NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

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

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

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

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.10) [1].

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

List of Figures

List of Tables

Acronyms

AGV	Automated Guided Vehicle *(pp. 4, 24)*
API	Application Programming Interface *(p. 8)*
CA	Component Agent *(p. 17)*
COM/DCOM	Microsoft Windows Distributed Component Object Model *(p. 10)*
CPPS	Cyber-Physical Production System *(pp. 1, 2, 4–7, 11–15, 17, 23, 52)*
CPS	Cyber-Physical System *(pp. 1, 4, 5, 52)*
DA	Deployment Agent *(pp. ix, x, 15, 16, 24, 25, 30, 33, 36–38, 41, 42, 45, 46, 49, 50)*
DF	Directory Facilitator *(pp. ix, 25, 26, 28, 30, 31, 33, 38, 39, 46, 47, 49)*
DWPS	Device Profile Web Services *(pp. 15, 17)*
FIPA	Foundation for Intelligent Physical Agents *(pp. 2, 10, 12, 16, 26, 28, 33, 42)*
GUI	Graphical User Interface *(pp. x, 24, 25, 36, 37, 41, 42, 45)*
HTTP	Hypertext Transfer Protocol *(pp. 40, 44, 50, 51)*
IntA	InterfaceAgent *(p. 13)*
JADE	Java Agent DEvelopment Framework *(pp. 10, 12, 16, 17, 33, 34, 36, 37, 39)*
MAS	Multi-Agent System *(pp. ix, x, 2, 4, 6–8, 10–12, 14–16, 19–24, 28, 30, 33, 36–40, 43, 45–50, 52)*
MQTT	Message Queuing Telemetry Transport *(pp. 44, 50)*
OEE	Overall Equipment Effectiveness *(p. 5)*
OPC Classic	Open Platform Communications Classic *(p. 10)*

OPC UA Open Platform Communications Unified Architecture *(pp. 7, 10, 11, 13, 14, 44, 51)*

PA Product Agent *(pp. ix, 12, 15, 16, 24–31, 33–35, 37, 45, 47)*
PLC Programmable Logic Controller *(pp. ix, 7, 10, 11, 14, 15, 17)*
PM Product Manager *(pp. ix, x, 24–26, 30, 33, 38, 41, 42, 45–47, 50)*
PMA Production Management Agent *(pp. 16, 17)*
PSA Prime System Agent *(p. 16)*

QoS Quality of Service *(p. 44)*

RA Resource Agent *(pp. ix, 12, 13, 15, 16, 24–31, 33, 35–37, 41, 45–47, 49, 50)*

SMA Skill Management Agent *(p. 17)*

TA Transport Agent *(pp. ix, 15, 16, 24–31, 33, 36, 37, 41, 45–47, 49, 50)*
TCP-IP Transfer Control Protocol - Internet Protocol *(p. 11)*

UDP User Datagram Protocol *(p. 11)*

WS4D-JMEDS Web Services 4 Devices - Java Multi Edition DPWS Stack *(p. 15)*

XML eXtensible Markup Language *(pp. 12, 37, 39, 40)*

1 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 CPS capable of operating in a production line. A Cyber-Physical System 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 costumer 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.

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.

1.2 Problem and Solution

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 and because they are generic, they can

be re-used 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.

2

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 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. As such, this system works in a loop, constantly exchanging data and instructions 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 hardware like robot arms, AGVs, conveyor belts and 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.

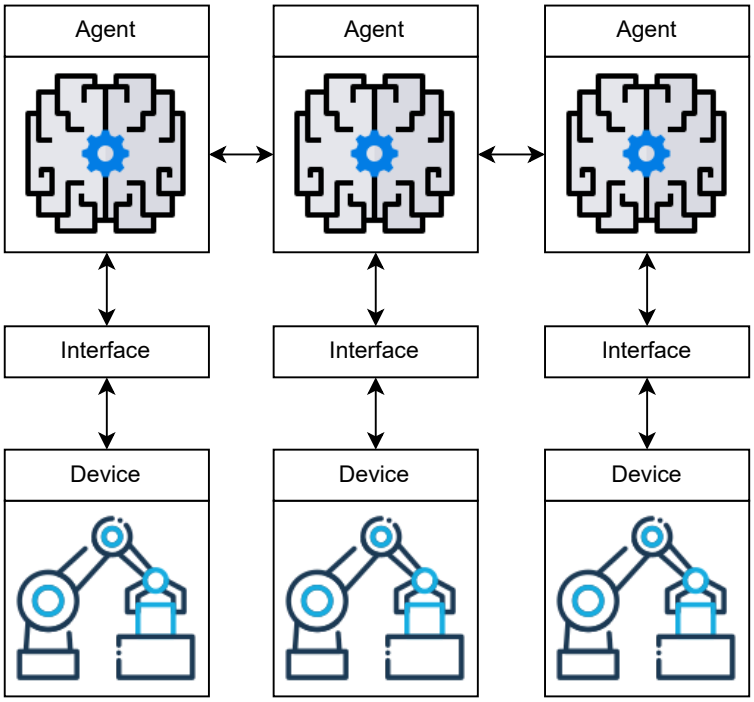


Figure 2.1: Industrial Multi-Agent System

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].

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.

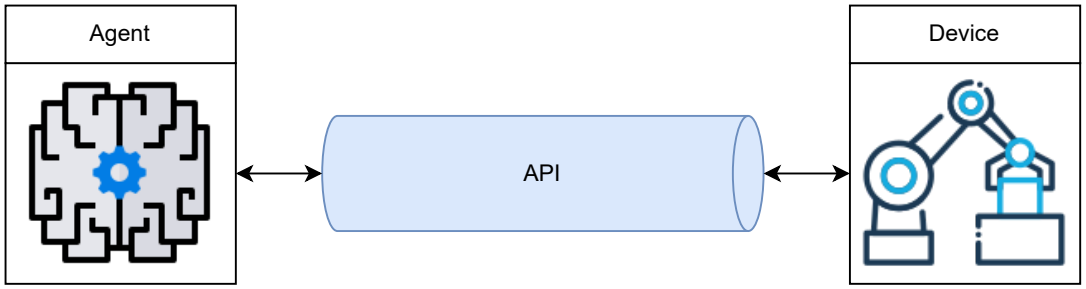


Figure 2.2: Tightly coupled Hybrid interface. Source: Adapted from [15]

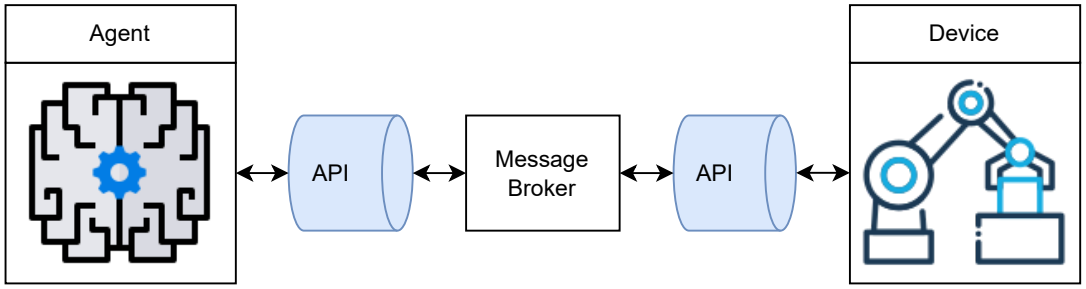


Figure 2.3: Loosely coupled Hybrid interface. Source: Adapted from [15]

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].

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].

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

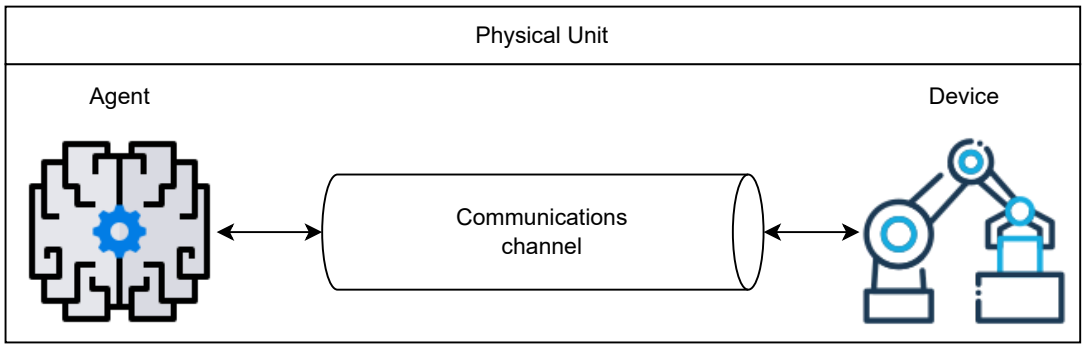


Figure 2.4: Tightly coupled On-device interface. Source: Adapted from [15]

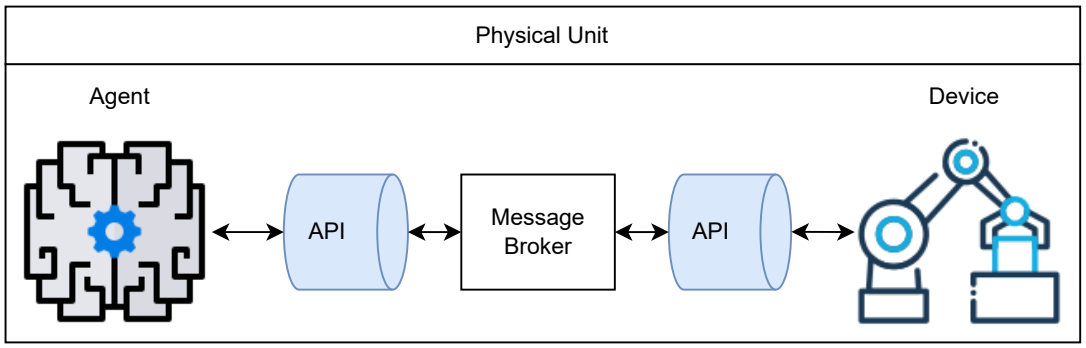


Figure 2.5: Loosely coupled On-device interface. Source: Adapted from [15]

the whole unit is dependent on the performance of the broker [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 it through the use of already made libraries. 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.

