



**DAVID RODRIGUES FERREIRA**

Master in Electrical and Computer Engineering

**MARKETPLACE-DRIVEN FRAMEWORK  
FOR THE DYNAMIC DEPLOYMENT AND INTEGRATION OF  
DISTRIBUTED, MODULAR INDUSTRIAL CYBER-PHYSICAL SYSTEMS**

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING  
NOVA University Lisbon  
September, 2024



## MARKETPLACE-DRIVEN FRAMEWORK

FOR THE DYNAMIC DEPLOYMENT AND INTEGRATION OF  
DISTRIBUTED, MODULAR INDUSTRIAL CYBER-PHYSICAL SYSTEMS

DAVID RODRIGUES FERREIRA

Master in Electrical and Computer Engineering

**Adviser:** André Dionísio B. da Silva Rocha

*Assistant Professor, NOVA University Lisbon*

### Examination Committee

**Chair:** Name of the committee chairperson  
*Full Professor, FCT-NOVA*

**Rapporteur:** Name of a rapporteur  
*Associate Professor, Another University*

**Members:** Another member of the committee  
*Full Professor, Another University*

Yet another member of the committee  
*Assistant Professor, Another University*

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon

September, 2024

**Marketplace-Driven Framework  
for the Dynamic Deployment and Integration of Distributed, Modular Industrial Cyber-  
Physical Systems**

Copyright © David Rodrigues Ferreira, NOVA School of Science and Technology, NOVA University Lisbon.

The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

To my friends and family.

## ACKNOWLEDGEMENTS

Acknowledgments are personal text and should be a free expression of the author.

However, without any intention of conditioning the form or content of this text, I would like to add that it usually starts with academic thanks (instructors, etc.); then institutional thanks (Research Center, Department, Faculty, University, FCT / MEC scholarships, etc.) and, finally, the personal ones (friends, family, etc.).

But I insist that there are no fixed rules for this text, and it must, above all, express what the author feels.

”

*“The most important step a man can take.  
It’s not the first one, is it?  
It’s the next one.  
Always the next step (...).  
”*

— **Brandon Sanderson**, Oathbringer

## ABSTRACT

The concept of Industry 4.0 revolutionized the manufacturing sector. It called for the use of Cyber-Physical Production Systems and the digitalization of many manufacturing processes. Even though the implementation of Industrial Multi-agent Systems as Cyber-Physical Production Systems comes with a lot of advantages, it has not seen much use by the industry.

As a consequence of this, Multi-agent Systems have not grown out of their infancy and haven't evolved like other technologies through their practical use. This may be because there is still some scepticism surrounding the concept of Multi-agent Systems, and whether or not they can perform to the same efficacy when compared to already existing systems.

In this work, an architecture which helps solve this problem is proposed. More precisely, this platform would allow for the flexible integration of agents with their respective hardware by proposing a method that selects one or more generic libraries based on the kind of hardware that is being integrated into the industrial Multi-agent System. Through this method, more devices are able to be integrated in less time and with less work, contributing for the flexibility and scalability that is characteristic of Multi-agent Systems.

This platform would facilitate the adoption of industrial Multi-agent Systems, because not only would it make integrating them easier, it would also mean that anyone could write these libraries, making it easily adaptable to already existing systems, which would reduce costs in the adoption of Multi-agent Systems for industrial applications.

**Keywords:** Industry 4.0, Cyber-Physical Production System, Multi-agent System, Manufacturing Systems, Hardware Integration

## RESUMO

O conceito de Indústria 4.0 revolucionou o setor de manufatura. Uma das características deste conceito é o uso de Sistemas de Produção Ciberfísicos e a digitalização de processos de manufatura. Apesar da implementação de Sistemas Industriais Multi-agente como Sistemas de Produção Ciberfísicos trazer várias vantagens, não tem sido muito adotada por parte da indústria.

Como consequência, Sistemas Multi-agente não passaram da sua fase inicial e não evoluíram através do seu uso prático como outras tecnologias. Isto pode ser atribuído ao ceticismo do qual o conceito de Sistemas Multi-agente ainda sofre, e à incerteza sobre se estes conseguem operar com a mesma eficácia quando comparado com sistemas já existentes.

Neste trabalho, é proposta uma arquitetura que poderá ajudar a atenuar este problema. Mais precisamente, esta plataforma permite a integração flexível de agentes com o seu respetivo hardware, propondo um método que seleciona uma ou mais bibliotecas genéricas baseadas no tipo de hardware que está a ser integrado no Sistema Multi-agente industrial. Com isto, mais dispositivos são capazes de ser integrados em menos tempo e com menos trabalho, contribuindo para a flexibilidade e escalabilidade que é característico dos Sistemas Multi-agente.

Esta plataforma facilitaria a adoção de Sistemas Multi-agente industriais, porque não só faria a sua integração mais simples, mas também permitia que qualquer desenvolvedor pudesse criar uma destas bibliotecas, tornando o sistema adaptável a sistemas já existentes, reduzindo custos na adoção de Sistemas Multi-agente para aplicações industriais.

**Palavras-chave:** Indústria 4.0, Sistemas de Produção Ciberfísicos, Sistemas Multi-agente, Sistemas de Manufatura, Integração de Hardware

# CONTENTS

<b>List of Figures</b>	<b>ix</b>
<b>List of Tables</b>	<b>xi</b>
<b>Acronyms</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problem and Solution . . . . .	2
<b>2 State of the Art</b>	<b>4</b>
2.1 Cyber-Physical Production Systems . . . . .	4
2.2 Multi-Agent Systems . . . . .	6
2.2.1 Best Practices and Common Architectures . . . . .	7
2.2.2 OPC UA . . . . .	10
2.2.3 Modbus . . . . .	11
2.2.4 MQTT . . . . .	11
2.3 Practical Uses . . . . .	12
2.3.1 Bottling Plant . . . . .	12
2.3.2 Agent-based Plug and Produce CPPS . . . . .	15
2.3.3 PRIME . . . . .	17
<b>3 Architecture</b>	<b>19</b>
3.1 System Architecture . . . . .	19
3.2 Usage Scenarios . . . . .	21
3.3 System Components . . . . .	23
3.4 Module Engine Operations . . . . .	24
3.5 Multi-Agent System . . . . .	25
3.6 System Overview . . . . .	31
<b>4 Implementation</b>	<b>34</b>

<b>4.1</b>	<b>System Implementation . . . . .</b>	<b>34</b>
4.1.1	Implementation Tools . . . . .	35
4.1.2	Deployment Agent . . . . .	36
4.1.3	Resource Agent . . . . .	37
4.1.4	Transport Agent . . . . .	38
4.1.5	Constants . . . . .	39
4.1.6	Directory Facilitator . . . . .	40
4.1.7	Product Manager . . . . .	40
4.1.8	Product Agent . . . . .	41
4.1.9	Module Engine . . . . .	42
4.1.10	Link Libraries . . . . .	43
<b>4.2</b>	<b>Interfaces . . . . .</b>	<b>44</b>
4.2.1	Human to Agent . . . . .	44
4.2.2	Agent to Agent . . . . .	46
4.2.3	Agent to Hardware . . . . .	46
<b>4.3</b>	<b>Multi-agent System Operations . . . . .</b>	<b>48</b>
4.3.1	Initial Setup . . . . .	49
4.3.2	Launching a Product . . . . .	49
4.3.3	Removing Agents . . . . .	51
<b>5</b>	<b>Tests . . . . .</b>	<b>53</b>
5.1	Performance Tests . . . . .	53
5.1.1	Deployment Test . . . . .	53
5.1.2	Execution Test . . . . .	54
5.2	Case Study . . . . .	55
<b>6</b>	<b>Conclusion . . . . .</b>	<b>59</b>
<b>Bibliography</b>		<b>60</b>

## LIST OF FIGURES

2.1	Industrial Multi-Agent System . . . . .	7
2.2	Tightly coupled Hybrid interface. Source: Adapted from [15] . . . . .	9
2.3	Loosely coupled Hybrid interface. Source: Adapted from [15] . . . . .	9
2.4	Tightly coupled On-device interface. Source: Adapted from [15] . . . . .	10
2.5	Loosely coupled On-device interface. Source: Adapted from [15] . . . . .	10
2.6	Interactions between PLCLinkAgent and PLC. Source: Adapted from [8] . .	15
2.7	Plug and Produce RA. Source: Adapted from [23] . . . . .	16
3.1	System architecture. . . . .	20
3.2	Deployment Agent activity diagram. . . . .	22
3.3	Agent replacement/integration scenario. . . . .	22
3.4	Component diagram of a cyber-physical entity using the Module Engine. .	23
3.5	Sequence diagram of Module Engine operations. . . . .	25
3.6	Deployment Agent activity diagram. . . . .	27
3.7	Product Manager activity diagram. . . . .	27
3.8	Contract Net interaction protocol. Adapted from [25]. . . . .	28
3.9	Contract Net protocol between a Product Agent and two Resource Agents. .	29
3.10	Request interaction protocol. Adapted from [26]. . . . .	30
3.11	Requests between agents. . . . .	30
3.12	Activity diagram of the Industrial Multi-Agent System. . . . .	31
3.13	Activity diagram of the final system. . . . .	33
4.1	Deployment Agent class diagram. . . . .	37
4.2	Resource Agent class diagram. . . . .	38
4.3	Transport Agent class diagram. . . . .	39
4.4	Constants class diagram . . . . .	39
4.5	Directory Facilitator class diagram . . . . .	40
4.6	Product Manager class diagram. . . . .	41
4.7	Product Agent class diagram. . . . .	41
4.8	Module Engine class diagram . . . . .	43

4.9	Link Library class diagrams . . . . .	44
4.10	Deployment Agent Graphical User Interface. . . . .	45
4.11	Product Manager Graphical User Interface. . . . .	45
4.12	Multi-Agent System startup. . . . .	48
4.13	Agent startup . . . . .	50
4.14	Agent termination . . . . .	51
4.15	Multi-Agent System operations . . . . .	52
5.1	Simulation kit. . . . .	55
5.2	Kit stations. . . . .	56
5.3	NodeRED flows. . . . .	57

## LIST OF TABLES

4.1	ACLMessages parameters . . . . .	46
4.2	ACLMessages performative types . . . . .	46
5.1	Deployment times. . . . .	54
5.2	Execution times. . . . .	55
5.3	MQTT configurations. . . . .	57
5.4	OPC UA configurations. . . . .	57
5.5	Product Types . . . . .	58

## ACRONYMS

<b>AGV</b>	Automated Guided Vehicle ( <i>pp. 4, 26</i> )
<b>API</b>	Application Programming Interface ( <i>p. 8</i> )
<b>CA</b>	Component Agent ( <i>p. 17</i> )
<b>COM/DCOM</b>	Microsoft Windows Distributed Component Object Model ( <i>p. 11</i> )
<b>CPPS</b>	Cyber-Physical Production System ( <i>pp. 1–7, 12, 13, 15, 17, 25, 59</i> )
<b>CPS</b>	Cyber-Physical System ( <i>pp. 1, 4, 5, 59</i> )
<b>DA</b>	Deployment Agent ( <i>pp. ix, x, 16, 17, 22, 26, 27, 31, 34–38, 40, 44, 45, 48, 49, 51, 57</i> )
<b>DF</b>	Directory Facilitator ( <i>pp. ix, 26–32, 34, 35, 40, 49, 51, 55, 56</i> )
<b>DWPS</b>	Device Profile Web Services ( <i>pp. 16, 18</i> )
<b>FIPA</b>	Foundation for Intelligent Physical Agents ( <i>pp. 2, 10, 13, 16, 17, 28, 29, 35, 38, 46</i> )
<b>GUI</b>	Graphical User Interface ( <i>pp. x, 26, 34, 36, 40, 44, 45, 48</i> )
<b>HTTP</b>	Hypertext Transfer Protocol ( <i>pp. 21, 42, 43, 47, 56–58</i> )
<b>IntA</b>	InterfaceAgent ( <i>pp. 13, 14</i> )
<b>JADE</b>	Java Agent DEvelopment Framework ( <i>pp. 10, 13, 16–18, 35–38, 40</i> )
<b>MAS</b>	Multi-Agent System ( <i>pp. ix, x, 2–4, 6–8, 10–17, 19–22, 25, 26, 30, 31, 35–37, 39, 40, 42, 46, 48–52, 55, 57, 59</i> )
<b>MQTT</b>	Message Queuing Telemetry Transport ( <i>pp. xi, 11, 43, 47, 56–58</i> )
<b>OEE</b>	Overall Equipment Effectiveness ( <i>p. 5</i> )

<b>OPC Classic</b>	Open Platform Communications Classic ( <i>pp. 10, 11</i> )
<b>OPC UA</b>	Open Platform Communications Unified Architecture ( <i>pp. xi, 7, 10, 11, 13–15, 43, 47, 56–58</i> )
<b>PA</b>	Product Agent ( <i>pp. ix, 12, 13, 16, 17, 26, 28–32, 34, 35, 40, 41, 48–51, 56</i> )
<b>PLC</b>	Programmable Logic Controller ( <i>pp. ix, 7, 11, 15, 16, 18</i> )
<b>PM</b>	Product Manager ( <i>pp. ix, x, 26, 27, 31, 34, 35, 40, 41, 44, 45, 48, 49, 51, 57</i> )
<b>PMA</b>	Production Management Agent ( <i>p. 17</i> )
<b>PSA</b>	Prime System Agent ( <i>p. 17</i> )
<b>QoS</b>	Quality of Service ( <i>p. 47</i> )
<b>RA</b>	Resource Agent ( <i>pp. ix, 13, 16, 17, 26, 28–32, 34–38, 40, 44, 45, 48–51, 55, 57</i> )
<b>SMA</b>	Skill Management Agent ( <i>p. 17</i> )
<b>TA</b>	Transport Agent ( <i>pp. ix, 16, 17, 26, 28–32, 34–36, 38–40, 44, 45, 48–51, 55, 57</i> )
<b>TCP-IP</b>	Transfer Control Protocol - Internet Protocol ( <i>p. 11</i> )
<b>UDP</b>	User Datagram Protocol ( <i>p. 11</i> )
<b>WS4D-JMEDS</b>	Web Services 4 Devices - Java Multi Edition DPWS Stack ( <i>p. 16</i> )
<b>XML</b>	eXtensible Markup Language ( <i>pp. 13, 36, 42, 44</i> )

# INTRODUCTION

## 1.1 Motivation

In 2011 the term Industrie 4.0 was introduced for the first time during a German conference. This term was later translated into Industry 4.0 and it quickly became synonymous with the fourth industrial revolution. With it, came the appearance of [Cyber-Physical Production Systems \(CPPSs\)](#). A [CPPS](#) is essentially, as the name implies, a [Cyber-Physical System \(CPS\)](#) capable of operating in a production line. A [CPS](#) is a physical system that has a digital representation. Both the physical and digital counterparts exchange data to maintain coherence and work in tandem to achieve a goal. These [CPPSs](#) brought about the digitalization of the manufacturing sector, and despite there being different models, most of them follow these design principles [2]:

- Services offered through an online platform
- Decentralized, enabling autonomous decision-making
- Virtualized, to allow interoperability
- Modular, making them flexible to changes in the system
- Real-time capabilities
- Ability to optimize processes
- Communication is done through secure channels
- Cloud service for data storage and management

This new way of operating a production chain revolutionized the industry, because it brought about the changes needed to make manufacturing processes more flexible, adaptable and re-configurable, bringing with it a solution for the ever increasing complexity needed to address the rapidly changing customer demands. In recent years, multiple companies have made the transition to this model. They have made diversified changes,

from the way they analyze and process data to the way they manufacture and distribute products, with very positive results as we can see in [3].

In light of this, [Cyber-Physical Production Systems](#) became a popular research subject. These [CPPSs](#) would allow for more efficient, robust and flexible systems, equipped with Big Data analyzing algorithms, Cloud storage to easily access data, service oriented manufacturing and interoperability.

This research was mainly done on the topic of [Multi-Agent Systems](#) [4] [5]. These [MASs](#) are a coalition of agents, all part of one single system but completely independent of each other. They are intelligent, social and capable of performing tasks on their own, however due to their social capabilities, they can also perform tasks cooperatively, making them powerful tools in goal-oriented networks. Because this system is, by nature, decentralized, these agents can leave and join a coalition of agents as needed, to complete selfish or collective objectives. Evidently, the system is also highly flexible and robust, since agents can be taken out of commission and new agents can be introduced as needed, either because the overall system specifications need to change or simply because an agent has become faulty [6].

[Multi-Agent Systems](#) ([MASs](#)) have actually been around for decades and as such, a lot of research and standardization already exists, like the [Foundation for Intelligent Physical Agents \(FIPA\)](#) specifications. [FIPA](#) as an organization have ceased operations, however their standard are still put to use in [MASs](#) nowadays [7]. An industrial [MAS](#) follows similar requirements as a non-industrial one, although it needs to take into consideration other factors, such as hardware integration, reliability, fault-tolerance and maintenance and management costs.

## 1.2 Problem and Solution

Despite all the advantages an [MAS](#) has for the manufacturing sector, it has not seen much success outside research fields. This could be due to the fact that there is still skepticism surrounding agent-based systems for industrial production [8]. Because it never left the prototyping stages, real-world applications never evolved past their infancy, therefore never gained much momentum in the industry at large [9]. Another consequence of this is the difficulty in designing a robust and scalable interface, that allows, for example, the addition and removal of agents without system reconfiguration.

In this work, a new approach to developing interfaces for an Industrial [MAS](#) is proposed. It consists of a framework that allows the creation of industrial agents through the selection of generic libraries for the interface between agent and device. These libraries are picked

based on the type of communication protocols the hardware implements and because they are generic, they can be re-used and replaced with minimal reconfigurations. The aim of this solution is to simplify the process of creating new agents for the system, enabling them to interface with any kind of hardware. This would minimize development time and costs when creating an Industrial [MAS](#), making them a more viable option when implementing a [CPPS](#).

## STATE OF THE ART

In this chapter we'll explore how researchers have dealt with the challenges of creating a **Multi-Agent System (MAS)** based **Cyber-Physical Production System (CPPS)**. We'll start by examining the concepts of **CPPS** and **MAS** in more detail and by taking a look at the most commonly used designs and tools, as well as the recommended practices for an industrial **MAS**. Finally, we'll do a brief analysis on some prototypes that were made to showcase the usefulness of an **MAS** in an industrial setting.

### 2.1 Cyber-Physical Production Systems

As stated before, a **Cyber-Physical System (CPS)** is a system composed of two main types of entities, the physical system that contains all the hardware and resources and the cyber system that represents the physical system in the digital world. These two systems work together, the physical system sends data taken from sensors from the environment to the digital system and the digital system processes and stores this data and instructs the physical system on what actions to perform through the use of actuators. A coupling of a physical entity and a digital one can be called a cyber-physical entity. As such, this system works in a loop, constantly exchanging data and instructions between the physical entities and the digital ones, to achieve a set goal.

A **Cyber-Physical Production System** is, as the name implies, a **Cyber-Physical System (CPS)** that operates in a manufacturing setting. The physical system is composed of a myriad of devices like robot arms, **AGVs**, conveyor belts and other specialized machinery for manufacturing. The cyber system might be as simple as a computer program or as complex as a full-on three dimensional model of the physical system. Vogel-Heuser and Hess [2] proposed that a **CPPS** should:

- Be service oriented, meaning that it should offer its services through the internet
- Be intelligent, with the ability to make decisions on its own

- Be interoperable, by having the capacity to aggregate and represent human-readable information and by providing a virtualization of the physical system
- Be able to flexibly adapt to changes in requirements and scale
- Have Big Data algorithms capable of processing data in real time
- Be capable of optimizing processes to increase Overall Equipment Effectiveness (OEE)
- Be able to integrate data across multiple disciplines and stages of the products life cycle
- Support secure communications to allow partnerships across companies
- Be capable of storing and accessing data in the Cloud

In summary, these CPPSs abstract the hardware resources in the physical environment into their digital counterparts, giving them a measure of autonomy and intelligence, which in turn allows for the fulfillment of most requirements. The remaining requirements are fulfilled by integrating the CPPS with other existing tools such as Big Data processing platforms, Cloud storage services and data-sharing services.

These systems have a virtual representation of the physical system, that updates in real-time according to the information received from the physical world. This virtualization allows for a more efficient and realistic analysis of the real world, enabling more complex behaviors, better error correction and the optimization of industrial processes.

Although advantageous in comparison to existing industrial systems, CPPSs are still fairly recent and therefore creating one of these systems from the ground up still has its challenges. Leitão, Colombo, and Karnouskos [10] have compiled key challenges, as well as their difficulty and priority.

For these reasons, companies have not made a huge push to create CPPSs from the ground up, instead opting to adapt their already existing systems to integrate CPPS elements and characteristics. This is essentially an integration of a CPS in an already existing production line. many of these adaptations depend on creating interfaces capable of interacting with many types of hardware, to allow an easier and more streamlined integration of new entities into the system.

Some examples of CPPS retrofitting can be seen in Cardin [11] and Arjoni et al. [12]. Cardin [11] have done an analysis on the whole process of retrofitting a robotic arm. Before the retrofit, this arm was controlled through a remote controller or a serial interface. After the retrofit, the arm was able to receive commands through a new interface, which

was able to communicate through standard Wi-Fi and Ethernet protocols, sending sensor data and receiving commands to and from the network. This network also integrated a cloud service to allow for the storage and access of information. Performance-wise, the arm showed better energy efficiency and less operation latency after the retrofit and all values were below the thresholds for the standards in an Industry 4.0 CPPS.

