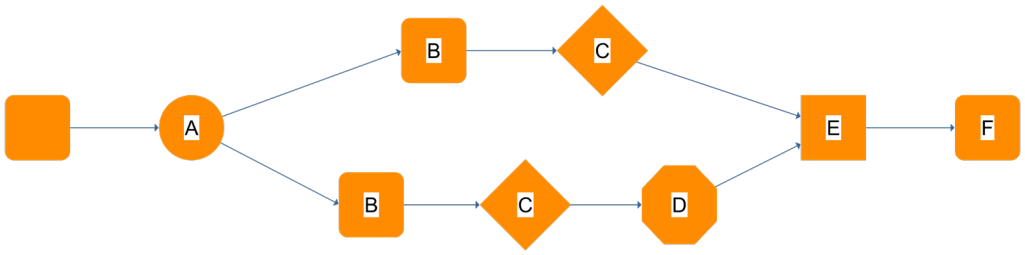

Figure 8: Parallel Production Paths

The advantages of this method of aggregation are thus clear: service handling and execution control for complex systems is simplified as the top-level of the holarchy is simplified by limiting the granularity of the services offered; and scheduling clashes and scheduling infeasibilities are eliminated by simplifying complex production lines. The one major disadvantage of this, however, is that the overall complexity of implementation may be increased, as additional new holons will be required, namely ART, AAT, and AAI holons.

5 ARTI-based HMES Implementation

This chapter details the implementation of the developed ARTI-based HMES on the case study system. Section 5.1 discusses the version of the implementation of the BASE architecture that is used for this implementation, while section 5.2 discusses the programming language which is used for this implementation. Section 5.3 discusses the low-level controllers' programming, section 5.4 discusses the scheduling strategy used, and section 5.5 discusses system communications. The development of the plugins used for each BASE holon is detailed in section 5.6, which is followed by a section discussing the system's user interface. Finally, the chapter concludes with a discussion of the implementation process.

5.1 BASE Architecture Version

The version of the BASE architecture implementation that was used for this case study implementation is called 'The BASE Factory' (Van Niekerk, 2021). This version of the BASE architecture added the functionality of large-scale scalability, the addition of a service directory, and improvements on robustness and resilience. This version also introduced 'basic resources' which is a method of adding non-BASE entities to the holarchy in order to provide services to the holons.

It must be noted, however, that this version of BASE uses the term *activity* for the description of a task that a single holon executes. This term can cause confusion when utilizing the BASE architecture for an ARTI architecture implementation, as the ARTI architecture includes *Activities*. To avoid such confusion, these BASE activities are simply referred to as *tasks* and the ARTI Activities remain *Activities.* It may be seen within the implementation code however, that the implementation still uses the term *activity* or the abbreviation *act* when referring to a BASE task.

A basic web dashboard was also added for the management of holons. These were all features that were found to be necessary for the implementation of this case study system, and this was also the most recent revision of the BASE architecture at the time of implementation, thus it was found to be a good choice for the implementation of the ARTI-based HMES.

5.2 Implementation Programming Language

The BASE architecture implementations have been developed using the Erlang/OTP programming language. Erlang is a functional programming language "used to build massively scalable soft real-time systems with requirements on high availability" and "has built-in support for concurrency, distribution and fault tolerance" (Ericsson Computer Science Labratory, 2021). Although the Erlang programming language was designed for use in the telecommunications industry, it has been shown that it can work well as a platform on which to implement holonic

systems (Kruger & Basson, 2017; Sparrow *et al.*, 2021). Implementing the BASE architecture on the Erlang/OTP platform allows it to be an easily scalable and fault-tolerant platform on which to build holonic systems. For these same reasons, and for improved interoperability, and reduced development effort, the Erlang/OTP programming language was selected for the implementation of this case study system.

5.3 Case Study System Low-Level Control

The hardware setup for the case study implementation remains largely unchanged from the default setup of the Mini-Factory. The only software changes that were made were that the 'Main' controller was not utilised for the case study, besides for its function as an MQTT broker. This is due to it not providing much valuable contribution to the case study, as it is solely responsible for the MQTT broker, the MQTT bridge connection, the surveillance camera, and the environmental sensor; most of which were not needed in the case study. The major changes that were made to the system, however, are all software-based changes. The Mini-Factory's default software setup uses an FSM on each controller for the control of the hardware resource. The controllers listen for messages from the main controller on certain MQTT topics for instructions to perform their pre-defined tasks. Once a task is triggered, the process is mostly automated on the low-level, with the main controller waiting for acknowledge messages from the other controllers before starting the next activity.

The low-level software on the resource controllers was thus rewritten to provide more granular control over the resources using MQTT messages. The topic names were also changed to be more suitable to the implementation. Tables 15 to 18 in Appendix B outline the new MQTT interfaces of each resource after the changes had been implemented, including the topics that it is subscribed to, the topics that it publishes to, the payload format, and a description of the payload.

These changes essentially removed the intelligence, decision making and automation from the low-level hardware control system in order to allow full control of the system from the high-level HMES. The changes also increased the granularity of the control possible from the high-level, such that more individual operations were possible. The low-level control system did however remain in control of individual hardware components and their actuations.

Further changes to the case study system were carried out during the testing and evaluation phase of the case study. These subsequent changes are discussed in section 6.2.

The high-level HMES implementation runs in an Erlang shell on a host PC which is connected to the Mini-Factory wireless network. This allows the host PC to be able to send/receive messages to/from the controllers on the network via the MQTT protocol.

5.4 Scheduling

This case study system uses a simple but effective method of scheduling. Scheduling consists of AIs contracting the services of RIs through the CNP for each process step that needs to be carried out. Once the first process step is completed, the CNP is followed again to secure a service provider for the second step and so on. Schedule slots are also booked sequentially, meaning that an activity cannot book its task between two other tasks in the resource's schedule. This is done to prevent the case of an activity taking longer than it is expected to and running into the booked schedule slot of the next task. Since the system state is constantly monitored, this scheduling method allows for the case where if one resource goes offline, that the system is able to detect the offline resource and carry on with the activity, provided that there is another resource of the same type in the system. A problem that could arise from this method, however, is that it is possible for a workpiece to become 'stuck' on the production line, if for some reason midway through the production cycle the next resource in line became unavailable. This problem is fortunately highly unlikely in the case of the Mini- Factory, as the workpiece follows a single, linear production line, where the next resource station in the production line could not be booked unless the previous station was booked as well. This could however happen if one of the resources went offline and the WP did not have an alternative path.

In the CNP followed by the AI when scheduling tasks, proposals are received from the RIs that contain their earliest estimated completion times. These estimated completion times are calculated by the RTs by using the resource's schedule and the "maximum task duration" attribute. The resource's schedule consists of a list of scheduled tasks, each with their expected starting and end times and, depending on the task's state, recorded start, and end times.

The latest task on the resource's schedule is found; if that task has already been completed, and there are no other tasks on the schedule, the proposed task's starting time is set as the current system time. The estimated completion time for the proposed task is calculated by adding the maximum task duration to the starting time.

If the latest task in the resource's schedule is still ongoing (thus the task is in execution), then the starting time is set as the estimated completion time of the ongoing task, which is calculated using the task's maximum duration attribute, added to the task's actual starting time. The proposed task's estimated completion time is then calculated by adding the proposed task's maximum duration to the starting time.

Lastly, if the latest task on the schedule has not started yet, then the start time of the proposed task is set as the ongoing task's estimated completion time (calculated from maximum duration attribute and estimated starting time).

These estimated completion times received in the proposals are then all compared in order to find the proposal with the earliest estimated completion time. This method of calculating the estimated completion time of the proposed task can however be very inefficient, as the resource utilization rate drops very low if there are big variances in individual task durations, as the schedule is based on maximum waiting times. This is fortunately not a significant problem for this case study as task durations are fairly consistent. For tasks that have higher variances in their durations, it is suggested that alternative scheduling methods be used.

This scheduling strategy is reactive and adaptive, where if a resource goes offline before a booking is made, the system will react and book an alternative resource for the required task. Scheduling is also proactive as the selection of which resource to use is based on the earliest estimated completion time for a certain task. The AI proactively selects the option that favours its own goal of completing its task in the shortest time possible, which also makes the scheduling system a rational decision-making component. Through the use of holon attributes, the system is also capable of learning from its experiences by analysing previous task durations in order to update the estimated task duration. This allows for scheduling to potentially improve over time, as the system learns what its resources are capable of.

More complex scheduling systems can be implemented for the system in order to improve scheduling and system performance, but for the purposes of this case study a simplistic and reliable scheduling system was favoured in order to keep the focus of the case study on the implementation of the ARTI-based HMES. A possible alternative method of scheduling is with the aid of a DMAS that would calculate the best schedule based on information from all holons that is constantly updated. The Exploring Ants would constantly ‘explore’ the holarchy for potential paths for the production process to follow and for the best path for the next production step and relay this information back to the scheduling holon (the AI in this case). Once the AI has selected its intended resource holon and schedule time, the Intention Ants spread these AIs intentions and decision to the other holons in the holarchy. The Feasibility Ants then constantly look for alternative paths in the background, and if a better or more suitable path is found (a path with an earlier estimated completion time) then they will inform the AI, and the schedule will be changed. This method of scheduling is potentially more efficient and performance improving, but it is also vastly more complex. Since the simpler scheduling system is still sufficient for the purposes of this case study and its goals and requirements, the simpler scheduling is selected.

5.5 Communication

System communications are detailed in this section. Communication types include: inter-platform communications, which are communications between the high-level HMES and the low-level hardware control platform; inter-holon communications, which are communications between different holons in the HMES; and intra-holon

communications, which refer to the internal communications between different components of a holon.

5.5.1 Inter-platform Communications

As briefly mentioned before, inter-platform communications messages are broadcast using the MQTT protocol. The high-level execution control system uses a MQTT client service provider which subscribes and publishes to the respective topics of the different resources, as shown in Tables 2 to 5. Each low-level controller also runs a MQTT client to subscribe to their respective topics to listen for control instructions. The low-level controller also publish state and acknowledge messages to their respective topics.

The MQTT Quality of Service (QoS) differs depending on what type of message is being sent. For control instructions that are being sent to the resources, as well as acknowledge messages sent by the resources, a QoS level 2 is used, to ensure that the message is received by the resource, and that the message is only received once, as duplicate messages could cause problems in the system by triggering the same task twice. For other types of messages, such as state messages and stock messages, a QoS level 1 is used to ensure that the message is sent at least once, but it is not catastrophic if the message is received more than once.

As a precautionary measure, and as a means of organising data, a timestamp is also included in each MQTT message sent. This timestamp is checked on the receipt of the message, and only processed if the timestamp matches the current time within a couple seconds.

In terms of security, for the purposes of this case study, the MQTT messages are transmitted unencrypted over socket 1883 as messages remain on the local network and there is a low chance of the messages being intercepted. TLS/SSL could also be used instead with relative ease if it were deemed necessary. The MQTT broker that all messages are transferred through is hosted on the Main TXT controller and uses the TCP/IP protocol over the local wireless network to transmit the messages.

The format of the MQTT messages vary depending on the type of message, but all messages are formatted using the JSON syntax. Tables 17 to 20 in Appendix B show the payload formats for all types of system MQTT messages. An example of a control instruction message is shown in the JSON snippet below:

```
1. {"ts":"2021-04-27T13:51:03.010Z",
2.  "code":4,
3.  "workpiece”: {"id": "0416ae4a616080",
4.              "state”: "RAW",
5.              "type": "BLUE"}}
```

5.5.2 Inter-holon Communications

Inter-holon communications are handled primarily by the BASE architecture's CM, which is a part of each BASE holon. Communication consists of synchronous messages between holons' CMs. These messages include the message type, as well as a service contract that is generated by the client.

Message types vary depending on the purpose of the message; primarily inter-holon messages are in the form of requests, and generally request a service from the other holon. These service contracts are mutual agreements between a service provider and a client for the service provider to provide a certain service at a certain time for the client. Contracts contain information such as client details, service provider details, service details, a service proposal, and a service package. The service details could contain any information that is relevant to the specific service but will always contain the type of service that is required, the starting time of the service and any additional request arguments. The service proposal is used for messages sent as part of the CNP and contain a proposal from the service provider to the client. This proposal remains a part of the contract for the duration of the contract's life. The service package is a package that is delivered to the client, as a part of the contract, once the requested service has been completed. This package contains the completion status of the service (whether it completed successfully or if it failed), the actual starting and completion times of the service, as well as any other relevant information. Included together with this service package is the method of delivery to the client that is necessary.

Another type of communication that occurs between holons is 'inform' messages. These are also synchronous messages and are purely for the purpose of delivering information to a holon. These messages contain the 'inform type' data field, as well as a data field which includes the message payload. These inform messages are primarily used by the MQTT service provider to notify a holon of a received MQTT message but can be used for a multitude of purposes.

5.5.3 Intra-holon Communications

Intra-holon communication varies widely. Primarily, these messages consist of synchronous calls between a plugin and a BASE core component. The BASE architecture also provides a comprehensive API for communication with the core components. Generally, when possible, API functions are used instead of direct calls to components, but for certain functionalities it is necessary to use direct calls. API functions are used for multiple functionalities, namely: service finding and service requests; starting activities; adding, removing, and retrieving activities from the schedule; retrieving, deleting, and updating attributes; retrieving and updating activity data; retrieving core component reception addresses; and retrieving and updating business cards. Since intra-holon communications are very specific to the function that is being performed, it will be discussed in greater depth in section 5.6.

5.6 Plugin Development

5.6.1 Overview

The plugins developed in this section were combined with instances of the BASE core to form the holons of the ARTI architecture. Each holon within the defined holarchy for this implementation consists of an instance of the BASE core, along with four plugins (SP, EP, RP and AP), an additional plugin functionality module, and in some cases a module with the behaviour of an FSM. Figure 9 illustrates the internal software architecture of a generic holon in the implementation, including the primary plugin modules, the additional plugin functionality module, as well as the FSM module.

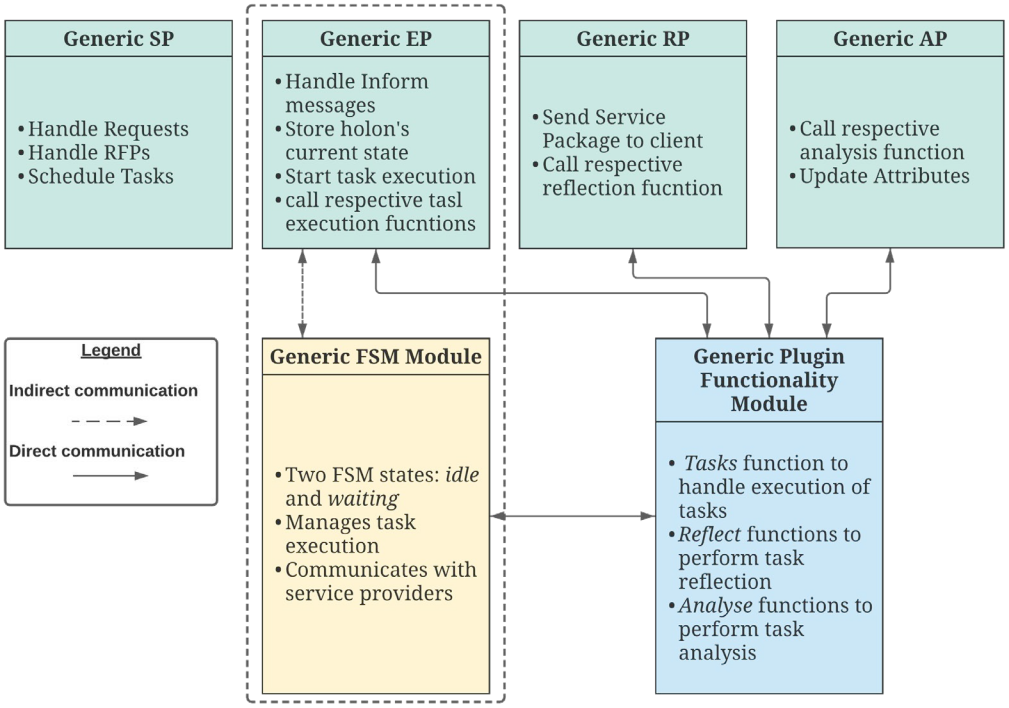

Figure 9: Generic Holon Plugin Modules

Figure 9 serves to provide an illustration of the plugins and interfacing modules of a generic holon as a reference for the remainder of this chapter. Individual holons may, however, differ in configuration.

