

2023-07-21

# Multi-Agent Modelling of Industrial Cyber-Physical Systems for IEC 61499 Based Distributed Intelligent Automation

Lyu, Guolin

---

Lyu, G. (2023). Multi-agent modelling of industrial cyber-physical systems for IEC 61499 based distributed intelligent automation (Doctoral thesis, University of Calgary, Calgary, Canada).

Retrieved from <https://prism.ucalgary.ca>.

<https://hdl.handle.net/1880/116791>

*Downloaded from PRISM Repository, University of Calgary*

UNIVERSITY OF CALGARY

Multi-Agent Modelling of Industrial Cyber-Physical Systems for IEC 61499 Based Distributed  
Intelligent Automation

by

Guolin Lyu

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES  
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE  
DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN MECHANICAL AND MANUFACTURING ENGINEERING

CALGARY, ALBERTA

JULY, 2023

© Guolin Lyu 2023

## Abstract

Traditional industrial automation systems developed under IEC 61131-3 in centralized architectures are statically programmed with determined procedures to perform predefined tasks in structured environments. Major challenges are that these systems designed under traditional engineering techniques and running on legacy automation platforms are unable to automatically discover alternative solutions, flexibly coordinate reconfigurable modules, and actively deploy corresponding functions, to quickly respond to frequent changes and intelligently adapt to evolving requirements in dynamic environments.

The core objective of this research is to explore the design of multi-layer automation architectures to enable real-time adaptation at the device level and run-time intelligence throughout the whole system under a well-integrated modelling framework. Central to this goal is the research on the integration of multi-agent modelling and IEC 61499 function block modelling to form a new automation infrastructure for industrial cyber-physical systems. Multi-agent modelling uses autonomous and cooperative agents to achieve run-time intelligence in system design and module reconfiguration. IEC 61499 function block modelling applies object-oriented and event-driven function blocks to realize real-time adaption of automation logic and control algorithms. In this thesis, the design focuses on a two-layer self-manageable architecture modelling: a) the high-level cyber module designed as multi-agent computing model consisting of *Monitoring Agent*, *Analysis Agent*, *Self-Learning Agent*, *Planning Agent*, *Execution Agent*, and *Knowledge Agent*; and b) the low-level physical module designed as agent-embedded IEC 61499 function block model with *Self-Manageable Service Execution Agent*, *Self-Configuration Agent*, *Self-Healing Agent*, *Self-Optimization Agent*, and *Self-Protection Agent*. The design

results in a new computing module for high-level multi-agent based automation architectures and a new design pattern for low-level function block modelled control solutions.

The architecture modelling framework is demonstrated through various tests on the multi-agent simulation model developed in the agent modelling environment NetLogo and the experimental testbed designed on the Jetson Nano and Raspberry Pi platforms. The performance evaluation of regular execution time and adaptation time in two typical conditions for systems designed under three different architectures are also analyzed. The results demonstrate the ability of the proposed architecture to respond to major challenges in Industry 4.0.

## **Acknowledgements**

There are many people whom I would like to acknowledge for the support they provided in my pursuit of the PhD degree and in making this dissertation possible.

First and foremost, I would like to thank my supervisor, Dr. Robert Brennan, for his guidance, advice, patience, and support in my research project. Thanks very much for providing this valuable opportunity and putting the much appreciated effort in my research, especially for the numerous discussions, reviews, and revisions of research articles, and for the work on building simulation models. For their generous financial supports, I would like to thank the Natural Sciences and Engineering Research Council of Canada, Spartan Controls, the Suncor Energy Foundation, the Faculty of Graduate Studies, and the Department of Mechanical and Manufacturing Engineering.

I would like to thank Dr. Paul Tu and Dr. Simon Li for being my supervisory committee members, Dr. Peter Goldsmith and Dr. Laleh Behjat for serving on the candidate examination committee, Dr. Mahdis Bisheban, Dr. Arne Dankers, and Dr. Petr Kadera for serving on the final examination committee. Thanks very much for putting efforts in the exam and offering advice on the research. I would also like to thank Dr. Deyi Xue for the research work on compute-aided product modelling and Dr. Alireza Fazlirad for helping the initial multi-agent modelling experimental testbed design.