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].

2.3 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 [21] 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 [21], 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 InterfaceAgentConfig 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.

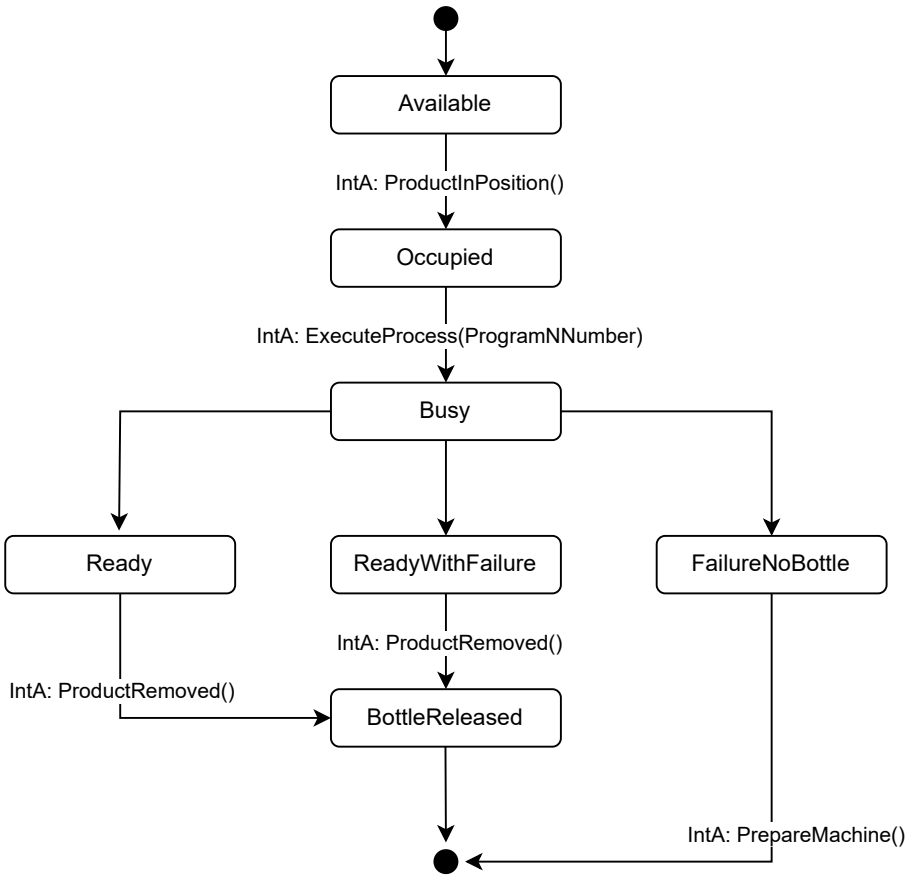


Figure 2.6: Interactions between PLCLinkAgent and PLC. Source: Adapted from [8]

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. [22] 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.

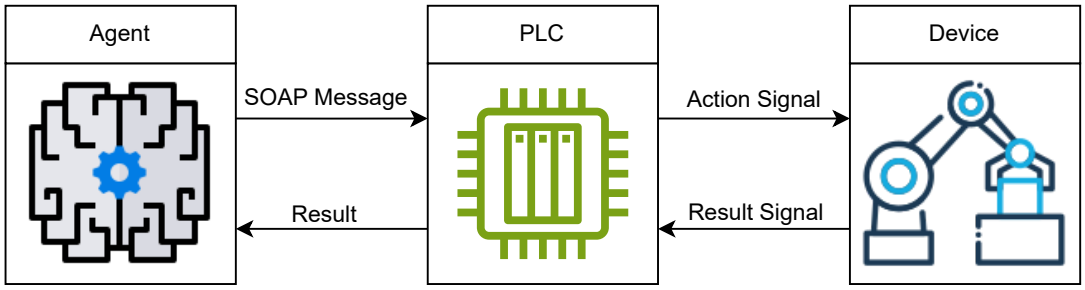


Figure 2.7: Plug and Produce RA. Source: Adapted from [22]

For the interface between hardware and agent, PLCs were chosen. These PLC 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 [23] 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 [23].

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 [23].

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 [23]. This last option is not flexible and uses a case-by-case configuration, making it the least desirable in a system.

3

Architecture

In this section, the developed architecture is explained. Starting with an overview of the system, we will take a look at the main architecture, followed by a few use cases. 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. The evolution over time is also shown, to explain how the system might change and what kind of processes it undergoes. 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 system proposed in this work consists of two main components, a Module Engine and the Link Libraries. The Module Engine operates between the layers of software, in this case the agent, and hardware. It allows the agent to interface with any kind of hardware through the use of Link Libraries.

These libraries are what actually communicates to the hardware below, and can 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. In Figure 3.1, we can see the four main layers of the system. As we can observe, it matters not what kind of agent or hardware the agent is using, with the Module Engine, it must be capable of interfacing with it.

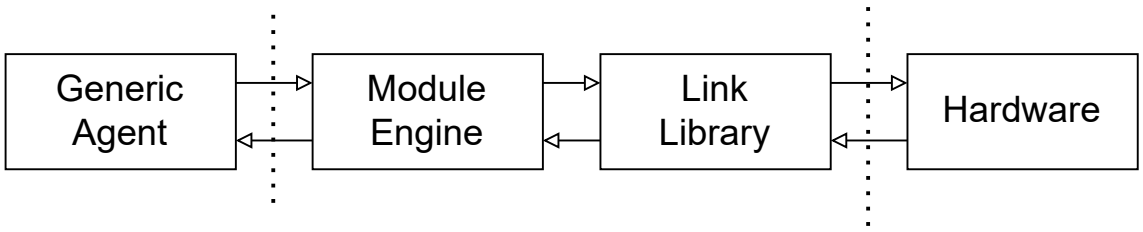


Figure 3.1: Simplified system architecture.

3.2 Use Cases

This system has three main use cases. 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 use case 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 is already developed. If not, then they need to create it. The creation of new Link Libraries will be explained below, in another use case.

The agent will be able to call the Module Engine at any point during its operation, whenever it need to communicate something to the hardware. Figure 3.2 shows a diagram representing this use case. As we can see, the development of the agent is unrelated to the hardware. 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. 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 use case can be seen in Figure 3.3.

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 more flexible than other hardware

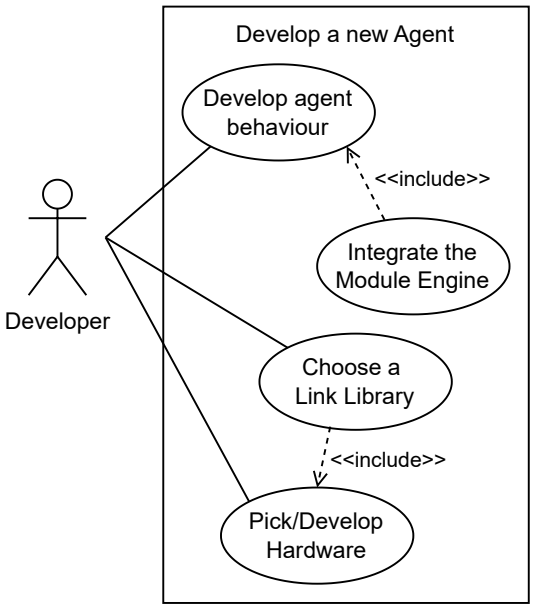


Figure 3.2: New agent development use case.

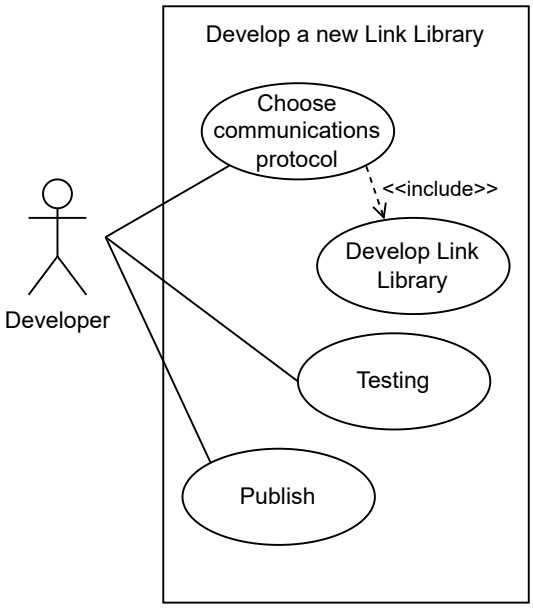


Figure 3.3: New agent development use case.

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 3.4.