5.6.2 Resource Type Plugins

5.6.2.1 Overview

The HMES included two RT holons. One for the RI holons on the top-level of the holarchy; and the second for the ARI holons. Besides for the change in service

names, these holons are essentially identical. As discussed previously, this is due to the hardware resources in the case study system being highly similar in functionality. For the purposes of differentiating between the two holons, the top-level RT will simply be referred to as the RT, and the Aggregate RT will be referred to as the ART. These holons each offer three services to other holons. The RT offers the following services to the top-level holarchy:

- *FIND_PROCESS_TIMES* – This service finds the earliest estimated completion time of a proposed task for a RI holon.
- *RESOURCE_INSTANCE_STATUS_MONITORING* – This service is used by the AT holon in order to request the RT to check its RIs' statuses and report these back to the AT.
- *RESOURCE_TYPE_STATUS* – This service is also used by the AT holon in order to request the status of the RT holon to be sent to the AT.

The ART offers similar services to the aggregate holarchy (the network of holons that the aggregate holons are aware of), with slightly altered names.

These services are each complemented with internal RT and ART holon tasks of the same names. These tasks are enabled through the use of scheduling, execution, reflection, and biography plugins. For the purposes of reducing repetition, only the RT holon plugin development will be detailed.

When the RT holon is started up, a plugin settings file is passed to the BASE core. This is a text file formatted using JSON with the following data: the names and locations of all the plugins that need to be started and added to the holon; and arguments for each plugin. A snippet of the RT plugin settings file for a single plugin (AP) is shown below:

```
1.    "1":{"args":{"config": "resource_type",
2.                 "attributes": []},
3.         "file_or_folder": "resource_type_ap",
4.         "type":"file"}}
```

The arguments that are specified for each plugin are passed to the plugin start-up functions upon plugin initiation, which allows the plugins to access these arguments and process them as needed.

5.6.2.2 Scheduling Plugin

The purpose of the SP in the RT is, as the name suggests, to handle the scheduling of the holon tasks. This includes handling requests from other holons and handling other administration related tasks.

In the *init* function, the resource configuration of the RT is read from its start-up arguments and saved to the plugin's variable map. For this SP the configuration could either be *resource_type* or *aggregate_resource_type*. This affects many functionalities within the plugin, but primarily the services that it provides and the tasks that it can perform. During the initialisation of the plugin, it must also register for service handling with DOHA, for its services to be added to DOHA's service registry, for other holons to use.

The RT SP handles requests from other holons by checking received contracts to find what the requested service is. If the requested service is a part of the holon's provided services, further information is fetched from the request arguments in the contract, and a new task is scheduled through the Schedule API. In this call, the Stage 1 activity data map (henceforth referred to as S1Data) is also added to the task, which includes the task's contract, and the activity's *scheduled* status.

Due to RT tasks not interacting with hardware, it is not necessary to follow the CNP for scheduling. Requested tasks are scheduled for immediate execution. In conventional programming languages this could become a bottleneck if many requests are received simultaneously, but thanks to Erlang's high concurrency, it is possible for a new process to be spawned for the handling of each call that the plugin receives.

5.6.2.3 Execution Plugin

The EP is responsible for receiving a call from the Schedule component of BASE core to signal that a scheduled task is ready to be started. The plugin then utilizes the Schedule API to start the activity. The execution of the activity is then started, after ensuring that the activity contains S1Data and that the S1Data contains a valid contract. The activity status is then updated to reflect that the activity has started. If the S1Data was not valid, the activity would be ended, and its status would be changed to *failed* and a reason for failure would be attached.

The EP then continues the execution of the task by calling a function in an auxiliary module called *resource_type*, as shown in the code snippet below:

```
1.  Reply = resource_type:tasks(Contract, Act, MyBC, maps:get(my_pid,
    VariableMap), State);
```

This module and the task execution is detailed in section 5.6.2.6. The execution of the task is then moved to this module in order to keep this EP as generic as possible so that it can be used by both the RT and the ART.

The *tasks* function can return one of two reply tuples: either *{ok, done}* or *{error, Error}*, with *Error* being a variable that contains the reason for the error. The task is then completed once this *tasks* function returns with a reply, and the Stage 2 activity data (henceforth referred to as S2Data) for the activity is updated, based on the outcome of the task.

The successful completion of a task is done by making a call to the *ActivityHandler* and updating the task's state to *done*. The unsuccessful completion of a task is also updated with the *ActivityHandler* as *done*, in order to stop the execution of the task. Note that the task status as well as the error reason is added to the Stage 2 data, so that this information can be sent to the client that requested the service. This is shown in the code snippet below:

```
1.  ok = gen_server:call(ActivityHandler, {done, #{task_status => failed,
    error => Error}});
```

The RT EP is also responsible for receiving the status of its RIs. Once a status message has been received, the resource configuration is retrieved from the message, and then the message is parsed based on the resource configuration. The parsed information from the status message is then saved to the RT variable map, where it can be accessed later when requested by the AT.

5.6.2.4 Reflection Plugins

The RT's RP is primarily responsible for recording completed task information to the BASE Biography. In order to do this, the RP has to register for the reflection of certain task types. This is done through the Execution component's API, with a list of task types, for which to register, as an argument.

When the plugin receives a call to notify it that there is a completed task that is ready to be reflected, the plugin calls a function in the *resource_type* module that handles the reflection of the respective task. The task data is passed to this function as an argument. This is done so as for the RP to remain as generic as possible, so that it can be used for both the RT and the ART.

5.6.2.5 Analysis Plugins

The AP has the primary function of analysing the Biography of the holon in order to update its attributes. However, for the case of the RT and ART plugins, no attributes are updated as in this implementation they do not have attributes that are used that are common to more than a single resource.

5.6.2.6 Resource Type Plugin Functionalities

The *resource_type* module is specific to the RT holon, however the ART holon's *aggregate_resource_type* module is extremely similar, besides for the names of the tasks. This module is used as a code container for functions used by the RT plugins. It contains functions for the execution, reflection, and analysis of tasks, as required for the RT holon. Its functionalities are:

- Execute, reflect on, and analyse the *find_process_times* task.
- Execute, reflect on, and analyse the *resource_type_status* task.

- Execute, reflect on, and analyse the *resource_instance_status_monitoring* task.

The *find_process_times* function receives a list of scheduled and executing tasks, as well as the *max_wait_time* attribute for that task from the RI as request arguments in the contract. This information is then used to calculate the earliest estimated completion time for the proposed task, using the method previously discussed in section 5.4. The calculated earliest estimated completion time and its corresponding starting time is then returned to the RI as a part of the service package.

The *resource_type_status* function sends the RT status to the requester, which includes statuses of all RI holons. The function receives the following arguments: a reply address from the service contract to which it sends the requested RT status; and the EP state blackboard. The statuses of all configurations of RIs are retrieved from the EP variable map, as well as the RT status. This information is then compiled into a status message and sent to the client via a call to the client's reply address.

The *resource_instance_status_monitoring* function finds all RIs, through DOHA, which advertise the service *resource_instance_status.* A request is then sent to them requesting their statuses. The list of business cards of these RIs is found through the DOHA API, as shown below:

```
1.  {ok,BCList}=doha_api:find_service_providers("resource_instance_status"),
```

For the case of the RT and ART holons, it is not necessary to add any additional information to the biography for tasks, besides the already captured S1Data and S2Data. As discussed before, the analysis of the biography for the RT is also not necessary. Thus, the reflection and analysis functions of the RP and AP do not serve any purpose in this holon.

5.6.3 Resource Instance Plugins

5.6.3.1 Overview

The HMES in this case study consists of five RI holons and two ARI holons. The five RI holons each use the same plugins for their BASE core, which have been designed to be generic to all the configurations of the hardware resources. These plugins interface with configuration-specific function modules, as with the RT and ART, for their customised functionalities. The ARI holons work in a similar manner, with a single set of plugins servicing the two ARI holons. The ARI and RI plugins are extremely similar, besides for tasks, services and configurations having different names. For the purposes of detailing the implementation of these holons, all seven of these holons are considered to use the same set of plugins for their instance of the BASE core. Key differences between the RI and ARI plugins are discussed. The RI and ARI holons also each make use of custom FSM modules for

controlling task execution. These modules are also discussed together with the function modules.

Similar to the RT holons, the RI holons' configurations are set when the holon is initialized, by reading in a configuration file. This configuration file specifies which plugins are to be used by the holon, what the holon's hardware configuration is, what MQTT topics should be used for communication with the resource, and then any additional arguments for the plugins, such as default attributes. A HBW RI configuration file snippet for the EP and AP modules is shown below as an example of these configuration files:

```
{"1":{"args":{"config": "hbw",
              "topics": ["hbw/ack", "hbw/stock", "state/hbw",
                         "vgr/ack"]},
              "file_or_folder": "resource_instance_ep",
              "type":"file"
  "4":{"args":{"config": "hbw",
              "attributes":[{"id": "MAX_WAIT_TIME_FETCH_WP",
                             "type": "PROCESS_TIME",
                             "context": "SECONDS",
                             "value": "45"}]},
              "file_or_folder": "resource_instance_ap",
               "type":"file"}}
```

5.6.3.2 Scheduling Plugins

The SP of the RI holons has two essential responsibilities: RFP handling; and request handling. The SP is also responsible for registering with DOHA for service handling. Since each of the RI holons provide unique services to the holarchy, their individual service definitions are contained within the holons' configuration-specific function modules. This service information is then retrieved by the SP for use in service registration and handling. The process followed for service registration is similar to that of the RT SP.

This SP also has to register to receive attributes of a certain type, which are used in the *handle_rfp* function to facilitate scheduling. The attribute type that the SP needs are defined as follows:

```
-define(ATRTYPE, ["PROCESS_TIMES"]).
```

This *ATRTYPE* definition is then used when the plugin registers itself, during initialization, to receive updates to all attributes of this type.

When the RI SP receives an RFP call from another holon (typically an AI holon), the *handle_rfp* function is called. This function retrieves the holon's scheduled and currently executing tasks for the next hour (through the Schedule and Biography

APIs), and then makes a request to the holon's RT to calculate its process times, with the Schedule and Biography information, as well as the proposed task's *max_wait_time* attribute, as arguments. The DOHA API is used to find the BC (and thus the address) of the RT holon, as having the RI permanently store the address of the RT could lead to a situation where the saved address is no longer the RT's address, as the RT's address had changed for some reason. Calculating process times involves calculating the earliest estimated task completion time, along with the corresponding starting time, using the Execution and Schedule information of the holon, as discussed previously.

Once these process times have been received from the RT, a proposal is created, indicating that the RI has accepted the RFP, and which includes the RI's proposed best task completion time. This proposal is attached to the contract received in the RFP and returned to the requester. This simple proposal is shown below:

```
1.  Proposal = #{reply => accept, proposed_finish_time => FinishTime},
```

If the requester decides to accept the proposal, they make a request, including the proposed contract, to the RI SP to add the proposed task to the RI's schedule. This is handled by the *handle_request* function. This function confirms the validity of the contract and proceeds to add the proposed task to the RI's schedule through the Schedule API. The S1Data map is also created before scheduling the task, which includes the contract, task-specific data, such as the task status, and a timeout value. This timeout value is configuration and task specific, and is used within the FSM, which handles the execution of the task, as a timeout for waiting for the completion of a specific task. An example of the scheduling of a HBW task is shown in the code snippet below:

```
1.  S1DataMap = #{contract=>Contract,activity_data=>#{activity_status =>
    scheduled}, timeout => hbw:get_timeout()},
2.  {ok,"success"}=sched_api:new_act(Sched, ServType, StartTime, S1DataMap)
```

5.6.3.3 Execution Plugins

The EP of the RI holons has multiple responsibilities: to subscribe to required MQTT topics for communication with the respective hardware stations; to receive MQTT messages from those subscribed topics, from the MQTT service provider; and to start the execution of tasks and update their status.

The EP subscribes to MQTT topics that are defined in the start-up arguments of the holon. It does this by using the DOHA API to search for a MQTT service provider in the system, and then sending a request to this service provider to subscribe to a certain topic. This MQTT service provider will then inform the EP of any new messages on this topic. The subscription to these topics is carried out in the initialization of the holon through a recursive function that receives a list of topics, as well as the service provider's BC as arguments. Each time that the function is recursively called, it makes a request to the MQTT service provider to subscribe to

a single topic. The contract that is created between the RI EP and the MQTT service provider includes the task that is being requested from the service provider (*subscribe*), the topic that needs to be subscribed to, as well as the Process Identifier (PID) of the EP as the delivery address for the received MQTT messages.

The initialization of the EP then also includes registration with DOHA for the execution of tasks of a certain type. This is done similarly to the RT EP where the task types that are being registered for are configuration-specific and are retrieved from the configuration-specific function module. An added step to the initialization of the RI EP is that of spawning and starting the configuration-specific FSM process that handles the execution process of tasks. The FSM process can be seen as another EP, as it is only utilized within the EP. Communications between the primary EP process and the FSM process are however not direct, as the communications first pass through the plugin functionality module's process.

The execution of tasks is handled in a similar manner that tasks are handled in the RT EP. Once the task is ready for execution, the EP is called to handle the task. This then starts the *handle_act* function, which subsequently calls the *execute* function. This function updates the S2Data of the task to reflect that the task is in progress. A MQTT service provider is then found through the DOHA API, which returns a BC.

This is done as the execution of RI tasks relies on MQTT communications with the hardware resource. If there are no MQTT service providers, the task fails, and the client is notified of the reason for the failure. If a service provider is found, the validity of the contract in the S1Data is checked, and if not valid, the task's S2Data is updated with the reason for failure. If the contract is valid, the configuration of the EP is checked via a simple case statement and the task is handed off to the *tasks* function of the respective configuration-specific function module.

This function, similarly to in the RT EP, replies with a tuple that either contains *{done, WP}* or *{error, Error}*, depending on the outcome of the task. The task's S2Data is then updated accordingly. The WP variable that is also returned in the reply from the function contains the WP information relating to the WP that the executing task was executed on. This WP data is included in the final S2Data and is eventually relayed to the AI which requested the task, which allows the AI holon to keep track of the WP as it progresses through the production line.

Received MQTT messages are handled by the *handle_call* function, which handles calls with the packet: *{inform, Content}*. This function is used for messages with the intent to inform the EP of external events. The Content variable is a map, and contains a key called *purpose*. The value related to this key is checked, and if it is *mqtt_received,* the function relays the message to the respective configuration-specific function module for further processing.

5.6.3.4 Reflection Plugins

The RI reflection plugin calls a function in the configuration-specific function module to handle the reflection of the task. The specific function depends on the task type that needs to be reflected. The *resource_instance_status* task is, however, reflected from within the RP, as it does not need any additional Stage 3 activity data (henceforth referred to as S3Data) added. Reflection functionalities for the remaining task types are discussed in each configuration-specific function module's section.

5.6.3.5 Analysis Plugins

The RI AP is responsible for analysing RI tasks that have been completed and moved into the biography, and using this analysis to update the holon's attributes. The primary attribute type that is analysed and updated within the RI holons is the *process_times* attribute type. This is due to the *process_times* attribute type encompassing the *max_wait_time* attribute, which is used in the scheduling of tasks. This attribute must be continuously updated after each task in order for the attribute to remain relevant and for the scheduling to remain functional. Another function that is performed in the AP is that of attribute inheritance from the RT upon initialization of the RI.