Many thanks go to Bethe and Johnny Andreasen for help and encouragement in returning to the PhD program in early 2018 with Dr. Brennan and for love and support to my family. I also appreciate all help from my friends, visiting scholars, research colleges, and support staff in my pursuit of the PhD degree at the University of Calgary.

Most importantly, my wife Angyue deserves special thanks for the love and support she provided which made all the difference in the past years. Thank our little girl Sophie for bringing so much joy to us since late 2020. I would also like to thank my parents, parents-in-law, and all family members for the constant love and support that they have provided.

Thank you all for being part of this journey and for the witness to every step.

## **Dedication**

To my grandparents, especially my maternal grandmother who passed away in 2021 while I was so far away from home.

## Table of Contents

Abstract .....	ii
Acknowledgements .....	iv
Dedication .....	vi
Table of Contents .....	vii
List of Tables .....	x
List of Figures .....	xi
List of Abbreviations .....	xiv
 CHAPTER ONE: INTRODUCTION .....	 1
1.1 Research Background .....	1
1.2 Thesis Structure .....	5
 CHAPTER TWO: RESEARCH MOTIVATION .....	 7
2.1 Problem Statement .....	7
2.2 Objectives and Methodologies .....	13
2.3 Anticipated Contributions .....	14
 CHAPTER THREE: LITERATURE REVIEW .....	 17
3.1 Introduction .....	17
3.2 Initiation of IEC 61499 Standards .....	19
3.2.1 IEC 61499 Theoretical Fundamentals .....	19
3.2.2 IEC 61499 Function Block Models .....	20
3.3 Execution of IEC 61499 Function Blocks .....	22
3.3.1 Execution Semantics of IEC 61499 Function Blocks .....	22
3.3.2 Semantic-Correct Mapping for IEC 61499 Function Blocks .....	23
3.4 Transition to IEC 61499 Based Systems .....	26
3.4.1 Challenges with Transition to IEC 61499 Based Systems .....	26
3.4.2 Methods of Transformation to IEC 61499 Based Systems .....	28
3.5 Integration with IEC 61499 Enabling Technologies .....	32
3.5.1 Design Paradigms for Modelling IEC 61499 Based Systems .....	32
3.5.2 Computing Paradigms for Modelling IEC 61499 Based Systems .....	36
3.6 Implementation of IEC 61499 Engineering Environments .....	43
3.6.1 Development of IEC 61499 Engineering Environments .....	43
3.6.2 Application of IEC 61499 Engineering Environments .....	45
3.7 Summary .....	47
 CHAPTER FOUR: ARCHITECTURE MODELLING FRAMEWORK .....	 49
4.1 Introduction .....	49
4.2 Modelling Framework .....	50
4.2.1 Multi-Layer Macro Architecture .....	51
4.2.2 Multi-Layer Micro Architecture .....	53
4.3 Summary .....	57
 CHAPTER FIVE: HIGH-LEVEL CYBER MODULE ARCHITECTURE MODELLING .....	 59
5.1 Introduction .....	59



5.2 Monitoring Agent Design .....	61
5.3 Analysis Agent Design .....	63
5.3.1 Analysis Agent Modelling.....	63
5.3.2 Self-Learning Agent Modelling .....	65
5.4 Planning Agent Design .....	68
5.5 Execution Agent Design .....	70
5.6 Knowledge Agent Design.....	72
5.7 Summary .....	74
<b>CHAPTER SIX: LOW-LEVEL PHYSICAL MODULE ARCHITECTURE MODELLING.....</b>	<b>75</b>
6.1 Introduction.....	75
6.2 Developing Low-level Control Systems in IEC 61499 Function Blocks .....	77
6.2.1 IEC 61499 Reference Architecture .....	77
6.2.2 Interface Declaration Model.....	79
6.2.3 Component Encapsulation Model .....	87
6.2.4 IEC 61499 Application Model Design .....	88
6.3 Self-Manageable Service Model for Architecture Design in IEC 61499 .....	90
6.3.1 Self-Manageable Service Model .....	90
6.3.2 Self-Manageable Service Execution Agent Design .....	92
6.3.3 Self-Manageable Agents Interface Design .....	92
6.3.4 IEC 61499 Function Block System Interface Design.....	93
6.3.5 Agent-Embedded Function Block Design Pattern .....	95
6.4 Summary .....	99
<b>CHAPTER SEVEN: ARCHITECTURE MODELLING EVALUATION .....</b>	<b>101</b>
7.1 Introduction.....	101
7.2 Illustrative Example Demonstration .....	102
7.2.1 Typical Industrial Scenario.....	102
7.2.2 High-Level Cyber Module Design .....	103
7.2.3 Low-Level Physical Module Design .....	106
7.3 Multi-Agent Simulation Model .....	108
7.3.1 Development of Agent-Based Model .....	108
7.3.2 Introduction of A New Part Type .....	113
7.3.3 Responding to A Conveyor Section Failure .....	115
7.3.4 Optimizing Part Routing.....	121
7.4 Experimental Testbed Design .....	125
7.4.1 Testbed Setup .....	125
7.4.2 Test Scenarios.....	128
7.5 Performance Evaluation Analysis.....	131
7.5.1 Regular Running Conditions .....	132
7.5.2 Adaptation Required Conditions .....	134
7.5.3 Performance Evaluation Estimation .....	140
7.6 Summary .....	143
<b>CHAPTER EIGHT: CONCLUSIONS AND FUTURE WORK.....</b>	<b>145</b>
8.1 Conclusions.....	145
8.2 Future Work .....	148