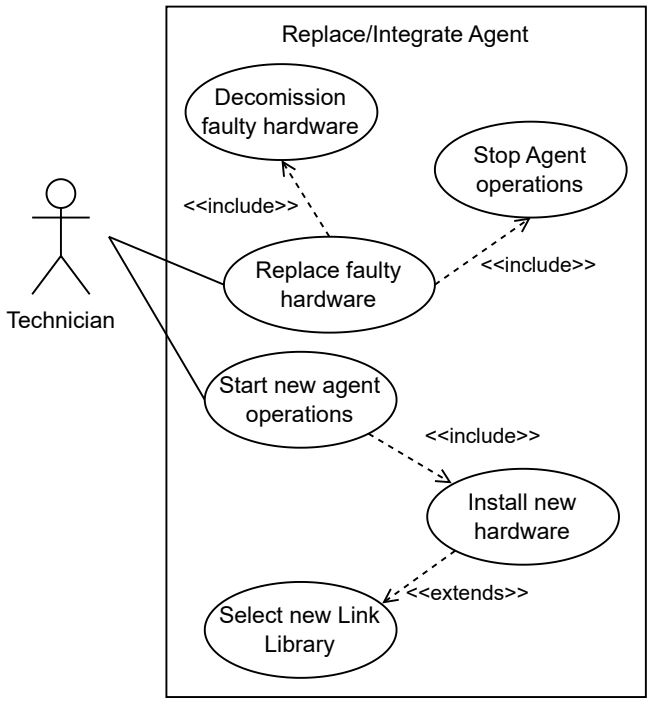


Figure 3.4: Agent replacement/integration use case.

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 3.5, we can also see a Deployment Entity. This component is part of the overall MAS and is detached from the agent. It is used to launch the agent and also provide it with the initial parameters. These parameters allow the Module Engine to select which Link Library is to be used and also give the Link Library its configurations needed to establish a connection to the hardware.

3.4 Module Engine Operations

The Module Engine is instantiated by the agent it is under. As explained before, the Module Engine needs three main components to operate. It needs the marketplace file, where all available Link Libraries are listed. From this list, the Module Engine picks the one to use based on the library type it also receives. Finally, after loading in the correct

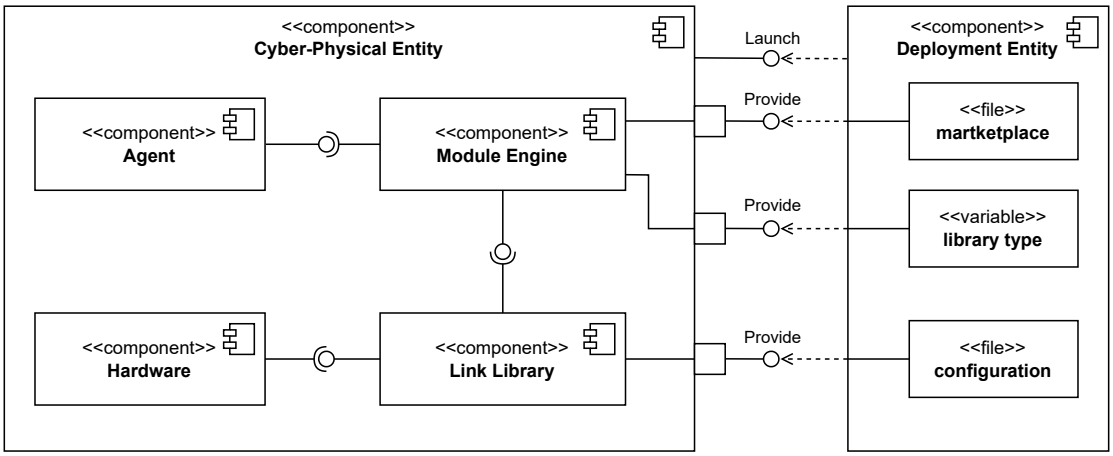


Figure 3.5: Component diagram of a cyber-physical entity using the Module Engine.

Link Library, it provides it with the configurations file.

The Link Library will 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.

Once communication with the hardware is established, the Module Engine will wait until a new command arrives from the agent. Once it does, it 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 3.6.

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

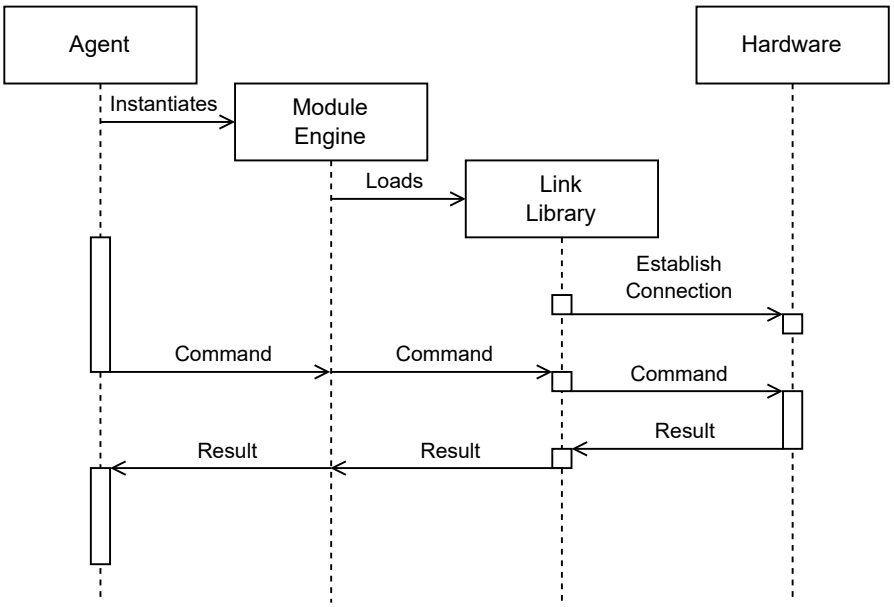


Figure 3.6: Sequence diagram of Module Engine operations.

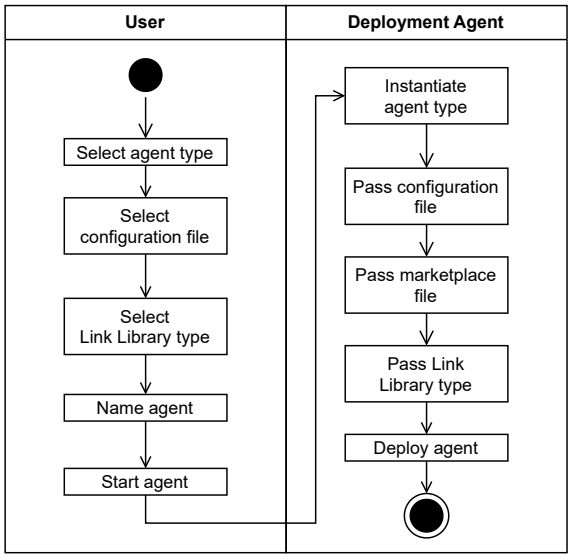
counterpart and therefore do not use the Module Engine.

The developed entities are:

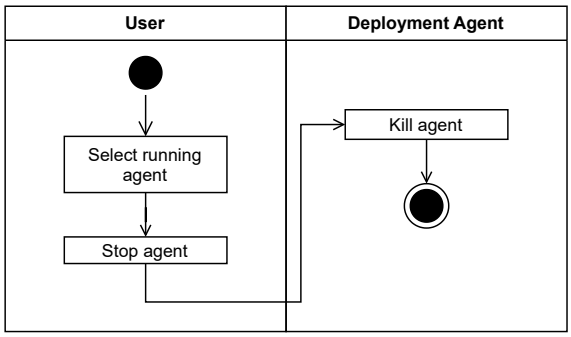
- The Resource Agent (RA), that represents any kind of physical component capable of performing processes, like a robotic arm or a bottle filling station;
- The Transport Agent (TA), that represent any kind of physical component capable of transporting products from one location on the shop floor to 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 3.5;
- 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 3.7

presents the process of deploying a new agent (3.7(a)) and stopping a pre-existing agent (3.7(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.



(a) Deployment of new agent.



(b) Stopping of running agent.

Figure 3.7: Deployment Agent activity diagram.

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 3.8 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

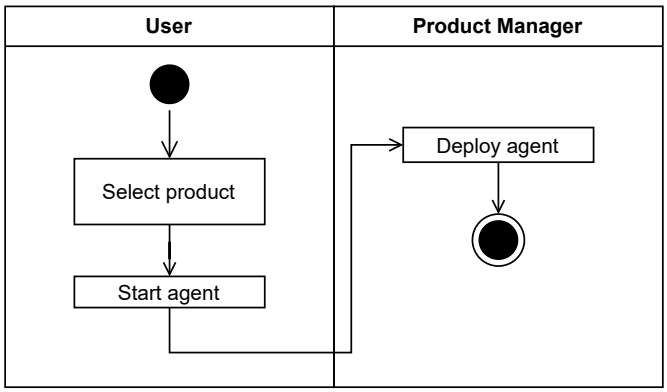


Figure 3.8: Product Manager activity diagram.

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 3.9.

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

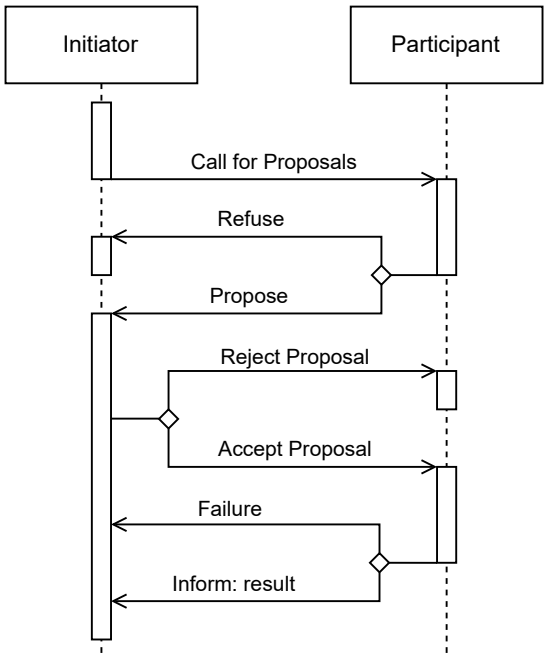


Figure 3.9: Contract Net interaction protocol. Adapted from [24].