The AP registers with the Biography so that completed tasks are sent to the AP for analysis. This is done through a Biography API function call, with the task type (*ActType*) of the task that the AP wants to register for as an argument:

```
1.  {Res,OldActs}=bio_api:register_analysis_plugin(BioPid, MyPid,
    hd(ActType)),
```

This function call returns with a response of whether the call was successful or not, and with a list of previously completed tasks that have not been analysed yet.

In order to keep the AP generic between RI configurations, the analysis of tasks is completed by a function in the configuration-specific function module, similarly to how reflection is done by the RP. This is done by calling the respective analysing function for the task, as shown below in the code snippet from the HBW configuration, for the *hbw_fetc_wp* task:

```
1.  {NewVariableMap, NewAttributeList} = hbw:analyse_hbw_fetch_wp(Act,
    VariableMap, AttributeList);
```

This function receives a completed task, the AP state variable map, and the AP attributes list as arguments, and returns a new variable map, containing a attributes list with the updated attributes.

Attribute inheritance for RI holons takes place in the following manner: the RT holon sends a request to the RI holon to start the attribute inheritance task (e.g.

hbw_inherit_attributes). The contract sent in this request contains a single attribute that needs to be either added to or updated in the RI. The task is then scheduled, and when executed, the task completes immediately, and passes through the RP to the Biography, without any processing being done. This is in order to get the task to the AP, where the attributes of the holon can be updated. The AP then retrieves the attribute elements from the contract and creates a new attribute record with these values. This new attribute is then either added to the attribute list, or the existing attribute is updated with the new values. This new/updated attribute list is then saved to the Attributes through the Attributes API:

```
1.  {ok,"success"} = atr_api:save(AtrRecep, AttributeList),
```

This process is then repeated for all attributes that need to be inherited.

5.6.3.6 Configuration-specific Function Modules and FSMs

Besides for task and function name changes, respective to the tasks offered as services by each configuration of RI, the configuration-specific function modules and FSMs are very similar to each other. For the sake of reducing repetition, only the function module and FSM for the HBW configuration will be detailed, as it is representative of all the other modules. The only exception to this is the PLA configuration module, as it is an aggregate and will be discussed in the following section.

The HBW RI module is called *hbw* and is responsible for the execution of most functionalities and tasks related to the HBW RI configuration. The HBW RI provides multiple services to the holarchy. These services are defined in the functions *get_serv_types( )* and *get_act_types( )*. These functions are used by plugins to retrieve HBW service information. Shown below are these functions, as well as the function *get_atr_types( )* used by the SP to find the attribute types that are needed for scheduling:

```
1.  get_serv_types()->
2.  ["HBW_FETCH_CONTAINER", "HBW_STORE_WP", "HBW_FETCH_WP",
    "HBW_STORE_CONTAINER","HBW_RESET_STORAGE","HBW_INHERIT_ATTRIBUTES",
    "RESOURCE_INSTANCE_STATUS"].
3.  get_act_types()->
4.  [ "HBW_FETCH_CONTAINER", "HBW_STORE_WP", "HBW_FETCH_WP",
    "HBW_STORE_CONTAINER","HBW_RESET_STORAGE","HBW_INHERIT_ATTRIBUTES",
    "RESOURCE_INSTANCE_STATUS"].
5.  get_atr_types()->
6.    ["PROCESS_TIMES"].
```

Executing tasks are handled by the function *tasks*. This function is called from the EP and takes the FSM PID, the contract, the task type, the MQTT service provider's BC, the EP's state and the hardware timeout, as arguments. The function checks the task type, and then depending on the type of task, calls the respective FSM function to start the execution of the task. Two exceptions are the *resource_instance_status*

task and the *hbw_inherit_attributes* task, which are executed from within the module. The former is executed by retrieving the RI's current state from the EP's state variable map and sending the created status message to the client.

The *hbw_inherit_attributes* task, as mentioned before, has no executing elements and is immediately completed so that the task can continue to the AP. This is done by having the function return *{done, #{}}* immediately.

The remaining service tasks are executed by the *hbw_fsm* FSM module. The HBW FSM has four states: *idle, notify_vgr, waiting_vgr* and w*aiting*. Figure 10 shows the state diagram for a generic FSM as used by multiple RIs in the HMES.

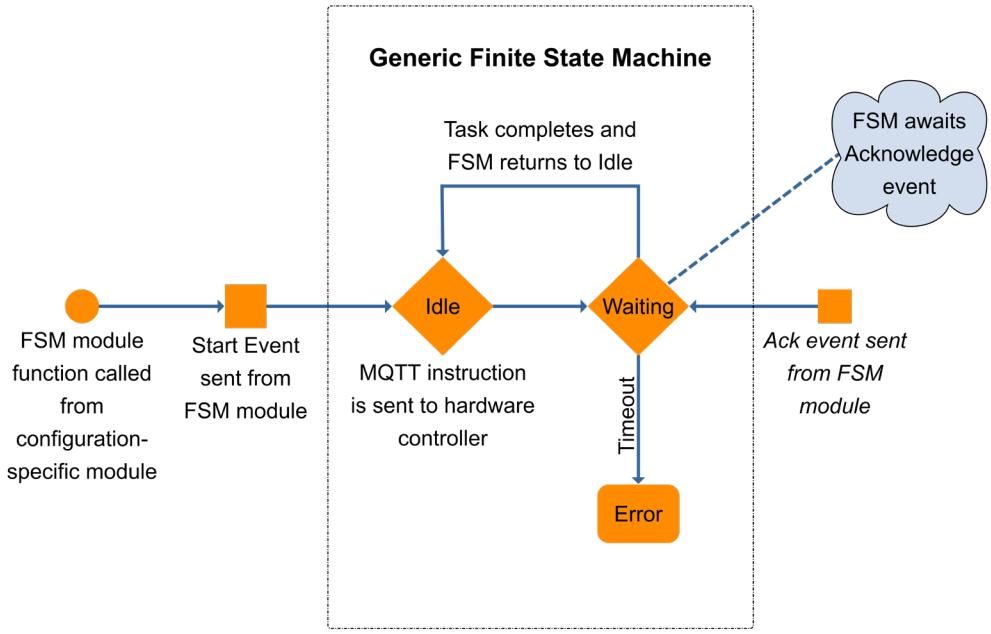

Figure 10: FSM State Diagram

The *idle* state waits for a synchronous event call to the FSM that initiates the start of a specific event. This call is made from within the *hbw_fsm* module from task-specific functions that are exported for use by the *tasks* function in the *hbw* module, in order to start the execution of the tasks that are offered as a service by the RI. An example of this for the *hbw_fetch_wp* task is shown below:

```
1.  hbw_fetch_wp(OwnPid, Contract, ServBC, Timeout) ->
2.    try
3.        Reply = gen_fsm:sync_send_event(OwnPid, {fetch_wp, Contract,
    ServBC}, Timeout*1000 )
4.    catch
5.              Error:Reason -> _Res = timeout(OwnPid),
6.                              {error, hbw_timeout}
7.    end.
```

This call then sends a synchronous event to the *gen_fsm*. If the FSM (and thus the resource) is in the *idle* state, the call is handled by the function:

```
1.  idle({fetch_wp, Contract, ServBC}, From, State) ->
```

This function then retrieves the request arguments from the contract and creates a new contract with the MQTT service provider that was included in the function arguments. This contract includes the following information as part of its request arguments:

```
1.  Mqtt_message = {"ts": Mqtt_time,
2.                  "code": Code,
3.             "workpiece":{"id": WP_ID,
4.                          "state" : WP_State,
5.                          "type" : WP_Type}},
6.  OutRequestArgsMap = #{task => publish_message, topic => Topic,
    message => Mqtt_message, qos => QoS},
```

The *Mqtt_time* variable is populated by the *hbw:mqtt_time( )* function which gets the current UTC time and formats it in the ISO-8601 format, which is supported by the low-level hardware controllers, and shown below:

“YYYY-MM-DDThh:mm:ssZ”

When the low-level hardware controllers receive a message on their respective MQTT topics, they check the *code* value in the MQTT message to determine which task needs to be carried out. Thus, the *code* variable in the above code snippet refers to the hardware tasks as specified in Appendix B. As previously discussed, the QoS for these instruction messages is set to 3 in order to ensure that the message is delivered, and that it is only delivered once. The *Mqtt_message* also includes the information on the WP related to the task. This contract is then included in the request that is sent to the MQTT service provider. If the MQTT message is successfully sent, the FSM is then moved to the n*otify_vgr* state. If the request is denied, the FSM replies to the initial calling function (the *hbw_fetch_wp* function) with the respective error. This is then passed back to the *hbw:tasks* function as a reply, which in turn passes the error message back to the RI EP as a reply. The executing task is then marked as *done*, with the task status *failed*.

The *notify_vgr* state then waits for the HBW hardware resource to send an MQTT acknowledge message that the WP has been fetched and is awaiting collection. The message is initially received by the RI EP and is then passed on to the *hbw:mqtt_received* function. This function is later discussed in more detail. This function then calls the *hbw_fsm:fetched( )* function to notify the FSM. The FSM then moves onto the *WaitingVGR* state, where an *inform* message is then sent to the client that requested the *hbw_fetch_wp* task to notify it that the WP has been fetched and is ready for collection. This allows the AI to start the next step, which in normal use would be for the VGR to pick up the WP. The FSM then transitions to the *waiting_vgr* state, where it awaits, until a timeout, for the VGR to notify it that it has picked up the WP. Once this message has been received, the FSM sends an MQTT message, in a similar manner to previously described, to the HBW hardware resource to store the now-empty container. If the VGR does not notify that the WP has been picked up within the allotted timeout time, the WP is stored again. The FSM then transitions to the *waiting* state, where it awaits the completion of either *store_container* or *store_wp.* Once the acknowledge message that the WP/container has been stored has been received from the hardware resource, the FSM replies to the original calling function (*hbw_fetch_wp)* with *{done, WP}*, where the WP variable is a map that contains the current WP's information. This in turn gets sent back to the *hbw:tasks* function as a reply, and then gets sent to the EP, again as a reply. The function is then successfully completed from the EP.

The other service tasks that can be performed by the HBW RI are executed in a similar manner, except that they do not utilize the *notify_vgr* and *waiting_vgr* states. Once the task is started from the *idle* state, the FSM transitions to the *waiting* state, where it awaits the acknowledge message that indicates that the task completed successfully.

The *mqtt_received* function in the *hbw* module receives any MQTT message that the EP received, along with the topic that the message was sent in, as arguments. The message is then decoded depending on the topic of the message. If the topic indicates that the message is an acknowledge (ack) message, then the JSON message is decoded and information such as the timestamp, ack code, and WP information is retrieved from the message. Then, depending on the ack code received, a message is sent to the FSM that the HBW hardware has either fetched a WP/container, or it has stored a WP/container. If the topic indicates that the message is a status update message, the message is decoded and information such as the timestamps, the state code, the active code, the description of the current state, and the station's name is retrieved and stored to the EP state blackboard. Stock updates are also received in this manner, with the received current stock being saved to the EP state blackboard. The HBW RI also subscribes to VGR ack messages, in order to listen for *Code 3* VGR ack messages which indicate that the VGR has picked up a WP from the HBW.

The *hbw* module also contains functions that are used by the RP to reflect on the various HBW RI tasks. These reflection functions are all similar in operation, only

using the different task names. The function is responsible for creating the service package of the completed task, sending the service package to the client, and updating the S3Data. This is done similarly to how it is done for the RT holons. An example of the service package and service package delivery for the *hbw_fetch_container* task shown below:

```
1. Contract = maps:get(contract,Stage1Data),
2. Package = #{act_name => "HBW_FETCH_CONTAINER", act_status =>
   Act#stage3Activity.execution_info#execution_info.s2data, completion_time
   => Act#stage3Activity.biography_info#biography_info.tend},
3. FinalContract = maps:update(return_package, Package, Contract),
4. spawn_monitor(fun()->
   contracts:deliver_package(FinalContract,Package)end),
```

The *hbw* module lastly contains functions for the analysis of completed tasks in the AP. The analysis that occurs is relatively simple for the RIs in this case study, as the only attributes that are used for the RIs are the task *process_times* attributes. These are updated by first calculating the duration of the completed activity, and then updating the *max_wait_time* attribute for that task by determining if the current task's duration was greater than the existing *max_wait_time* value for that task, and if so, updating the attribute with the new duration value.

5.6.3.7 PLA Function Module and FSM

The PLA RI is a unique holon in that it is essentially a broker for the services of the aggregate system. It thus does not require any functionality of its own besides relaying requests from the top-level system to the aggregate system. Due to the plugins used by the other RI holons being generic between configurations, the PLA RI uses some of the same plugins, albeit with a few changes.

The aggregate system only provides two services: *process* and *pla_maintenenace,* and since the maintenance activity does not encompass much functionality, the *process* activity is used as an example for this discussion.

Similar to the other RI holons, the SP of the PLA advertises the services that the aggregate system provides to the holarchy and handles requests and RFPs from other holons. The inner workings of these functions differ to the other RIs however, as requests and RFPs are relayed to the AAT holon for processing. As it would not be feasible to spawn an entire new AAI holon to send out RFP calls to the ARIs, this responsibility falls on the AAT. The AAT thus advertises an *aggregate_process_rfp* activity to the PLA RI. When receiving an RFP for the execution of a certain task, the PLA RI SP requests the AAT to perform the *aggregate_process_rfp* activity, in a similar manner that the AT receives a request to start a new activity (more on this in the next section). This RFP process is illustrated in Figure 11.

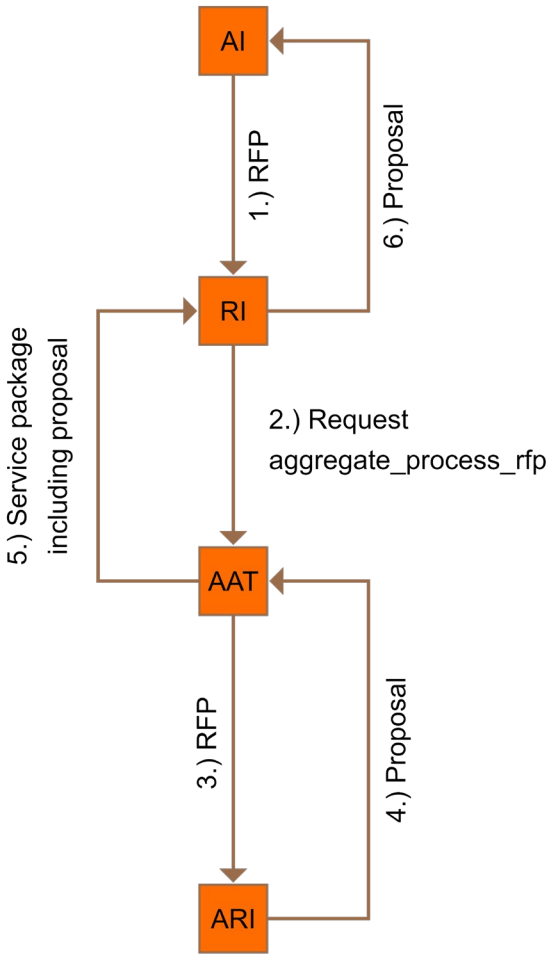

Figure 11: RFP Process for Aggregated HMES

RFPs are then sent by the AAT to the ARIs for each process step in the *process* activity, in order to calculate the earliest estimated completion time for the proposed *process* activity. The result of this activity is then sent back to the PLA RI SP to be returned to the client as a part of the proposal.

When the PLA RI SP then receives a request, the SP schedules a *pla_process* task for immediate execution. The execution of this task then follows a similar method as for the other RIs, by passing the task from the EP to the *pla* module and then to the *pla_fsm* module, with the only change being that instead of sending an MQTT message to a hardware resource, a synchronous call is made to the AAT to request the start of the proposed *process* activity at the agreed upon time, as stated in the service contract. This activity then executes in the same manner that a top-level activity would execute.