REFERENCES .....	151
APPENDICES .....	163
Appendix A Multi-Agent Simulation Model.....	163
Appendix A.1 NetLogo Agent Based Simulation Model.....	163
Appendix A.2 NetLogo Routing Optimization Model.....	186
Appendix A.3 Input for Agent-Based Simulation Model .....	190
Appendix A.4 Procedures for Low-Level Self-Manageable Agents.....	191
Appendix A.5 Communication for Agent-Based Model.....	193
Appendix B Experimental Testbed Setup.....	196
Appendix B.1 General Steps for Testbed Setup.....	196
Appendix B.2 High-Level JetBot Modelling Test .....	196
Appendix B.3 Low-level Devices Modelling Test.....	208

## **List of Tables**

Table 3-1: IEC 61499 review/keynote papers and their scopes.....	18
Table 3-2: Overview of the IEC 61499 standard publications .....	20
Table 3-3: Comparison of key aspects of IEC 61499 and IEC 61131-3 .....	21
Table 3-4: IEC 61499 FB execution semantics .....	23
Table 3-5: Main challenges for industrial adoption of IEC 61499 .....	26
Table 3-6: Projects of developing IEC 61499 engineering environments .....	43
Table 3-7: Applications of typical IEC 61499 engineering environments .....	46
Table 5-1: Algorithm for the proposed multi-agent MAPLE-K model.....	60
Table 6-1: Algorithm for the proposed self-manageable service model.....	91
Table 7-1: Execution time comparison in regular running conditions .....	134
Table 7-2: Execution time comparison in adaptation required conditions .....	138

## List of Figures

Figure 1-1: Structure of the thesis.....	6
Figure 2-1: A simplified and typical industrial automation scenario .....	7
Figure 3-1: IEC 61499 function block model .....	21
Figure 4-1: Multi-layer system architecture modelling framework.....	51
Figure 5-1: The proposed high-level architecture modelling framework .....	60
Figure 5-2: The <i>Agent_Monitoring</i> data model .....	62
Figure 5-3: The <i>Agent_Analysis</i> data model .....	65
Figure 5-4: The <i>Agent_Planning</i> data model.....	69
Figure 5-5: The <i>Agent_Execution</i> data model .....	71
Figure 5-6: The <i>Agent_Knowledge</i> data model .....	73
Figure 6-1: The proposed low-level architecture modelling framework .....	76
Figure 6-2: The IEC 61499 reference architecture .....	77
Figure 6-3: The IEC 61499 function block class diagram .....	79
Figure 6-4: IEC 61499 SIFB <i>requester/responder</i> models.....	80
Figure 6-5: Communication patterns of IEC 61499 SIFB <i>requester/responder</i> models .....	81
Figure 6-6: An example of IEC 61499 SIFB models .....	82
Figure 6-7: IEC 61499 adapter <i>plug/socket</i> models.....	83
Figure 6-8: IEC 61499 FB application without adapters .....	85
Figure 6-9: IEC 61499 adapter design .....	86
Figure 6-10: IEC 61499 FB application with adapters .....	86
Figure 6-11: Comparison of IEC 61499 CFB and SubApp models .....	88
Figure 6-12: Design of an IEC 61499 application model for distributed automation .....	89
Figure 6-13: The low-level self-manageable architecture modelling framework .....	91
Figure 6-14: The <i>SelfManageableServiceExecutionAgent</i> data model.....	92