Arjoni et al. [12] created a small manufacturing plant using old devices. They consisted of two robotic arm, a CNC machine and a welder arm. All three devices communicate through different protocols, but by using off-the-shelves hardware like an Arduino UNO and a Raspberry Pi 3, Arjoni et al. were able to give these devices the ability to be integrated into a more complex system. Although they recognize the system could still be optimized, they considered the result promising in the retrofit of old devices into a CPPS.

## 2.2 Multi-Agent Systems

To understand what should be the best practices in an MAS based CPPS, we first need to understand its base characteristics. An MAS is, in essence a system composed by many entities called agents that have their own capabilities. These agents communicate and collaborate together by exchanging data among themselves and acting on that data to achieve a common goal. They are capable of adapting their behavior and of autonomous decision-making to determine the best course of action. This system is by nature decentralized and has no hierarchy, making it highly flexible and modular. At any point an agent can leave or join the system, without significant changes in architecture [6].

Industrial agents inherit all the qualities of software agents, like the intelligence, autonomy and cooperation abilities, but in addition are also designed to operate in industrial settings, and need to satisfy certain industrial requirements such as reliability, scalability, resilience, manageability, maintainability and most important of all, hardware integration [13]. An example of a base architecture for an MAS can be seen in Figure 2.1.

These requirements are generally tough to fulfill, especially so because despite the theoretical potential MAS have shown in supporting them, there aren't a lot of agent-based production systems outside the prototyping phase. This has stopped the growth of Industrial MAS due to the lack of practical knowledge in this field [14]. Another problem seen in Industrial MAS is the lack of models that can represent these systems. One of the key elements of an MAS are the changes made in structure and logic as the system operates. Thanks to the decentralized nature of the system, it is possible to add, remove or reconfigure modules freely to better adjust it to the systems needs [14].

Now that we have an idea of the characteristics and requirements for Industrial MAS, we can take a look at the most commonly used architectures, followed by what is

recommended by the IEEE Standard.

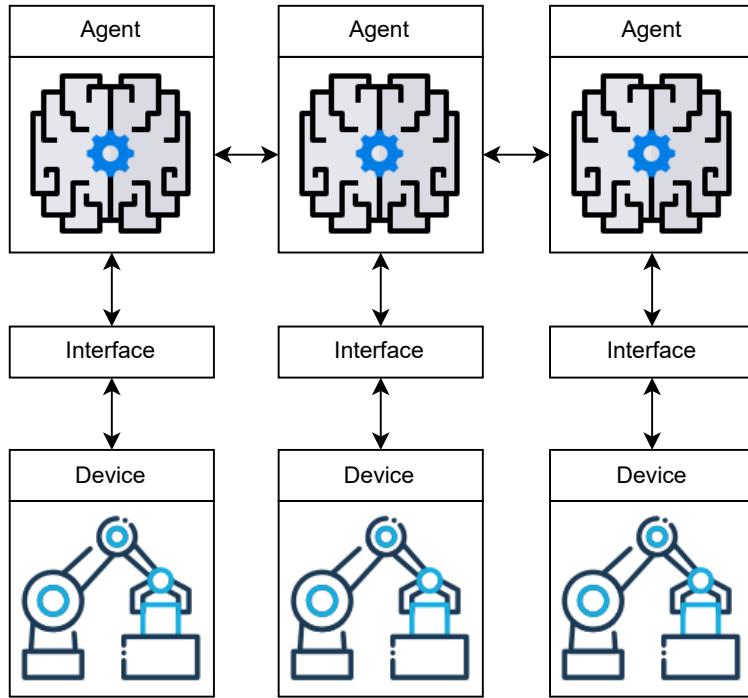


Figure 2.1: Industrial Multi-Agent System

### 2.2.1 Best Practices and Common Architectures

Leitão et al. [13] analyzed the IEEE 2660.1 Standard for the recommended practices in integrating software agents and low-level automation functions. They described the use of an **MAS** as a **CPPS**, where control is decentralized, emerging from the interactions of agents that are part of the system. And as we've discussed before, one of the biggest problems is creating an interface between the agent and the device associated with it. Because of this, the IEEE 2660.1 Standard was created, defining the best practices in designing an appropriate interface.

As an example, the authors mention three main types of interfaces. An interface for a smart sensor, to acquire measurements. An interface for a **PLC**, to control simple devices like conveyor systems. And finally, an interface for a robot controller to control more complex functions in the **CPPS**. These three interfaces present different challenges on a development level, because each requires consideration on which architecture to follow, with different consequences to the evolution of the manufacturing plant over time.

The authors then created multiple scenarios, one of them being factory automation. They then proposed that the most valuable criterion was the response time of the system. As a secondary criterion scalability was chosen, but with a lesser importance. From this scenario the authors then concluded that a **Tightly coupled Hybrid OPC UA** interface

was preferable according to the IEEE 2660.1 standard. This means that the interface should have a client-server approach and be running remotely, a Tightly coupled Hybrid approach. However the authors also mention that this setup has a relatively low score, meaning that many of the other proposed practices are still viable, with testing needed to be done in order to pick the best one based on each specific scenario.

In [14], it is proposed that one of the key requirements in the design of interfaces for MAS is interoperability. This comes with other challenges associated, like re-usability and scalability. In an MAS, the authors identified two main types of interfaces, the interface between agents, which normally is provided through the framework of the agent-based system, and the interface between agent and device.

Leitão et al. [15] analyzed a study performed under the IEEE P2660.1 Working Group [16] and concluded that most approaches followed a two-layer convention. The upper layer contained the agents part of the MAS and the lower layer the hardware associated with the physical production system. These two layers can interact in two ways [15]:

- Tight coupling, where the two layers communicate either through shared memory or through a direct network connection. This communication is synchronous and more direct.
- Loose coupling, where the two layers communicate through a queue or a pub/sub channel. This communication is asynchronous and less direct.

These layers can also be hosted in different setups [15]:

- Hybrid setup, where the two layers run in different devices.
- On-device setup, where the two layers run in the same device.

This means that there can be four different interfaces, Tightly coupled Hybrid, Loosely coupled Hybrid, Tightly coupled On-device and Loosely coupled On-device.

A Tightly coupled Hybrid interface (Fig. 2.2) is characterized by having the upper layer where the agent operates running remotely and accessing the lower layer through an API. This API is responsible for translating the instructions given by the agent into commands the hardware can interpret. It is also responsible for the opposite, translating the hardware output, such as error codes or function results into data the agent can use. This approach is limited by the channel through which both layers communicate since both agent and device operate on two different computing platforms. This channel is affected by the amount of traffic in the network, more connections implies a lesser quality of service, namely in response time [15].

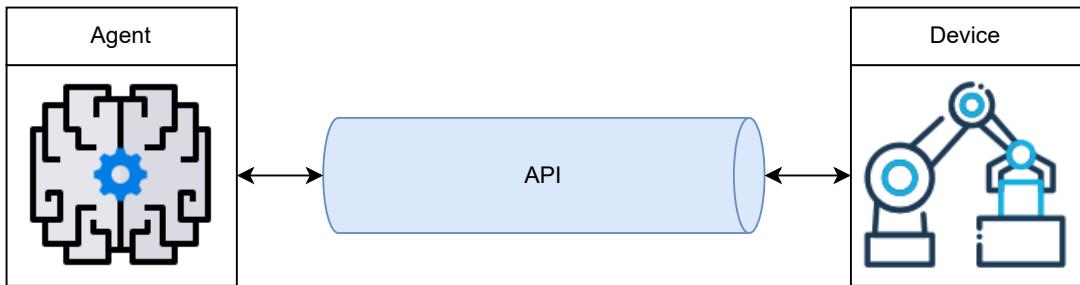


Figure 2.2: Tightly coupled Hybrid interface. Source: Adapted from [15]

A Loosely coupled Hybrid interface (Fig. 2.3) also sees both agent and device running on different computing entities. The difference is that instead of each agent having a direct connection to the corresponding device, they communicate through a message broker. Since the system still runs on two different computers it still suffers from the quality of the connection between layers, making this somewhat inappropriate for systems highly dependent on real time action. However, this approach sees better results in complex systems, where the agent layer needs to publish information to a large amount of devices at once. It also sees good results when it comes to scaling the system, since both layers are very independent of each other [15].

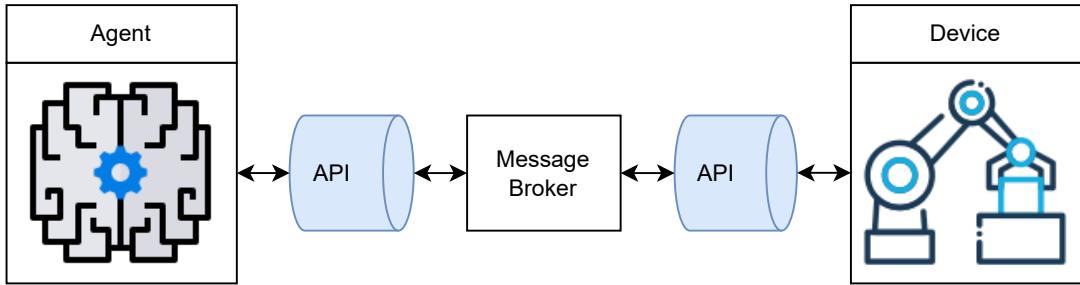


Figure 2.3: Loosely coupled Hybrid interface. Source: Adapted from [15]

A Tightly coupled On-device interface (Fig. 2.4) on the other hand follows an architecture where both devices share the same physical platform and can be done in two different ways. The first one, and far less common, has both agent code and device code compiled into a single binary running in the same computing element. This solution provides far better results in very demanding real time applications, however it also removes some flexibility from the system and is far more complicated to design due to the lack of development tools. The second option has the computational resources shared through a software library, where communication is done through software functions while abstracting some elements. This option still holds good results in real time control, but not as good as the first option [15].

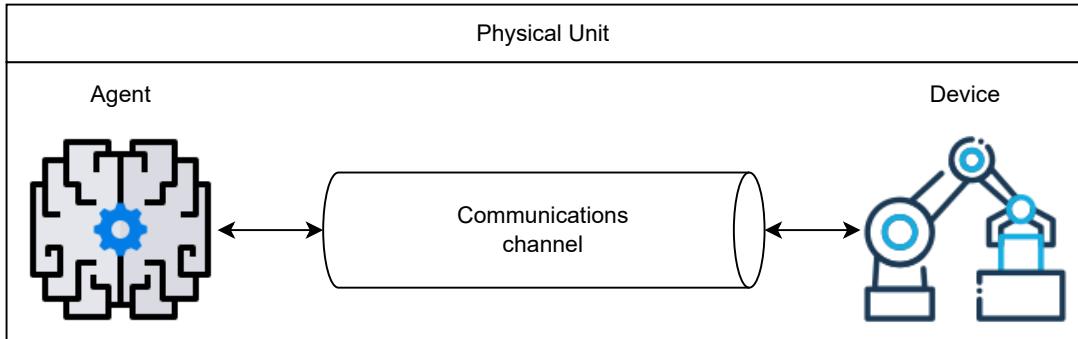


Figure 2.4: Tightly coupled On-device interface. Source: Adapted from [15]

Finally, a Loosely coupled On-device interface (Fig. 2.5) is characterized by having the agent embedded in the device and communication is done through a broker. Both layers share a physical unit but do not share computational resources. The utilization of a broker between the two layers offers some flexibility, since the agent and hardware are less dependent of each other. This comes with the caveat that the real time response of the whole unit is dependent on the performance of the broker [15].

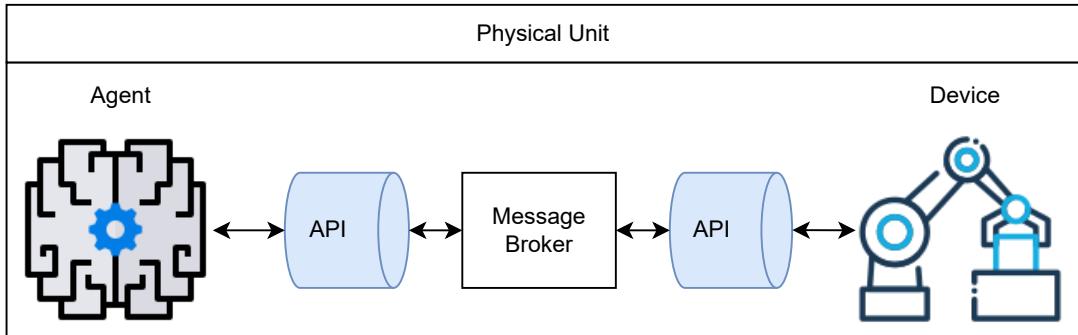


Figure 2.5: Loosely coupled On-device interface. Source: Adapted from [15]

The most common programming language to codify agents is Java, most likely due to the existence of **JADE** followed by C++ [15]. **JADE** helps developers in the implementation of **MASs** with the **FIPA** specifications. It also allows deployment for different machines, due to Java supporting multiple devices [17]. For the device part, preexisting hardware is used in the majority of cases because it can be integrated into an **MAS** by using protocols such as **OPC UA** [15], which is a platform independent data exchange standard. It allows for both server-client and publish/subscribe communications as we'll see in the following section [18].

### 2.2.2 OPC UA

Another way to integrate hardware into an **MAS** is through the use of already made tools. The **Open Platform Communications Unified Architecture (OPC UA)** was created based on **Open Platform Communications Classic (OPC Classic)**, which in turn is based on

the **COM/DCOM** for the exchange of data. **OPC UA** is an evolution of **OPC Classic** but with extra functionalities, like added security, extensibility and platform independence. This last feature allowed it to run on many more platforms such as cloud-based servers, micro-controllers and **PLCs** [18]. As mentioned in Section 2.2.1, it is a recommended way to implement an interface for an **MAS**. **OPC UA** can be used with as a tightly coupled or loosely coupled interface, because it supports both direct client-server connections and pub/sub message transmissions, making it a strong option to integrate hardware into an **MAS**.

An **OPC UA** server is needed, and it can be hosted anywhere on the network. Cyber-physical entities can connect to it using its address, and it supports encryption and users with different levels of permissions for added security. A server hosts a group of files and folders, called nodes. Each node can be accessed independently of other nodes, and values can be written to and read from a node. A node can represent anything, from a machine to a single sensor. **OPC UA** also supports subscription based communications and events, to lessen the load of the system on a network [18].

Because **OPC UA** already implements communication protocols and communication infrastructure, as well as an information format, all participants in the network know how to communicate. This means that the **MAS** can be running on any kind of platform or be programmed in any kind of language, increasing system flexibility. Many modern **PLCs** already provide an embedded **OPC UA** server facilitating their integration into a new system [19].

### 2.2.3 Modbus

Another way to integrate hardware into an **MAS** is by using Modbus. Modbus is an industrial communications system that has been around since 1979. It follows a master/slave architecture, where the master is usually an application capable of acquiring data and the slave a **PLC**. This master application can be substituted for an agent from an **MAS**, integrating it with the slave hardware. Modbus supports standard **TCP-IP** and **UDP**. It is simple, has high levels of compatibility and is decentralized due to the use of master/slave communications, making it flexible. Modbus could be integrated with any programming language of choice, by using software libraries to interpret and send commands to the Modbus layer [20].

### 2.2.4 MQTT

**MQTT** is a widely used protocol for the Internet of Things (IoT). It is a pub/sub protocol, with a minimal network bandwidth and a small code footprint. It is dependent on a **MQTT broker**, which serves as a hub to the messages flowing between clients. A client is any entity part of the **MQTT** network, and they can both publish messages or subscribe

to topics. A topic is an identifier that comes with every message published to the broker and all clients that currently subscribe to a topic of an incoming message will receive that message [21].

It has seen a lot of use by, not only the manufacturing industry, but also in logistics, smart home applications and more, making it a valuable tool in the integration and communication of smart devices.

## 2.3 Practical Uses

Adoption of industrial oriented **MAS** has been slow. According to Karnouskos and Leitão [9], the technology was still in its infancy almost two decades ago, with an incremental progress at best being made since then. In [14], Karnouskos et al. claim that agent-based applications in the industry is still limited. This is because despite the potential shown, these systems have not been implemented in real-world applications, where they would have the chance to evolve and leave the prototyping phase as new research is being done to make them more suitable for these applications.

There are, although, many research prototypes of **MAS** used for an industrial applications, and we'll take a look at some of them in this section.

### 2.3.1 Bottling Plant

For the design of the bottling plant in [8], a research paper [22] was first done by Marschall, Ochsenkuehn, and Voigt, with the collaboration of multiple partners from different backgrounds. This was done to define the base requirements for the **CPPS** as well as the associated **MAS**. These requirements are:

- Easy Scalability and Functional Expansion
- Manufacturer-neutral Resource Representation
- Robust Production Control by the Product
- Lot Size One without Identification

Summarizing, the system needed to be easily scalable and it must be able to expand its functions to handle changes in production, it must be able to represent the resources in the **CPPS** as a generic and manufacturer independent representation, products must be able to handle their own production steps, handling errors and reacting robustly to those errors and finally be able to produce lot sizes of one without needing an identification system for each individual product.

In consequence of these requirements, Marschall, Ochsenkuehn, and Voigt designed a base solution consisting of two generic agents, a **Product Agent (PA)** that represents

individual products being manufactured, in this case the beverage bottles, and a **Resource Agent (RA)** that represents the resources used to manufacture the beverage bottles.

The **MAS** was made using **JADE**, and as such all agents in this system are **FIPA** compliant and communicate through **FIPA** Requests and the **FIPA** Contract Net.

Each agent inherits from a generic class called **Agent**. The **RA** and **PA** then inherit from this class, with added functionalities and variable fields. The **PA** is an agent with the capabilities of performing autonomous and goal-driven behaviors. The **RA** is an agent capable of controlling the physical resources by using sensor data and messages with hardware instructions.

A **PA** differs from the generic **Agent** by implementing a configuration file **ProductConfig**. This file is in the **XML** format and has information on the product represented by it. It also has a **PlanningModule**, which is essentially a **StateMachine** with all the products processing instructions.

Like the **PA**, the **RA** also mirrors the **Agent** class. It must be generic in order to be implemented on multiple resources, so for each **ResourceType** there exists a **ResourceConfig** file which is loaded onto the **RA** in order to implement its interface. To give an example, if the resource has the **ResourceType Machine**, the **ResourceConfig** file **MachineConfig** must be loaded.

The authors identified five different **ResourceTypes** by classifying many resources from renowned machine manufacturers. **RAs** not only include robots, transport systems and stations but also databases, as a data management method. The authors recognize that not all types of possible resources for a **CPPS** have been considered. This means that for every new resource type that needs to be added to the system, a new **ResourceConfig** must be created in order to integrate the new **RA** into the system. In addition, a new resource that communicates differently from the ones already in the system would need a new interface, and thus a new **ResourceConfig** would also need to be created, even if the new resource performs similar functions in the manufacturing plant.

In [8], Marschall et al. created the service oriented bottling plant using the industrial agent system designed in [22], with positive results. To allow for extensibility of the system by additional resources a new agent class was created called **InterfaceAgent (IntA)**. The **ResourceConfig** files for each type of **RA** now have a new field which contains an **InterfaceAgentConfig**. This new **IntA** is instantiated by the **RAs** following the configurations of their respective **InterfaceAgentConfig** and it extends either a **DBLinkAgent**, used to connect to a database, and a **PLCLinkAgent**, used to connect to a **OPC UA** client. All the

information, like server address, port and authentication, is stored in the InterfaceAgent-Config file. By using [OPC UA](#), the system can now interface with the hardware in the manufacturing plant, through a Loosely coupled Hybrid interface. The authors recognize that according to the IEEE P2660.1 Working Group the scalability is considered weak, although they claim the flexibility is worth this loss in scalability.

To implement a new [IntA](#), the following must be provided:

- The communications protocol and the address, ports and authentication
- The way the interactions between the agent and resource should proceed
- The rules for the interpretation of internal resource states

Communication between the PLCLinkAgent and the machine is done through the use of simple commands. They specify the command to perform and the program type to use and must also include a unique name, the NodeID which identifies the type of the given input on the [OPC UA](#) server, the data type, all possible values, the read/write direction and the description. The program type is chosen based on the size of the bottle and the command has to be one of the five commands already defined:

- NoCommand, in case no action is needed
- ProductInPosition, in case the product is in position and ready
- ExecuteProcess, in case the process should start
- ProductRemoved, in case the product needs to be removed from the machine/station
- PrepareMachine, in case the machine needs to be prepared for a process to start

All commands have a number code associated to them to facilitate communications and at any time the PLCLinkAgent can retrieve the state of a machine by observing a NodeID with the machine state code. In Figure 2.6 we can see the state machine representing the interactions between the PLCLinkAgent and a machine. To handle manufacturer specific machine state codes, the authors mapped said codes onto generic [MAS](#) states, with the possibility of having multiple codes mapped to a single state. This was done to make the system capable of handling machines from different manufacturers without sacrificing flexibility in the [MAS](#).