When this activity has been completed, the service package is delivered to the AAT, as it is seen as the client within the aggregate system, and the AAT relays this

service package back to the PLA as a part of the service package delivery for the *pla_process* activity, along with the status of the activity (either completed or failed). The task then follows a similar process to other RI tasks, until it is eventually completed in the PLA RI EP. The task then follows the normal procedure of going through the RP, where the service package is created to reflect the status of the aggregate activity and is then sent to the original client. The task is then sent to the AP where the task duration is used to update the *max_wait_time* attribute of the *process* task. This attribute is then seen as the aggregated information of the lower-level aggregate system.

The top-level system thus does not know of the existence of the lower-level aggregate system, which simplifies operations and hides complexity.

5.6.4 Activity Type Plugins

Three AT holons and two AAT holons exist in this HMES case study system. The three AT holons are: Order Type, Maintenance Type and Delivery Type holons. The two AAT holons are the Aggregate Process Type and the Aggregate Maintenance Type holons. Although the AT and AAT holons differ slightly in function, they are very similar once again in implementation. Due to this, and for the purpose of reducing repetition, this section details the implementation of the AT holons, and provides overviews of how the AAT holons differ in implementation.

Similar to RI holons, the AT holons all share a single set of plugins that have been implemented to be generic amongst AT holons, with configuration-specific functionalities handled by individual function modules and the execution of tasks handled by FSM modules. Also, similarly to the RI and RT holons, the AT holons' configurations are defined in a configuration file which is loaded when the holons are initialized.

5.6.4.1 Scheduling Plugins

The scheduling plugins of the AT holons are very similar to the RT holons, in that their primary responsibility is to handle requests to perform tasks that they are registered to provide as a service. The Order AT registers to provide three services: *order_type*, *resource_type_status_monitoring*, and *activity_type_status* services. The *order_type* activity receives the requested WP colour in the request arguments of the contract. This could also have been implemented as three different activities (*order_red*, *order_blue*, *order_white*), but this was decided against as it would lead to unnecessary code repetition as the activities would be virtually identical besides for the colour of the WP. If the activities differed more, it might have been justifiable to implement these as different activities. The Maintenance AT registers *maintenance* and *resource_type_status_monitoring* services, the Delivery AT registers a *delivery_type* service, the Process AAT registers a *process* service and, lastly, the Maintenance AAT registers *aggregate_maintenance* and *aggregate_resource_type_status_monitoring* services. The function within the SP that handles requests schedules the requested activity to start immediately.

The requests that the ATs receive primarily come from the Dashboard Service basic resource, which handles interactions between the web dashboard and the HMES (discussed further in sections 5.6.5.2 and 5.7), and the requests that the AATs receive are primarily from the PLA RI holon.

5.6.4.2 Execution Plugins

The EP of the AT is mostly similar to the EP of the RT but differs slightly in some areas. An additional type of call that the AT EP needs to handle is a *service_package_delivery* call. This call is made from a service provider, indicating that a requested task has been completed. The EP handles this call by relaying the call to the respective configuration's FSM where the delivery package is processed. The arguments passed to the EP on initialization are also unique, in that they include a list of the individual process steps that are required to complete the respective activity that the AT represents. The execution of a task from the EP standpoint is similar to that of the RT holons in that the task is handed to the configuration-specific module for execution, albeit with one change, the list of process steps is also passed as an argument to the *tasks* function. The EP also sends a message to the dashboard when an activity completes with the status of that activity.

5.6.4.3 Reflection Plugins

The reflection plugin works in the same way as the RT RP, with different function names for the function calls to the configuration-specific module to reflect on the tasks.

5.6.4.4 Analysis Plugins

The analysis plugin is similar to that of the RI and RT APs, in that it is responsible for analysing completed tasks and updating the holon's *process_times* attributes, based on the analysis of completed activities. This analysis is done in the configuration-specific module, as before.

5.6.4.5 Configuration Specific Function Modules and FSMs

The configuration-specific modules are similar to the RI configuration specific modules in that they encompass task execution functionalities, reflection functionalities and analysis functionalities. They also make use of *gen_fsm* modules to carry out the execution of the tasks. To detail the workings of the configuration-specific modules, the Order AT *order_type* module will be used as an example, as the other configuration-specific modules work in a similar manner.

The *tasks* function handles the execution of the tasks and separates the execution functionalities based on the task type. The *resource_type_status* and *activity_type_status* tasks work similarly to the *resource_instance_status_monitoring* and *resource_instance_status* tasks in the

RT holons. The *order_type* task works in a different manner, as the AT holon spawns AI holons to handle the activities of the AT.

The function starts by creating a unique name for the AI holon that is about to be spawned. It does this by appending the Unix time to the word "ORDER." This is done to ensure a unique name of the AI holon as its name is used for identification. The holon's type is then set as "*activity_instance"* and the holon is spawned using the *spawn_activity_instance* function. This function reads in the configuration file for an *order* configuration AI, creates a new clean BC for the holon, and then creates a new holon with this information, using the BASE resource creator API. This process is shown in the code snippet below:

```
1. PluginSettingsFile = "activity_instance_order.txt",
2. Addr = #{ipv4 => "pending", erl_addr => Resource_ID},
3. Services = #{},
4. {ok, ChildBC} = business_cards:create_new_bc(Resource_ID, Resource_ID,
   "activity_instance", Addr, Services, pending, pending, []),
5. Response = base_resource_creator:new_base_resource(top_sup, ChildBC,
   PluginSettingsFile),
```

The response from the API function process is then checked to see if the AI started up correctly. This response is then sent as a reply to the calling *tasks* function process.

If the spawning of the AI was not successful, the task fails and notifies the user that the task failed for reason: *spawn_ai_failed*. However, if the spawning of the AI was successful, the task continues by saving the AI holon's BC to the task contract as the service provider. An instance of the *order_type_fsm* is then spawned to handle the NEU protocol communications between the AT and the AI. The order WP colour is then retrieved from the contract and a WP is then found from the RI's stock. This is done by finding the WP, from the HBW stock status, of the right colour nearest to the position A1 in the HBW (which is closest to the 'home' location of the HBW robot). This is done by simply looping through the stock map, column by column, until a WP of the right colour is found. If no WP is found, the activity is failed with the reason: *no_wps_in_hbw.* This WP's information, along with the contract, is then included in the arguments of the *order_type_fsm:start_order* function, which sends a synchronous event call to the AI's FSM instance.

The *order_type_fsm* has two states, *idle* and *process*. The *idle* state has an entry point for an event called *start_order*. This event starts the execution of the first process step in the process step list for the *order_type* activity. This is done by the AT requesting the AI to perform the *order* task. This task receives the name of the process step task that the AI needs to perform, and then starts the CNP process for the requested process step task. This process step is then removed from the list of process steps, and the FSM transitions to the *process* state, until such a time that it

receives a service package delivery indicating that the requested task has been completed.

The task status is retrieved from the service package, if the task failed, the failed activity state and error reason from the service package is returned to the AT EP where the activity is then updated as *done,* along with the status of *failed*. From the EP the dashboard is then also notified of the activity (*order*) failure. If the task completed successfully, the process steps list is checked to see if there are any remaining process steps required for the activity. If there are none, the current WP information along with the successfully completed activity status is returned to the AT EP where the activity is updated as *done*, along with the status of *completed*, and the dashboard is notified of the successful completion of the activity.

If there are remaining process steps in the list, the same procedure as in the *idle* state is followed, and the FSM transitions again to the *process* state, waiting for the delivery of the executing task's service package, and repeating the process until all process steps have been completed (or until a single task fails). Once all of the process steps have been completed, or one of the steps have failed, the FSM replies to the original calling *tasks* function process with the results of the execution. These results are then included in the reply to the AT EP, where the activity is updated as *done,* along with the status of the activity. The FSM is then terminated from the *order_type* configuration-specific module. Once that has completed, the AI is also terminated.

The other configurations of AT holons are implemented in a similar manner, with the main differences being that the function, task, and activity names are different, and that the process steps specified in the configuration file are different. The AAT holons are also implemented in a similar manner, however for the AAT holons, the client is the PLA RI instead of the Dashboard Service.

5.6.5 Activity Instance Plugins

Since AI holons are dynamically spawned and terminated as activities are started and completed, there is no set number of AIs in the system at one time, but there are four different configurations of the AI and AAI holons. These configurations are tied to the configurations of the AT and AAT holons, as they are their instances. There are three AI configurations: *order*, *delivery*, and *maintenance*; and there are two AAI configurations: *process* and *maintenance.* As with all other holons, plugins and plugin arguments are specified in the configurations' respective start-up configuration files. Similar to before, due to the similarity between configurations of the AI and AAI holons, only the AI holons' implementation will be detailed, with key differences discussed.

5.6.5.1 Scheduling Plugins

The SP of the AI holons are responsible for handling requests from the AT holons and scheduling these requested tasks for immediate execution. The AI SP is thus very similar to the SP of the AT.

5.6.5.2 Execution Plugins

The EP of the AI is mostly similar to the EP of the AT but differs slightly in some areas. The EP also receives service package delivery calls and relays them to the FSM, where the service packages are received from the RIs. The EP start-up settings files are also different, as they do not include a list of the individual process steps that are required to complete the respective activities. Process steps are received one at a time from the AT via the NEU protocol. The EP also spawns a single FSM at initialization to handle execution, similarly to the RI holons. The execution of a task from the EP standpoint is similar to that of the AT holons in that the task is handed to the configuration-specific function module for execution.

5.6.5.3 Reflection Plugins

The RP works in the same way as the AT RP, with different function names for the function calls to reflect on the completed tasks.

5.6.5.4 Analysis Plugins

The AP is similar to that of the AT AP, in that it is responsible for analysing completed tasks and updating the holon's *process_times* attributes based on the analysis. This analysis is done in the configuration-specific module, as before.

5.6.5.5 Configuration Specific Modules and FSMs

Once again, due to a large similarity between the AI and AAI configuration-specific modules, only the implementation of the *order* AI configuration will be detailed, with key differences highlighted. The *order* function module is very similar to the AT configuration-specific modules, however with one major difference: the AI holons do not spawn instance holons or FSM instances.

The *order_fsm* is a module with the behaviour of a *gen_fsm* (as with all of the FSMs in this system implementation), and it has two states: *idle* and *process*. The process of a task executing within the AI is similar to that of a RI, except that the AI communicates with RI service providers and schedules tasks via the CNP, instead of communicating with an MQTT service provider.

The CNP process between the AI and potential service providers, for the service requested by the AT, is handled in the *idle* state of the FSM. A list of BCs of potential service providers is first obtained through the use of the DOHA API. If no service providers are found, the task fails and the AI sends a service package to the

AT with the failed task status and the reason for failure: *no_service_provider*. If service providers are found, the proposed contract is sent to all service providers as part of an RFP, as shown in the code snippet below:

```
1.  {ok, ReceivedProposedContract} =
    gen_server:call(FirstBcAddr,{rfp,Contract}, 1000),
```

A one second call timeout was chosen for the timeout of the call, as it was unknown what the maximum time would be between an RFP call being sent and the reply, containing the proposal, being received. It was evident that an experiment needed to be performed to determine what the average duration of the CNP process was. The received contract, including proposal, that is received as a reply to this call is then examined to see whether the RFP was accepted or rejected. If the proposal was accepted by the RI, the proposal is added to a list of received proposals. Once all the proposals have been received, the best proposal is selected by examining which proposes the earliest estimated completion time for the requested task. The service provider responsible for this proposal is then selected as the service provider for the requested task. The start time and service provider in the proposed contract is then updated with the start time and service provider specified in the selected proposal, and this contract is then sent as a part of a service request to the selected service provider. This step acts as the *accept* step from the client, in the CNP. The FSM then transitions to the *process* state.

Similar to the *process* state in the AT, the FSM awaits a service package delivery from the service provider of the executing task. The task status of the service package is then examined to determine if the task has successfully completed or if it had failed. If the task failed, the task status, and any other information that the AI would like to return to the AT before its termination (such as attribute information), is returned to the EP, where the task status is updated as *failed* and the information is added to the S2Data. This data is then returned to the AT by the RP, as a part of the service package. A similar process is followed for a successfully completed activity, where the current WP information is also added to the service package, for use by the AT. The FSM then returns to the *idle* state, awaiting the start of the next process step.

5.6.6 Basic Resources

5.6.6.1 MQTT Service

The MQTT Service basic resource is implemented as a generic server process and utilises the EMQTT library (Lee, 2012) for MQTT communications. It provides the *mqtt* service to the HMES. The MQTT broker network IP address as well as the broker's username and password are defined for use in connecting to the broker. During initialization, the *emqtt* server is started and connected to the MQTT broker. The *emqtt* server then communicates with the *emqtt_service* module to provide its functionalities as a service to the HMES. It allows for the subscription and publication to MQTT topics and saves the IDs of each holon that is subscribed to a

certain topic. When a message is received on a topic, the subscription list is checked to see which holons are subscribed to that topic, and the message is sent to that holon via the inter-holon network as an *inform* message.

5.6.6.2 Dashboard Service

The Dashboard service is also a basic resource that provides the *dashboard* service to the HMES. This service includes delivering *order, delivery,* and *maintenance* requests from the dashboard to the HMES, which is done by using the DOHA API to find either *order_type, delivery_type,* or *maintenance_type* service providing holons in the HMES and sending them a request to perform the respective task. The service also caters to *alert_dashboard* requests, in which a holon can request the *dashboard_service* basic resource to display an alert on the web dashboard with the requested content. The dashboard service also requests status updates from the holarchy periodically (in the case study set to 1 Hz) in order to provide the system status to the user, via the web dashboard, in near real-time. This is done by the dashboard requesting the *activity_type_status, resource_type_status_monitoring,* and *resource_instance_status_monitoring* services from all respective holons in the holarchy.

5.7 User Interface

The system includes a web-based user interface dashboard which is built on top of the holarchy management web-dashboard created in the 'BASE-Factory' implementation of the BASE architecture. The initial dashboard included pages for viewing existing holons (BASE holons), viewing individual holon information, adding, or removing holons, adding or removing basic resources, adding or removing users, as well as a page for managing plugins. Functionality was added to the holon viewing pages to enable the functionality to restart a resource if necessary. A new page was also added named "Order Dashboard" which allowed the user to interact with, view, and control the implemented system. A list of functionalities of the dashboard can be found in Appendix C. Screenshots of the Order Dashboard are shown in Figure 12 and Figure 13. The dashboard runs on an HTTP server which was implemented using the Cowboy library (Hoguin, 2012). The web page for the dashboard was developed using a combination of HTTP, CSS, and JavaScript, and communicates with an Erlang web socket module, which in turn communicates with the Dashboard Service basic resource.