Figure 6-15: The <i>SelfManageableAgents</i> data model .....	93
Figure 6-16: The <i>IEC61499FunctionBlockSystem</i> data model .....	94
Figure 6-17: The agent-embedded IEC 61499 FB model .....	96
Figure 6-18: Interface and FB network design of the agent-embedded FB module .....	97
Figure 6-19: FB network design of the agent-embedded FB module with adapters .....	98
Figure 7-1: An extended industrial automation scenario .....	103
Figure 7-2: The automated conveyor system .....	109
Figure 7-3: Agent-based simulation model for the automated conveyor system .....	111
Figure 7-4: Typical tests in performed three experiments .....	112
Figure 7-5: Introduction of a new part type: Test 1 .....	114
Figure 7-6: Introduction of a new part type: Test 2 .....	115
Figure 7-7: Responding to a conveyor section failure: Test 3 .....	116
Figure 7-8: Responding to a conveyor section failure: Test 4 .....	117
Figure 7-9: Responding to a conveyor section failure: Test 5 .....	118
Figure 7-10: All agent interaction diagram in Test 6 .....	120
Figure 7-11: The part routing agent-based model: Test 7 .....	122
Figure 7-12: The final positions of the <i>diverter</i> agents in Test 7 .....	123
Figure 7-13: Wait time performance for the ordered and optimized routing options (mean and 95% confidence intervals) in Test 7 .....	124
Figure 7-14: Experimental testbed design .....	126
Figure 7-15: The full system configuration in Eclipse 4diac .....	126
Figure 7-16: FB network design for LEDs in Eclipse 4diac .....	127
Figure 7-17: FB network design for Motor1 in Eclipse 4diac .....	127
Figure 7-18: FB network design for Motor2 in Eclipse 4diac .....	128
Figure 7-19: Communication network design for JetBot in Eclipse 4diac .....	128
Figure 7-20: Test scenario with two LEDs and one motor in Eclipse 4diac .....	129

Figure 7-21: Test scenario with four LEDs and two motors in Eclipse 4diac .....	130
Figure 7-22: Estimated performance evaluation under different system design architectures ...	142

## List of Abbreviations

<b>Symbol</b>	<b>Definition</b>
iCPS	Industrial Cyber-Physical Systems
IIoTS	Industrial Internet of Things and Services
PLC	Programmable Logic Controller
IEC	International Electrotechnical Commission
IL	Instruction List
ST	Structured Text
LD	Ladder Diagram
FBD	Function Block Diagram
SFC	Sequential Function Chart
HMI	Human-Machine Interface
IPMCS	Industrial Process Measurement and Control System
FB	Function Block
BFB	Basic Function Block
SIFB	Service Interface Function Block
CFB	Composite Function Block
ECC	Execution Control Chart
POU	Program Organization Unit
RTFM	Real-Time For the Masses
XML	eXtensible Markup Language
XSD	XML Schema Definition
OOD	Object-Oriented Design
CBD	Component-Based Design
SOA	Service-Oriented Architecture
SubApp	Sub-Application
MAS	Multi-Agent Systems
WSN	Wireless Sensor Networks
HLC	High-Level Control
LLC	Low-Level Control
NIST	National Institute of Standards and Technology
SaaS	Software as a Service
PaaS	Platform as a Service
IaaS	Infrastructure as a Service
OPA	Open Process Automation
DCS	Distributed Control Systems
DICS	Distributed Intelligent Control Systems
SCADA	Supervisory Control and Data Acquisition Systems
PLM	Product Lifecycle Management
MES	Manufacturing Execution Systems
ERP	Enterprise Resource Planning
SPADE	Smart Python Agent Development Environment