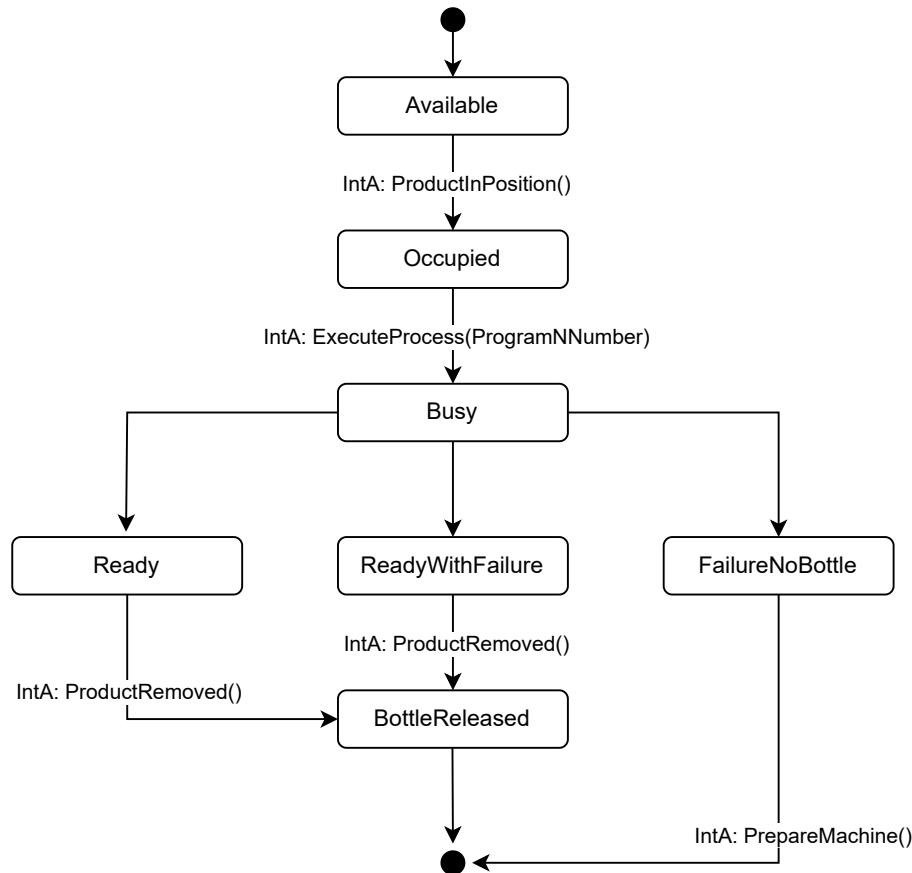


Figure 2.6: Interactions between PLCLinkAgent and PLC. Source: Adapted from [8]

The use of **OPC UA** makes this system very flexible. New hardware that needs to be implemented in the system can be configured to use these commands through the use of a **PLC**. The use of generic states make it scalable and configuration files also help fulfilling this necessity. However, it is still dependent on these files, with individual configurations needed for different manufacturers and machine types. The system also needs to be adapted to recognize the machine states from the new implemented hardware.

Nevertheless this system performed as intended, producing customized beverage bottles according the customers specifications. It is a good example of an **MAS** based **CPPS**, flexible, scalable, robust, with decentralized control and autonomous intelligent agents.

### 2.3.2 Agent-based Plug and Produce CPPS

Rocha et al. [23] have created a Plug and Produce **CPPS**. It is capable of integrating new agents on the fly, as the system operates. First, the authors divided the system into three layers. The upper layer is where the **MAS** operates, called the fog layer. The middle layer is where the interface between the **MAS** and the hardware is, called the edge layer. And finally, the lower layer is where the hardware components of the **CPPS** are, called the

physical layer.

The agents were also categorized into **PA** and **RA**, performing similar functions to the ones in Section 2.3.1 and in addition the authors also considered **Transport Agents (TAs)** which abstract all resources whose function is to move a product from point A to point B. They also considered a **Deployment Agent (DA)** tasked with managing the existence of all other agents. This **DA** should create a new agent or remove an existing one whenever a physical resource is plugged into or unplugged from the system.

To accomplish this the authors considered the grouping of a physical resource with its agent a Module. Modules are the components of the whole production system, and they can be plugged and unplugged at any time. The system need to be able to operate to the best of its abilities at all times. In Figure 2.7 we can see an example of one of these Modules.

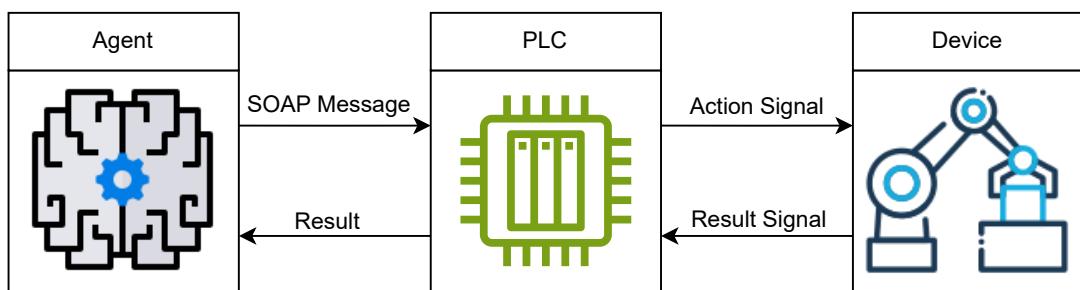


Figure 2.7: Plug and Produce RA. Source: Adapted from [23]

For the interface between hardware and agent, **PLCs** were chosen. These **PLC**s are able to communicate using **Device Profile Web Services (DWPS)**. For the **DA**, a Java class was implemented using the **Web Services 4 Devices - Java Multi Edition DPWS Stack (WS4D-JMEDS)** framework, which searched for the devices in the network and obtained information about them. It was able to detect all devices connected to the network and add and remove them as needed.

Each resource is able to perform certain procedures on the products, represented in the **MAS** as skill which **RAs** perform on to **PA**. This is how the authors simulate the system virtually, whenever a product needs a certain procedure to be performed on itself, the corresponding **PA** asks an **RA** for that skill. This is relevant because the system may or may not have a resource able to perform a specific skill. In the case of a **PA** that needs an non-existing skill to be performed, it waits until an **RA** capable of performing the needed skill to be plugged into the system.

The **MAS** was developed in the **JADE** framework, all agents are therefore **FIPA** compliant and communicate through **FIPA** Requests and **FIPA** Contract Net. More precisely,

the **FIPA** Requests are used for **PA-TA** communications and the **FIPA** Contract Net used for **PA-RA** communications. Whenever a new **PA** enters the system, it asks through the **FIPA** Contract Net which **RAs** are can perform the needed skill. After getting responses from all the **RAs**, the **PA** picks the best one and receives its location. It then sends a **FIPA** Request to the **TA** to transport it to this location. After arriving, it then requests that the skill be performed to the **RA**.

This system was used to simulate a simple conveyor belt line with brushing capabilities. The authors where able to successfully plug and unplugged Modules from the system during its operation. The **DA** correctly connected and disconnected hardware components from the system and deployed and kicked agents from the system accordingly. This shows that an **MAS** built from the ground up with these functionalities is very powerful in its scalability, and although this was only a prototype it shows a lot of promise for a dynamic system.

### 2.3.3 PRIME

Rocha, Barata, and Santos [24] have done a demonstration on the PRIME architecture as an agent based framework with plug and produce functionalities. PRIME is a project developed thanks to the European FP7 program, and it proposes a solution to allow plug and produce using any kind of computational devices. This means that it is no longer needed to restrict a manufacturing plant to specific controllers to provide to it some sort of reconfigurability and flexibility. It allows any kind of controller or to be integrated into the system while still using the same framework.

All agents in this framework were developed using **JADE**, therefore all agents are **FIPA** compliant. This adds to the plug and produce functionality of the system. The PRIME agent system is composed of eight different agents:

- **Prime System Agent (PSA)** is the highest level agent in the framework. It manages the current state of the system.
- **Production Management Agent (PMA)** is responsible to combine all resources and tasks in the same space to abstract certain functionalities
- **Skill Management Agent (SMA)** works in tandem with a **PMA**, combining the lower-level skills into higher-level ones according to pre-defined rules, that the associated **PMA** then provides to the system
- **Component Agent (CA)** is the agent that abstracts the hardware in the **CPPS**. It is able to read and write data to and from the hardware
- Local Monitoring and Data Analysis

All devices to be integrated into the system were categorized into three groups according to their characteristics:

- Fully intelligent resources, capable of running the necessary agents locally, typically a machine with the ability to run the Java environment, setup in a Tightly or Loosely Coupled On-device architecture
- Semi-intelligent resources, capable of announcing their existence to the system and of reconfiguring themselves but without the ability to run the Java environment, setup in a Tightly or Loosely Coupled hybrid architecture
- Passive resources, without any computational abilities and dependent on a controller to connect them to the main system

For the first type of device, PRIME was able to very easily connect to it. The device ran all the necessary components to enable communications and to expose itself to the network. It ran the necessary agents related to the hardware locally and the main framework was able to configure the hardware by interacting with the local agents, which in turn interfaced with the device. This is the type of architecture preferred by PRIME, since it provides all services autonomously [24].

For second category of devices an INICO PLC capable of running DWPS was needed to connect the device to the framework since they can't run the agents locally. An auxiliary computer running the DWPS software was responsible for detecting the PLC and connecting to it. The necessary agents running on this computer launched the hardware related agents whenever they detected a new device running DWPS, and these local agents were able to connect to the main PRIME network [24].

The last category of devices is the most complicated one because they have no easy way to interface with the main system. In this case a more primitive PLC was used to simulate this limitation. An auxiliary computer running the JADE framework with all necessary agents is needed to create this interface and the system communicates with the PLC on a case-by-case scenario, with each PLC possibly needing different configurations. When the relevant agent detect a new device, it launches a new agent to interface with this device. The new agent then uses a case specific library to interface the PLC using standard protocols [24]. This last option is not flexible and uses a case-by-case configuration, making it the least desirable in a system.

# ARCHITECTURE

In this section, the designed architecture is explained. Starting with an overview of the system, we will take a look at the main architecture, followed by a few usage scenarios. Then we will go into more detail as to how each component of the system works and how they are connected through the interfaces it uses to communicate with other components. We will also look at how the system is supposed to operate normally. Finally, the supporting [Multi-Agent System](#) that was built to showcase the Module Engine is explained. Every agent is explained in detail and a full system architecture is presented at the end.

## 3.1 System Architecture

The main objective of this architecture is to bridge the gap between software and hardware. More specifically, to allow for a cyber-physical entity to interface with any kind of hardware in a flexible manner. To allow this flexibility, the system needs to be modular to an extent, enough to allow compatibility for many types of hardware, but not so much to make it hard to work with. A main entity must allow for the interfacing with the modular part of the system, to enable support of multiple physical entities.

In addition, some supporting architecture is also needed to deploy the cyber-physical entities. This architecture must provide a list of all modules available for usage to the main entity that loads the modules. It must also provide a way to supply the modules with the right configurations so that they are able to interface with the hardware.

The system proposed in this work then consists of two main components, a Module Engine and the Link Libraries, with a deployment entity capable of providing these two components with the right resources. The Module Engine is the main entity that operates between the layers of software, in this case the agent, and hardware. It allows the agent to interface with any kind of hardware by loading Link Libraries.

These Libraries are what actually communicates to the hardware below, and must be

able to use any kind of communication protocols. Due to their modularity, any Link Library can be used at any point, independent of the characteristics of the agents above them. They are made with flexibility in mind to allow for the integration of any kind of hardware.

The Module Engine must be equally as flexible to allow for switching on the fly, during system operations. This allows for a more flexible and adaptable MAS, since communications protocols can be switched fairly easy, the system becomes more robust also. To accomplish this, the Link Libraries need to be loosely linked to the Module Engine.

As described above the auxiliary architecture must provide the Module Engine with a list of all the available Link Libraries. It must also provide the Link Libraries with the correct configurations so that they can interface with the hardware. This could be done by passing all necessary resources to the agent on launch, and then the agent passes them along to the Module Engine and Link Libraries. This support architecture can be a simple entity that, upon deploying the agents, supplies them with the correct resources that are in turn passed to the Module Engine and Link Libraries. In this case, the deployment entity approach was chosen, but it would be possible to deploy an agent using the Module Engine through other means. As long as the Module Engine is supplied with the resources it needs to operate, it is able to perform its function. In Figure 3.1, we can see the three main layers of our cyber-physical entity, along with the deployment entity. There is an agent, which will take care of the decision making, the Module Engine and Link Library which will provide an interface between agent and device, and the hardware itself, that will actuate on the physical world and provide sensor information to the agent above. As we can observe, it should matter not what kind of agent or hardware the agent is using, with the Module Engine, it must be capable of interfacing with it.

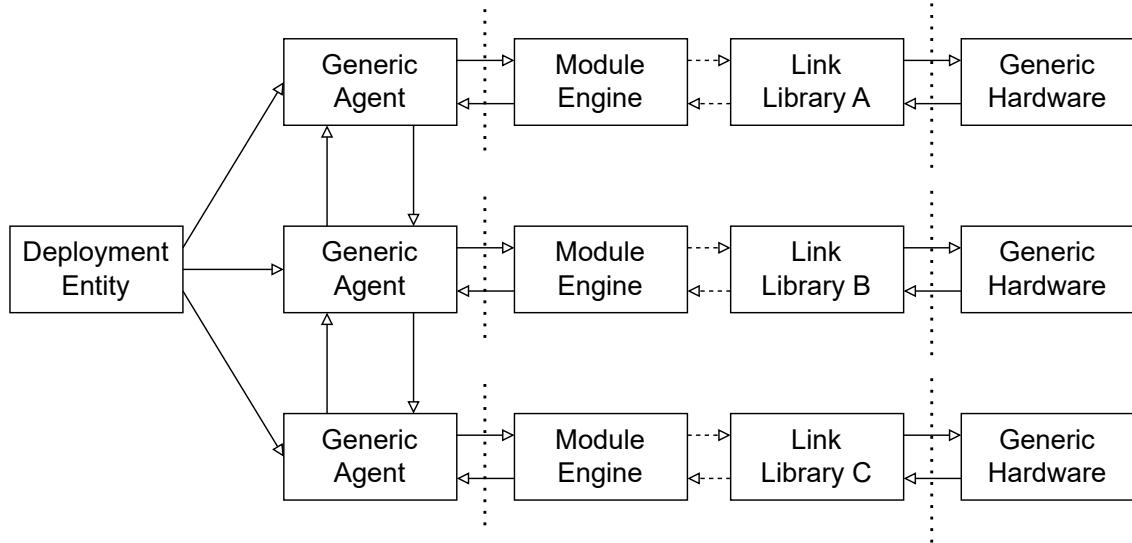


Figure 3.1: System architecture.

To summarize, the main objective of this project is to find a generic architecture that allows an agent to interface with any kind of hardware. The proposed architecture consists of the Module Engine and Link Libraries. The Module Engine serves as an intermediary between the agent and the Libraries. The Libraries in turn are tasked with interfacing with the hardware. To support this system, a deployment architecture is needed, to supply the Module Engine with the correct resources to load and utilize the Link Library correctly.

## 3.2 Usage Scenarios

This system has three main usage scenarios. The first is when a developer is creating a new agent to add to either a new system or a previously existing one. The second is when a developer wants to create a new Link Library to integrate a new type of hardware, or to use a different communications protocol. The third is when the already operating **MAS** runs into an hardware problem, and needs to use other types of hardware to reduce system downtime. This scenario can also be applied when a new agent is integrated into the already operating **MAS**, since it is in part what is happening when an agent needs replacement.

When a developer is creating a new agent, they need to make sure to use the Module Engine together with the agents that need to interface to the hardware. For obvious reasons, those agents that do not use hardware do not need the Module Engine. They also need to make sure that the Link Library they wish to use to communicate with the hardware is already developed. If not, then they need to create it. The creation of new Link Libraries will be explained below, in another scenario.

The agent should be able to call the Module Engine at any point during its operation, whenever it need to instruct the hardware. It should also be able to retrieve the result of the operation from the hardware. Figure 3.2(a) shows a diagram representing this scenario. As we can see, the development of the agent is unrelated to the hardware, since the Module Engine allows a more loose connection between the two. This is one of the advantages of the Module Engine. They can both be done in parallel to speed up development time.

Developing a new Link Library is fairly easy in comparison. The developer only needs to pick a new protocol and start development. Testing must be part of it, to make sure the new Link Library can in fact communicate with the Module Engine. In addition the hardware also needs to implement the same communication protocol. If the hardware supports **HTTP** requests, for instance, the Link Library would need to implement a way to communicate through **HTTP** requests. When development is complete, this new Link Library can be published in some software distribution system, GitHub for instance, to be freely available to other developers wishing to use it. This scenario can be seen in Figure 3.2(b).

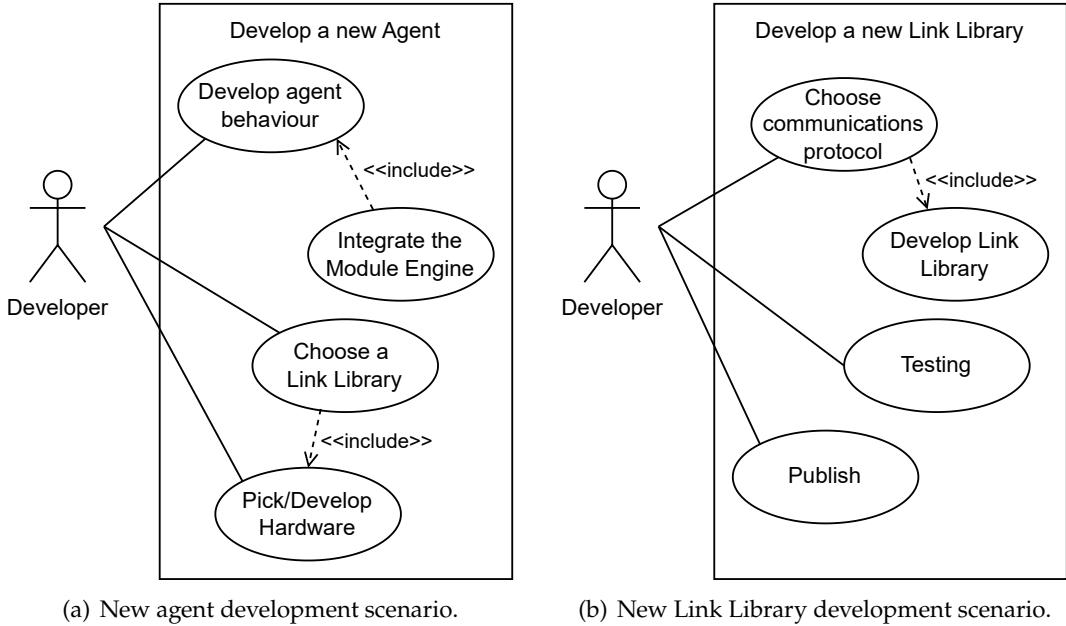


Figure 3.2: Deployment Agent activity diagram.

Finally, performing maintenance on a system using the Module Engine reduces system downtime, since if and when there is a fault, new hardware can easily be integrated. The Module Engine allows this, since Link Libraries are more flexible than other hardware integrations, they can be replaced easily by selecting a different library from the already available ones. If there is not an adequate Link Library, a new one can be developed. Since they are relatively small, it might be faster to create a new library than to create a new interface for the hardware. It also follows that integrating new agents into the pre-existing MAS is as easy, since both processes share similarities, as shown in Figure 3.3.

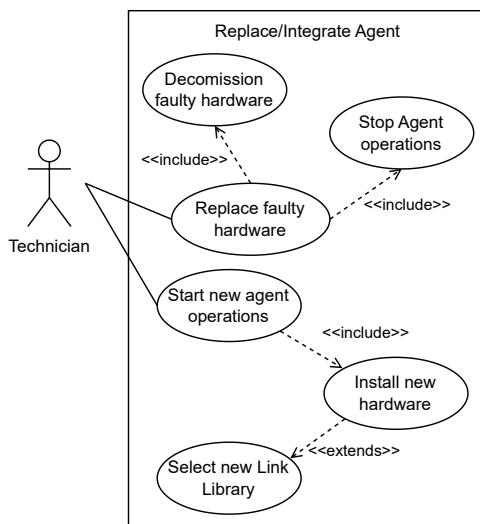


Figure 3.3: Agent replacement/integration scenario.

### 3.3 System Components

The developed system is to be inserted between agent and hardware layers to allow for a more flexible interface. The cyber-physical entity of which the agent is part of is composed of four main components. The agent itself, the Module Engine tasked with loading the Link Libraries, the Link Libraries capable of interfacing with the hardware, and the hardware represented by the agent. The agent, the Module Engine and the Link Library can be considered the cyber part of the cyber-physical entity, with the hardware being the physical part.

In Figure 3.4, we can also see a Deployment Entity. This is the entity tasked with launching an agent using the Module Engine and also provide it with the initial parameters. Once again, this entity could be replaced with any other system, as long as the Module Engine receives its parameters, it should operate normally. These parameters allow the Module Engine to know what Link Libraries exist, which one is to be used and also give the Link Library its configurations needed to establish a connection to the hardware. Both the marketplace file and the Library type are parameters used by the Module Engine, to load a Link Library. The configurations need to be passed to the Link Library, as they are Library specific.