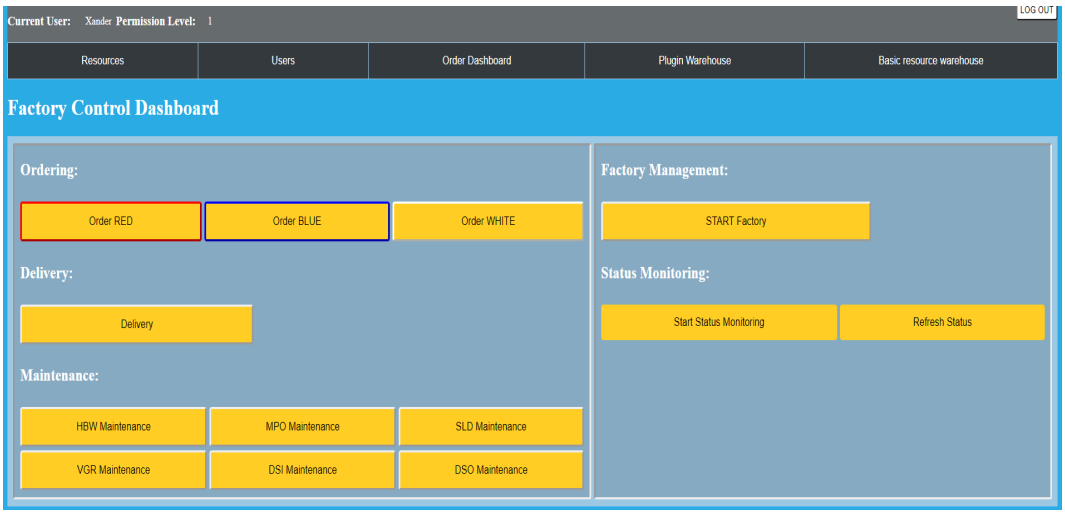

Figure 12: Factory Control Dashboard Screenshot (Image 1 of 2)

Figure 13: Factory Control Dashboard Screenshot (Image 2 of 2)

5.8 Discussion

This section aims to briefly discuss a few points of interest in the implementation of the HMES. Most of these points are not evaluated in Chapter 6, but are deemed significant enough to warrant mentioning.

Although the implementation of the BASE architecture is thorough, there were two functionalities that were found to not be present in the implementation that was used. This included checking of the validity of contracts used when requesting the services of basic resources, and a method with which to get the BC of a holon from the EP. Both of these functionalities were added for use in this implementation. However, it must also be noted that these issues have been resolved in the most recent implementation of the BASE architecture.

The BASE architecture aided the implementation of the HMES by providing a comprehensive system with which the ARTI holons could be implemented. The Communications Manager assisted greatly with communication between holons, especially sending/receiving RFPs, service requests and basic service requests. Request handling was also assisted greatly by the CM as checks were done to ensure contracts were valid and that the requested service provider is capable of that particular service. Task scheduling and the initiation of task execution was greatly aided by the BASE Schedule component, which handled this in the background. The BASE Attributes component also greatly aided with the management and use of attributes, which included adding additional attributes, and globally updating attributes. Lastly, the DOHA greatly reduced the difficulty of holons registering their service, and for other holons to be able to find these services.

It is believed that the implementation of this ARTI-based HMES was effectively supported through the use of the BASE architecture implementation as a backbone and administration shell for the ARTI holons that the system consisted of. The next chapter aims to quantify this.

6 Case Study Evaluation

This chapter aims to evaluate the implemented system as detailed in the previous chapter, against the requirements set out in section 4.3. This process includes the formulation of evaluation criteria that will be used in the testing of the system, the design and execution of experiments that aim to evaluate the system based on the evaluation criteria, and the interpretation of the experiment results.

6.1 Evaluation Criteria

The requirements set out in section 4.3 are predominantly qualitative requirements, as quantitative requirements are largely not applicable to the implementation requirements of this case study system. For evaluation purposes, it is however helpful to have quantitative measures with which the performance of the system can be measured. Thus, it is necessary to formulate a set of criteria by which the system requirements can be quantitatively evaluated. These metrics were obtained largely from literature, as was discussed in section 2.2.2 and 2.2.3 of the Literature Review. Table 6 provides a relationship matrix between system requirements and the selected evaluation criteria. The remaining system requirements are evaluated qualitatively.

Table 6: Relationship Matrix between Requirements and Evaluation Metrics

		Evaluation Metrics:							
		Quantitative							Qualitative
		New Lines of Code	Changed Lines of Code	Development Time	Code Reuse Rate	Production Time	Resource Utilization Rate	Throughput	Error Handling
Requirements:	Reconfigurability	X	X	X	X		X	X	
	Robustness								X
	Decision Making						X		
	Goal-Orientated						X		
	Self-Learning						X	X	
	Throughput Maintained					X	X	X	X

These evaluation criteria were selected to be as quantifiable as possible, while maintaining the ability to, completely and accurately, evaluate the performance of

the implemented system. Some criteria are however still qualitative, as some requirements, such as that of being goal-orientated, robust, and autonomous, are not suited to being quantifiably measured. The evaluation metrics, and how they were measured, is discussed in Appendix D. Additional qualitative metrics are also discussed within each experiment.

6.2 Experiments

The experimenting phase of the evaluation aimed to gather evaluation metric data, which could then be used to evaluate the implemented system. The experimentation of the system was carried out over three experiments in order to collect sufficient data. The first experiment was a baseline experiment, which collected baseline data that was used for comparison against data from the remaining experiments. The second experiment was a reconfiguration experiment, which primarily tested the system's ability to be reconfigured, and the third experiment was a robustness experiment, which primarily tested the system's error/fault handling and tolerance.

For the experimentation phase of the evaluation, tests were only conducted on the *order* activity process. This was done to provide a consistent testbed for the evaluation. The *order* activity was chosen specifically for two reasons: it utilises all resources within the Mini-Factory, providing a good overview of the performance of the entire system; and it is the most emphasised activity that is of importance in the manufacturing environment.

6.2.1 Baseline Experiment and Evaluation

The baseline experiment was a vital experiment, as the data collected in this experiment was used to compare to the data collected in the reconfigurability and robustness experiments, providing a baseline from which to evaluate the system's performance. The baseline experiment was split into two parts: baseline testing of the system in a default configuration, similar to the default configuration of the Mini-Factory; and baseline testing of the system with the MPO and SLD resources aggregated into a single aggregate resource, the PLA. The *order* process steps list needed to be standardised for the baseline test, to ensure that the tests were repeatable and relatable to each other. Thus, the standard *order* process steps list was defined as: *hbw_fetch_wp, vgr_order, mpo_produce, sld_sort* and finally *vgr_dispatch.*

6.2.1.1 Default Resource System – Experiment 1a

6.2.1.1.1 Experiment Description

The system was initially tested in its default configuration without aggregation. This gave a 'true' baseline, from which the effect the aggregation of the MPO and SLD RIs could be assessed. The baseline tests, that this experiment consisted out of, involved ordering three WPs, one of each colour, and waiting for the process to

complete. It was ensured that the HBW was fully stocked before each experiment, with the WPs organised in columns, by colour. This ensured consistent times for the HBW robot to fetch the three required WPs, for each test.

The first measured metric for this test was the individual order start times and end times, as recorded by the system. The order durations were then calculated from these times, with system throughput, measured in WPs per minute, calculated from the total number of WPs produced (three WPs, if all were completed successfully) divided by the average duration of a test. These tests were each repeated three times, and the average times from the three tests were calculated. This was done to ensure the validity and repeatability of the data. The second measured metric during the baseline test was that of the average duration of the execution of the CNP. This provides valuable insight into possible delays caused by the scheduling technique used, where the CNP is carried out after each completed process step. Another requirement that was tested for in this experiment, was that of robustness. Thus the error/fault handling of the system during these tests was observed and noted. Resource utilization was also measured in order to determine to what extent each the resources were being used during the testing process. This was measured as a percentage of the total test duration that each resource was busy. Lastly, source lines of code were counted.

6.2.1.1.2 Results

The results of the baseline experiment with no aggregation are shown below in Table 7 and Table 8. Table 7 shows the average duration of a test, system throughput in WPs per minute, the average measured CNP duration, and the source lines of code for the implementation. Lastly, average resource utilization rate is shown in Table 8.

Table 7: Experiment 1a Results

Average Test Duration	System Throughput	CNP Average Duration	Source Code
00:06:36	0,455 *WPs/min*	126.3 *ms*	13,138 *lines*

The test durations for this experiment were highly consistent, with the durations ranging from 00:06:31 to 00:06:37. This indicates a stable system implementation and scheduling system. CNP duration was measured from the time that the first RFP was sent, until the time that the selected RI replied to the final *request_service* call. This time varied between 63ms and 203ms depending on the number of currently executing and scheduled tasks, which increased scheduling difficulty. Source lines of code were manually counted, and included all code written for the implementation, and excluded BASE core code.

Table 8: Experiment 1a Resource Utilization

Resource	HBW	MPO	SLD	VGR
Average Utilization	65.74%	45.09%	46.80%	53.15%

It can be seen from the resource utilization that the HBW resource was a bottleneck in the system, with the VGR also being a secondary bottleneck. It can also be noted that overall, the resource utilization rate is relatively low, which indicates that optimizations can be made to the scheduling of system tasks.

6.2.1.2 Aggregated Resource System – Experiment 1b

6.2.1.2.1 Experiment Description

The second part of the baseline experiment tested the system in its aggregated configuration, where the MPO and SLD RIs were aggregated into a single PLA RI. This served as the baseline to which the remaining experiments were compared. This baseline experiment was conducted in the same manner as the first baseline experiment, with the same metrics being measured, along with additional development metrics such as development time, and code reuse rate. The *order* process steps list needed to be updated for the second baseline experiment, as the services provided by the MPO and SLD were now aggregated into a single PLA service. Thus, the *order* process steps list was set as: *hbw_fetch_wp, vgr_order, pla_process* and finally *vgr_dispatch.*

6.2.1.2.2 Results

The results of the baseline experiment with aggregation are shown in Table 9 and Table 10. Table 9 shows the average duration of a test, system throughput in WPs per minute, the average measured CNP duration, the source lines of code for the implementation, as well as other development metrics. Lastly, average resource utilization rate is shown in Table 10.

Table 9: Experiment 1b Results

Average Test Duration	System Throughput	CNP Average Duration	Source Code
00:06:33	0,458 *WPs/min*	145.38 *ms*	20,410 *lines*
Additional Lines of Code	**Development Time**	**Changed Lines of Code**	**Code Reuse Rate for New Lines**
8,077 *lines*	32 *hours*	382 *lines*	91.16%

From these results it can be seen that the implementation of aggregation within the HMES did not lead to any reduction in performance in the system as the *system throughput* remained relatively constant, with the new average test duration being

on average three seconds faster, which is within margin of error. The new measured CNP average duration was approximately 19 ms higher than previously. This higher CNP duration was however expected, as the aggregated CNP process is more complex and involves more messages being sent. It can also be seen that the reconfiguration of the system to incorporate an aggregate system required a considerable number of additional lines of code, as new RI, ART, AAT, AAI and ARI holons were needed to be added to the system. A minimal number of lines also needed to be changed in existing plugins and modules, which showed that these plugins/modules were sufficiently generic.

Table 10: Experiment 1b Resource Utilization

Resource	HBW	PLA	VGR
Average Utilization	68.62%	79,39%	56,23%

As seen from the resource utilization, the HBW and VGR resources were very similar to the measured values in Experiment 1a. This once again shows that the introduction of aggregation did not negatively impact the system's performance. It can be seen that the utilization rate for the PLA aggregate resource was high, which was expected as this shows the utilization for the combination of the MPO and the SLD.

6.2.1.2.3 Discussion of Results

As can be seen from the results of these two baseline experiments, the system operated consistently and predictably, which provides for a good baseline from which to evaluate the upcoming experiments. These results also showed that it was possible to implement aggregation into the HMES without noticeably sacrificing performance, which is useful for simplifying complex processes and interactions. These experiments also showed preliminarily that it is possible to reconfigure the system without needing to redevelop the entire system. Full results of the baseline experiments can be seen in Appendices E.1 and E.2.

In terms of fault detection and error handling, no errors or faults occurred during these experiments, however two faults occurred during the pre-testing for the experiments. Both faults were SLD resource hardware faults, where the hardware went offline during the execution of an activity. This caused the respective RI holon to timeout during execution, which caused the currently executing process step to fail. This in turn caused the order activity to fail, and the user was notified via the web dashboard. Error/fault detection, handling and reporting is handled in more depth in section 6.2.3.

6.2.2 Reconfigurability Experiment and Evaluation

The second experiment was aimed at testing the system's ability to be reconfigured. This was done as reconfigurability is commonly known as one of the major benefits/advantages of HMESs, and it would be valuable to demonstrate the developed system's ability to be reconfigured with relative ease. It would also be beneficial to quantify the extent to which using the ARTI architecture, as well as the BASE architecture, assists and lessens the development effort of system reconfiguration. This experiment consisted out of two parts, the first was to develop a new type of resource and add it to the system, and the second was to add an additional instance of an existing resource to the system.

6.2.2.1 New Resource Addition – Experiment 2a

6.2.2.1.1 Experiment Description

For the first part of the experiment, a new type of resource was added in series to the existing production line. The new resource was arbitrarily chosen as a Packaging Station (PKG) which would simulate the packaging of WPs after they came off of the sorting line. It would be the VGR's responsibility to move the WP from the SLD to the PKG, and from the PKG to the DSO. This however meant that a new process steps list was required for the *order* process. The new process steps list was: *hbw_fetch_wp, vgr_order, pla_process, vgr_pla2pkg, pks_package,* and finally *vgr_dispatch_pkg.*

The additional resource was added in order to fully test the system's reconfigurability, as a new set of plugins had to be developed for the PKG RI, and additions had to be made to the RT to include the new resource. The ease of reconfigurability of the system was also quantified by this experiment, as various metrics such as development time and code reuse rate were measured. For the purposes of the experiment, the physical hardware of the PKG was not developed/implemented due to spatial constraints and constraints on the VGR's reach. The existing 'reject' bin in the Mini-Factory was repurposed to be used as a temporary PKG resource. An ESP32 microcontroller with a proximity sensor was then added to the reject bin to detect the presence of a WP and was connected to the factory's MQTT broker as a client. The ESP32 was programmed to send an acknowledge message and start a timer for an arbitrary amount of time (15 seconds in this case) when it detected a WP being placed in the bin, and then when the timer completed, send another acknowledge message, indicating that the process had completed.

Adding the ESP32 MQTT client to the system also proved to be an experiment in itself, as it demonstrates the system's ability to be platform agnostic, allowing it to interface with any type of device that communicates via the correct protocols. This would be a valuable attribute for an implementation in the real world, as often systems are comprised out of a variety of different makes and types of controllers.

Once the new PKG resource was added to the system, the same test as used in the baseline was then run. The same metrics were captured from the experiment, with the addition of measuring the number of lines of code that needed to be changed or added for the additional resource, the code reuse rate for the additional resource, and the development/implementation time. RI attributes were also recorded during this experiment, to provide an indication of the system's ability for self-learning.

6.2.2.1.2 Results

The results of experiment 2a are shown below in Table 11 and Table 12. Table 11 shows the average duration of a test, system throughput in WPs per minute, the average measured CNP duration, the source lines of code for the implementation, as well as other development metrics. Lastly, average resource utilization rate is shown in Table 12.

Table 11: Experiment 2a Results

Average Test Duration	System Throughput	CNP Average Duration	Source Code
00:07:59	0.376 *WPs/min*	142.90 *ms*	21, 017 *lines*
Code Reuse Rate	**Development Time**	**New Lines of Code**	**Changed Lines of Code**
71.99%	7 *hours*	607 *lines*	9 *lines*

As expected, the average test duration increased, and the system throughput decreased in this experiment, due to two additional process steps in the production process. It is noteworthy that the average CNP duration remained relatively constant, indicating that the CNP duration is not significantly affected by increased system complexity on the same hierarchical level. The development time required to develop and implement the new resource was much lower than the time required to implement aggregation in the baseline tests, which shows that simple reconfiguration, such as adding a new type of resource, is greatly aided by the provisions of the ARTI architecture and the ability for BASE plugins to be generic between resource types. This is evidenced by the number of new lines of code, the number of changed lines of code, and the code reuse rate. The relatively low number of new lines of code indicates that the ARTI architecture allows for the addition of new resources, and thus functionality, with a minimal amount of added code. The low number of changed lines of code indicates that the BASE architecture allows for additions to the system without greatly affecting the rest of the system. Lastly, the relatively high code reuse rate indicates that, due to the BASE architecture allowing for plugins to be relatively generic amongst resources, the majority of added lines of code are reused from other plugins. The RI attributes were recorded after each order in each test, and it was observed that the resource's attributes successfully updated to reflect *max_wait_time* as well as *average_wait_time*. These attributes, as well as full results for this experiment can be found in Appendix E.3.