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 3.10. 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.

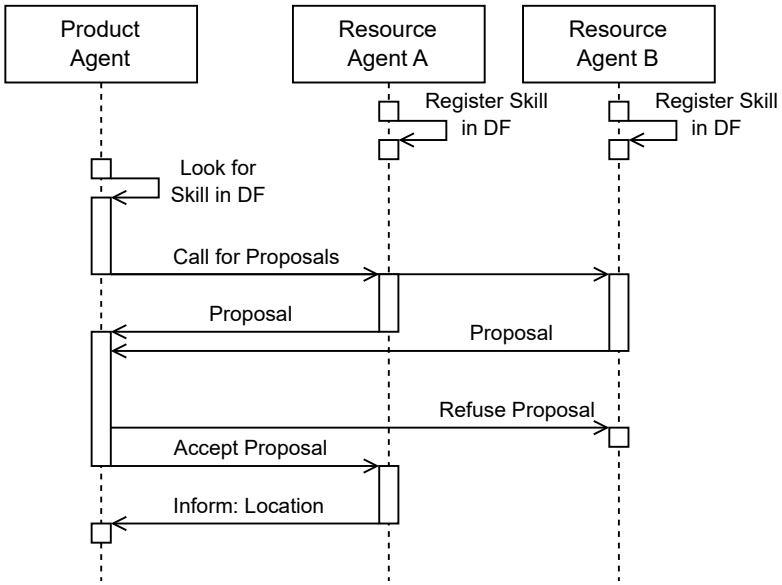


Figure 3.10: Contract Net protocol between a Product Agent and two Resource Agents.

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 3.11.

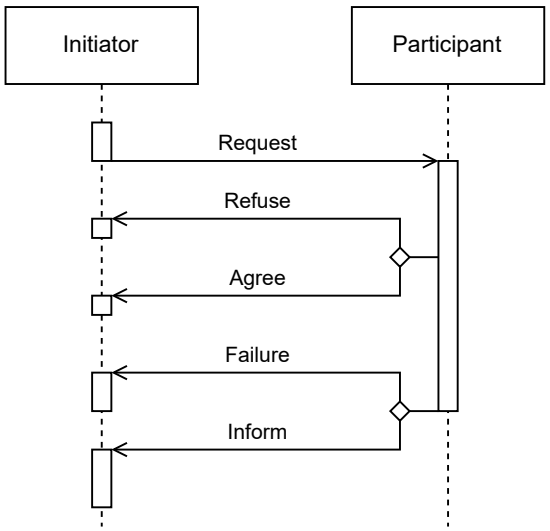
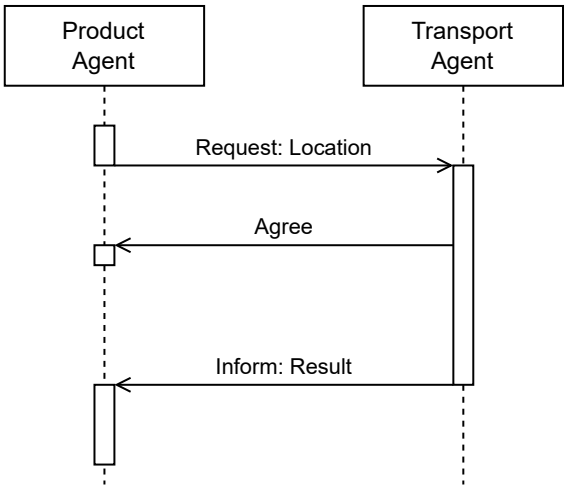


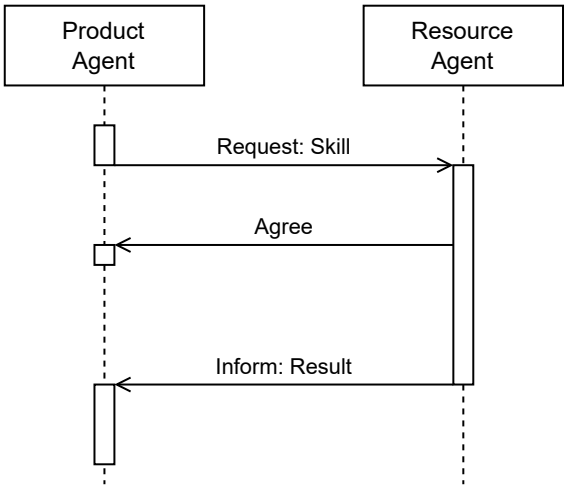
Figure 3.11: Request interaction protocol. Adapted from [25].

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 3.12. In 3.12(a) we can see the Requests protocol between a PA and TA, and in 3.12(b) between a PA and RA.

When the RA finishes it 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 3.13. It only shows the full interactions between agents, their actions and decisions without the Module Engine.



(a) Product Agent and Transport Agent.



(b) Product Agent and Resource Agent.

Figure 3.12: Requests between agents.

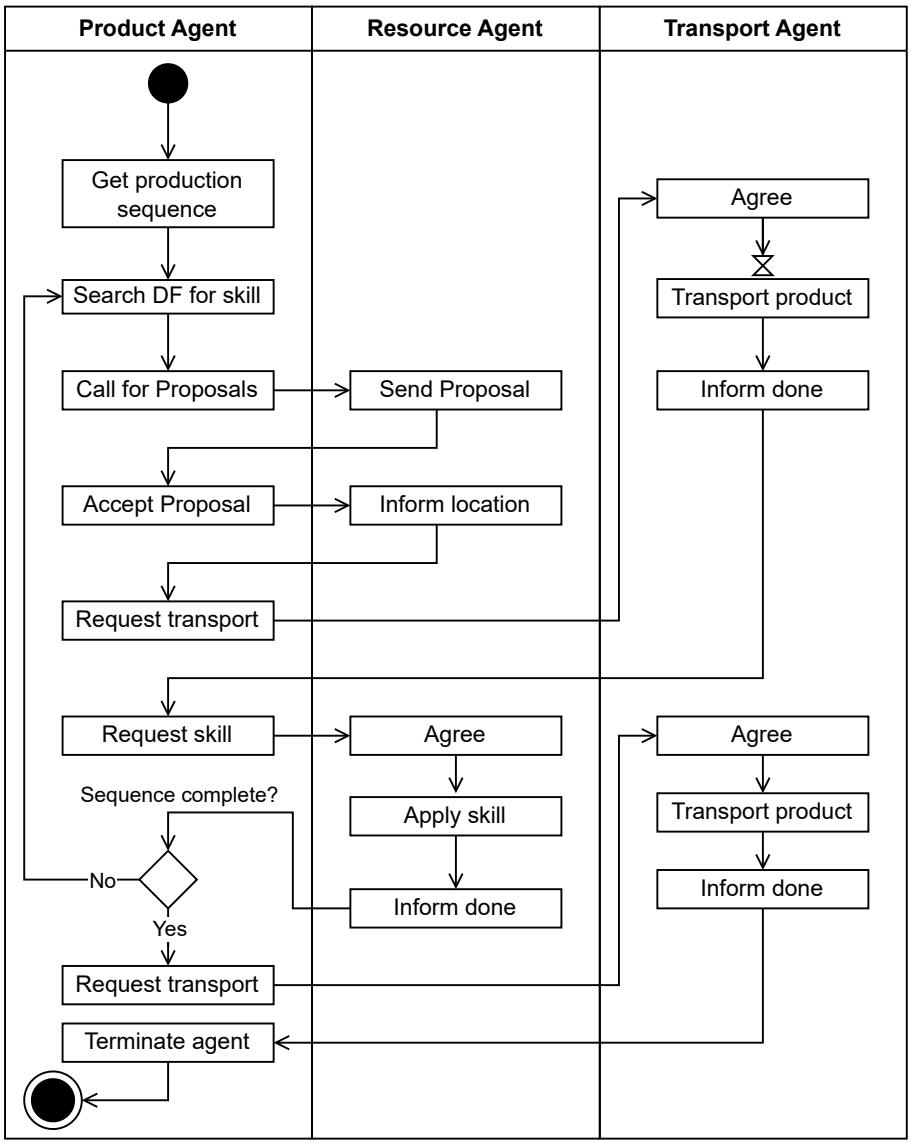


Figure 3.13: Activity diagram of the Industrial Multi-Agent System.

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 send 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 3.14, this whole process is shown, from start to finish.