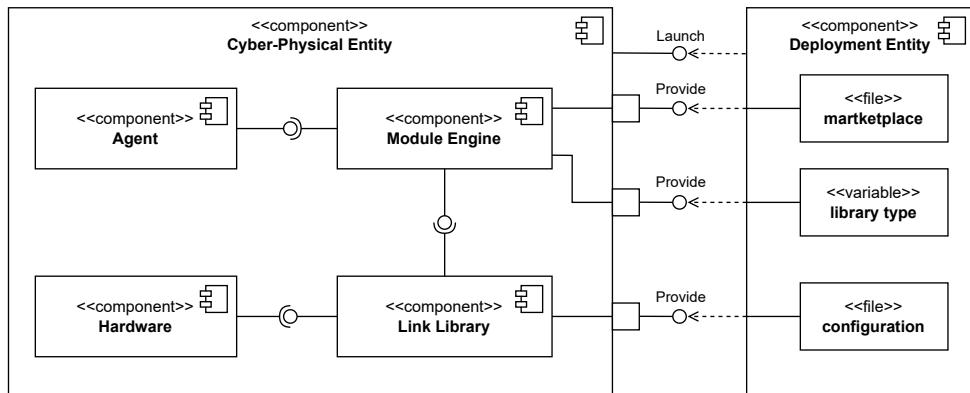


Figure 3.4: Component diagram of a cyber-physical entity using the Module Engine.

Because the Link Libraries are the modular part of the system, these would need to be designed and implemented by other developers working with the Module Engine, wishing to create Libraries with new protocols. To create a new one, a few specifications need to be followed in order to make sure the Library is able to communicate with the Module Engine without problems. The Link Library must establish the connection with the hardware upon being loaded, through the protocol it implements. It also must be able to close this connection at any point, since these Libraries can be switched arbitrarily. Lastly, a Link Library needs to be able to communicate to the hardware through the protocol it implements. This means that the hardware would also need to implement the same

protocol in order to establish a connection. It would also be possible to develop a Library capable of directly interfacing with the hardware through a more specific communication channel, by hosting the agent on the same device as the hardware, for instance. As long as a Link Library implements it, it should be possible to use any communication channel. No information is exchanged exclusively between the loaded Link Library and the Module Engine, making development easier.

The Module Engine simply serves as a loader and interface between the agent and the Link Library. All messages sent to the Link Library by the agent need to go through the Module Engine, and they will arrive at the Library untouched. From the point of view of the agent, it is as if it is communicating directly to the Link Library. All data that flows from the agent to the Link Library must be in the format of a string of text. This Library can now transform the message in any way if needed, to allow for the correct usage of the protocol, in order to communicate with the hardware.

In this architecture, the marketplace is a simple file that holds the location and names of the Link Libraries. It must be organized in pairs, each Link Library name must be associated with a Library path, to allow the Module Engine to load the Library. The marketplace could be developed into a more complex application that fetches the Link Libraries from a remote database, akin to an online platform for the distribution of packages. However, to reduce the project scope, this simpler design was chosen.

### 3.4 Module Engine Operations

The first entity that launches when the system is started must be the deployment entity, since this is the entity that will launch the cyber-physical entities in the system. It will deploy the agents using the Module Engine, and provide them with the parameters it needs to operate. A cyber-physical entity needs to create its own instance of the Module Engine upon deployment, and give it the three parameters received from the deployment entity. Therefore, the Module Engine would need to be included in its development process.

As explained before, the Module Engine needs the marketplace file, where all available Link Libraries are listed, to select the right Link Library. It is picked based on the Library type it also receives. Finally, after loading in the correct Library, it provides it with the configurations file.

The Link Library must now read this file and search for the configurations of the protocol it is implementing. It is possible to include more than one type of protocol in each configuration file. That is, a single configuration file may include different configurations for different protocols, but never more than one group of configurations per protocol. This

file may include things like server addresses, ports, namespaces, topics, etc. Anything that the Link Library needs to establish a connection through the protocol it is implementing must be included in the configurations file. Since this Library can be created by other developers, it offers a lot of liberty in what it can do. As long as it follows the specifications described above, it should operate without problems.

Once communication with the hardware is established, the Module Engine will wait until a new command arrives from the agent. To accomplish this, an agent only needs to call the Module Engine and provide it with the command. Once it does, the Module Engine will pass that command downward to the Link Library, which will pass it along to the hardware. This process is needed to ensure modularity. Any Link Library can be associated with any agent, so the way to achieve this is to have an interface in the middle, the Module Engine, to pass instructions along.

Once the command is executed by the hardware, the result is passed along through the inverse path. From the hardware, to the Link Library, to the Module Engine and finally to the agent. This whole sequence of events is depicted in Figure 3.5. It is also possible to disconnect a Link Library by using the right functionality. The Module Engine would need to be restarted to change the Library.

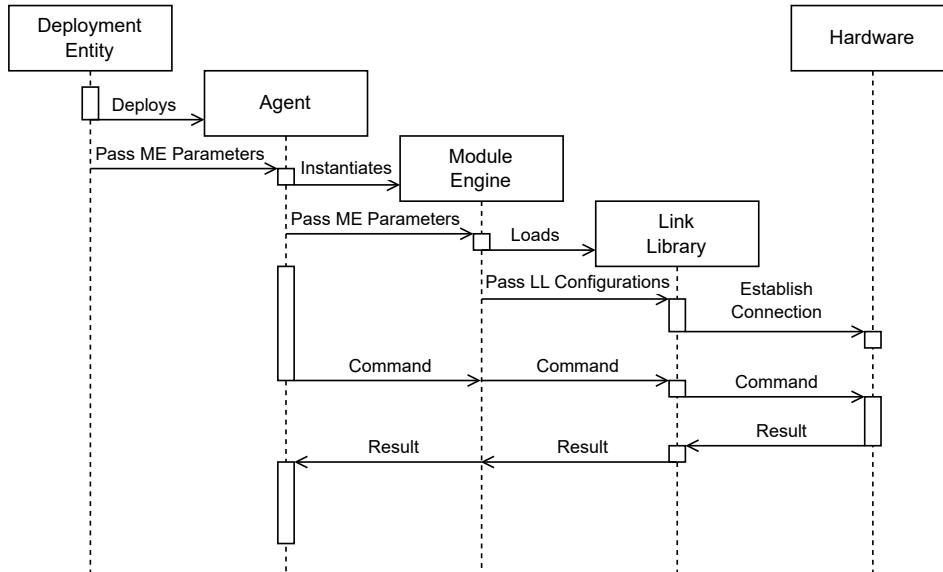


Figure 3.5: Sequence diagram of Module Engine operations.

## 3.5 Multi-Agent System

To showcase the functionalities and perform tests on the Module Engine, it was necessary to develop an Industrial Multi-Agent System (MAS). This Cyber-Physical Production System is composed of three main types of cyber-physical entities, with two other entities

performing more of a management role, used for agent deployment. The two management entities are not considered cyber-physical entities because they do not have a physical counterpart and therefore do not use the Module Engine.

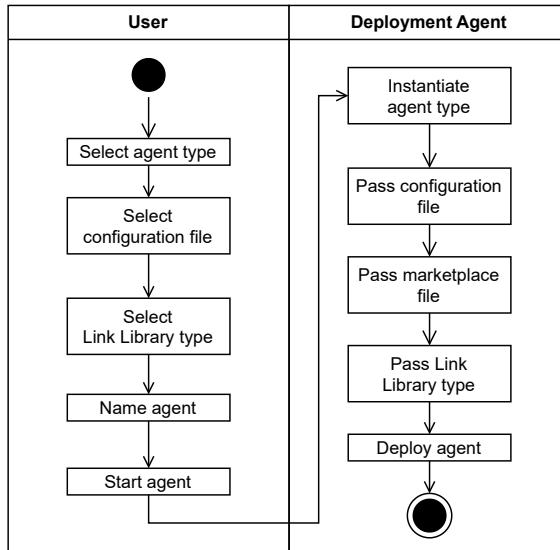
The developed entities are:

- The **Resource Agent (RA)**, that represents any kind of physical component capable of performing processes, like a robotic arm or a bottle filling station;
- The **Transport Agent (TA)**, that represent any kind of physical component capable of transporting products from one location on the shop floor to another, components like a conveyor belt or a **Automated Guided Vehicle (AGV)**;
- The **Product Agent (PA)**, that represent any kind of product to be manufactured by the Industrial **MAS**;
- The **Deployment Agent (DA)**, that will launch **RAs** and **TAs** and provide them with the necessary parameters. This is the entity seen in Figure 3.4;
- And the **Product Manager (PM)**, that will launch **PAs** and provide them with their production sequence.

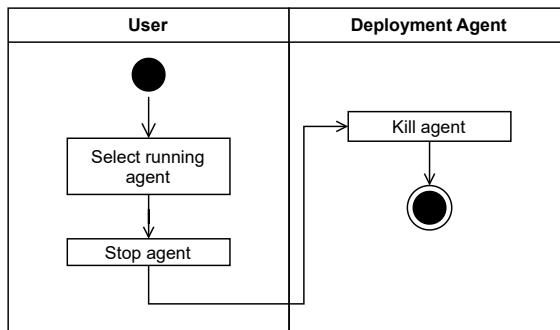
When the system is first launched, the agents that start up immediately are the **Deployment Agent (DA)** and **Product Manager (PM)**, since both of these agents are responsible for the launch and management of all other agents. Both of them need to be capable of receiving input from a human user through a **Graphical User Interface (GUI)**. The user creates the necessary agents through the **DA** by providing it with the Link Library type and configuration file. The **DA** is capable of launching both **RAs** and **TAs** because these are the agents that need the Module Engine to operate as mentioned above. Figure 3.6 presents the process of deploying a new agent (3.6(a)) and stopping a pre-existing agent (3.6(b)). The marketplace file is not provided by the user, but could instead be provided by some kind of web application. For this work however, it is provided locally and the **DA** has access to its contents directly.

The **Product Manager** is simpler in its activities, since its only task is to launch **Product Agents**. A user only has to start the agent through the **GUI** of the **PM**. When a **PA** is started, it does not need any other external parameters. Figure 3.7 presents this simple operation. Since the **PM** does not terminate the **PAs**, it only has the functionality to deploy agents.

After deployment, **Resource Agents** and **Transport Agents** will instantiate the Module Engine and pass it the marketplace and configuration files, along with the Link Library type. Then they will proceed to register themselves in the **Directory Facilitator**, which is a way for agents to search for other agents with specific characteristics. This **DF** works a little bit like a phone book, presenting all registered agents along with their agent ID, a



(a) Deployment of new agent.



(b) Termination of running agent.

Figure 3.6: Deployment Agent activity diagram.

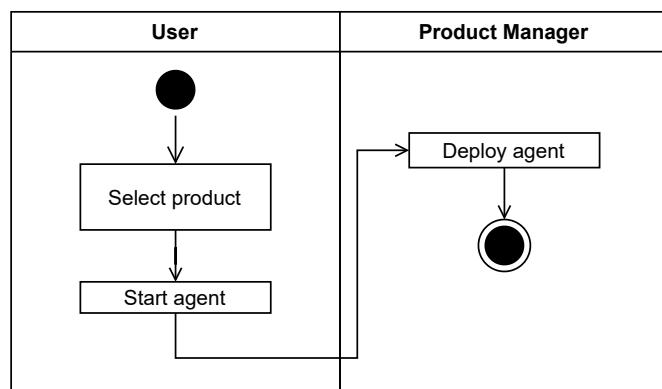


Figure 3.7: Product Manager activity diagram.

unique identifier that corresponds to each agent. Any agent can access the DF and search for agents with certain capabilities, called skills. These skills show what actions an agent can perform. For example, an agent with a skill called "Move\_to\_Storage" might move itself or a load to storage, or an agent with the skill "Staple\_tag" might be able to staple a

tag on a piece of clothing, and so on. A production sequence is list of the skills a product needs to be performed in order to complete it fabricated. In the designed system, **Product Agents** can use the **DF** to look for **RAs** and **TAs** capable of performing the needed skills.

When all **RAs** and **TAs** have been launched and their initial setup completed, the system is now ready to work. **PAs** are launched as new products enter the production line, and after getting their production sequences, they will search the **DF** to find a **RA** capable of performing the first skill. When they find it, they will issue a Call for Proposals from all relevant agents through the Contract Net. This interaction protocol was developed by **FIPA**, and it works as follows:

In this protocol the agent that starts the communication is called the Initiator, and all other are the Participants. The Initiator sends a Call for Proposals or CFP message to all Participants preselected by the Initiator. After receiving the message, the Participants can either accept the call by sending a Proposal, or refuse altogether. On refusal, this particular Participant is out of communications from now on. Proposals might include some data to help the Initiator decide which Participants it wants to keep communicating with. Upon deciding this, the Initiator will send an Accept Proposal message to the desired Participants, Rejecting all other Proposals. Finally, the Participants whose Proposal was accepted might perform some process and inform the Initiator of the result, with a Failure message or a Inform message. This interaction protocol can be visualized in Figure 3.8.

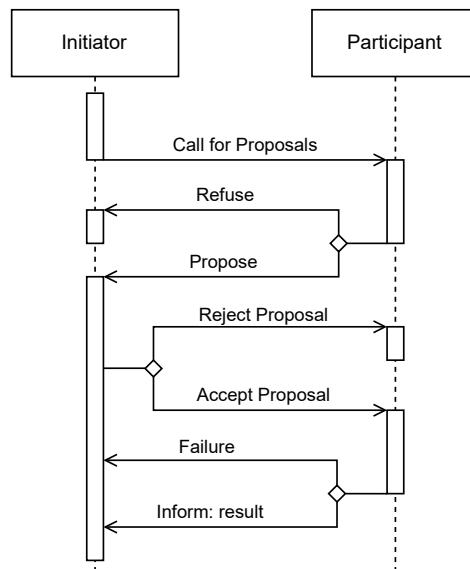


Figure 3.8: Contract Net interaction protocol. Adapted from [25].

After receiving the Call for Proposals, **RAs** will respond with a Proposal. This simply signifies that the agent is available to perform the needed skill, and it does not contain any extra data. Then the **PA**, the Initiator, will Accept the first Proposal on the responses list for simplicity. Finally the selected **RA** will respond with an Inform message, in which the

contents of the message contain the location of the **RA** on the shop floor. This location will be used by the **PA** to ask for transportation from a **TA**. An example of a communication through the Contract Net can be seen in Figure 3.9. In this example, two **RAs** have the relevant skill for the **PA**. It selects the first one and rejects the other. Then the **RA** Informs the **PA** of their location.

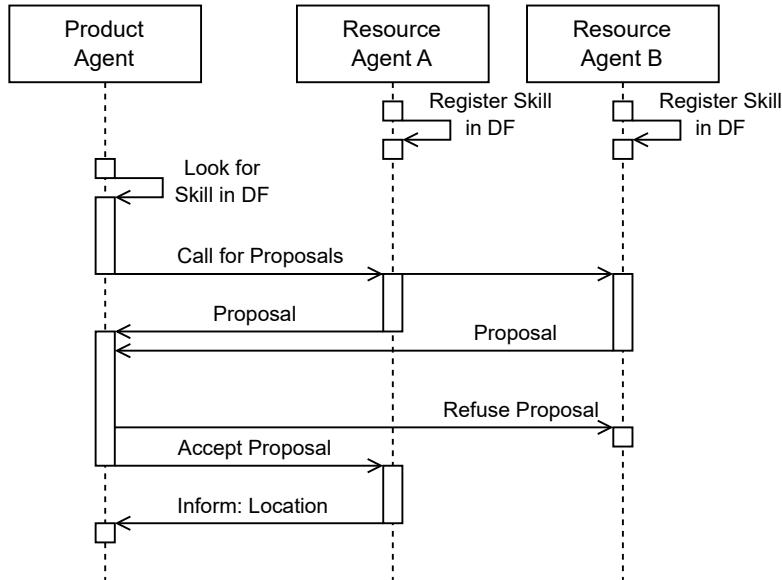


Figure 3.9: Contract Net protocol between a **Product Agent** and two **Resource Agents**.

Upon finding a **Resource Agent**, the **Product Agent** now needs to be transported to its work station. For this it needs a **Transport Agent**. So the **PA** will once again look into the **DF** for a **TA**, by searching for a skill once again. It was decided that for simplicity, only one transport agent will be used in the simulated system. So instead of asking for an agent through the Contract Net, we can skip straight away to the next step, which is to ask for a skill to be performed through a Request. This protocol is also one created by **FIPA**, and its less complex than the Contract Net:

This protocol also includes an Initiator, but only one Participant. This is a one to one communication. The Initiator starts by sending a Request to the Participant. At this point the Participant may Refuse, and communications are terminated. If it Agrees however, it will start performing the desired process immediately. Upon completion, it will notify the Initiator with either a Failure or Inform message. This protocol is shown in Figure 3.10.

The **PA** sends a Request with its starting and ending positions, so the **TA** from where the **PA** departs from, and what is its destination. After Agreeing with the Request and before the communication terminates, the **TA** will transport the **PA** to the destination and send an Inform message signalling the transportation is complete. After arriving at the location where the previously found **RA** is located, the **PA** will use the same protocol to Request the skill from the **RA**. Both of these sequences of messages are shown in Figure 3.11. In 3.11(a)

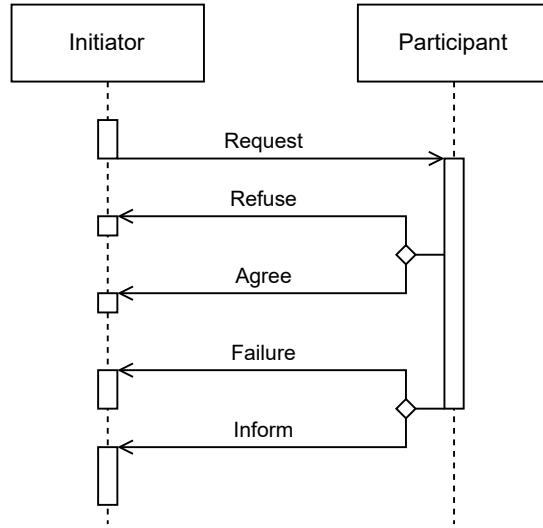


Figure 3.10: Request interaction protocol. Adapted from [26].

we can see the Requests protocol between a PA and TA, and in 3.11(b) between a PA and RA.

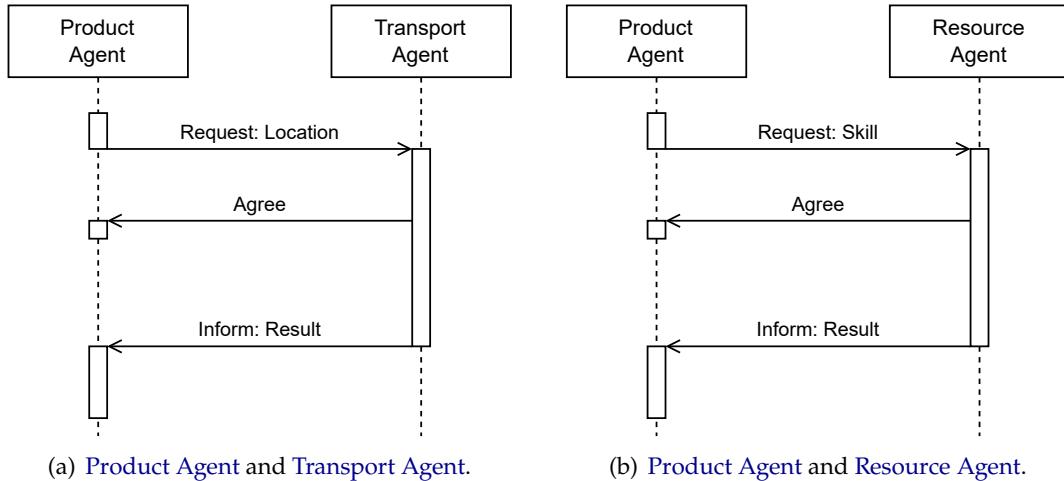


Figure 3.11: Requests between agents.

When the RA finishes its skill and informs the PA, the PA will check if its production sequence is complete. If it is, it will Request transportation from a TA to storage. If the sequence is not complete, it restarts the process by checking the DF for agents capable of performing the next skill, Calling for Proposals, and so on. The whole evolution of the MAS can be visualized in Figure 3.12. It only shows the full interactions between agents, their actions and decisions without the Module Engine.