# **Chapter One: Introduction**

## **1.1 Research Background**

Industrial automation plays a fundamental role in global industry. From a broad view, industrial automation systems have been evolving through a series of industrial revolutions: beginning with water- and steam-powered mechanical manufacturing systems, and followed by electricity-powered mass production systems, then developing into information technology enabled mechatronic systems, until now integrated with communicating and computing capabilities to form industrial cyber-physical systems [1]-[2]. During these revolutions, industrial systems have gradually been automated and empowered from simply replacing heavy and repetitive labor work by machines, to applying computerized procedures and processes to control machines, to machines that are capable of intelligent behaviours. One of the most critical motivations for companies to continuously develop, deploy, and advance their automation systems is to remain competitive (e.g., balancing conflicts of mass customized product varieties and short production lead times, or low cost and high quality) in the global market. Especially over the last few decades, industrial automation systems empowered by information technology and intelligent electronics have significantly improved companies' performances in meeting challenges and achieving goals (e.g., business, societal, and environmental). In the past decade, the manufacturing industry has marched into a new era and is leading the way to the fourth industrial revolution (i.e., Industry 4.0 [1]), of which some key features, e.g., integration of industrial cyber and physical systems (iCPS), application of industrial internet of things and services (IIoTS), are required for the development of next-generation industrial systems.



Traditional industrial automation systems developed in centralized architectures (e.g., several automation processes controlled by a single controller) are statically programmed with determined procedures (e.g., system functions and module interactions designed at early stages considering limited available requirements) to perform predefined tasks in structured environments (e.g., system operating as initially designed and difficult to adapt during runtime). However, current industrial environments create many challenges for these systems, especially when viewed in the context of Industry 4.0. In particular, systems designed under traditional engineering techniques and running on legacy automation platforms are unable to automatically discover alternative solutions, flexibly coordinate reconfigurable modules, and actively deploy corresponding functions, to quickly respond to frequent changes and intelligently adapt to evolving requirements in dynamic environments. To address this challenge, research is required on the design and modelling of automation architectures that are responsive to frequent changes and adaptive to evolving requirements in a distributed and intelligent way during runtime. Frequent changes and evolving requirements result from both the supply chain aspect (e.g., customer requirements and manufacturing resources) and the industrial system aspect (e.g., software update and hardware maintenance). A distributed and intelligent solution requires that systems are flexible in self-managing distributed architectures (e.g., dynamic configuration of decentralized modules and scalable solutions to meet challenges) and are adaptable in self-organizing intelligent behaviours (e.g., balancing limited available resources and complex assigned workloads to improve overall utilization and to ensure required priority) in response to frequent changes and evolving requirements.

In order to realize the promise of Industry 4.0, an autonomous, distributed, and cooperative approach to automation and control is required, that matches the intelligent,

concurrent, and stochastic nature of next-generation industrial systems. Recently, industrial systems have been evolving into a new form (i.e., industrial cyber-physical systems, iCPS), in which cyber and physical components collaborate with each other and are empowered for intelligence by communicating and computing cores [2]. This new type of iCPS appears to hold the most promise of achieving modern industrial automation systems in the Industry 4.0 era, to be flexible in reconfiguration of distributed system architectures and to be intelligent in adaptation to changes in dynamic environments. Central to this work on distributed intelligent automation, is the academic research on and industrial application of the standards IEC 61131-3 [3]-[4] and IEC 61499 [5]-[6]. Furthermore, recent work in software design and hardware development have also helped reshape industrial automation systems [7]. In past decades, the IEC 61131-3 standard has been widely used in developing industrial automation systems, mainly focusing on the design of scan-based centralized and closed system architectures. However, as envisioned for the next-generation industrial automation systems to be portable, interoperable, and configurable, a new standard IEC 61499 was proposed for programming distributed industrial automation solutions [8]-[9]. IEC 61499 has been a promising alternative to IEC 61131-3 as it offers some key features, e.g., application-based distributed architecture design, object-oriented function block modelling, and event-driven control application execution, to address challenges (e.g., scan-based and device-centered centralized architecture, implementation/vendor dependent feedback connection and device communication) IEC 61131-3 is facing under current industrial environments.

In this research, the core objective is to explore the design of multi-layer automation architectures to enable real-time adaptation at the device level and run-time intelligence throughout the whole system under a well-integrated modelling framework. Central to this goal

is the research on the integration of multi-agent modelling and IEC 61499 function block modelling, together with other enabling techniques, to form a new automation infrastructure for iCPS. Multi-agent modelling uses autonomous and cooperative agents to achieve run-time intelligence in system design and module reconfiguration. IEC 61499 function block modelling applies object-oriented and event-driven function blocks to realize real-time adaption of automation logic and control algorithms. In this thesis, a multi-agent architecture modelling framework to realize IEC 61499 based distributed intelligent automation is proposed. The design will focus on a two-layer self-manageable architecture modelling.