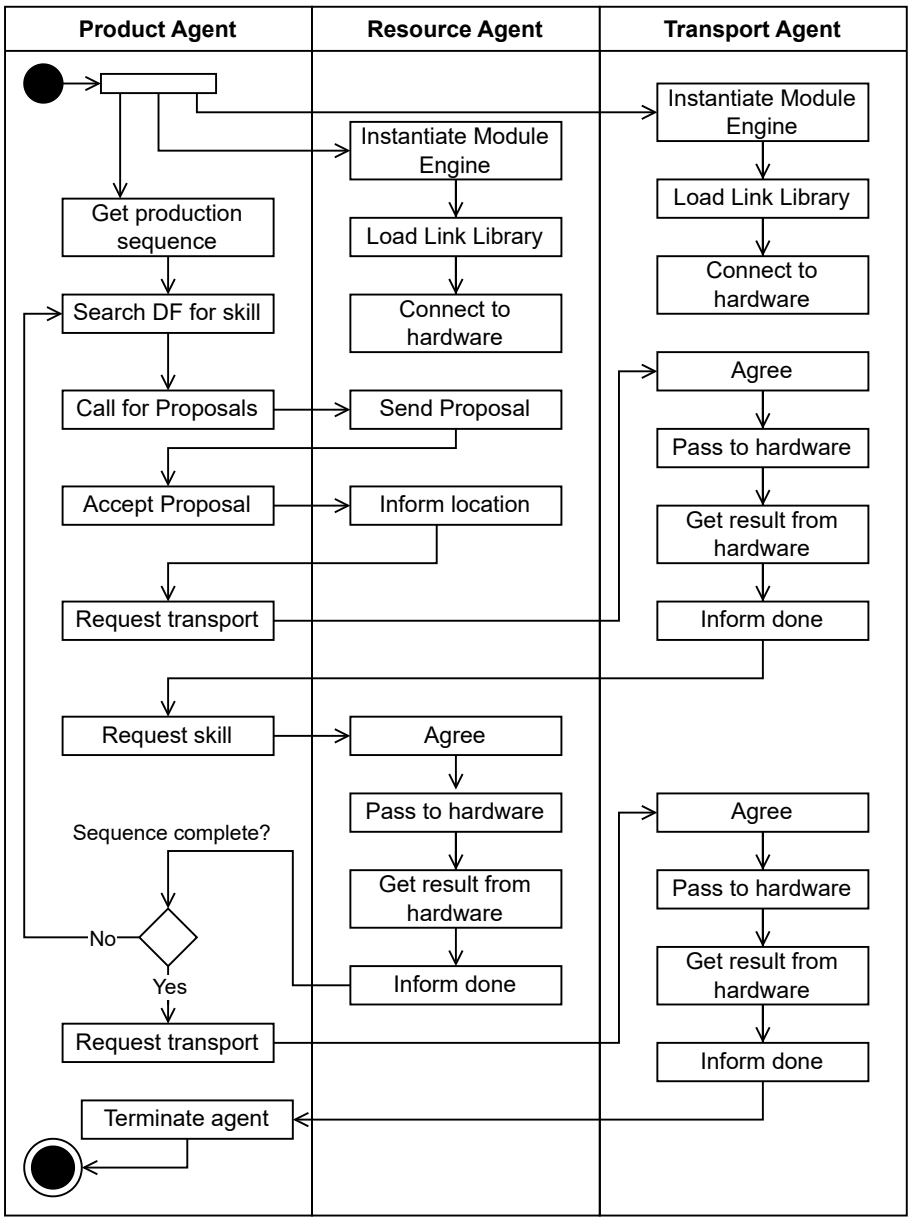


Figure 3.14: Activity diagram of the final system.

4

Implementation

Now that the architecture of the Module Engine and the accompanying MAS has been laid out and well defined, it is time to implement it. First we will see exactly how each class was implemented, their data models and methods. Then we will look at how the different module 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 Class Implementations

For the implementation of the whole 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 this language was going to be used at length 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 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 developed 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. Other 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 anoter 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.

4.1.1 Product Agent Class

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 that are explained in 4.1.6 and 4.1.7, respectively.

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 "AchieveREInitator" class. These classes and methods are all represented in Figure 4.1, along with the variables they use.

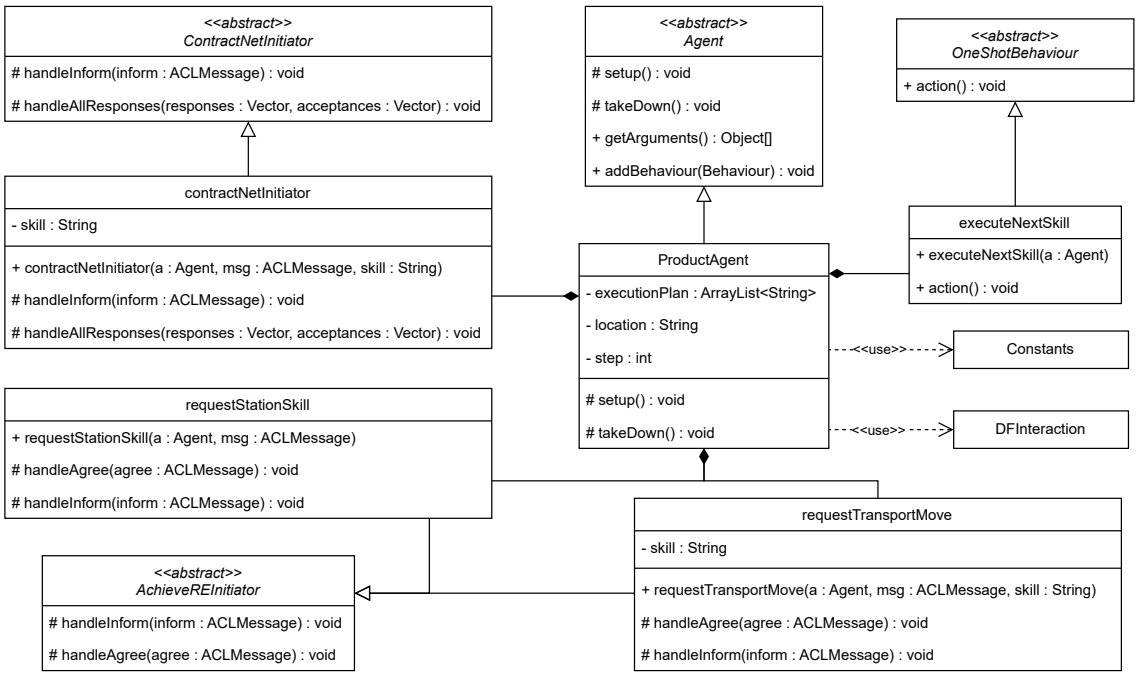


Figure 4.1: Product Agent class diagram.

4.1.2 Resource Agent Class

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

Like the PA, it also makes use of the auxiliary "Constants" and "DFInteraction" classes. It has two objects of type File called "xmlConfigFile" and "xmlMarketplaceFile that store the corresponding files. The String "libType" 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 is using, and the ArrayList of Strings "associatedSkills" holds the list of all skills this agent is able to perform.

The "requestResponder" class extends the "AchieveREResponder" class and the "contractNetResponder" class extends the "ContractNetResponder" class. 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 4.1.8 in more detail.

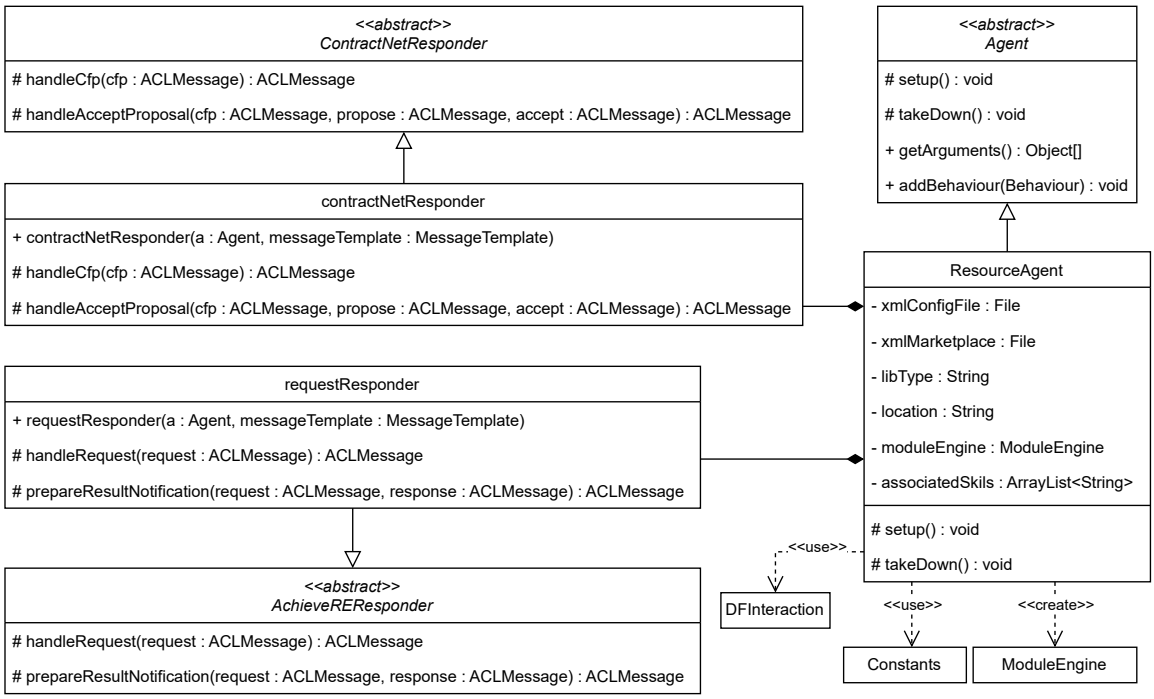


Figure 4.2: Resource Agent class diagram.

4.1.3 Transport Agent Class

The Transport Agent class is very similar to the Resource Agent. The only differences are in variables it uses and the classes it implements. It holds the same variables except for the "location", since the Transport Agent does not have a location in the physical system. It contains both "setup" and "takeDown" methods. This class also makes use of the "Constants" and "DFInteraction" classes.

It implements a single "requestResponder" class which extends the "AchieveREResponder" class provided by JADE. Figure 4.3 has a representation of its implementation. Once again the classes related to the Module Engine have been omitted.

4.1.4 Deployment Agent Class

The Deployment Agent class 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.