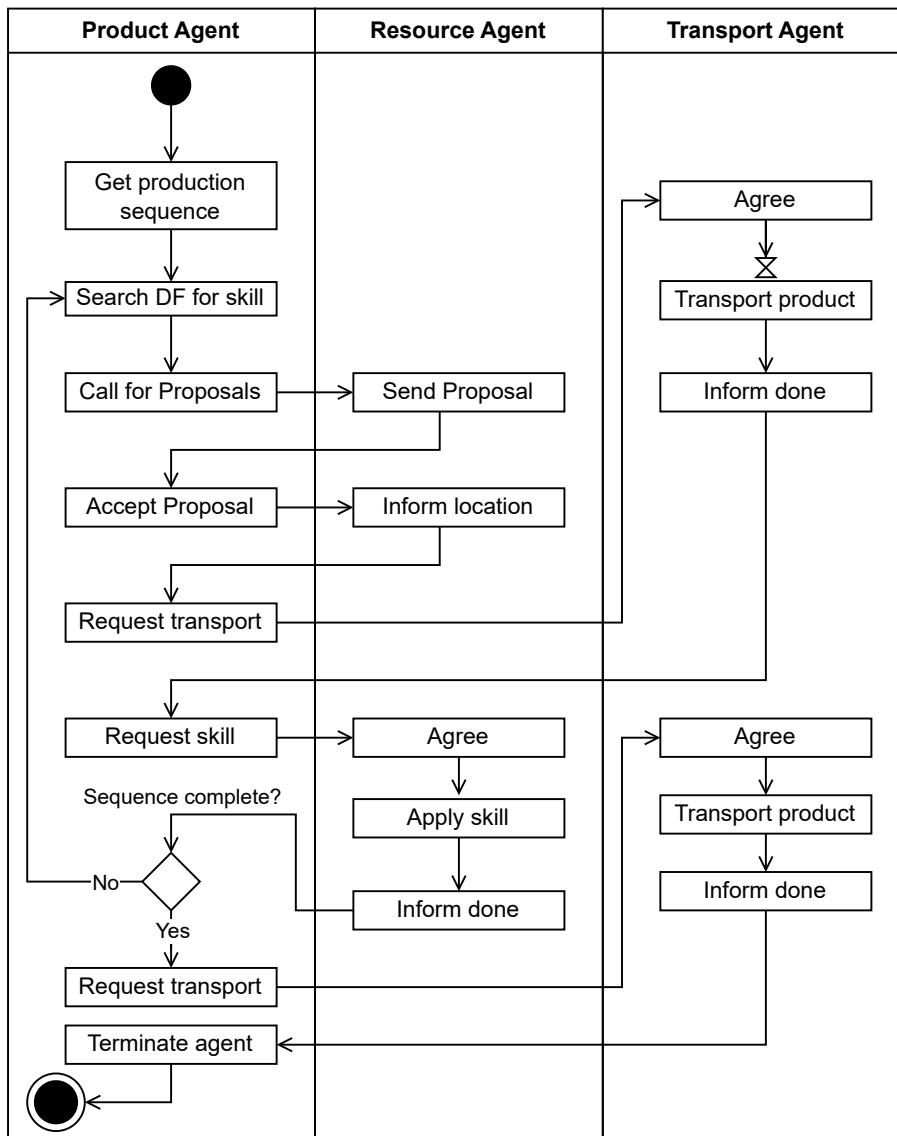


Figure 3.12: Activity diagram of the Industrial Multi-Agent System.

## 3.6 System Overview

Now that all parts of the system have been explained, we will do a brief overview of the behaviour of the system has a whole. When the MAS is launched, the first two agents that start up immediately are the Deployment Agent and the Product Manager. Then a user needs to launch the Resource Agents and the Transport Agent and select the right configurations for each agent.

When these agents are launched, they will register themselves in the DF with their skills and instantiate their Module Engine, which will in turn load the corresponding Link Library. These libraries will establish connection to the hardware. After all of these agents are deployed and ready to operate, the user can now launch the Product Agent.

Upon being launched, the **PA** will look in the **DF** and it will find the **RAs** capable of performing the first skill in its production sequence. Then it will establish contact with all of them through the Contract Net. After receiving the Proposals, it will select one and refuse the all others. In the last message, the **PA** will receive the location of the station where the **RA** is located.

The next step is to Request transportation from a **TA** to this location. The **PA** will issue a Request with its current location and its destination. The **TA** will answer this Request to signal it received the message and will immediately start the process of moving the product. For this, it will send the command through the Module Engine and through the Link Library, to the hardware. Once the hardware finishes executing it, it will return the result back up through the Link Library and through the Module Engine to the agent.

The **TA** will now notify the **PA** that transportation is complete. The **PA** then issues another Request, this time to the previously chosen **RA**. This agent will execute its skill by using the same process as the **TA**. Through the Module Engine and Link Library down to the hardware. After the result arrives back at the agent, it will notify the **PA**. If this is the last skill in this products production sequence, the **PA** will now Request another transportation, but this time to storage, terminating the agent and ending its production. If, however, there are still skills in the production sequence, then the process restarts. The **PA** will search again for a **PA** capable of performing the new skill in the sequence and so on. In Figure 3.13, this whole process is shown, from start to finish.

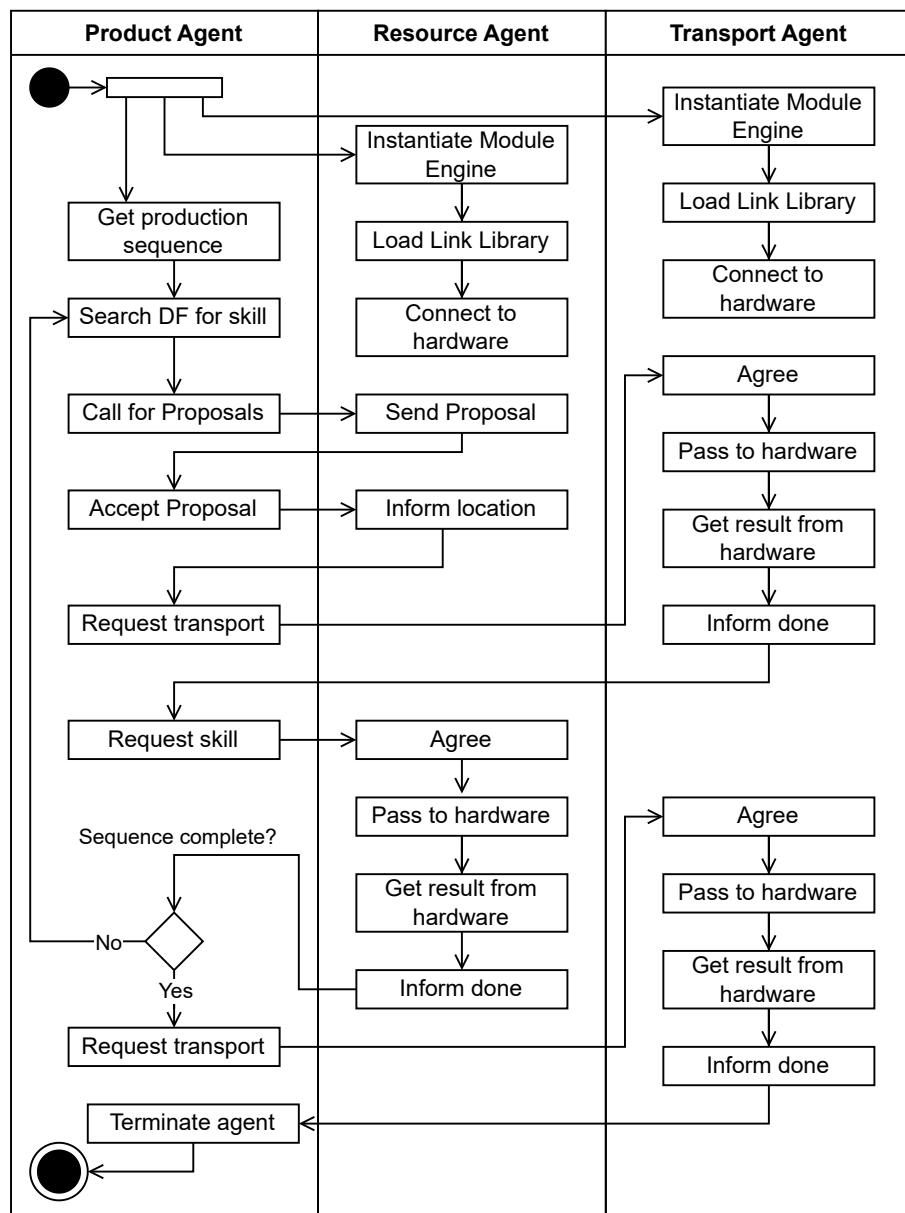


Figure 3.13: Activity diagram of the final system.

# IMPLEMENTATION

Now that the proposed architecture of the system has been laid out and well defined, it is time to implement it. First we will see what tools were used and exactly how each class was implemented, their data models and methods. Then we will look at how the different modules in the system communicate through interfaces, explaining how a human operator can interact with the system, what behaviours each agent makes use of and the developed Link Libraries for hardware interfacing. Finally, we will see how the system operates and what behaviours and methods are called for each operation.

## 4.1 System Implementation

The implementation needs to start with the deployment agents, since these are responsible for launching all other agent types. The **Deployment Agent (DA)** is tasked with deploying **Resource Agents (RAs)** and **Transport Agents (TAs)**. The **DA** needs a **Graphical User Interface (GUI)** to allow a user to launch these agents with the right configurations. These must include the agent type to be launched, the type of Link Library the agent must use and the configurations for the Link Library. In addition to the **DA**, the **Product Manager (PM)** also needs to launch agents, more specifically, **Product Agents (PAs)**. This should also be done through a **GUI** but this one should only show buttons per product type, and some information on the already deployed **PAs**.

The **RAs** and **TAs** will operate somewhat similarly to each other, with the **RAs** needing an extra step before a skill executes, as we will see. They will register themselves in a registry of agents called the **Directory Facilitator (DF)**. This **DF** works like a phone book, where agents can register themselves and find other agents. This is mainly so that the **Product Agents** can find the right agent for the operation they need to accomplish. For skill execution, **RAs** and **TAs** are dependent on the Module Engine.

The **Product Agents** need to be constantly on a loop. They look at their production

sequence and search for a **RA** capable of performing the current skill. Upon finding it, they will request transportation to the station. Upon arrival, they will call for the skill to be performed on them. If the production sequence has been completed, they request transportation out of the production line. If not, the process must repeat until this condition is met.

The Module Engine will need to operate between the layers of agent and hardware. It needs to be able to use its configurations to load any developed Link Library. It must also be capable of relaying messages from the agent to the hardware and vice versa. Multiple Link Libraries need to be developed in order to test whether or not the Module Engine is capable of switching Libraries during runtime.

#### 4.1.1 Implementation Tools

For the implementation of the Module Engine framework and accompanying **MAS**, the Java programming language was selected, more specifically version 21.0.2 of the openjdk. As mentioned in 2.2.1, **JADE** was built with the **FIPA** specifications in mind and its Java version is well supported, so this framework was chosen to implement the **Multi-Agent System (MAS)**.

The **JADE** framework provides a lot of tools for agent development, communication and management. It provides a lot of classes and methods useful for agent setup, communication, **DF** registration and more. Since the Java language was going to be used for the **MAS** and is one of the best supported in the world, the Module Engine and the Link Libraries were also developed using it.

As mentioned in Chapter 3, three main cyber-physical agent types were developed, **Resource Agents (RAs)**, **Transport Agents (TAs)** and **Product Agents (PAs)**. Along with these, two more agents were created, **Product Manager (PM)** and **Deployment Agent (DA)**, with the purpose of launching and managing the other three agent types. All of these agents extend an interface "Agent" provided by **JADE**.

This interface contains a lot of useful methods and variables, although not all of them are used in this **MAS**. The methods used are the "setup" method, executed once when an agent is launched, and the "takeDown" method, executed when a agent is terminated. These methods must be overwritten to provide the agent with instructions on launch and termination.

Because the agents are launched through **JADE**, the class constructor cannot have any initial arguments. To launch an agent with arguments, the method "getArguments" present in every agent can be used. It returns a generic object array the developer can then use to retrieve the arguments an agent is launched with.

JADE also provides a lot of other classes that define agent behaviour. These classes must be defined inside the agents class to add new behaviours. Some of them must be used in order to make use of the different JADE functionalities, like the Contract Net. Others simply exist to add a bit more flexibility during development.

The behaviour classes used in this project were:

- The "OneShotBehaviour" class, that adds a single behaviour to the agents behaviour sequence. It is executed once, and then is excluded from the sequence;
- The "ContractNetInitiator" class, that allows an agent to start interacting with other agents as the Initiator in the Contract Net protocol;
- The "ContractNetResponder" class, that allows an agent to respond to messages as one of the Participants in the Contract Net protocol. Once it finishes communications, it is excluded from the behaviour sequence;
- The "AchieveREInitiator" class, that allows an agent to start interacting with another agent as the Initiator in the Requests protocol;
- And the "AchieveREResponder" class, that allows an agent to respond to messages as a Participant in the Requests protocol. This class is also excluded from the behaviour sequence after communications are done.

#### 4.1.2 Deployment Agent

Starting with deployment, the Deployment Agent was designed to allow a human user to deploy and terminate agents during the execution of the MAS. In its constructor the functionalities of the GUI are defined and some initial values are set. This class extends the "Agent" class, and only makes use of the "setup" method. All other functionalities are called on button presses, through events.

The "selectedAgent" specifies which type of agent (RA or TA) is to be launched. This can be changed through a radio button on the interface.

The "xmlMarketplace" is loaded during "setup" with the method "getMarketplaceLibraries", using the file pointed to by the "marketplaceXMLPath" String. It holds the path of the XML file that contains the available Link Libraries.

The "agentContainer" field holds a reference to the container where the agents deployed will be hosted. Finally, the "xmlConfigPath" is a user loaded value that is used to pass the Link Library configurations file.

In Figure 4.1, we can see some of the classes the agent uses to draw the GUI. These classes are part of the "javax.swing" package, useful to draw graphic interfaces. as the "ContainerController" used to create a container where other agents are deployed. This

is a JADE class. We can also see the auxiliary "Constants" class, that stores constants useful for the correct operations of the MAS. This class will be explained in more detail in 4.1.5.

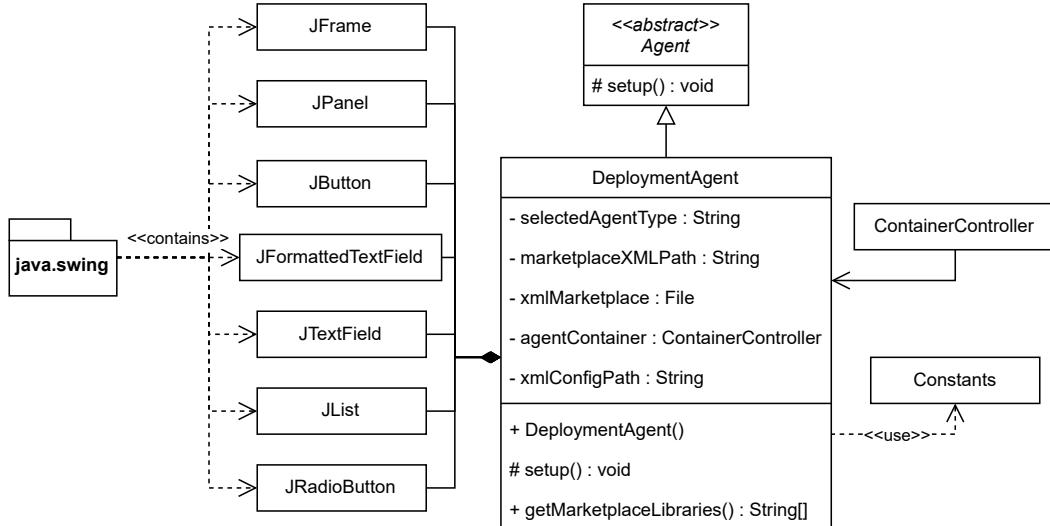


Figure 4.1: Deployment Agent class diagram.

### 4.1.3 Resource Agent

This agent is launched by the DA because it is dependent on the parameters provided by it.

The **Resource Agent** class also extends the "Agent" abstract class, and overrides the "setup" and "takeDown" methods. It makes use of the "addBehaviour" and "getArguments" methods, as well as the "ContractNetResponder" and "AchieveREResponder" classes.

It makes use of the auxiliary "Constants" and "DFInteraction" classes, which will be explained below. It has two objects of type File called "xmlConfigFile" and "xmlMarketplaceFile", given as a parameter by the DA, that store the corresponding files. The String "libType", also given as a parameter by the DA, contains the type of Link Library this agent is using and the String "location" contains the position of this agent in the physical system.

Finally, the object "moduleEngine" of type ModuleEngine holds the Module Engine instance this agent instantiates, and the ArrayList of Strings "associatedSkills" holds the list of all skills this agent is able to perform. This last object is initialized during setup, with the configurations provided by the DA as well.

The "requestResponder" class extends the "AchieveREResponder" class and the "contractNetResponder" class extends the "ContractNetResponder" class, explained in 4.1.1. Figure 4.2 shows a class diagram of the agent. In this Figure, the Module Engine and all subsequent classes are omitted, as they will be explained in 4.1.9 in more detail.

## CHAPTER 4. IMPLEMENTATION

---

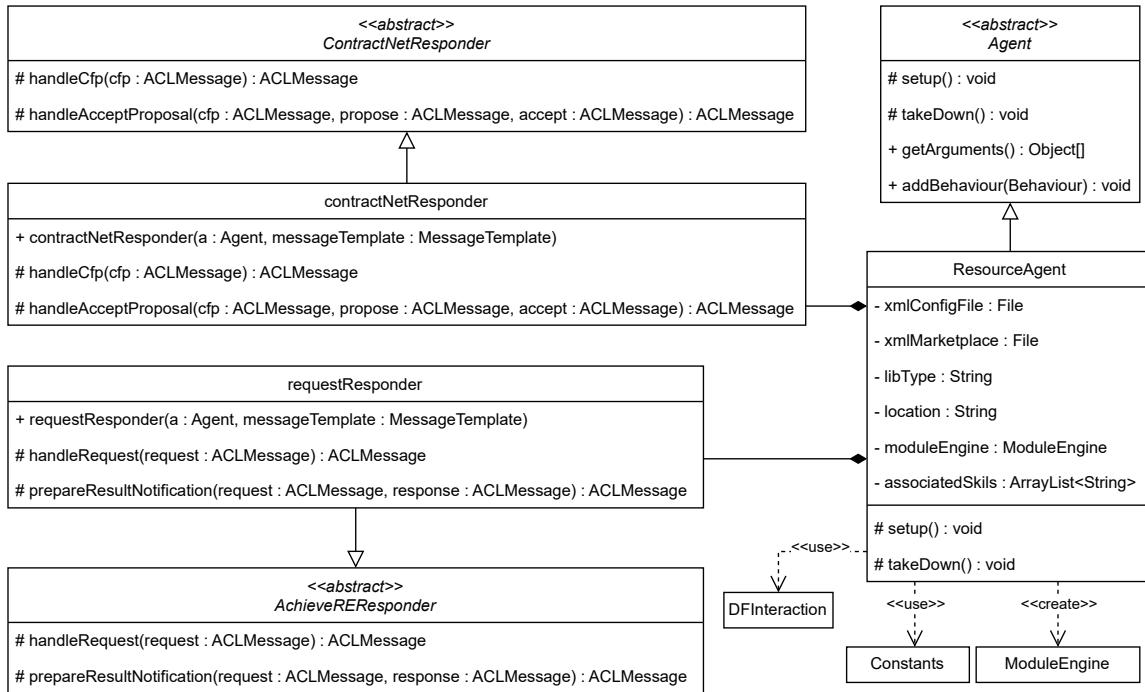


Figure 4.2: Resource Agent class diagram.

### 4.1.4 Transport Agent

Also launched by the [DA](#), the [Transport Agent](#) class is very similar to the [Resource Agent](#). It also extends the "Agent" class and contains both "setup" and "takeDown" methods.

The only differences are in the variables it uses and the classes it implements. It uses the "getArguments" method provided by [JADE](#) to get the parameters set by the [DA](#). It also needs the "xmlConfigFile" and the "xmlMarketplaceFile" for the same purposes, since it also makes use of the Module Engine. The String "libType" defines which Link Library to use, like in the [RA](#), and it also has the ArrayList of Strings "associatedSkills" to hold the list of skills and an object "moduleEngine" that holds its instance of the Module Engine.

The only exception is the variable "location", since the [Transport Agent](#) does not have a static location in the physical system. This class also makes use of the "Constants" and "DFInteraction" classes.

It implements a single "requestResponder" class which extends the "AchieveREResponder" class, used to respond to [FIPA](#) Requests, provided by [JADE](#). This is because this agent does not make use of the Contract Net, therefore does not need any of the classes that allow its use.

Figure 4.3 has a representation of its implementation. Once again the classes related to the Module Engine have been omitted.

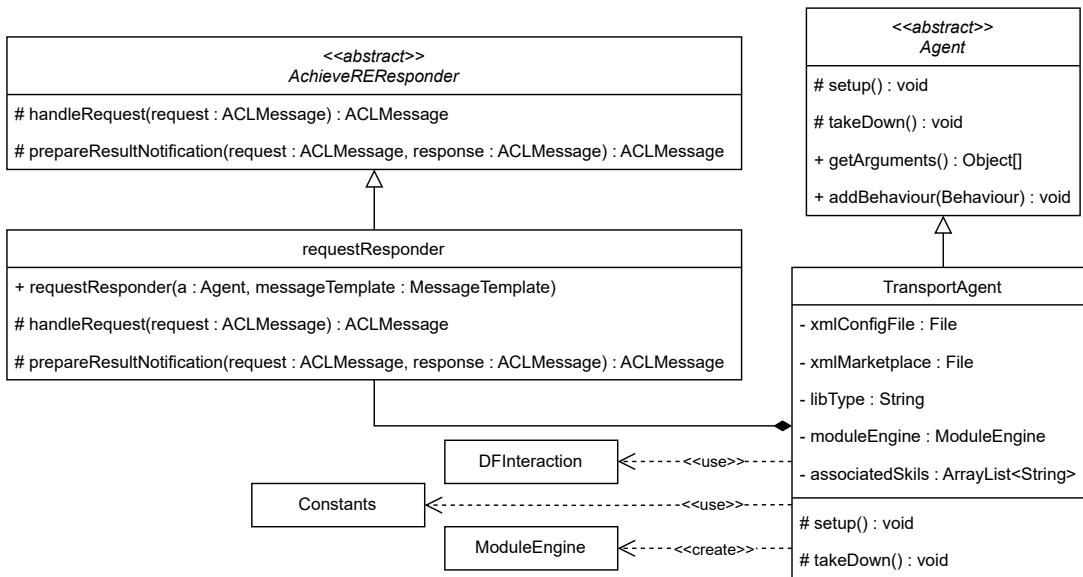


Figure 4.3: Transport Agent class diagram.