Table 12: Experiment 2a Resource Utilization

Resource	HBW	PLA	VGR
Average Utilization	61.64%	71.52%	61.80%

The resource utilization results are as expected. The HBW and PLA resources saw lower overall utilization as the HBW continued to be a bottleneck in the system, limiting the throughput, and thus utilization, of the PLA, and due to the extended test duration, the utilization rates fell. The VGR, on the other hand, saw increased utilization, as it had two additional tasks that it needed to perform per order.

6.2.2.1.3 Discussion of Results

This experiment showed that system reconfiguration is not only possible in this HMES implementation, but it is possible to do with relatively little development effort. This experiment reconfigured the system by adding a new resource to the system that included new functionalities. It also reconfigured the product, as the AT was also reconfigured, through changing the process steps in the configuration file, which led to a new product being produced by the system. This experiment also provided evidence to support the theory that the ARTI and BASE architectures decrease the development difficulty of HMESs. It must be noted here that the reduced development time per new lines of code compared to experiment 1b is due to the decreased amount of software design needed for adding an additional resource to the system, whereas a significant amount of system design was needed in experiment 1b to implement aggregation.

6.2.2.2 Resources in Parallel – Experiment 2b

6.2.2.2.1 Experiment Description

For the second experiment, a new PLA resource was added in parallel to the existing PLA resource. This was done to demonstrate that the system is capable of utilizing additional resources to increase the overall throughput of the system. For the purposes of the experiment, the physical hardware of the PLA (MPO and SLD resources) was not reproduced due to spatial constraints and constraints on the VGR's reach. The existing 'reject' bin in the Mini-Factory was repurposed to be used as a temporary PLA resource, in a similar manner to experiment 2a. The duration that the WP needed to stay in the simulated PLA2 station was set at a fixed time of ten seconds less than the average processing time of the PLA1. This would give the system the opportunity to make a decision to use the PLA resource that has the shortest production time.

Once the second PLA RI (PLA2) was added to the system, the same test as used in the baseline was then run, including the new PLA RI. The same metrics were captured from the experiment, with the addition of measuring the number of lines

of code that needed to be changed or added for the additional resource, the code reuse rate for the additional resource, and the development/implementation time.

6.2.2.2.2 Results

The results of this second reconfigurability experiment are shown below in Table 13 and Table 14. Table 13 shows the average duration of the tests, system throughput, CNP average duration, lines of code metrics, and development time. Lastly, average resource utilization rate is shown in Table 14.

Table 13: Experiment 2b Results

Average Test Duration	System Throughput	CNP Average Duration	Source Code
00:06:20	0.474 *WPs/min*	152.92 *ms*	20,859 *lines*
Additional Lines of Code	**Development Time**	**Changed Lines of Code**	**Code Reuse Rate for New Lines**
48 *lines*	1 *hour*	41 *lines*	85.42%

Table 14: Experiment 2b Resource Utilization

Resource	HBW	PLA1	PLA2	VGR
Average Utilization	71.92%	0.00%	79.21%	57.48%

As seen by the average test duration being approximately ten seconds less than the baseline, and by the PLA1 utilization rate being zero, the system made the decision to use the PLA2 resource for each order, as it had the shorter production time. Although this result does increase the system throughput, it does not do so by making use of both PLA resources, but by purely using the resource with the shorter production time. The reason for the system not making use of both PLA resources is that there is never a situation in the order process where two WPs are able to use the PLA at the same time, as the HBW creates a throughput bottleneck. Production can only move as quickly as the HBW can fetch WPs, which coincidentally only allows for one WP to travel through the PLA at a time. Although the HBW is a bottleneck, it is seen that its utilization rate is not 100%. This is due to poor scheduling efficiency and could be improved with a more advanced scheduling technique. The results also show that the development time of adding an additional instance of a resource to the HMES is minimal, with the new code that needed to be added having a high code reuse rate. The average CNP duration increased by approximately 8 *ms*, which is expected, as an RFP must be sent to and received from an additional resource in the third production step. Full results for this experiment can be found in Appendix E.4.

6.2.2.2.3 Discussion of Results

This experiment showed that it is possible to reconfigure the system to incorporate an additional instance of a resource with relative ease. The experiment also showed that the system is capable of making goal orientated decisions in terms of resource selection. This experiment did, however, fail to provide evidence that the system is capable of utilizing two of the same resources simultaneously. Theoretically this experiment would have been better performed by creating a second HBW RI, which would alleviate the bottleneck. This was, however, unfortunately not practically feasible. This experiment did however still provide valuable data, as it can be seen that the system is capable of making use of multiple resources of the same type, and that the system's decisions are goal orientated.

6.2.3 Robustness Testing and Evaluation

The last experiment was an experiment to test the robustness of the HMES implementation. Robustness in this case referred to the system's ability to detect, handle, and recover from, faults and/or errors. Robustness is considered as a critical attribute of HMESs as these systems often control the production of high-value or time sensitive products and cannot afford for the whole system to crash when a fault or error occurs. Thus, it is valuable to test the system's robustness in order to determine if the system's fault and error handling works as expected.

6.2.3.1 Experiment Description

For this experiment the system setup as used in experiment 2b was used again, with the two PLA resources in parallel. This provided the opportunity to be able to manually disable one of these resources during production and observe whether the system accurately determined the resource's offline state and observe if the system reacted in a favourable manner, by scheduling the production of the WP with the remaining PLA resource. The results expected from this experiment were purely qualitative. These metrics included the result of the order (completed or failed), if the system informed the user of the error, if the system accurately determined the fault and displayed that to the user, and what the effect was of the disturbance to the rest of the system and orders. This was also followed by the offline resource being brought online again and observing the system's reaction to this.

6.2.3.2 Results

The experiment consisted out of three tests: in the first test the SLD resource hardware (as a part of PLA1) was turned off while the WP from the second order was busy being processed by the PLA; in the second test the SLD was turned off while the WP was being fetched by the HBW; and in the third test, the PLA2 was off for the start of the test, but then switched back on during the production of the first order. It was seen in the first test that the second order failed, this was due to the WP being stuck on the offline SLD conveyor, and thus the SLD timeout being

reached. This fault was reported to the user as an *sld_timeout* error, and the activity status was changed to *failed*. The remaining order completed successfully with the remaining PLA2 resource. In the second test, production carried on as normal, however with the PLA1 resource offline. This then led to the system only receiving a proposal from the PLA2 resource during the CNP phase, and thus the system only used this resource. In the third test, while the PLA2 resource was offline, the first order was produced as normal with the PLA1 resource. Just as the second order started, the PLA2 resource was turned back on. The second and third orders then utilized the PLA2 resource, as its processing duration is shorter than that of the PLA1. The results of the orders in this experiment are shown in Table 31 in Appendix E.5.

6.2.3.3 Discussion of Results

This experiment showed that the developed HMES was resilient to disturbances as well as faults/errors. When a resource went offline, the system accurately detected the issue, and handled it by diverting the following orders through an alternate resource. The failure of an order in the first test resulted in a WP becoming stuck on the production line. This is however acceptable, as in a real-world system it would be necessary for a worker to inspect the fault and ensure that the system is safe to resume operations, including removing failed products from the production line. This experiment also showed that the system is capable of readopting resources which have come back online during the production process. It must be noted that the system took a long time to detect a failure, as a failure would only be detected once the resource had timed out (the timeout value is equal to the maximum production time plus five seconds). A more advanced activity status monitoring system would allow for the failure of a production step to be detected sooner. This could be done by having the resource send acknowledge messages at more stages during production steps.

6.3 Discussion

These experiments aimed to provide the results necessary to determine whether the implemented ARTI-based HMES satisfied the system requirements as set out in section 4.3. Experiments 1b, 2a and 2b demonstrated the system's ability to be reconfigured with relative ease. These tests included: reconfiguring the system to incorporate aggregation; adding new resources to the system; and adding more instances of a single resource. Minimal additional lines of code were necessary for the addition of a new resource, and very few were necessary for the addition of an additional instance of a resource. The code reuse rate for these experiments was also high, which indicated that most of the functionality already existed elsewhere in the system and could be reused. Additionally, very few lines of code had to be changed elsewhere in the system when these experiments were run, indicating that it is possible to reconfigure the system without affecting the rest of the system. The development time of these experiments was also relatively low, as the combination of the previously mentioned metrics resulted in decreased development difficulty.

Experiment 3 demonstrated the system's robustness. It was demonstrated that the system is fault tolerant, being able to detect and handle most hardware faults. It does so by checking if a task has been completed within a certain maximum time, and if not, an error is registered. This method allows for the detection of faults, but not in a very timely manner. When a fault is detected, it is accurately handled, and the respective task is failed. An error is presented to the user to inform them of the fault. The system purposefully does not handle faults by restarting the failed task, as often in a manufacturing environment it is necessary for a user to check the hardware for physical problems/issues before manufacturing can continue.

The system's decision-making and goal orientation was demonstrated in all experiments, but primarily in experiment 2b. When presented with a choice of two production paths to take, the system consistently chooses the path that leads to the fulfilment of its goal: completing the required production steps in the shortest time possible. This goal-orientation and decision making is captured in the system's ability to follow the CNP when scheduling production steps with resources. The system was also shown to possess the capability of self-learning. The system demonstrated this through updating its attributes, as shown in Experiment 2a. The attribute of primary importance that was updated was the *max_wait_time* attribute for each service type, as this constantly updating time value was used in the scheduling of tasks.

Throughout the experiments, the throughput of the system was measured, as this would be the requirement of the most importance to a manufacturing plant. Transitioning between baseline experiments 1a and 1b showed no noticeable reduction in throughput, even though system complexity was increased dramatically. Experiment 2a showed a slight reduction in throughput as expected, as an additional process step was added to the production process, and experiment 2b showed an increase in the system throughput, but not as a dramatic increase as one would hope. This was due to the MPO and SLD throughput being bottlenecked by the HBW's throughput. It is expected that if this experiment were repeated, except with an additional HBW instead of PLA, the system throughput would increase substantially. Throughout all experiments the system's ability for collaboration and autonomy was demonstrated through having multiple resources work together to achieve a common goal, as well as being capable of performing certain tasks independently.

Overall, the system performed well and achieved all of the set-out requirements for the implementation of an ARTI-based HMES using the BASE architecture. The primary improvements that could be made to the system would be with the scheduling and status monitoring subsystems. The overall low resource utilization rate can be attributed to low efficiency scheduling which doesn't make use of the resources' full capacities. The status monitoring system could also be improved to more accurately and timeously detect a resource's hardware status and react faster to faults.

7 Conclusions and Recommendations

This thesis presents an ATRI architecture based HMES that was implemented using the BASE architecture as a building tool for development of the holonic system. This implemented system was used for the fulfilment of the objectives as set-out in section 1.2 of this thesis.

An ARTI-based HMES implementation for a case study system was completed on the Fischertechnik Industry 4.0 Training Factory. This system was implemented using the BASE architecture for the development of the system holons. Once the HMES implementation was completed, it was evaluated using evaluation criteria and performance metrics compiled from literature. The system was developed with the primary requirements of being reconfigurable and robust, while not sacrificing throughput. It was found in the evaluation of the system that the system met all requirements, however there were improvements that could be made.

The ARTI architecture proved to be a valuable reference framework for use in the implementation of HMESs as it assists in functionally and logically splitting the system into its respective holons, and it guides the developer in the implementation thereof. The structure in which the ARTI architecture separates Resources and Activities, and their Types and Instances allows for the development of systems which are reconfigurable, scalable, intelligent, and robust. This structure also aids the developer in implementation as it breaks down a highly complex system with complex interactions into smaller, less complex, and manageable components (holons), which can each be tested individually during development.

This case study HMES implementation also provided the opportunity to evaluate the BASE architecture's suitability for use in a manufacturing environment. During reconfiguration experiments, the BASE architecture demonstrated its ability to greatly reduce the development time and effort by allowing for holon plugins to be greatly generic between the holons, which lead to favourably high code reuse rates during the evaluation experiments. The BASE architecture also aided development by making use of holonic principles, in that when the system was reconfigured, the changes hardly affected the other holons in the system. The BASE architecture also greatly simplified the implementation of all the ARTI holons, as the administration functionalities were already a part of the BASE architecture implementation. From the case study implementation and evaluation, it can be concluded that the BASE architecture is indeed suitable for the manufacturing industry and is capable of being used as the foundation for all ARTI holons.

Multiple recommendations can be made for the continuation of the work carried out in this thesis:

- The implementation of a DMAS within the HMES for use in advanced scheduling and disturbance handling. It is believed that the overall system performance could be increased through the use of a more complex

scheduling system such as DMAS, as it was seen during system evaluation that the scheduling efficiency of the system is poor. It could also improve fault and disturbance handling as the DMAS constantly looks for feasible/infeasible production pathways and optimizes scheduling based on this information.

- The implementation of a similar HMES on the same case study system, however, without using any reference architectures. This would provide a more informative baseline for the evaluation of to what the extent reference architectures assist the development of HMESs.

In conclusion, it has been demonstrated that the ARTI and BASE architectures are both individually suitable for the implementation of HMESs, but, when used together, can become a powerful toolset with which the development difficulty of HMES implementations can be greatly reduced, and thus, lessening one of the major barriers to widespread I4.0 enabling technology adoption.

8 References

Adolph, L., Anlahr, T., Bedenbender, H., Bentkus, A., Brumby, L., Diedrich, C., Dirzus, D., Elmas, F., Epple, U., Friedrich, J., Fritz, J., Gebhardt, H., Geilen, J., Hecker, C., Heidel, R., Hemberger, K., Hiensch, S., Hilgendorf, E., Hörcher, G., Klemm, E., Mehrfeld, J., Metzger, T., Middelkamp, S., Mosch, C., Nickel, P., Pichler, R., Prinz, C., Rauchhaupt, L., Rolle, I., Sasaki, F., Seidel, U., Stein, J., Tieves-Sander, D., Ullrich, C., Weber, I., Wei, W., Winkel, L., 2016. German Standardization Roadmap - Industry 4.0. *German Institue for Standardization & German Commission for Electrical, Electronic & Information Technologies of German Institute for Standardization and Association for Electrical, Electronic & Information Technologies*. 1–76.

Adolphs, P., Bebenbender, H., Dirzus, D., Ehlich, M., Epple, U., Hankel, M., Heidel, R., Hoffmeister, M., Huhle, H., Kärcher, B., Koziolek, H., Pichler, R., Pollmeier, S., Schewe, F., Walter, A., Waser, B., Wollschlaeger, M., 2015. *Status Report -Reference Architecture Model Industrie 4.0 (RAMI4.0)*.

Almeida, F.L., Terra, B.M., Dias, P.A. & Gonçalves, G.M. 2010. Adoption issues of multi-agent systems in manufacturing industry. *Proceedings - 5th International Multi-Conference on Computing in the Global Information Technology, ICCGI 2010*. (September):238–244.

Bi, Z.M., Lang, S.Y.T., Shen, W. & Wang, L. 2008. Reconfigurable manufacturing systems: The state of the art. *International Journal of Production Research*. 46(4):967–992.

Bigliardi, B., Bottani, E. & Casella, G. 2020. Enabling technologies, application areas and impact of industry 4.0: A bibliographic analysis. *Procedia Manufacturing*. 42(2019):322–326.

Borangiu, T., Oltean, E., Răileanu, S., Anton, F., Anton, S. & Iacob, I. 2019. Embedded digital twin for ARTI-type control of semi-continuous production processes. in *Studies in Computational Intelligence* Vol. 853. Springer. 113–133.

Derigent, W., Cardin, O., Trentesaux, D., Derigent, W., Cardin, O. & Trentesaux, D. 2021. Industry 4.0: contributions of holonic manufacturing control architectures and future challenges. *Journal of Intelligent Manufacturing*. 32(7):1797–1818.