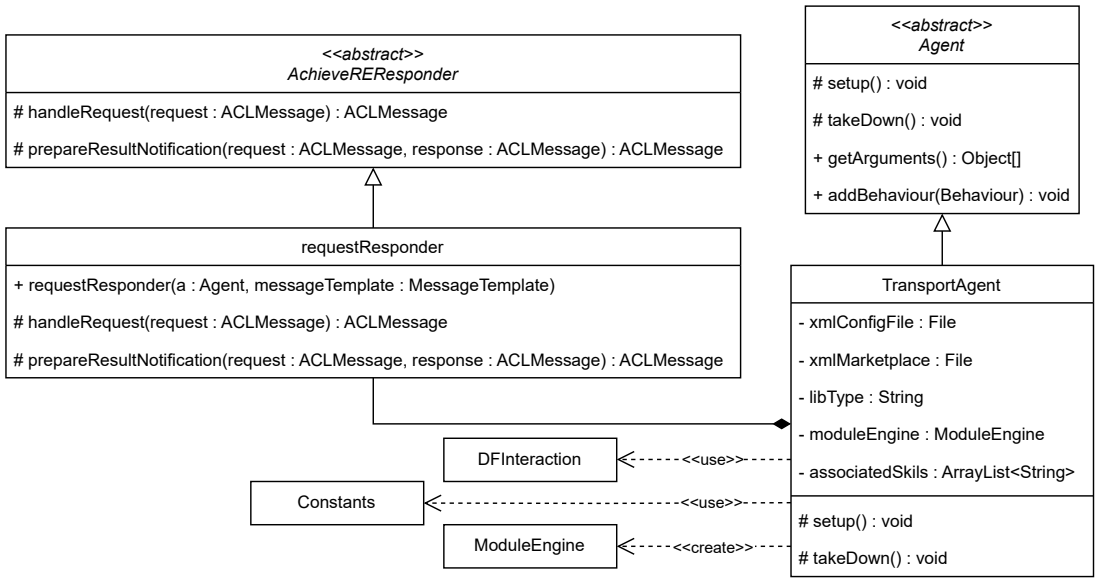


Figure 4.3: Transport Agent class diagram.

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 4.4, 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 is defined in 4.1.6.

4.1.5 Product Manager Class

This class 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.

In Figure 4.5, the classes that it uses to draw the GUI are shown, once again the "ContainerController" class 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.

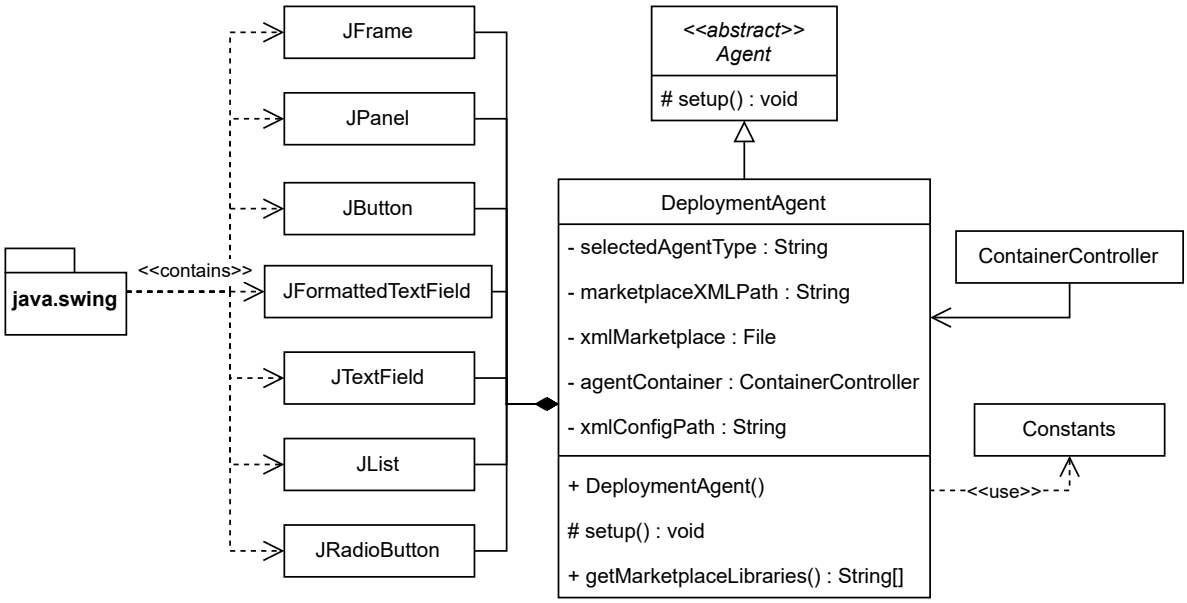


Figure 4.4: Deployment Agent class diagram.

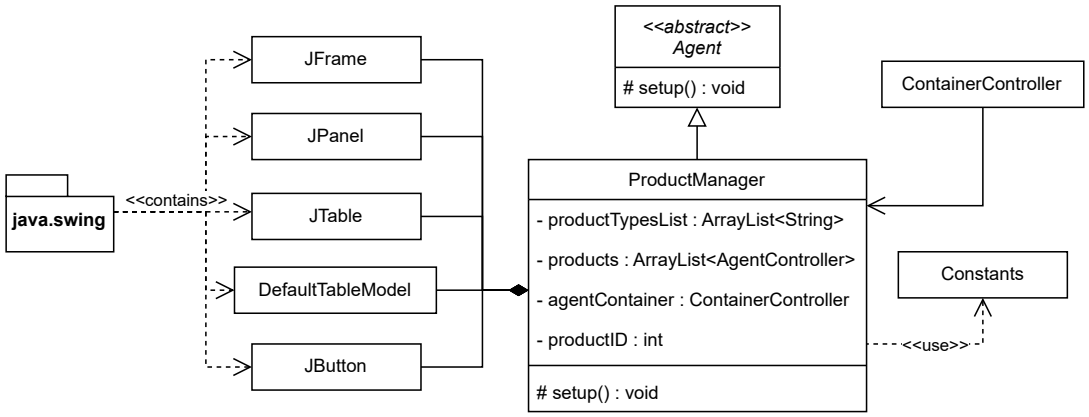


Figure 4.5: Product Manager class diagram.

4.1.6 Constants Class

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 4.6.

4.1.7 Directory Facilitator Class

The Directory Facilitator class is another auxiliary class. It provides methods that can register, remove or search information on the DF. Figure 4.7 shows its diagram. The method "RegisterInDF" has two version, one allows for the registry of an agent with a single skill,

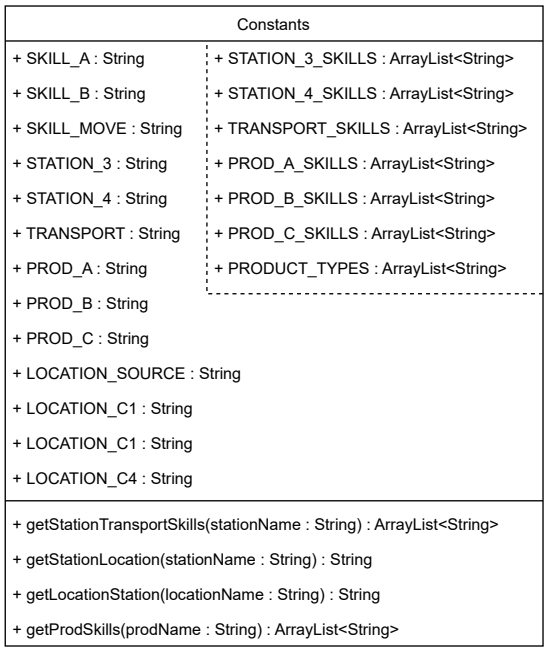


Figure 4.6: Constants class diagram

the other for an agent with multiple skills. The "DFService" class is implemented by JADE and it is what allows for the interactions with the DF.

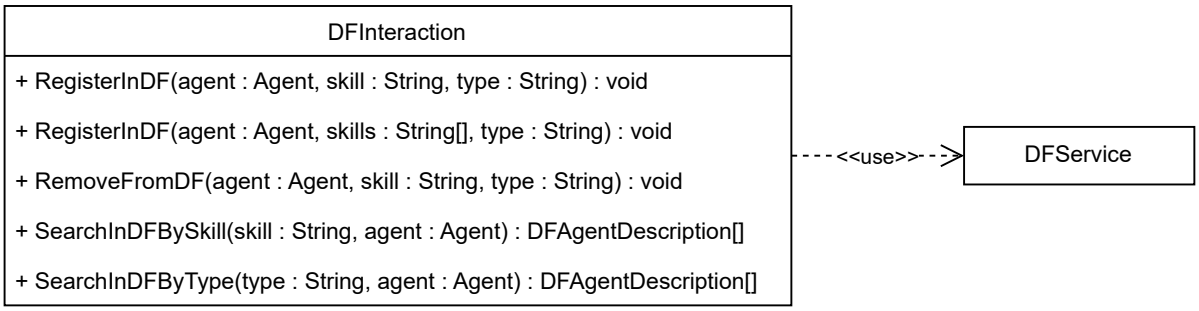


Figure 4.7: Directory Facilitator class diagram

4.1.8 Module Engine Class

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 hold the currently loaded Link Library. It also has a "classesToLoad" Hashmap, which holds all the available Link Libraries, for quick access. This Hashmap is loaded with the method "parseMarketplaceXML".

The method "parseMarketplaceXML" does exactly that, it parses the file and loads the Link Libraries in the "classesToLoad" hashmap. This marketplace file is in the XML format, and holds both the name of the Link Library and the class file that implements

it. These are loaded 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.

The method "createObject" 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, also received from the agent.

To execute a skill the method "executeSkill" is called, 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 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.