#### 4.1.5 Constants

To help define all of the constants needed for the MAS, an auxiliary class of constants was created. This "Constants" class also contains a few methods to help with information retrieval. It is mostly composed of String fields. Other classes can reference it to check production sequences, locations, skills, etc. All of this information is visible in Figure 4.4.

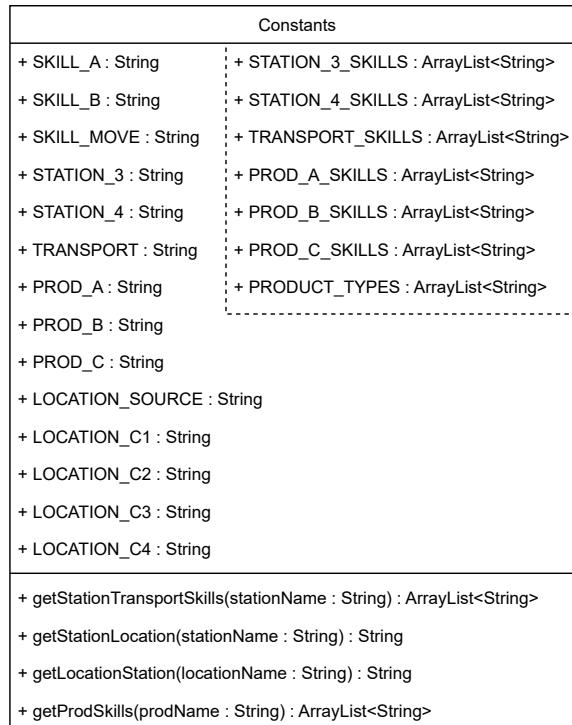


Figure 4.4: Constants class diagram

#### 4.1.6 Directory Facilitator

The "DFInteraction" is an auxiliary class. It provides methods that can register, remove or search information on the **DF**, which works as a registry for agents. In it, agents can register themselves, along with the skills they provide for the **MAS**. Other agents can then search agents by the skills they perform, and establish contact with those they find appropriate for the task, in order to request skill executions. Figure 4.5 shows the diagram of the auxiliary class implementation. The method "RegisterInDF" has two versions, one allows for the registry of an agent with a single skill, the other for an agent with multiple skills. The "DF-Service" class is implemented by **JADE** and it is what allows for the interactions with the **DF**.

**Resource Agents** and **Transport Agents** register themselves in the **DF** during their setup, with the skills they are able to perform. During production, **Product Agents** can use this registry to look up agents capable of performing their needed skills, and establish contact with those agents.

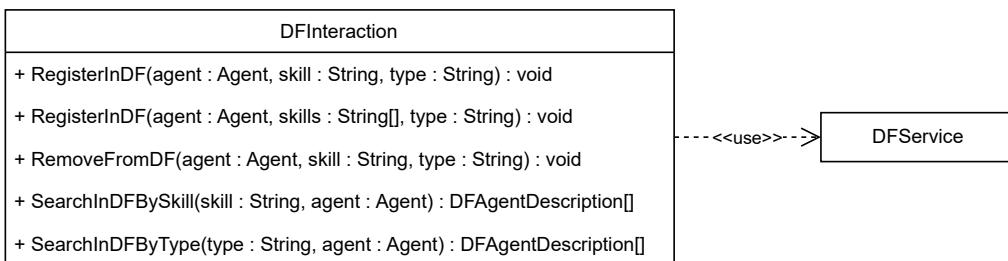


Figure 4.5: **Directory Facilitator** class diagram

#### 4.1.7 Product Manager

The **Product Manager (PM)** is another deployment entity. It also presents a **GUI** to allow a human operator to launch **Product Agents**, although it cannot terminate them. It operates similarly to the **DA**. This class also extends the "Agent" class, and also only makes use of the "setup" method, the other functionalities called on button presses.

It generates a button per product type based on the information present in the **ArrayList** of **Strings** "productTypesList". This file is populated by referencing the "Constants" class, which holds all the product types present in the **MAS**.

The "productID" is an integer appended to the name of each agent launched under the **PM**, because **JADE** does not allow agents with the same name to exist. The "products" **ArrayList** of **Strings** holds a list of product agents that have been launched so far.

In Figure 4.6, the classes that it uses to draw the **GUI** are shown, once again the "ContainerController" used to instantiate a container where **Product Agents** are launched. The "DefaultTableModel" class helps with the definition of the **GUI**, and is a class in the "javax.swing" package.

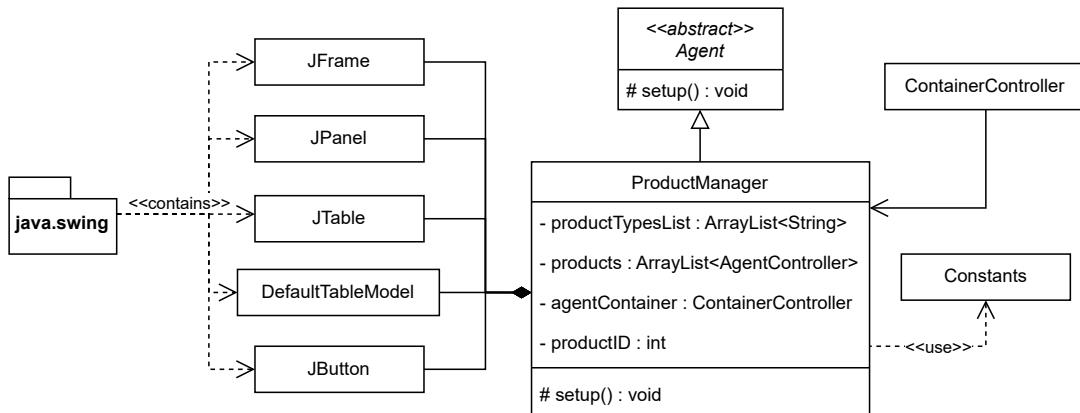


Figure 4.6: Product Manager class diagram.

#### 4.1.8 Product Agent

The **Product Agent** class extends the "Agent" abstract class and overrides the "setup" and "takeDown" methods. It makes use of the "addBehaviour" and "getArguments" methods, as well as the "OneShotBehaviour", "ContractNetInitiator" and "AchieveREInitiator" classes. It also uses the auxiliary "Constants" and "DFInteraction" classes explained above.

It has an `ArrayList` of `Strings` called "executionPlan" to store the whole skill sequence, an `int` called "step" to store the current step in that sequence and a `String` "location" that stores the agents current location on the physical system.

The "executeNextSkill" class extends the "OneShotBehaviour" class, the "contractNetInitiator" class extends the "ContractNetInitiator" class and the "requestTransportMove" and "requestStationSkill" classes extend the "AchieveREInitiator" class. These classes and methods are all represented in Figure 4.7, along with the variables they use.

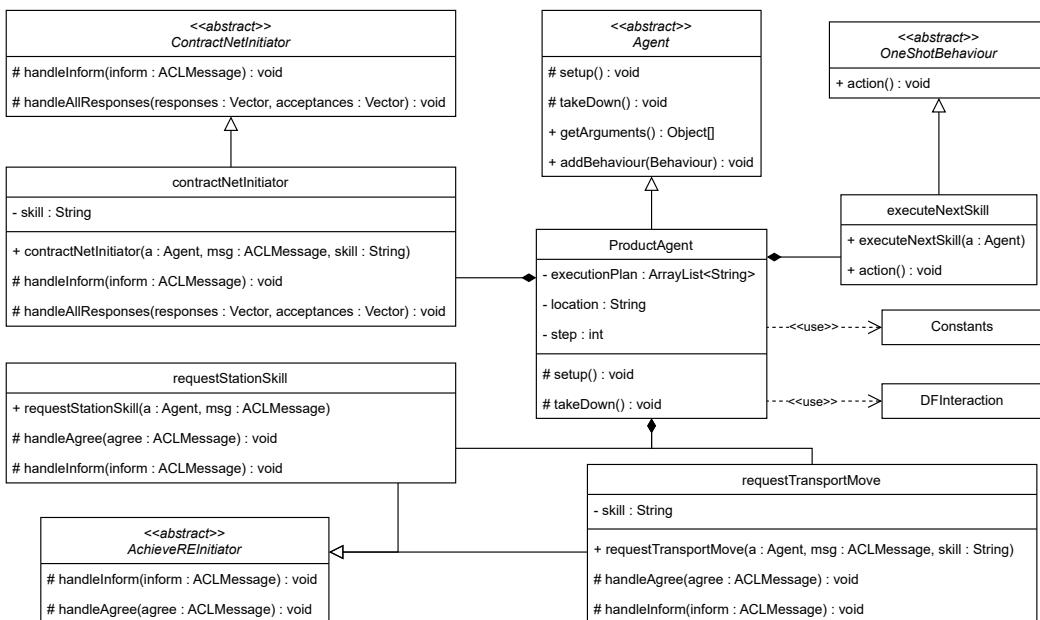


Figure 4.7: Product Agent class diagram.

#### 4.1.9 Module Engine

With the whole [MAS](#) already defined, we can now take a look at the hardware interface. The Module Engine class is the main framework that was developed for this. It contains a generic object "linkLibrary" that holds the currently loaded Link Library. This object needs to be generic since, at the start of runtime, the Module Engine does not know which type of Library will be loaded.

The method "parseMarketplaceXML" does exactly that, it parses the marketplace file with all the names and paths of all available Link Libraries and places them in the "classesToLoad" Hashmap. The name is used as the key, since it is unique, and the path as the value. This marketplace file is in the [XML](#) format, and holds both the name of the Link Library and the file path of the class that implements it. Evidently, if a Link Library has not been previously registered in the marketplace file, the Module Engine will not be able to find it, and therefore, load it. Currently, this file needs to be created and maintained by the developer, although a more complex solution could be implemented, making use of a package distribution application to download and update Link Libraries.

These Link Libraries are loaded inside the Module Engine by using the Reflections feature of the Java language. It allows for a program to inspect itself, and more importantly, to load classes and call their methods during runtime. This is what allows the Module Engine to load any kind of library at any point, while the [MAS](#) is running. This is important, because normally all the classes Java makes use of need to be compiled. However, since the Libraries that might end up being used may or may not have been created by the same developer, or exist at the time of compilation, the Module Engine needs this feature to accomplish its goal of being modular and flexible.

The method "createObject" takes the String "libType" from the agent that contains the type of Library to load. This library is then fetched from the "classedToLoad" Hashmap and attributed to a generic object of type "Class", or "Class<?>". This is then loaded using Reflections by creating a new object of that class and passing along the [XML](#) configurations file received from the agent as a parameter.

To execute a skill the method "executeSkill" is called with the skill as the argument, which will in turn get the method "ExecuteSkill" from the "linkLibrary" object and place it in a "Method" type object. This method is then invoked to execute the skill, which is passed to the Link Library also as an argument, and its return message is returned as a String.

Finally, the method "shutdown" is used to disconnect the hardware from the Link Library, and is usually called when the agent, and consequently the Module Engine, is terminated. Figure 4.8 shows how this class is implemented, along with an example [HTTP](#) Link Library.

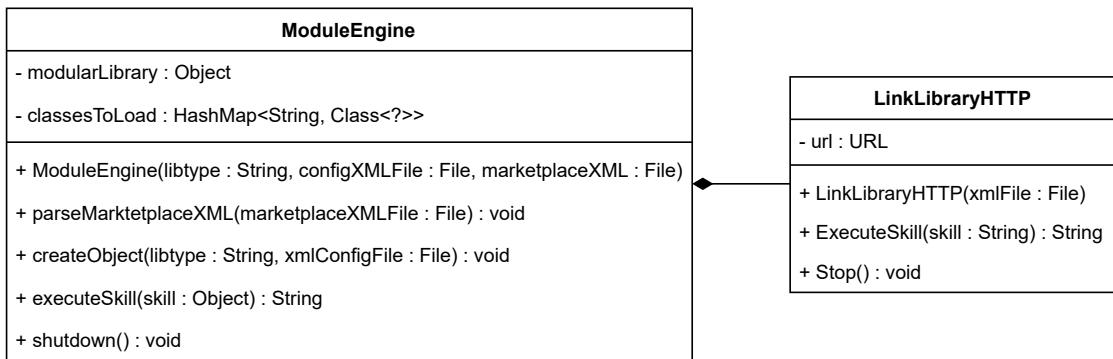


Figure 4.8: Module Engine class diagram

#### 4.1.10 Link Libraries

To showcase the functionalities of the Module Engine, three different Link Libraries were developed, each one implementing a different communication protocol. An abstract class "LinkLibrary" was extended to implement the "LinkLibraryHTTP", "LinkLibraryMQTT" and "LinkLibraryOPCUA" classes.

The "LinkLibraryHTTP" was implemented with the classes and methods provided in the "java.net" package. It communicates through [HTTP](#) Requests. When it receives a message from the Module Engine, it will relay it to the hardware through a POST message. It will now wait until the response of this POST gets returned, with the result of the operation in it.

The "LinkLibraryMQTT" is dependent on the "org.eclipse.paho.client.mqttv3" package to implement its [MQTT](#) protocol. When the Library is started, it will immediately try to connect to the broker specified in its configurations, and immediately subscribe to a response topic, also present in the configurations. When a skill needs execution, the Library publishes the skill in the requests topic, also specified in the configurations. Then, it will wait until the response topic receives a message, which it will return as the result to the operation.

Finally, the "LinkLibraryOPCUA" used the "org.eclipse.milo.opcua" package to implement an [OPC UA](#) communication. It will connect to the [OPC UA](#) server in the configuration file, and subscribe to the response node of the corresponding device. When a skill needs to be executed, the response node is cleared, to make sure the result is read. Then the skill is written on the requests node and the Library will wait until the response node value changes. When it does, the requests node is cleared and the result is returned to the agent.

All Link Libraries must extend the "LinkLibrary" class, because this is the basis of a Link Library and it defines the methods on which the Module Engine depends on for correct execution. All further Link Libraries also need to take this base, to ensure correct integration with the Module Engine.

Figure 4.9 has a representation of these classes and the respective packages they use to implement their communication protocols.

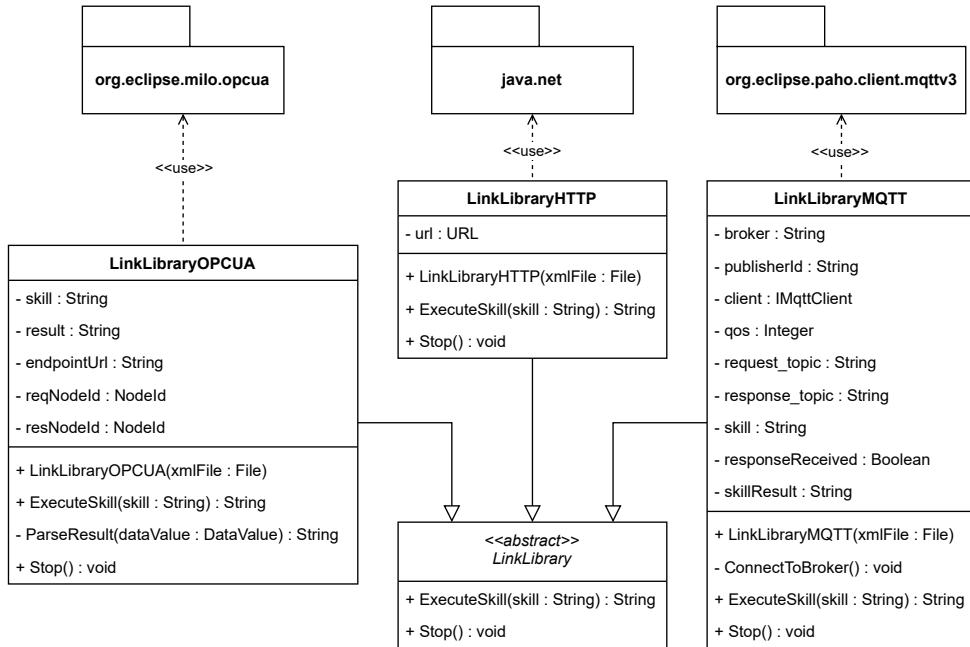


Figure 4.9: Link Library class diagrams

All developed Link Libraries were registered in the marketplace file, one Library per [XML](#) tag, each tag with two attributes. The first attribute is the exact name of the class and the second attribute the file path that points to where the given Link Library is located. With these two attributes, the Module Engine is capable of finding and loading the correct Link Library.

In order to use these Link Libraries however, the hardware that would work underneath the agent would also need to implement these three protocols in some way. In addition, a configuration file must always be provided to the Link Library for it to communicate properly.

## 4.2 Interfaces

### 4.2.1 Human to Agent

As mentioned before, there are two agents with [Graphical User Interfaces](#). The [Deployment Agent](#) and [Product Manager](#) both present a human user with an interface for agent deployment.

On the left side the [DA](#) interface has a button to open a configuration file and a text box where the path to the currently chosen file appears. Then it has a text field where the agent name is written and two radio buttons where the agent type can be selected, [Resource Agent](#) and [Transport Agent](#). Finally there is a list of all available libraries that

are fetched from the marketplace file and a button that starts the agent. On the right side there is a list with the currently running **RAs** and **TAs**. These agents can be selected by clicking on them and stopped by pressing the stop agent button below the list. Figure 4.10 shows this interface.

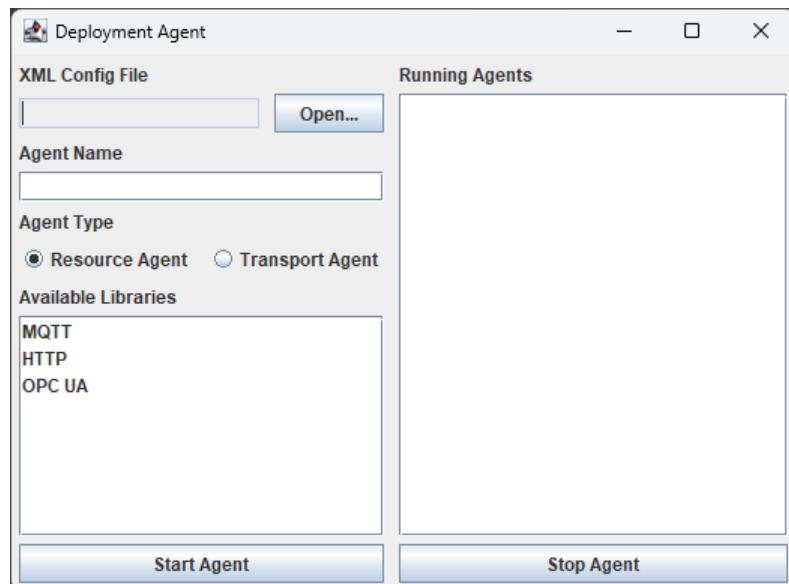


Figure 4.10: Deployment Agent Graphical User Interface.

The **PM** is more simple. It has buttons equivalent to the number of product agents specified in the "Constants" class and a list with the already launched agents. These agents have an ID represented by an integer, their product type and the skill sequence they need executed to complete the production process. We can see this interface in Figure 4.11.

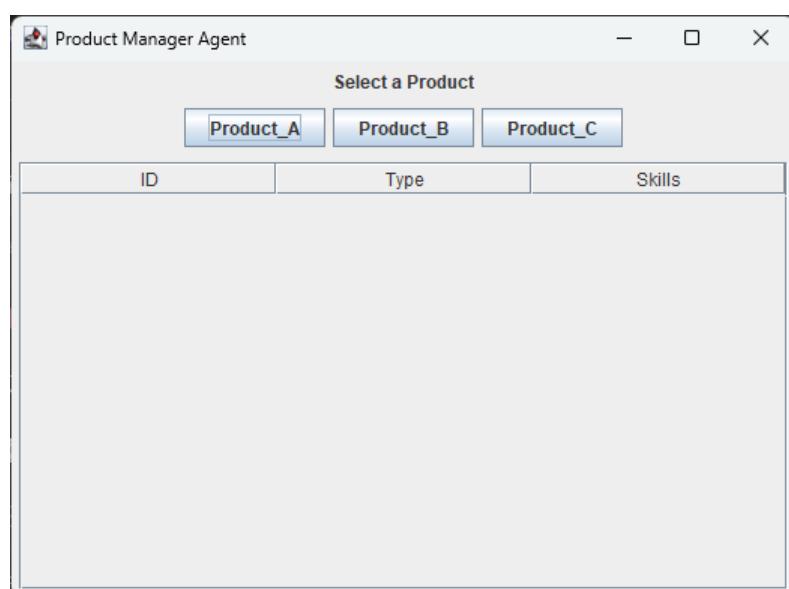


Figure 4.11: Product Manager Graphical User Interface.

### 4.2.2 Agent to Agent

Agent to agent communication is done through ACLMessages. These have been defined by [FIPA](#) [27]. These messages have many parameters that help define communications. Table 4.1 shows the parameters used by the agents of the [MAS](#), along with their utility.