Ericsson Computer Science Labratory. 2021. *Erlang*. [Online], Available: https://www.erlang.org/ [2021, September 13].

Galati, F. & Bigliardi, B. 2019. Industry 4.0 : Emerging themes and future research avenues using a text mining approach. *Computers in Industry*. 109:100–113.

Giret, A. & Botti, V. 2005. Analysis and design of holonic manufacturing systems. in *18th International Conference on Production Research*. 2–6.

Giret, A. & Botti, V. 2006. From system requirements to holonic manufacturing system analysis. *International Journal of Production Research*. 44(18–19):3917–3928.

Grieves, M. & Vickers, J. 2017. *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems*. Springer International Publishing Switzerland.

Hoguin, L. 2012. *Cowboy*. [Online], Available: https://ninenines.eu/ [2021, October 31].

Jones, D., Snider, C., Nassehi, A., Yon, J. & Hicks, B. 2020. Characterising the Digital Twin: A systematic literature review. *CIRP Journal of Manufacturing Science and Technology*. 29(2019):36–52.

Kotak, D., Wu, S., Fleetwood, M. & Tamoto, H. 2003. Agent-based holonic design and operations environment for distributed manufacturing. *Computers in Industry*. 52(2):95–108.

Kruger, K. & Basson, A. 2017. Erlang-based control implementation for a holonic manufacturing cell. *International Journal of Computer Integrated Manufacturing*. 30(6):641–652.

Kruger, K. & Basson, A. 2018. JADE Multi-Agent System Holonic Control Implementation for a Manufacturing Cell. Technical Report, Stellenbosch University.

Kruger, K. & Basson, A.H. 2019. Evaluation criteria for holonic control implementations in manufacturing systems. *International Journal of Computer Integrated Manufacturing*. 32(2):148–158.

Lee, F. 2012. *EMQ*. [Online], Available: https://emqtt.io/ [2021, October 22].

Lee, J., Bagheri, B. & Kao, H.A. 2015. A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems. *Manufacturing Letters*. 3:18–23.

Leitão, P., Colombo, A.W. & Karnouskos, S. 2016. Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges. *Computers in Industry*. 81:11–25.

Leitão, P. & Restivo, F. 2006. ADACOR: A holonic architecture for agile and

adaptive manufacturing control. *Computers in Industry*. 57(2):121–130.

Leitão, P. & 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*. 38(5):699–709.

McKinsey Digital. 2016. *Industry 4.0 after the initial hype: Where manufacturers are finding value and how they can best capture it*. McKinsey&Company.

Paolucci, M. & Sacile, R. 2005. *Agent-Based Manufacturing and Control Systems. New Agile Manufacturing Solutions for Achieving Peak Performance*. CRC Press.

Raj, A., Dwivedi, G., Sharma, A., Beatriz, A. & Sousa, L. De. 2020. Barriers to the adoption of industry 4 . 0 technologies in the manufacturing sector : An inter-country comparative perspective. *International Journal of Production Economics*. 224(October 2019):107546.

Redelinghuys, A., Basson, A. & Kruger, K. 2019. A six-layer digital twin architecture for a manufacturing cell. *Studies in Computational Intelligence*. 803(April):412–423.

Rossouw, J.J. 2021. An ARTI Holonic Architecture Implementation for Table Grape Production Management. Masters Thesis, Stellenbosch University.

Scholz-Reiter, B. & Freitag, M. 2007. Autonomous Processes in Assembly Systems. *CIRP Annals - Manufacturing Technology*. 56(2):712–729.

Schuh, G., Anderl, R., Gausemeier, J., Hompel, M. ten & Wahlster, W. 2017. Industrie 4.0 Maturity Index. *National Academy of Science andEngineering, Berlin.*

Sparrow, D.E. 2021. *The BASE architecture for the integration of human workers into an Industry 4.0 environment.* PhD Dissertation, Stellenbosch University.

Sparrow, D., Kruger, K. & Basson, A. 2020. Activity lifecycle description for communication in human-integrated industry 4.0 environments. *Studies in Computational Intelligence*. 853(January):85–97.

Sparrow, D.E., Kruger, K. & Basson, A.H. 2021. An architecture to facilitate the integration of human workers in Industry 4.0 environments. *International Journal of Production Research*. 1–19.

Valckenaers, P. 2020. Perspective on holonic manufacturing systems: PROSA becomes ARTI. *Computers in Industry*. 120:103226.

Valckenaers, P. & Van Brussel, H. 2005. Holonic manufacturing execution systems. *CIRP Annals - Manufacturing Technology*. 54(1):427–432.

Valckenaers, P. & De Mazière, P.A. 2015. Interacting holons in evolvable execution systems: The NEU Protocol. in *Proceedings of 7th International Conference, HoloMAS 2015* Vol. 9266. V. Mařík, A. Schirrmann, D. Trentesaux, & P. Vrba (eds.). Cham: Springer International Publishing V. Mařík, A. Schirrmann, D. Trentesaux, & P. Vrba (eds.). 120–129.

Van Brussel, H., Wyns, J., Valckenaers, P., Bongaerts, L. & Peeters, P. 1998. Reference architecture for holonic manufacturing systems: PROSA. *Computers in Industry*. 37(3):255–274.

Van Niekerk, D.J. 2021. *Extending the BASE architecture for complex and reconfigurable Cyber-Physical Systems using holonic principles.* Masters Thesis, Stellenbosch University.

Vyatkin, V. 2015. *Function Blocks for Embedded and Distributed Control Systems Design*. International Society of Automation.

Yin, Y., Stecke, K.E. & Li, D. 2018. The evolution of production systems from Industry 2.0 through Industry 4.0. *International Journal of Production Research*. 56(1–2):848–861.

Zheng, P. & Sivabalan, A.S. 2020. A generic tri-model-based approach for product-level digital twin development in a smart manufacturing environment. *Robotics and Computer-Integrated Manufacturing*. 64(January):101958.

Appendix A Case Study

A.1 Fischertechnik Hardware Components

Each of the six Fischertechnik Industry 4.0 Training Factory stations consist of a different combination of the following sensors and actuators:

- 9V DC motors
- Rotary encoders
- Air compressors
- Pneumatic piston cylinders
- DC solenoid actuated valves
- Limit Switches
- Photoelectric barrier sensors
- Phototransistors
- Photo-resistors
- Environmental sensors
- Near-Field Communication (NFC) readers
- NFC tags
- Colour detection sensors
- Fischertechnik TXT controllers

A.2 Default Order Process Steps

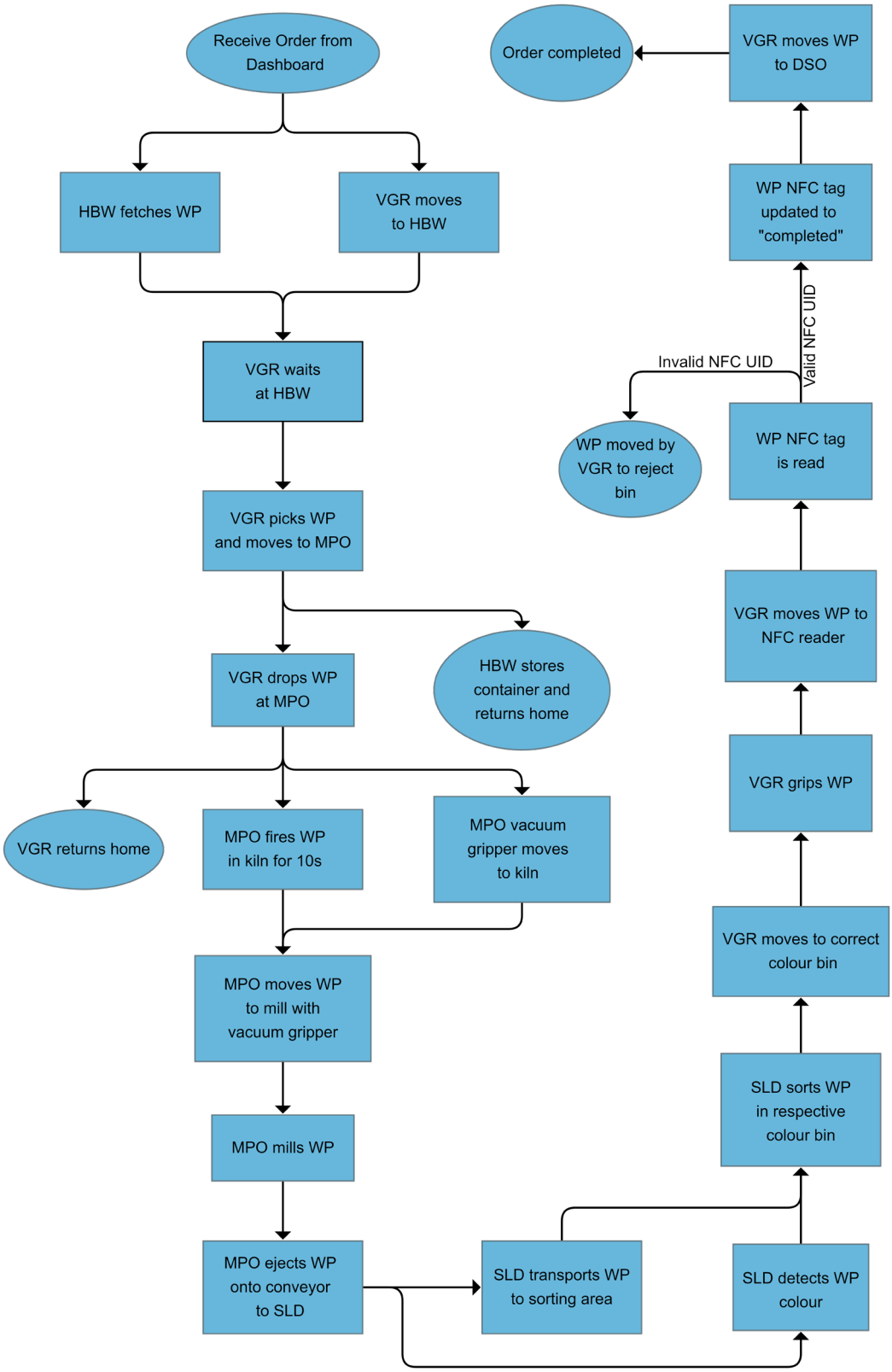

Figure 14: Default Mini-Factory Order Process Steps

A.3 Default Delivery Process Steps

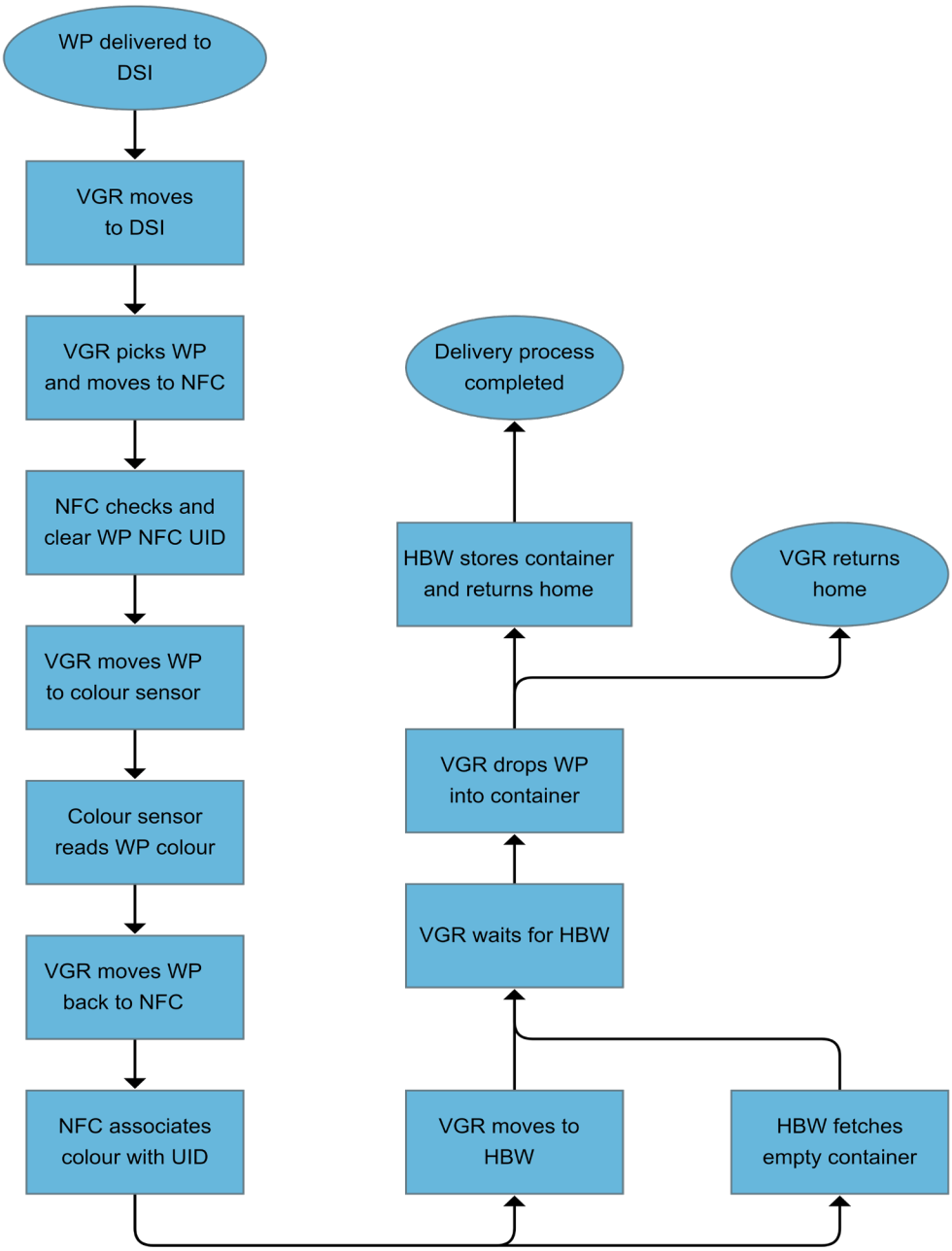

Figure 15: Default Mini-Factory Delivery Process Steps

A.4 Fischertechnik TXT Controller Features

The TXT controllers are designed by Fischertechnik, and their main features are:

- ARM Cortex A8 processor
- 128MB flash storage
- 8 universal digital and analogue inputs
- 4 quick counter digital inputs
- 4 9V PWM motor outputs
- Touch screen display
- WLAN/Bluetooth module
- USB connection port
- 10-pole pin row to increase I/O
- I2C-port

Appendix B MQTT Hardware Interface

Table 15: MPO MQTT Interface

TXT Factory MPO			
Subscribe	**Topic**	**Payload**	**Description**
TOPIC_DO_MPO	base/do/mpo	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "code":0, "workpiece":{...} }	**Code:** 0=MPO_QUIT, 1=MPO_PRODUCE
Publish	**Topic**	**Payload**	**Description**
TOPIC_STATE_MPO	state/mpo	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "station":"hbw", "code":0, "description":"text", "active":1, "target":""}	
TOPIC_MPO_ACK	mpo/ack	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "code":0 }	**code**: 1=MPO_STARTED, 2=MPO_PRODUCED

Table 16: HBW MQTT Interface

TXT Factory HBW			
Subscribe	**Topic**	**Payload**	**Description**
TOPIC_DO_HBW	base/do/hbw	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "code":0, "workpiece":{...} }	**code**: 0=HBW_EXIT, 1= HBW_FETCH_CONTAINER, 2= HBW_STORE_WP, 3= HBW_FETCH_WP, 4= HBW_STORE_CONTAINER, 5= HBW_RESETSTORAGE
Publish	**Topic**	**Payload**	**Description**
TOPIC_HBW_ACK	hbw/ack	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "code":0, "workpiece":{...} }	**code**: 1=HBW_FETCHED, 2=HBW_STORED
TOPIC_STOCK	hbw/stock	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "stockItems": [{ "workpiece": { "id":"123456789ABCDE", "type":"<BLUE/WHITE/RED>", "state":"<RAW/PROCESSED>" }, "location":"A1" },{ ... },{ "workpiece":null, "location":"B3" }] }	