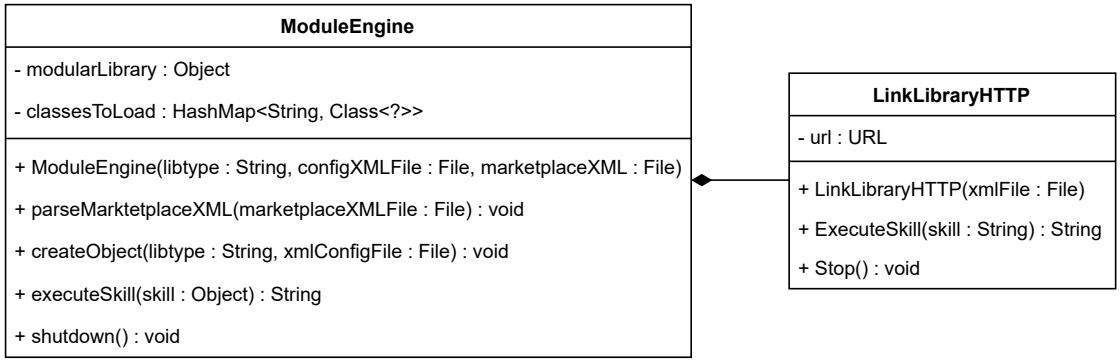


Figure 4.8: Module Engine class diagram

4.1.9 Link Library Classes

To showcase the functionalities of the Module Engine, three different Link Libraries were developed. An abstract class "LinkLibrary" was extended to implement the "LinkLibrary-HTTP", "LinkLibraryMQTT" and "LinkLibraryOPCUA" classes.

The "LinkLibraryHTTP" was implemented with the classes and methods provided in the "java.net" package. The "LinkLibraryMQTT" used the "org.eclipse.paho.client.mqttv3" package to implement its protocol and the "LinkLibraryOPCUA" used the "org.eclipse.milo.opcua" package.

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.

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

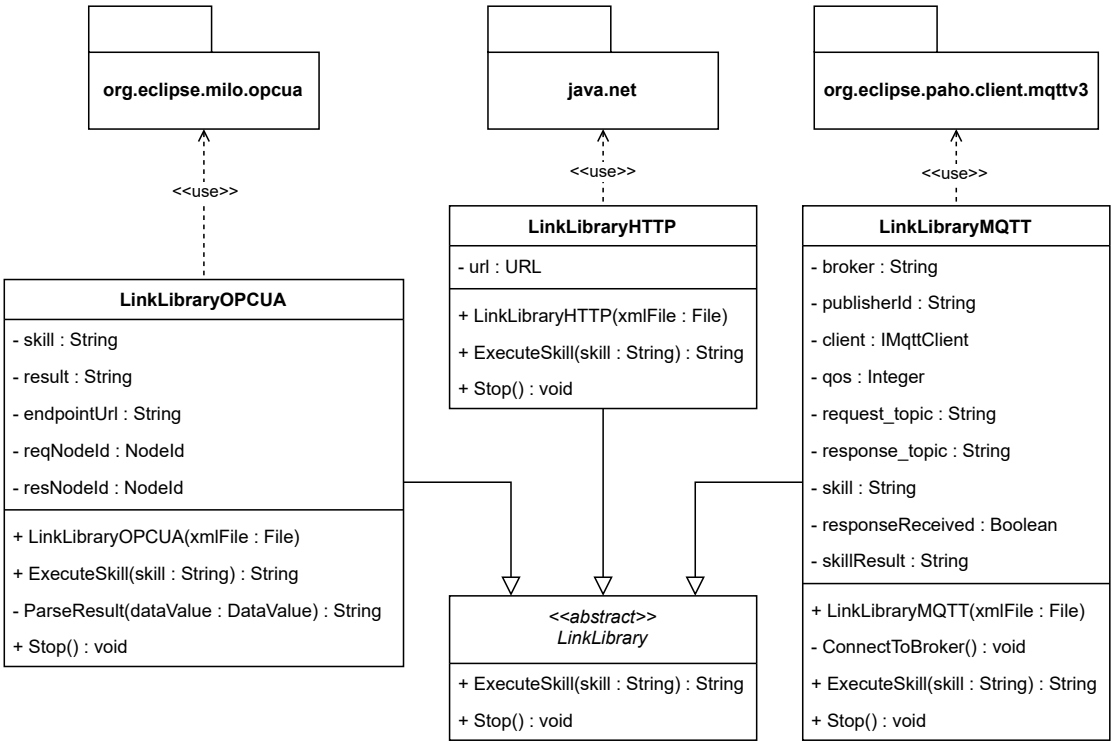


Figure 4.9: Link Library class diagrams

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.

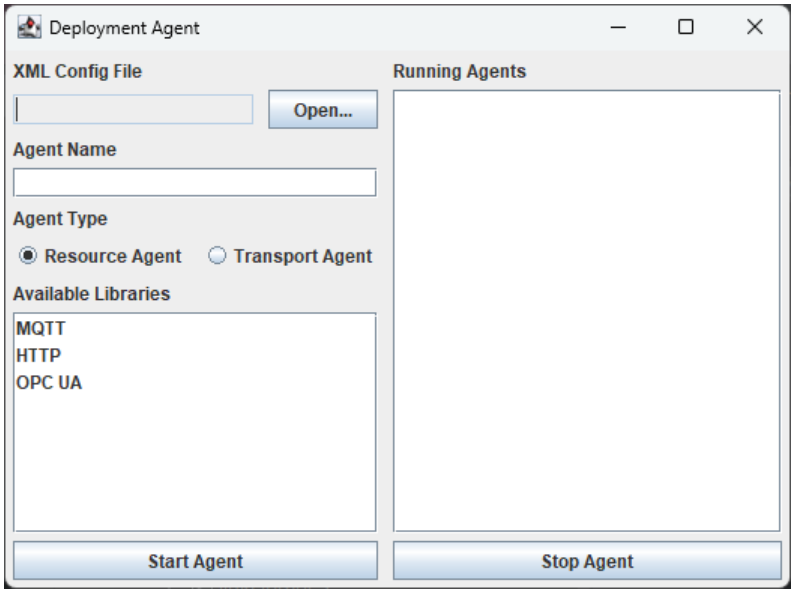


Figure 4.10: Deployment Agent Graphical User Interface.

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.

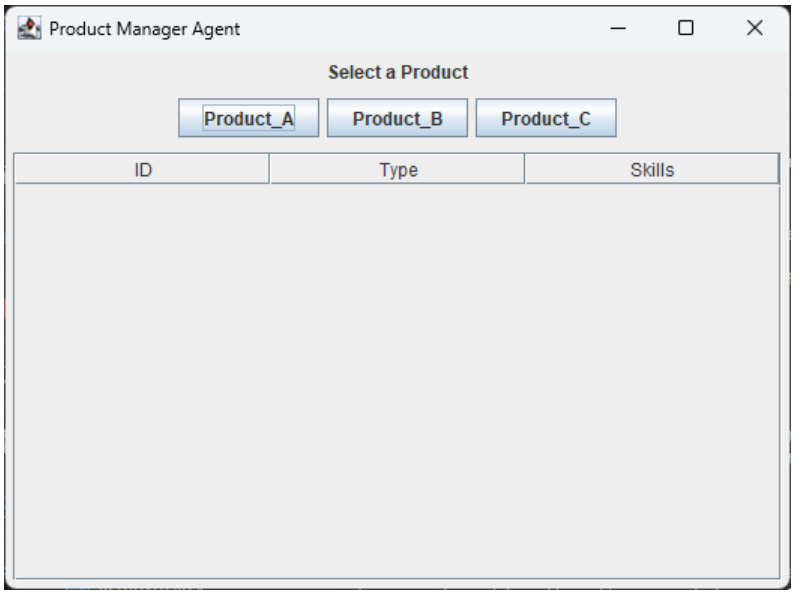


Figure 4.11: Product Manager Graphical User Interface.

4.2.2 Agent to Agent

Agent to agent communication is done through ACLMessages. These have been defined by FIPA [26]. 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

	Protocol	
Parameter	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 4.1.8.

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 4.1.9. 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 and 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.

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 it 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.

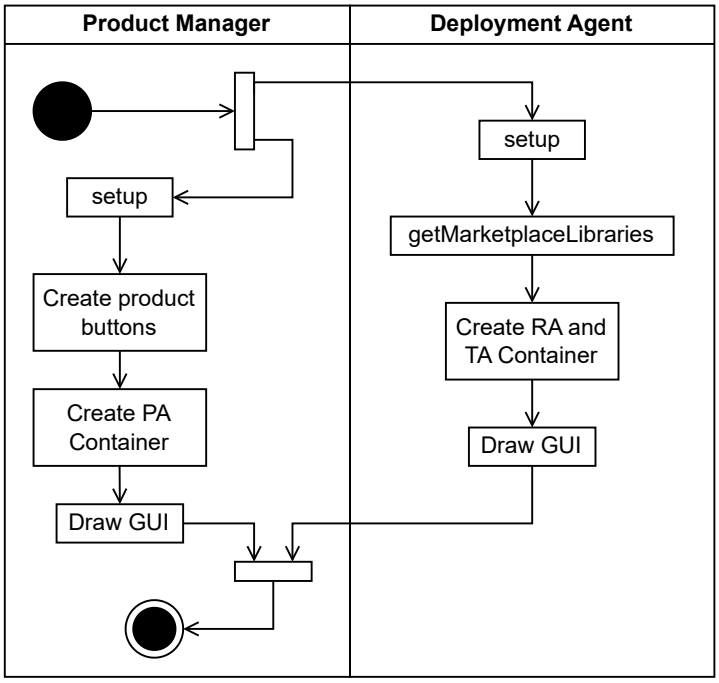


Figure 4.12: Multi-Agent System startup.

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 it name and file path added to the marketplace file. Upon selecting a Link Library,

the corresponding configuration file 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 agents 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.