Table 4.1: ACLMessage parameters

Parameter	Purpose
performative	Type of communicative act
sender	Sender of the message
receiver	Receiver of the message
content	Content of the message

The performative of a message can take multiple values. For the Requests protocol, messages of type "request", "refuse", "agree", "failure", "inform-done" and "inform-result". The ContractNet protocol makes use of messages of type "Call For Proposals" or "CFP", "refuse", "propose", "reject-proposal", "accept-proposal", "failure", "inform-done" and "inform-result". These message types and their respective protocols can be seen in Table 4.2.

Table 4.2: ACLMessage performative types

Parameter	Protocol	
	ContractNet	Requests
Request		✓
Refuse	✓	✓
Agree		✓
Failure	✓	✓
Inform-done	✓	✓
Inform-result	✓	✓
CFP	✓	
Propose	✓	
Reject-proposal	✓	
Accept-proposal	✓	

### 4.2.3 Agent to Hardware

To interface with the hardware, an agent must call the "executeSkill" method of the Module Engine. In its parameters it should send the skill as a String. The Module Engine will then call the "ExecuteSkill" of the Link Library loaded by it. All Link Libraries must contain the method "ExecuteSkill", otherwise calling it would be impossible. This method is implemented by the abstract class "LinkLibrary", as described in 4.1.9.

The Link Library will now forward the skill through the protocol it is implementing, in whatever format it was designed for. A developer could reformat this message to whatever type they want, but the Link Library must always return a result of type String. As long as these rules are obeyed, Link Libraries can be used in the way most suitable to the system they are in. The result is passed to the Module Engine as a String, which is then passed to the agent.

Three Link Libraries were implemented, seen in [4.1.10](#). The [HTTP](#) library works by creating an [HTTP](#) connection, every time a skill is to be executed, using the address provided in the configurations. It sends the skill as a payload in a POST request. It then waits for an OK message with code 200 as per the [HTTP](#) protocol.

This response must include the result of the operation that is converted into a String and passed upward to the Module Engine. This library does not need to disconnect from the [HTTP](#) server, since this protocol does not depend on a persistent connection.

The implemented [MQTT](#) library needs more parameters to work. It requires an [MQTT](#) broker address, a [Quality of Service \(QoS\)](#) value, a request topic and a response topic. These topics are differentiated to allow for a simpler implementation of the library.

When the library is loaded, it immediately establishes a connection to the broker and subscribes to the response topic. This means that every time a response arrives, the callback function is called. This function simply stores the message as a String.

Whenever the "ExecuteSkill" method is run, the library will publish the skill in the request topic. It then waits until the response topic gets a message. Upon receiving it, it will return it to the Module Engine. When this library needs to disconnect, it just disconnects from the broker.

The [OPC UA](#) library is a bit more complex than the other two. It also needs a server address, or endpoint, and a namespace. This namespace identifies which container the node representing the hardware is. In this node, two variable nodes are found, one for incoming requests and one for outgoing responses. To summarize, the library needs an endpoint address, a namespace, the identifier of the request node and the identifier response node.

When the Link Library is loaded, it immediately connects to the [OPC UA](#) server and creates two node objects locally with the namespace and identifiers, one with the request and one with the response identifier. It also creates a subscription to the responses node.

When a skill is to be executed, first the response node is cleared, to make sure the library gets notified of a new message. Then the skill is written to the requests node. The library waits until a response arrives and when it does it saves it and clears the requests node for the next cycle of communications. The result message is sent to the Module Engine to be returned to the Agent.

### 4.3 Multi-agent System Operations

With the whole system now defined, we can proceed to review how it operates. Before the system can be started however, the marketplace file needs to contain the names and paths of the Link Libraries the **MAS** is dependent on.

When the **MAS** is first executed, both the **Deployment Agent** and **Product Manager** are launched. These agents will run their own setups.

The **PM** will run its "setup" method. It will look into the "Constants" class and find what **Product Agent** are setup there and create buttons that correspond to each **PA**. It will setup the container where product agents will be deployed and draw the **GUI**.

The **DA** will also run the "setup" method. It will try to find a marketplace file and list the Link Libraries there listed by running the "getMarketplaceLibraries" method. Then the agent container is created and the interface will be setup with its own default values and drawn to the screen. The system is now ready to launch agents.

Figure 4.12 starts when the system is first executed. It shows the steps and methods until it is operational. Since no more agents are running at this point, only the **DA** and **PM** are shown.

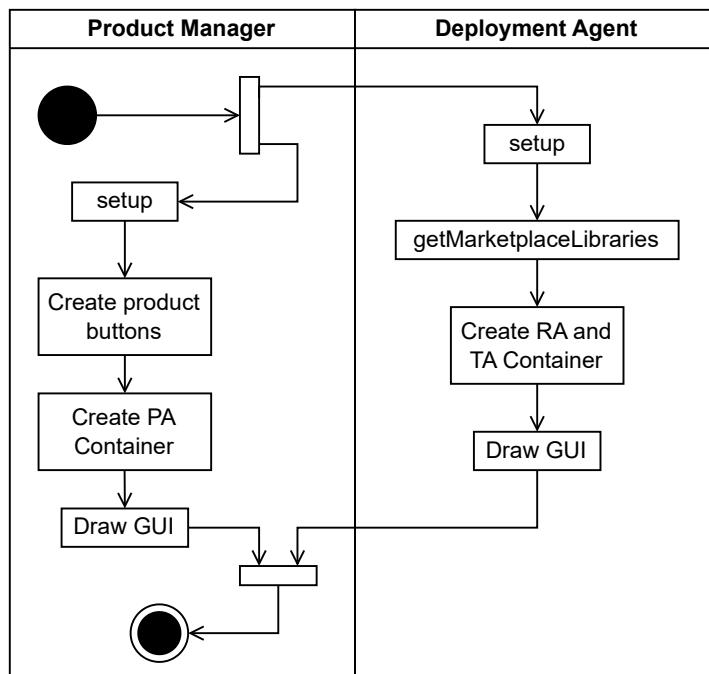


Figure 4.12: **Multi-Agent System** startup.

Before any product can be produced however, a human operator needs to deploy the **Resource Agents** and **Transport Agents** that compose the system.

### 4.3.1 Initial Setup

Resource Agents and Transport Agents agents need to be deployed through the DA. From the available Link Libraries, the operator will chose which one needs to be launched with the agent to interface with the hardware. This Link Library must have been previously developed according to the specifications so it is compatible with the Module Engine, and its name and file path added to the marketplace file. Upon selecting a Link Library, the corresponding configuration file for this agent must be selected. Then the agent must be given a name and its type selected. Now the agent can be deployed by pressing the "Start Agent" button. This process now needs to be repeated for every agent, until all the agents have been launched.

When an agent is launched, its "setup" method is run. In it the agent will start the Module Engine with the Link Library of the type selected, with the configurations provided to it. It will create a Link Library object by getting all the Link Libraries in the marketplace file with "parseMarketplaceXML" and create a Link Library object with "createObject". The Link Library will now connect to the hardware if needed, in its constructor. The agent registers itself in the DF using the method "RegisterInDF". Then the first behaviour is added to the agents operating sequence. In the case of the RA, it will be the "contractNetInitiator" and the "requestResponder". In the case of the TA, it will only be the "requestResponder".

Figure 4.13 shows this process for either a Resource Agent or Transport Agent. In the case of the RA, the "contractNetInitiator" behaviour needs to be added as well. The DA is omitted here for simplicity, since it would only deploy the agent.

With the MAS now built, new products can be launched by pressing the corresponding button in the PM.

### 4.3.2 Launching a Product

When a product is deployed, it will get its production sequence from the PM and set its own location as the starting position. It then will add the "executeNextSkill" to the behaviour sequence. This behaviour is executed immediately, its "action" method run, and it will search in the DF, using "DFInteraction", for an agent capable of performing its first skill. Upon finding it, it proceeds to send a Call for Proposals to all found agents.

For this the "contractNetInitiator" class is added to the behaviours and communications are established. Upon receiving a message, the RA runs "handleCfp" which generates a proposal. The PA receives it in "handleAllResponses", picks one if there are multiple and answers it. "handleAcceptPorposal" from the RA informs the PA of its location. Finally "handleInform" in the PA looks for a transport agent in the DF.

Now the product will ask the found TA for transportation using the "requestTransportMove" class. This starts the Requests protocol and after the TA gets the first message through "handleRequest" it creates an Agree message that is forwarded to the PA and

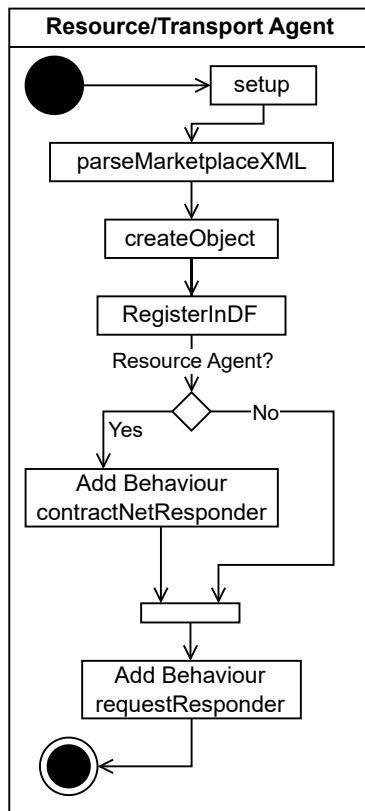


Figure 4.13: Agent startup

immediately starts the "prepareResultNotification", which executes the skill through the Module Engine and provides a result as an inform message.

The Module Engine, on receiving the skill in its "executeSkill", will execute the "ExecuteSkill" method in the Link Library. Depending on the Link Library, the message might be sent through different channels, explained in [4.2.3](#). When the skill finishes execution, the result is sent back to the **TA**, which Informs the **PA** of it completion.

The **PA** now updates its location and initiates another Requests communications with the class "requestStationSkill". This follows the same logic as the transportation requests, but with a **Resource Agent** instead.

After being informed of the skills execution, the "handleInform" method increments the "step" field, which represents the execution step in the skill sequence", an checks if there are more skills to be executed. If there are, it adds a new "executeNextSkill" to the behaviour sequence and the process restarts. If there are not, the transport agent is requested again through "requestTransportMove", but this time to the deposit location in the **MAS**, defined in the "Constants" class. After the move is complete, the agent will terminate itself.

In Figure [4.15](#), at the end of this chapter, we can observe the normal system operations. The diagram specifies which class executes what method. If no class is specified then it is

the agent class. The Module Engine has been initialized at this point and only needs to execute skills. All agents have been registered to the **DF** and the **PM** as been omitted since it would only launch the **Product Agent**.

### 4.3.3 Removing Agents

When a **Resource Agent** or **Transport Agent** needs to be removed from the **MAS**, the agent needs to be selected from the **DA** window and the "Stop Agent" button pressed. This will stop the corresponding agent by running its "takeDown" method. This method will in turn run the "shutdown" Module Engine method, which will run the "Stop" Link Library method.

The Link Library will disconnect from the hardware and the agent will remove itself from the **DF**. Finally, the agent will be terminated. This process is shown in Figure 4.14.

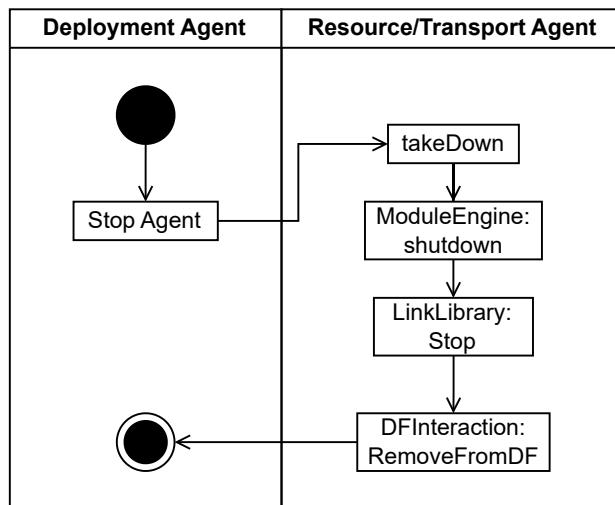


Figure 4.14: Agent termination

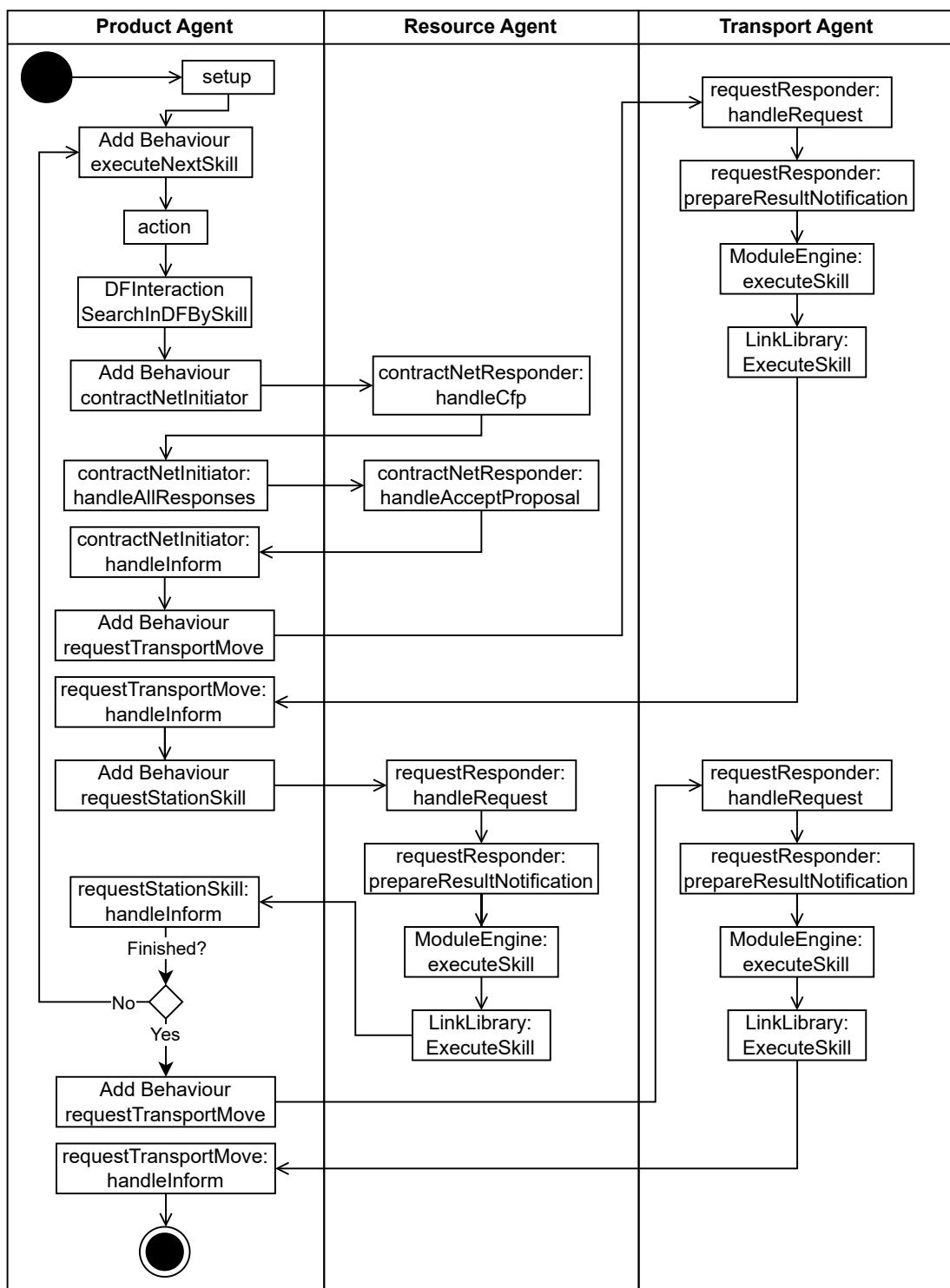


Figure 4.15: Multi-Agent System operations

# TESTS

With the implementation done, it is time to test the system. First, a few performance tests were executed to ascertain the efficiency of the Module Engine when compared to a standard implementation. Agent deployment times and skill execution times were measured and compared. A brief explanation of the physical system used to simulate a production line is given. Then the tests on this system are described. Finally, a few deployment and skill execution tests are described to ascertain the efficiency of the Module Engine.

## 5.1 Performance Tests

The performance tests were done on a machine with an AMD Ryzen 7 5700X processor and 32 gigabytes of RAM.

### 5.1.1 Deployment Test

The first test is the deployment test. In it, an increasingly amount of agents are launched to measure how long it takes to launch each agent. An agent using the Module Engine and a normal agent are compared to ascertain how much the Module Engine impacts agent launch times.

This test was repeated three times, the first a single agent of each type is launched, the second ten agents of each type are launched, and the third a hundred agents are launched. Measurements are made by starting a timer when the "setup" method of an agent starts, and then stopped once the setup finishes. An agent running the Module Engine should take longer since it needs to get the parameters from the deployment entity and needs to instantiate its Module Engine and wait for its setup to finish. The comparison is made between an agent that needs to follow these procedures versus and agent that does not. The times are written to a file that gets read to extract the values and calculate results.

As expected, an agent using the Module Engine takes more time before it is ready to operate. The one agent took 15171400 nanoseconds, or 15 milliseconds, to launch. An agent without Module Engine took around 1300 nanoseconds.

Ten agents with the Module Engine took 203 milliseconds total, with an average time of 20340230 nanoseconds, or 20 milliseconds, for each agent while the agents without took 3100 nanoseconds total, with an average of 310 nanoseconds per agent.

Finally, a hundred agents needed 894 milliseconds, with an average time of 9 milliseconds per agent. The sole agents needed 22104 nanoseconds, with 221 nanoseconds per agent.

Although the launch times increased considerably, we must take into account that the agents without the Module Engine did not have, effectively, any operations in their setup. A normal agent in an industrial application would need a setup of sorts to operate. Even though the duration of the setup for an agent making use of the Module Engine is higher, it is still under acceptable values, since none have too great a magnitude. We can also observe that an agent using the Module Engine does not impact other agents, so it will not have a great impact on the system as a whole.

Table 5.1: Deployment times.

Number of agents	with Module Engine		without Module Engine	
	Total	Average	Total	Average
1	15 ms	15 ms	1300 ns	1300 ns
10	203 ms	20 ms	3100 ns	310 ns
100	894 ms	9 ms	221044 ns	221 ns

### 5.1.2 Execution Test

The second test is the skill execution delay test. Here, a skill is executed through the Module Engine. A special Link Library was developed for this test, and its only function is to return a String to simulate the result of a skill. This result is then written on the console. The execution time of this operation is compared to the time it takes for the system to write the same message to the console, without going through the Module Engine.

This test was repeated sixty times, the five best and worst results excluded from the measured values to control for variations. The average time was calculated and compared. The operation without the Module Engine takes an average time of 12576 nanoseconds, or around 0.013 milliseconds. The operation with the Module Engine takes an average time of 28976 nanoseconds, or 0.029 milliseconds, also less than a millisecond.

The difference between executing with or without the Module Engine is around 16400 nanoseconds. It takes almost double the time for the message to be written, however, since the time is still under a millisecond, the impact caused by the Module Engine and Link Libraries is still minimal for an industrial application. The trade-off between skill execution time and modularity and flexibility is, arguably, worth it.

Table 5.2: Execution times.

	with Module Engine	without Module Engine
Time	28976 ns	12576 ns

## 5.2 Case Study

The [MAS](#) was simulated on a physical system composed of a conveyor belt and two processing stations. The conveyor belt is divided into four sections, each with their own sensor. This means that products can stop in the middle of any of the four sections. They are also independent, meaning that they can move and stop individually. The stations can move forward, backward, up and down, and their tool spins to simulate the execution of a skill. This model is illustrated in Figure 5.1.

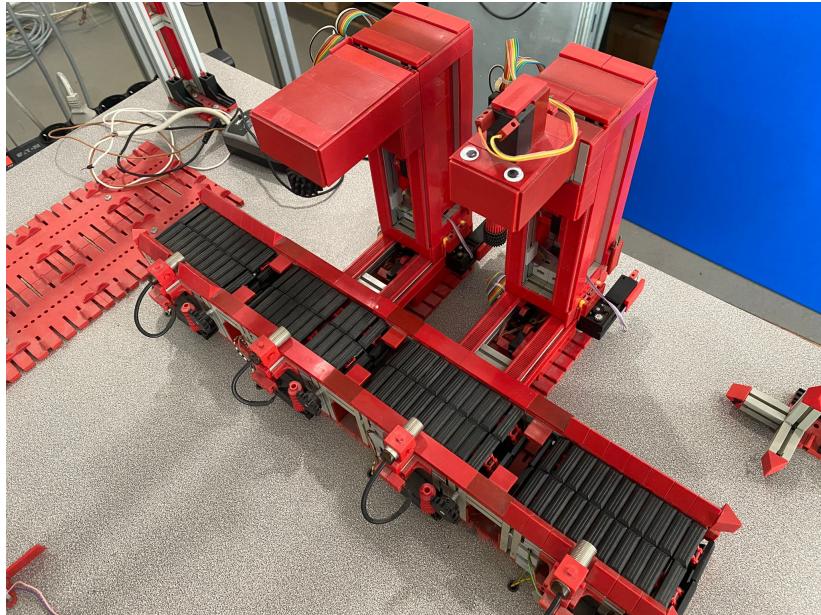


Figure 5.1: Simulation kit.