TOPIC_STATE_HBW	state/hbw	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "station":"hbw", "code":0, "description":"text", "active":1, "target":""}	

Table 17: VGR MQTT Interface

TXT Factory VGR			
Subscribe	**Topic**	**Payload**	**Description**
TOPIC_DO_VGR	base/do/vgr	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "code":0, "workpiece":{...} }	**code**: VGR_HOME = 0, VGR_MOVE_DELIVERY_IN_GRIP = 1, VGR_MOVE_NFC = 2, VGR_MOVE_COLOR = 3, VGR_MOVE_HBW = 4, VGR_HBW_PICKUP = 5, VGR_MOVE_MPO_RELEASE = 6, VGR_MOVE_SSD1 = 7, VGR_MOVE_SSD2 = 8, VGR_MOVE_SSD3 = 9, VGR_MOVE_DISPATCH_RELEASE = 10, VGR_NFC_READ = 11, VGR_NFC_DELETE = 12, VGR_MOVE_REJECT_RELEASE = 13, VGR_QUIT = 14, VGR_GRIP = 15, VGR_RELEASE = 16, VGR_SLD_SORTED = 17, VGR_HBW_FETCHED = 18, VGR_ORDER = 19, VGR_MPO_STARTED = 20, VGR_PLA1_SORTED = 21, VGR_PLA2_SORTED = 22, VGR_MOVE_PLA1 = 23, VGR_MOVE_PLA2 = 24
Publish	**Topic**	**Payload**	**Description**
TOPIC_VGR_ACK	vgr/ack	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "code":0, "workpiece":{...} }	**code**: VGR_STARTED = 1 VGR_COMPLETED = 2
TOPIC_STATE_VGR	state/vgr	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "station":"vgr", "code":0, "description":"text", "active":1, "target":"hbw"}	

TOPIC_STATE_DSI	state/dsi	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "station":"dsi", "code":0, "description":"text", "active":1}	
TOPIC_STATE_DSO	state/dso	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "station":"dso", "code":0, "description":"text", "active":1}	

Table 18: SLD MQTT Interface

TXT Factory SLD			
Subscribe	**Topic**	**Payload**	**Description**
TOPIC_DO_SLD	base/do/sld	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "code":0, "workpiece":{...} }	**code**: SLD_QUIT=0, SLD_SORT=1
Publish	**Topic**	**Payload**	**Description**
TOPIC_STATE_SLD	state/sld	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "station":"sld", "code":0, "description":"text", "active":1, "target":"hbw"}	
TOPIC_SLD_ACK	sld/ack	{"ts":"YYYY-MM-DDThh:mm:ss.fffZ", "code":0, "type":<>, "colorValue":<> }	**code**: 1=SLD_STARTED, 2=SLD_SORTED

Appendix C User Interface Functionalities

The user interface ‘Order Dashboard’ had the following functionalities:

- Button to start all system holons.
- Button to start continuous system status monitoring.
- Button to refresh dashboard status display in case background refresh of webpage did not work correctly.
- Buttons to order red, blue, or white WPs.
- Button to start delivery of ‘RAW’ WP to the Mini-Factory.
- Button for each RI to schedule maintenance.
- Display status of activities in progress.
- Display status of completed activities.
- Display Mini-Factory WP stock.
- Display Mini-Factory RI states. Green indicates online, yellow indicates currently busy, and grey indicates offline.

Appendix D Evaluation Metrics

An explanation of the formulated evaluation metrics is provided in Table 19 in this appendix. This will additionally include an explanation of how the metrics are measured during experimentation.

Table 19: Evaluation Metric Descriptions

Metric:	**Description:**	**Measurement:**
New Lines of Code [lines]	This is the number of new Source Lines of Code that were added to the programme. This helps quantify the development and reconfiguration effort. Lower is better.	This is measured by physically counting new lines of code that are added to the programme during the development of the experiment.
Changed Lines of Code [lines]	This is the number of existing Source Lines of Code that needed to be changed for the development and implementation of the experiment. This provides an indication of the modularity of the system, as well as the reconfigurability of the system. Lower is better.	This is measured by physically counting changed lines of code during the development of the experiment.
Development Time [hh:mm:ss]	This is the number of hours, rounded up, that were required for the development and implementation of the features/function required for the experiment. This helps quantify the development and reconfiguration effort. Lower is better.	This is measured through timing the actual development time using a stopwatch.

Code Reuse Rate [%]	This is the percentage of reused lines of code compared to new lines of code. This helps quantify the development and reconfiguration effort. Higher is better.	$\frac{reused\ lines\ of\ code}{new\ lines\ of\ code} * 100$
Production Time [hh:mm:ss]	This is the processing duration of a single WP order. This is used in the calculation of factory throughput. Lower is better.	Calculated using the difference between the system's recorded ending time and start time for an individual order.
Resource Utilization Rate [%]	This is the percentage of the total test duration that a resource is busy working on/with a WP. This metric effects throughput. This provides an indication of the system's reconfigurability, decision making, goal-orientation, and ability for self-learning. Higher is better.	$\frac{time\ busy}{test\ duration} * 100$
Throughput [WPs/min]	This is the number of WPs that the system can produce per minute. This provides an indication of the system's reconfigurability and ability for self-learning. Higher is better.	$\frac{\#\ of\ WPs\ produced}{test\ duration}$

Error Handling	This is the ability of the system to handle faults and errors accurately and in a desirable manner. This includes detecting errors/faults in a timeous manner. This is a qualitative metric used to determine robustness. This also effects throughput.	Qualitative.

Appendix E Experiment Results

E.1 Experiment 1a

Table 20: Experiment 1a Order Results

	Activity Type	Start Time	End Time	Result	Order Duration	Test Duration
TEST 1	ORDER WHITE	12:20:01	12:23:47	COMPLETED	00:03:46	
	ORDER BLUE	12:20:04	12:25:31	COMPLETED	00:05:27	
	ORDER RED	12:20:08	12:26:38	COMPLETED	00:06:30	0:06:37
TEST 2	ORDER RED	12:55:15	12:59:01	COMPLETED	00:03:46	
	ORDER BLUE	12:55:18	13:00:44	COMPLETED	00:05:26	
	ORDER WHITE	12:55:21	13:01:52	COMPLETED	00:06:31	0:06:37
TEST 3	ORDER BLUE	13:07:56	13:11:41	COMPLETED	00:03:45	
	ORDER WHITE	13:07:59	13:13:26	COMPLETED	00:05:27	
	ORDER RED	13:08:02	13:14:27	COMPLETED	00:06:25	00:06:31
TEST 4	ORDER BLUE	13:18:48	13:22:33	COMPLETED	00:03:45	
	ORDER WHITE	13:18:50	13:24:17	COMPLETED	00:05:27	
	ORDER RED	13:18:51	13:25:25	COMPLETED	00:06:34	00:06:37
Average Duration					00:05:14	0:06:36
System Throughput						0,455

Table 21: Experiment 1a Resource Utilization

Resource	Active time	Test Duration	Utilization %
HBW	00:04:21	0:06:37	65,74
MPO	00:02:59	00:06:37	45,09
SLD	00:03:03	00:06:31	46,80
VGR	00:03:31	00:06:37	53,15

Table 22: Experiment 1a CNP Durations

CNP Duration Non-aggregated:					
Test 1	step 1	63	Test 2	step 1	63
	step 2	94		step 2	94
	step 3	110		step 3	187
	step 4	141		step 4	121
	step 5	187		step 5	203
AVERAGE:					**126,3 ms**

E.2 Experiment 1b

Table 23: Experiment 1b Order Results

	Activity Type	Start Time	End Time	Result	Order Duration	Test Duration
Test 1	ORDER WHITE	10:46:18	10:50:02	COMPLETED	00:03:44	
	ORDER BLUE	10:46:20	10:51:41	COMPLETED	00:05:21	
	ORDER RED	10:46:22	10:52:51	COMPLETED	00:06:29	0:06:33
Test 2	ORDER WHITE	11:10:36	11:14:22	COMPLETED	00:03:46	
	ORDER BLUE	11:10:38	11:16:00	COMPLETED	00:05:22	
	ORDER RED	11:10:40	11:17:08	COMPLETED	00:06:28	0:06:32
Test 3	ORDER WHITE	11:25:12	11:28:55	COMPLETED	00:03:43	
	ORDER BLUE	11:25:14	11:30:37	COMPLETED	00:05:23	
	ORDER RED	11:25:16	11:31:45	COMPLETED	00:06:29	0:06:33
Average Duration					00:05:12	0:06:33
System Throughput						0,436

Table 24: Experiment 1b Resource Utilization

Resource	Active time	Test Duration	Utilization %
HBW	00:04:29	0:06:32	68,62
PLA	00:05:12	00:06:33	79,39
VGR	00:03:41	00:06:33	56,23

Table 25: Experiment 1b CNP Durations

CNP Duration Aggregated:					
Test 1	step 1	63	Test 2	step 1	63
	step 2	110		step 2	141
	step 3	141		step 3	187
	step 4	203		step 4	150
AVERAGE:					**129,25 ms**

E.3 Experiment 2a

Table 26: Experiment 2a Order Results

	Activity Type	Start Time	End Time	Result	Order Duration	Test Duration
Test 1	ORDER WHITE	18:57:45	19:02:16	COMPLETE	00:04:31	
	ORDER BLUE	18:57:47	19:04:24	COMPLETE	00:06:37	
	ORDER RED	18:57:49	19:05:42	COMPLETE	00:07:53	0:07:57
Test 2	ORDER WHITE	19:14:52	19:19:23	COMPLETE	00:04:31	
	ORDER BLUE	19:14:55	19:21:34	COMPLETE	00:06:39	
	ORDER RED	19:14:57	19:22:53	COMPLETE	00:07:56	0:08:01
Test 3	ORDER WHITE	19:33:10	19:37:42	COMPLETE	00:04:32	
	ORDER BLUE	19:33:12	19:39:50	COMPLETE	00:06:38	
	ORDER RED	19:33:14	19:41:09	COMPLETE	00:07:55	0:07:59
Test 4	ORDER WHITE	19:44:35	19:49:05	COMPLETE	00:04:30	
	ORDER BLUE	19:44:37	19:51:13	COMPLETE	00:06:36	
	ORDER RED	19:44:39	19:52:32	COMPLETE	00:07:53	0:07:57
Average Duration					00:06:21	0:07:59
System Throughput						0,376

Table 27: Experiment 2a Resource Utilization

Resource Utilization			
Resource	**Active time**	**Test Duration**	**Utilization**
HBW	00:04:54	0:07:57	61,64
PLA	00:05:44	00:08:01	71,52
VGR	00:04:56	00:07:59	61,80

Table 28: Experiment 2a CNP Durations

CNP Duration Non-aggregated:								
Test 1			Test 2			Test 3		
	step 1	62		step 1	62		step 1	32
	step 2	78		step 2	78		step 2	94
	step 3	141		step 3	234		step 3	172
	step 4	156		step 4	78		step 4	78
	step 5	172		step 5	188		step 5	141
	step 6	235		step 6	188		step 6	125
	step 6	328		step 7	187		step 7	172
AVERAGE:								**142,90**

Table 29: RI Attribute Updating

Resource Instance:	Attribute ID:	Starting Value:	Post 1st Order Value:	Post 2nd Order Value:	Post 3rd Order Value:
HBW	MAX_WAIT_TIME_FETCH_WP	25,00	27,00	27,00	31,00
	AVG_WAIT_TIME_FETCH_WP	25,00	26,00	26,33	27,50
MPO	MAX_WAIT_TIME_PRODUCE	40,00	45,00	45,00	45,00
	AVG_WAIT_TIME_PRODUCE	40,00	42,50	43,33	43,75
	MAX_WAIT_TIME_SORT	10,00	11,00	13,00	13,00
SLD	AVG_WAIT_TIME_SORT	10,00	10,50	11,33	11,75
VGR	MAX_WAIT_TIME_ORDER	10,00	10,00	10,00	10,00
	AVG_WAIT_TIME_ORDER	10,00	10,00	10,00	10,00
	MAX_WAIT_TIME_PLA2PKG	5,00	6,00	8,00	9,00
	AVG_WAIT_TIME_PLA2PKG	5,00	5,50	6,33	7,00
	MAX_WAIT_TIME_DISPATCH_PKG	5,00	9,00	9,00	9,00
	AVG_WAIT_TIME_DISPATCH_PKG	5,00	7,00	7,67	8,00
	MAX_WAIT_TIME_PACKAGE	15,00	15,00	20,00	20,00
PKG	AVG_WAIT_TIME_PACKAGE	15,00	15,00	16,67	17,50

E.4 Experiment 2b

Table 30: Experiment 2b Order Results

	Activity Type	Start Time	End Time	Result	Order Duration	Test Duration
Test 1	ORDER WHITE	12:43:10	12:46:44	COMPLETED	00:03:34	
	ORDER BLUE	12:43:12	12:48:24	COMPLETED	00:05:12	
	ORDER RED	12:43:14	12:49:31	COMPLETED	00:06:17	0:06:21
Test 2	ORDER WHITE	13:03:42	13:07:17	COMPLETED	00:03:35	
	ORDER BLUE	13:03:44	13:08:58	COMPLETED	00:05:14	
	ORDER RED	13:03:46	13:10:02	COMPLETED	00:06:16	0:06:20
Test 3	ORDER WHITE	13:17:10	13:20:44	COMPLETED	00:03:34	
	ORDER BLUE	13:17:12	13:22:25	COMPLETED	00:05:13	
	ORDER RED	13:17:14	13:23:30	COMPLETED	00:06:16	0:06:20
Test 4	ORDER WHITE	13:35:35	13:39:09	COMPLETED	00:03:34	
	ORDER BLUE	13:35:37	13:40:51	COMPLETED	00:05:14	
	ORDER RED	13:35:39	13:41:56	COMPLETED	00:06:17	0:06:21
Average Duration					00:05:01	0:06:20
System Throughput						0,474

Table 31: Experiment 2b Resource Utilization

Resource Utilization			
Resource	**Active time**	**Test Duration**	**Utilization**
HBW	00:04:34	0:06:21	71,92
PLA1	00:00:00	00:06:20	0,00
PLA2	00:05:01	00:06:20	79,21
VGR	00:03:39	00:06:21	57,48

Table 32: Experiment 2b CNP Durations

CNP Duration Non-aggregated:								
Test 1			Test 2			Test 3		
	step 1	58		step 1	63		step 1	62
	step 2	110		step 2	78		step 2	128
	step 3	235		step 3	260		step 3	265
	step 4	170		step 4	234		step 4	172
AVERAGE:								**152,92**

E.5 Experiment 3

Table 33: Experiment 3 Test Results

	Activity Type	Start Time	End Time	Result	Order Duration	Test Duration
Test 1	ORDER WHITE	15:13:10	15:16:44	COMPLETED	00:03:34	
	ORDER BLUE	15:13:12	15:18:24	FAILED: sld_timeout	00:04:43	
	ORDER RED	15:13:14	15:19:31	COMPLETED	00:06:17	0:06:21
Test 2	ORDER WHITE	16:13:41	16:17:16	COMPLETED	00:03:35	
	ORDER BLUE	16:13:43	16:18:57	COMPLETED	00:05:14	
	ORDER RED	16:13:45	16:20:01	COMPLETED	00:06:16	0:06:20
Test 3	ORDER WHITE	16:27:15	16:31:01	COMPLETED	00:03:46	16:31:01
	ORDER BLUE	16:27:17	16:32:30	COMPLETED	00:05:13	16:32:30
	ORDER RED	16:27:19	16:33:36	COMPLETED	00:06:17	16:33:36
Average Duration					00:04:58	0:06:21