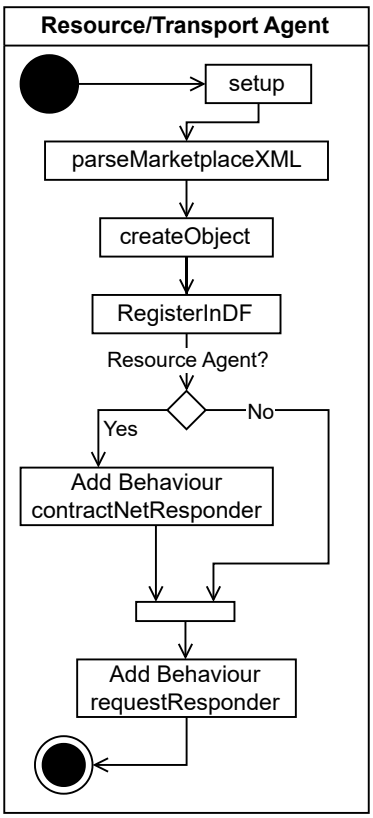


Figure 4.13: Agent startup

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 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 4.14 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.

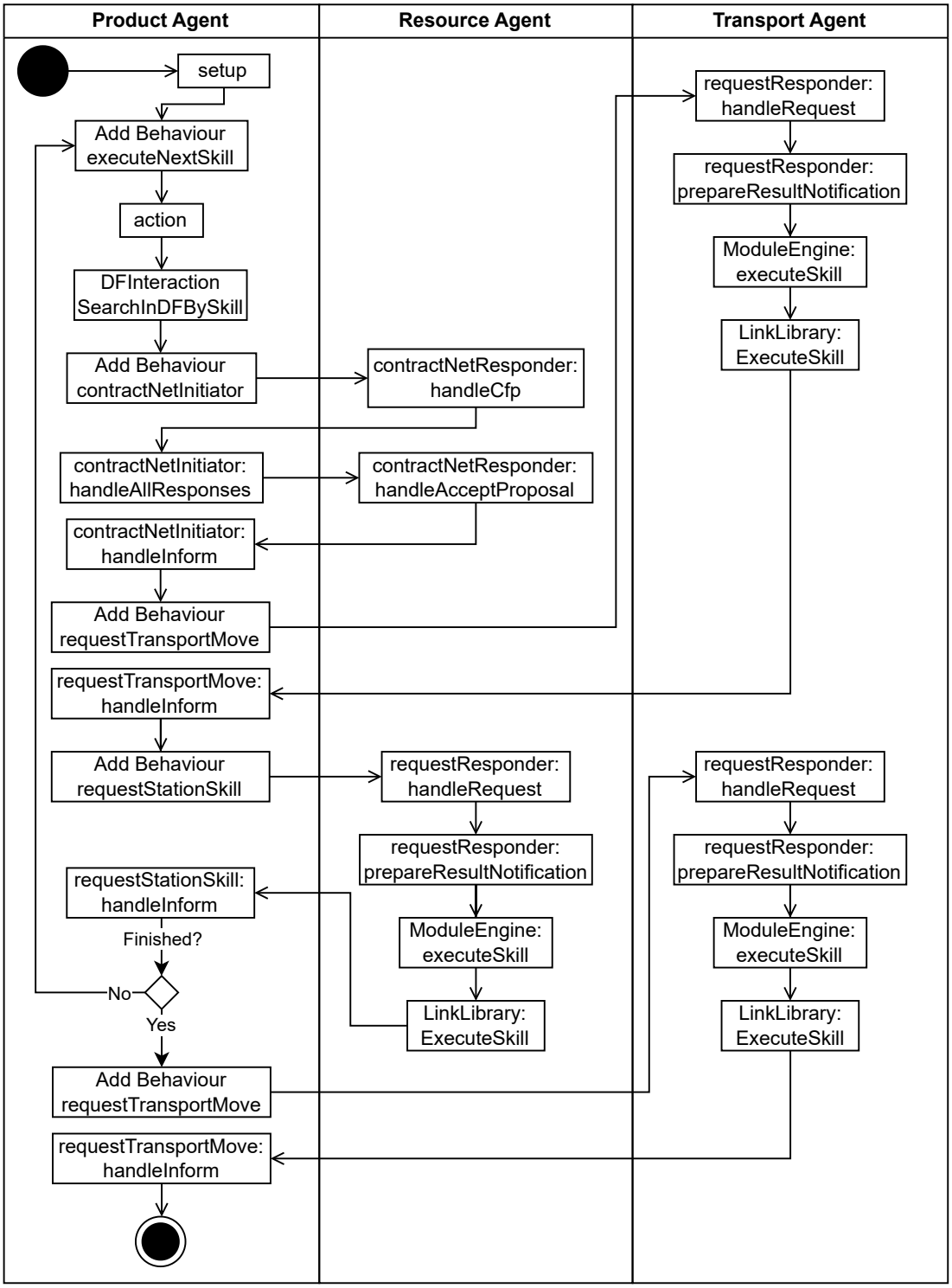


Figure 4.14: Multi-Agent System operations

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 4.15.

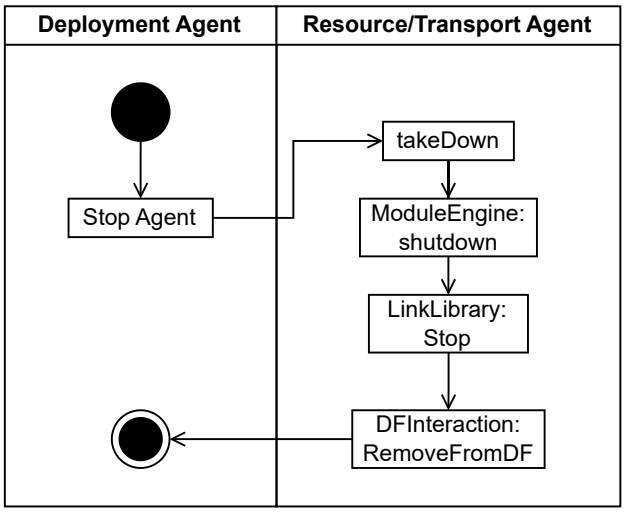


Figure 4.15: Agent termination

5

Tests

To test the system, both NodeRED and the MAS systems were launched. The Product Manager and Deployment Agent startup and wait for agents to be launched. After this, two Resource Agents are launched, corresponding to Station 3 and Station 4, aptly named Station_3 and Station_4. Along with those agents, a Transport Agent is also launched, corresponding to the Conveyor Belt, called Conveyor. The right configuration files are selected for each agent and they are launched, one by one. To start off, 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 described in Chapter 4.

After agents are done with their initial setups, a product agent can be launched. For the purposes of showcasing the systems capabilities, three different Product Types were created. These products are shown in 5.1, along with their production sequences and the sequence of Conveyor Belt sections they need to go to in order to complete their process.

Table 5.1: 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. The system handled their production without any problems, with all RAs and TA communicating to the hardware through the MQTT protocol.

After the last product finished its process, the Resource Agent for Station 3 was stopped and restarted with the HTTP Link Library. The same test was run again, and no significant changes to the production steps were noted, even though Station 3 was now communicating through HTTP Requests to the hardware. Similar tests were performed, where an

agent would be stopped and restarted again, with a different Link Library.

The only changes in processing time were noticed when the OPC UA protocol was used, but this is likely due to the server being hosted on the RevPi itself, which could slow down its operation speed somewhat, since multiple processes were running in parallel. This could've easily been mitigated by hosting the OPC UA server on another machine.

To ascertain if there was a difference in processing time in a system using the Module Engine and a system communicating directly to the hardware, a simple test was done. A dummy agent was created for this and in its setup code a simple loop was implemented that would send two HTTP POSTs. The first through the Module Engine and the other directly. This loop ran for a hundred thousand iterations and the time each communication took was taken, and the average of the differences between times was calculated. All average times measured were under a millisecond, which mean that, on average, the system using the Module Engine takes about the same amount of time as system communicating directly to the hardware.

This result was to be expected, since the Link Library used by the Module Engine is loaded like any other Java class and treated like any other Java object. This makes it behave like it was there since the projects compilation, even though it wasn't, which should not impact the time performance significantly, as the test demonstrates.

6

Conclusion

The concept of Industry 4.0 revolutionized the manufacturing sector through the digitalization of manufacturing processes. Industrial CPS or CPPSs 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 MAS were suggested as a model for the implementation of a CPPS 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.

There have been a lot of proposed architectures and methods of interfacing throughout the years to try and make MASs more accessible. In this work a new way of interfacing is proposed, enabling more seamless integration of new industrial agents into already existing hardware through the use of the developed Module Engine powered by the Reflections feature of the Java programming language. This allows for the creation of highly flexible libraries, that can integrate any king of hardware with an MAS. This in turn would help the integration of MASs in industrial settings, increasing their use by the manufacturing sector and consequently allowing for the development of even better technologies due to their practical use.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX 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 (cit. on pp. 1, 4).

[3] *Industry 4.0 Case Studies (curated)*. URL: https://amfg.ai/2019/03/28/industry-4-0-7-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 (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 (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 (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 (cit. on pp. 2, 11, 13, 14).

[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 (cit. on pp. 2, 11).

[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 (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 (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 (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 (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 (cit. on pp. 6, 8, 11).

[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 (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 (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 p. 10).

[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 (cit. on p. 11).

[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 (cit. on p. 11).

[21] 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 (cit. on pp. 11–13).

[22] 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 (cit. on p. 15).

[23] 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 (cit. on pp. 16–18).

[24] 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. 27).

[25] 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. 28).

[26] 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. 42).