Each element of this system can be represented by an agent. The conveyor belt fits the definition of a [Transport Agent](#) and both stations can be [Resource Agent](#). All of these entities were defined in the "Constants" class, the [TA](#) as "Conveyor" and both stations as "Station\_3", for the station on the left (Figure 5.2(a)), and "Station\_4", for the station on the right (Figure 5.2(b)).

Both stations are able to perform a skill. To reduce complexity, each station was attributed a single unique skill. "Station\_3" is able to perform "Skill\_A" and "Station\_4" can perform "Skill\_B". Both of these were also defined in the "Constants" class. These skills are registered by the respective agent in the [DF](#). The act of performing the skill corresponds to the device moving forward, activating its tool for about three seconds and then moving to the starter position.

The "Conveyor" agent is capable of performing the skill "Skill\_Move", also defined in

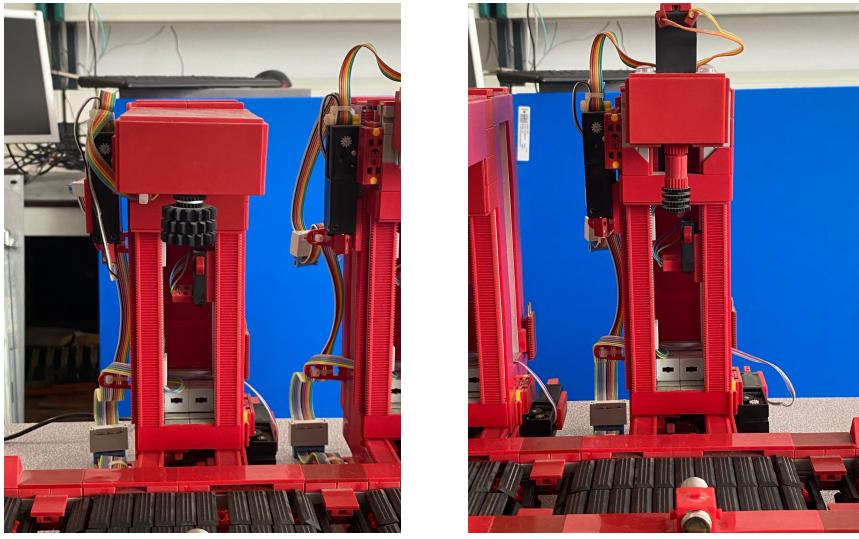


Figure 5.2: Kit stations.

the "Constants". This is the skill with which the agent registers itself in the [DF](#). When a [Product Agent](#) needs transportation, it must send a message to the "Conveyor" agent through the aforementioned channels in the format "CX#TOKEN#CY". Here, X represents the source location and Y represents the destiny location. The conveyor is split into four segments, numbered one to four from the left to the right. Each segment has a sensor that detects whether or not a product is on it. So the instruction "C1#TOKEN#C3" makes the conveyor turn the first, second and third segments to the right. Similarly, the instruction "C3#TOKEN#C2" makes the conveyor turn the second and third segments to the left.

A RevPi running an instance of NodeRED was used to control the system, its inputs and outputs connected to the conveyor and stations. A RevPi is a Raspberry Pi equipped with an shield board that allows it to interface hardware, like sensors and actuators. This RevPi was hosting the NodeRED instance, and it depended on an official RevPi NodeRED library to control the physical system.

NodeRED is an open source programming tool, capable of integrating hardware. It works by connecting nodes, which send messages to other connected nodes to complete processes. These groupings of nodes are called flows. Three flows were created, one for each cyber-physical entity. These flows are capable of receiving messages in [HTTP](#), [MQTT](#) and [OPC UA](#), and return the result through the same protocol. Figure 5.3 shows the developed nodes. It also shows the [OPC UA](#) server that was needed for [OPC UA](#) communications. The [MQTT](#) broker was hosted on a different machine.

Three configuration files were created, one for each cyber-physical entity. A single

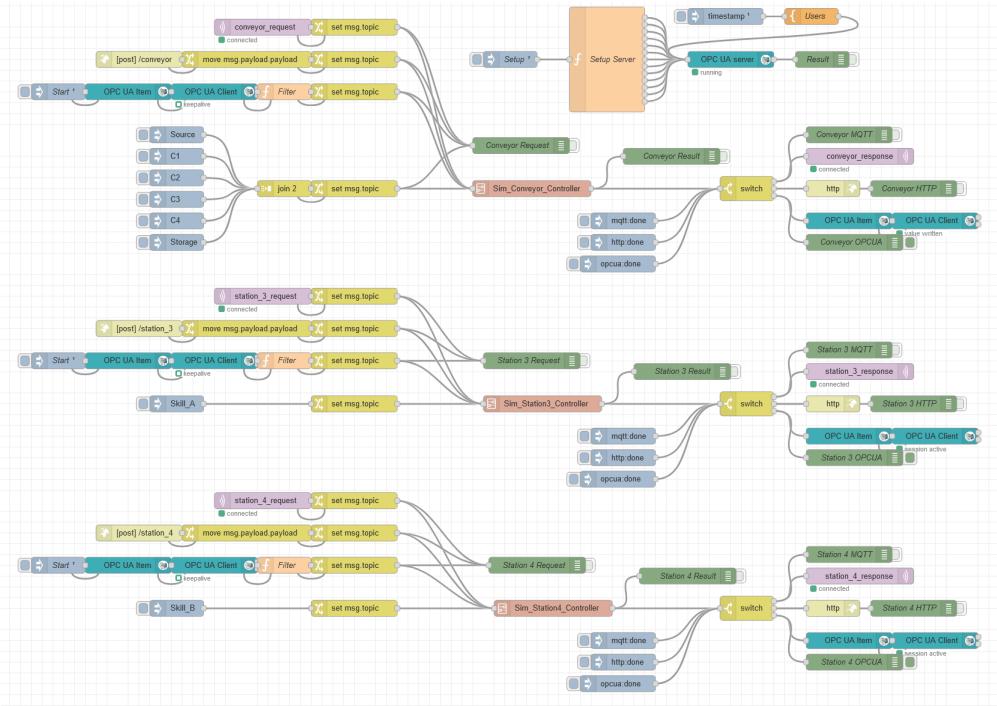


Figure 5.3: NodeRED flows.

file contains the configurations needed for the three developed Link Libraries. Table 5.3 shows the configurations for the **MQTT** protocol and Table 5.4 for **OPC UA**. The field for the address is not shown as well as the configurations for **HTTP** since the only element is the server address.

 Table 5.3: **MQTT** configurations.

	Station_3	Station_4	Conveyor
Quality of Service	2	2	2
Request topic	station_3_request	station_4_request	conveyor_request
Response topic	station_3_response	station_4_response	conveyor_response

 Table 5.4: **OPC UA** configurations.

	Station_3	Station_4	Conveyor
Namespace index	1	1	1
Requests identifier	Station3_Req	Station4_Req	Conveyor_Req
Response identifier	Station3_Res	Station4_Res	Conveyor_Res

To simulate the system, both NodeRED and the **MAS** systems were launched. The **Product Manager** and **Deployment Agent** are immediately deployed, finish their initial setup explained in 4.3.1 and wait for agents to be launched. After this, the two **Resource Agents** were launched, "Station\_3" and "Station\_4", with their respective configurations. Along with those agents, the "Conveyor" **Transport Agent** is also launched, also with its configuration file. At first, all agents are using the Link Library that communicates

through [MQTT](#), and thus the configurations for this protocol are used. These agents are connected to the broker on launch, and will communicate to the corresponding topics.

After the agents were done with their initial setups, a product agent was launched. For the purposes of showcasing the systems capabilities, three different Product Types were created. These products are shown in [5.5](#), along with their production sequences and the sequence of Conveyor Belt sections they need to visit in order to complete their process. All of these values were also defined in the "Constants" class.

Table 5.5: Product Types

Product Type	Production Sequence	Location Sequence
A	[Skill_A]	[C1; C2; C4]
B	[Skill_B]	[C1; C3; C4]
C	[Skill_B; Skill_A]	[C1; C3; C2; C4]

All three product types were launched, one at a time. Product A was first, and the system was able to complete the production sequence without any problems. The product first went to Station 3, the station applied the skill, and then moved on to storage. The other two products followed a similar result, their production sequence executed as planned. This means the Module Engine was working as intended. Even though it wasn't hard-coded to the [MQTT](#) protocol, it was able to use the Link Library that integrates it to communicate to the machines.

To further test the Module Engine, "Station\_3" was taken down and restarted, this time with the [HTTP](#) Link Library. The same procedure was performed, with the same result. No noticeable difference was observed in the three products sequences. The Module Engine was once again integrating a new protocol.

The only difference noted was when [OPC UA](#) was used, the stations wouldn't respond as fast to the instructions. However, this was attributed to the fact that the RevPi was running both the [OPC UA](#) server and the NodeRED instance. This could easily explain the delay observed. Despite this, the Module Engine was also capable of integrating this protocol.

Many combinations of the three protocols were used, with the same results. The Module Engine was integrating all three protocols as if it was hard-coded with them. The Link Libraries developed were acting as interfaces between the Module Engine and NodeRED, but this interface could be replaced at any point to integrate new hardware.

This outcome was expected, since through the use of the Reflections feature of the Java programming language, a class can be loaded during runtime as if the program was compiled with it. Some kind of delays were to be expected, since there are extra instructions running in the background due to the Module Engine serving as an intermediary between agent and Link Library , but their impact is minimal, since the Module Engine is fairly lightweight. This is also compensated by the flexibility it provides.

## CONCLUSION

The concept of Industry 4.0 revolutionized the manufacturing sector through the digitalization of manufacturing processes. Industrial [Cyber-Physical System](#) or [Cyber-Physical Production Systems](#) enable the transition to the Industry 4.0 standard. These systems are service oriented, can process Big Data and have cloud integration for storage.

They are also able to interface with the real world, by using sensors, and to act on it, by using actuators. In essence these systems are composed of two counterparts, a physical one and a digital one. This makes them very robust and efficient, since they rely on the digital version to extract information on how to act on their environment.

Industrial [Multi-Agent System](#) were suggested as a model for the implementation of a [Cyber-Physical Production System](#) due to the advantages they could bring to the industry, such as the decentralization of the system, allowing for the autonomous behavior of each individual agent to accomplish a common goal. This makes the system very robust and flexible because errors don't propagate throughout the different layers of the system and agents can join and leave the system as needed. They, however, have not seen practical uses outside research prototypes. This may come from the skepticism that they won't perform to the same capabilities as existing systems.

In this work, a method to facilitate the development of Industrial [Multi-Agent System](#) was presented. Through the use of the Module Engine, it is possible to integrate any previously developed Link Library with any kind of agent. These Link Libraries could be developed to support all kinds of protocols and hardware, facilitating the integration of a [Cyber-Physical Production System](#).

The prototype presented in this work shows a lot of promise, with minimal drawbacks to performance, it is a great alternative to the methodologies employed nowadays for the integration of hardware on a [Multi-Agent System](#). It could be improved in many ways, like the ability to change Link Libraries without having to terminate an agent or by creating a web application capable of providing and updating Link Libraries remotely and automatically. Still, this kind of framework is a step in the right direction of the wide adoption of [Multi-Agent System](#) with industrial applications.

## BIBLIOGRAPHY

- [1] J. M. Lourenço. *The NOVAtesis L<sup>A</sup>T<sub>E</sub>X Template User's Manual*. NOVA University Lisbon. 2021. URL: <https://github.com/joaomlourenco/novathesis/raw/main/template.pdf> (cit. on p. i).
- [2] B. Vogel-Heuser and D. Hess. "Guest Editorial Industry 4.0–Prerequisites and Visions". In: *IEEE Transactions on Automation Science and Engineering* 13.2 (2016), pp. 411–413. DOI: [10.1109/TASE.2016.2523639](https://doi.org/10.1109/TASE.2016.2523639) (cit. on pp. 1, 4).
- [3] *Industry 4.0 Case Studies (curated)*. URL: <https://amfg.ai/2019/03/28/industry-4-0-real-world-examples-of-digital-manufacturing-in-action/> (visited on 2024-07-29) (cit. on p. 2).
- [4] L. Sakurada and P. Leitão. "Multi-Agent Systems to Implement Industry 4.0 Components". In: *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*. Vol. 1. 2020, pp. 21–26. DOI: [10.1109/ICPS48405.2020.9274745](https://doi.org/10.1109/ICPS48405.2020.9274745) (cit. on p. 2).
- [5] S. Karnouskos et al. "Industrial Agents as a Key Enabler for Realizing Industrial Cyber-Physical Systems: Multiagent Systems Entering Industry 4.0". In: *IEEE Industrial Electronics Magazine* 14.3 (2020), pp. 18–32. DOI: [10.1109/MIE.2019.2962225](https://doi.org/10.1109/MIE.2019.2962225) (cit. on p. 2).
- [6] R. Unland. "Chapter 1 - Software Agent Systems". In: *Industrial Agents*. Ed. by P. Leitão and S. Karnouskos. Boston: Morgan Kaufmann, 2015, pp. 3–22. ISBN: 978-0-12-800341-1. DOI: [10.1016/B978-0-12-800341-1.00001-2](https://doi.org/10.1016/B978-0-12-800341-1.00001-2) (cit. on pp. 2, 6).
- [7] The Foundation for Intelligent Physical Agents. *Welcome to the Foundation for Intelligent Physical Agents*. URL: <http://www.fipa.org/> (visited on 2024-07-29) (cit. on p. 2).
- [8] B. Marschall et al. "Design and Installation of an Agent-Controlled Cyber-Physical Production System Using the Example of a Beverage Bottling Plant". In: *IEEE Journal of Emerging and Selected Topics in Industrial Electronics* 3.1 (2022), pp. 39–47. DOI: [10.1109/JESTIE.2021.3097941](https://doi.org/10.1109/JESTIE.2021.3097941) (cit. on pp. 2, 12, 13, 15).

- [9] S. Karnouskos and P. Leitão. "Key Contributing Factors to the Acceptance of Agents in Industrial Environments". In: *IEEE Transactions on Industrial Informatics* 13.2 (2017), pp. 696–703. doi: [10.1109/TII.2016.2607148](https://doi.org/10.1109/TII.2016.2607148) (cit. on pp. 2, 12).
- [10] P. Leitão, A. W. Colombo, and S. Karnouskos. "Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges". In: *Computers in Industry* 81 (2016). Emerging ICT concepts for smart, safe and sustainable industrial systems, pp. 11–25. issn: 0166-3615. doi: [10.1016/j.compind.2015.08.004](https://doi.org/10.1016/j.compind.2015.08.004) (cit. on p. 5).
- [11] O. Cardin. "Classification of cyber-physical production systems applications: Proposition of an analysis framework". In: *Computers in Industry* 104 (2019), pp. 11–21. issn: 0166-3615. doi: [10.1016/j.compind.2018.10.002](https://doi.org/10.1016/j.compind.2018.10.002) (cit. on p. 5).
- [12] D. H. Arjoni et al. "Manufacture Equipment Retrofit to Allow Usage in the Industry 4.0". In: *2017 2nd International Conference on Cybernetics, Robotics and Control* (CRC). 2017, pp. 155–161. doi: [10.1109/CRC.2017.46](https://doi.org/10.1109/CRC.2017.46) (cit. on pp. 5, 6).
- [13] P. Leitão et al. "Recommendation of Best Practices for Industrial Agent Systems based on the IEEE 2660.1 Standard". In: *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*. Vol. 1. 2021, pp. 1157–1162. doi: [10.1109/ICIT46573.2021.9453511](https://doi.org/10.1109/ICIT46573.2021.9453511) (cit. on pp. 6, 7).
- [14] S. Karnouskos et al. "Key directions for industrial agent based cyber-physical production systems". In: *Proceedings - 2019 IEEE International Conference on Industrial Cyber Physical Systems, ICPS 2019* (2019-05), pp. 17–22. doi: [10.1109/ICPHYS.2019.8780360](https://doi.org/10.1109/ICPHYS.2019.8780360) (cit. on pp. 6, 8, 12).
- [15] P. Leitão et al. "Integration Patterns for Interfacing Software Agents with Industrial Automation Systems". In: *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*. 2018, pp. 2908–2913. doi: [10.1109/IECON.2018.8591641](https://doi.org/10.1109/IECON.2018.8591641) (cit. on pp. 8–10).
- [16] "IEEE Recommended Practice for Industrial Agents: Integration of Software Agents and Low-Level Automation Functions". In: *IEEE Std 2660.1-2020* (2021), pp. 1–43. doi: [10.1109/IEEESTD.2021.9340089](https://doi.org/10.1109/IEEESTD.2021.9340089) (cit. on p. 8).
- [17] *Jade Site | Java Agent DEvelopment Framework*. url: <https://jade.tilab.com/> (visited on 2024-07-29) (cit. on p. 10).
- [18] The OPC Foundation. *Unified Architecture - OPC Foundation*. url: <https://opcfoundation.org/about/opc-technologies/opc-ua/> (visited on 2024-07-29) (cit. on pp. 10, 11).
- [19] M. Seitz et al. "Automation platform independent multi-agent system for robust networks of production resources in industry 4.0". In: *Journal of Intelligent Manufacturing* 32 (7 2021-10), pp. 2023–2041. issn: 15728145. doi: [10.1007/S10845-021-01759-2](https://doi.org/10.1007/S10845-021-01759-2) (cit. on p. 11).

## BIBLIOGRAPHY

---

- [20] F. L. Alejano et al. "Enhancing the interoperability of heterogeneous hardware in the Industry: a Multi-Agent System Proposal". In: *2023 6th Conference on Cloud and Internet of Things (CIoT)*. 2023, pp. 157–162. doi: [10.1109/CIoT57267.2023.10084891](https://doi.org/10.1109/CIoT57267.2023.10084891) (cit. on p. 11).
- [21] MQTT.org. *MQTT: The Standard for IoT Messaging*. URL: <https://mqtt.org/> (visited on 2024-09-07) (cit. on p. 12).
- [22] B. Marschall, D. Ochsenkuehn, and T. Voigt. "Design and Implementation of a Smart, Product-Led Production Control Using Industrial Agents". In: *IEEE Journal of Emerging and Selected Topics in Industrial Electronics* 3.1 (2022), pp. 48–56. doi: [10.1109/JESTIE.2021.3117121](https://doi.org/10.1109/JESTIE.2021.3117121) (cit. on pp. 12, 13).
- [23] A. D. Rocha et al. "Agent-based Plug and Produce Cyber-Physical Production System – Test Case". In: *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*. Vol. 1. 2019, pp. 1545–1551. doi: [10.1109/INDIN41052.2019.8972169](https://doi.org/10.1109/INDIN41052.2019.8972169) (cit. on pp. 15, 16).
- [24] A. D. Rocha, G. Barata Diogo and Di Orio, and J. Santos Tiago and Barata. "PRIME as a Generic Agent Based Framework to Support Pluggability and Reconfigurability Using Different Technologies". In: *Technological Innovation for Cloud-Based Engineering Systems*. Ed. by L. M. Camarinha-Matos, T. A. Baldissera, and F. Di Orio Giovanni and Marques. Vol. 450. Cham: Springer International Publishing, 2015, pp. 101–110. ISBN: 978-3-319-16766-4. doi: [10.1007/978-3-319-16766-4\\_11](https://doi.org/10.1007/978-3-319-16766-4_11) (cit. on pp. 17, 18).
- [25] The Foundation for Intelligent Physical Agents. *FIPA Contract Net Interaction Protocol Specification*. URL: <http://www.fipa.org/specs/fipa00029/SC00029H.pdf> (visited on 2024-08-09) (cit. on p. 28).
- [26] The Foundation for Intelligent Physical Agents. *FIPA Request Interaction Protocol Specification*. URL: <http://www.fipa.org/specs/fipa00026/SC00026H.pdf> (visited on 2024-08-10) (cit. on p. 30).
- [27] The Foundation for Intelligent Physical Agents. *FIPA ACL Message Structure Specification*. URL: <http://www.fipa.org/specs/fipa00061/SC00061G.pdf> (visited on 2024-08-14) (cit. on p. 46).



# Marketplace-Driven Framework for the Dynamic Deployment and Integration of Modular Industrial Cyber-Physical Systems

This research project aims to develop a marketplace-driven framework for the dynamic deployment and integration of modular industrial cyber-physical systems (CPS). The proposed framework will enable the efficient and flexible deployment of CPS components across various industrial domains, such as manufacturing, energy, and transportation. The key features of the framework include:

- Modularization:** The system will be designed as a collection of modular components, allowing for easy integration and deployment.
- Dynamic Deployment:** The framework will support the dynamic addition and removal of components, enabling the system to adapt to changing requirements.
- Integration:** The framework will facilitate the integration of different CPS components, ensuring seamless communication and collaboration.
- Marketplace-Driven:** The framework will be based on a marketplace model, allowing users to discover and purchase components from a wide range of providers.

The proposed framework will be evaluated through extensive simulations and real-world experiments, demonstrating its effectiveness in supporting the dynamic deployment and integration of modular industrial CPS.

Project Lead: David Ferreira  
Project Number: 2024-IND-001  
Funding Source: National Science Foundation  
Start Date: October 1, 2024  
End Date: September 30, 2027