## 1.2 Thesis Structure

The thesis is structured as follows (Figure 1-1). Following this introduction, the research motivation is described in Chapter Two, which outlines the research problem, objectives, methodologies, and expected contributions. Chapter Three is a literature review of five major topics on IEC 61499 function block modelling for distributed automation systems, in which design methods (e.g., object-oriented modelling technique), computing paradigms (e.g., autonomic computing framework), and engineering environments (e.g., Eclipse 4diac) are analyzed and will be applied in this thesis. The following chapters focus on deploying autonomic computing in the proposed architecture modelling framework, including: a) the reference architecture employed in the high-level cyber module and implemented as multi-agent systems, b) the self-managing properties employed in the low-level physical module and implemented as agent-embedded IEC 61499 function blocks. The design results in a new computing module for high-level multi-agent based automation architectures and a new design pattern for low-level function block modelled control solutions. In Chapter Seven, the proposed architecture modelling framework is demonstrated and evaluated through various experiments on the multi-agent simulation model and the designed experimental testbed. Chapter Eight closes the thesis with conclusions and future work.

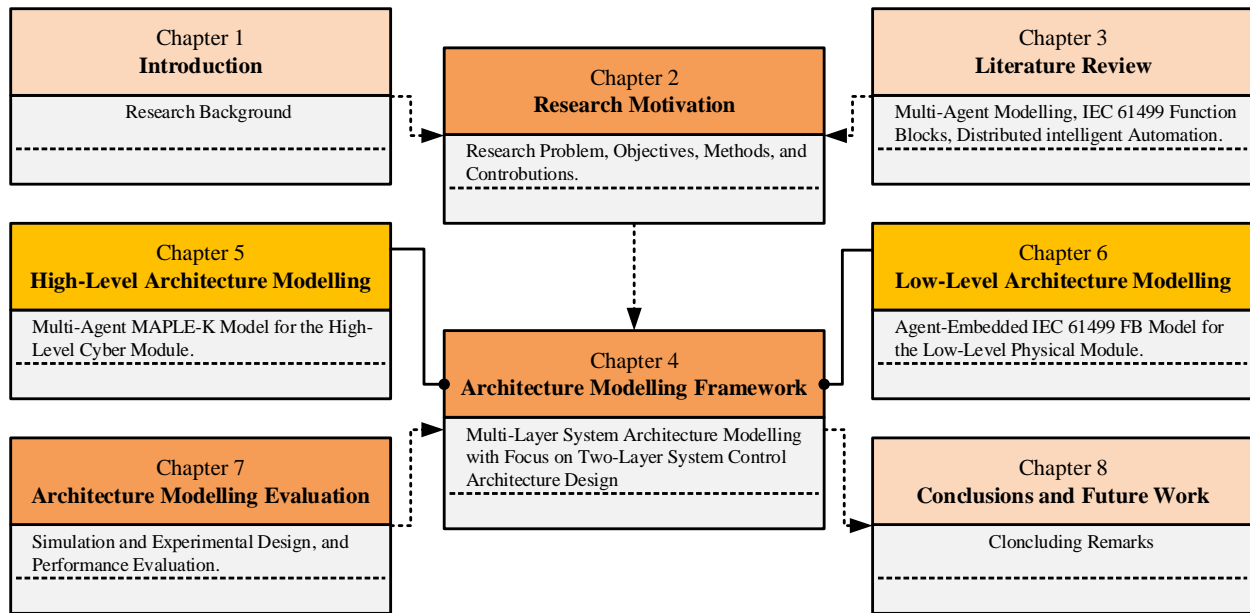


Figure 1-1: Structure of the thesis

## Chapter Two: Research Motivation

### 2.1 Problem Statement

Traditional industrial automation systems developed in centralized architectures are statically programmed with determined procedures to perform predefined tasks in structured environments. Figure 2-1 shows a simplified but typical industrial automation scenario, in which the system is originally programmed to sort specific blocks into corresponding bins. Details are described below to explain the research motivation:

- a programmable robotic arm can rotate and translate to grasp and place blocks from a conveyor into bins on the fly on a workbench;
- the task for the robotic arm is to pick up one type of block from the conveyor and then place them into the corresponding type of the bin on the workbench; and
- engineers pre-program specific working procedures for robotic arms to finish specific tasks.

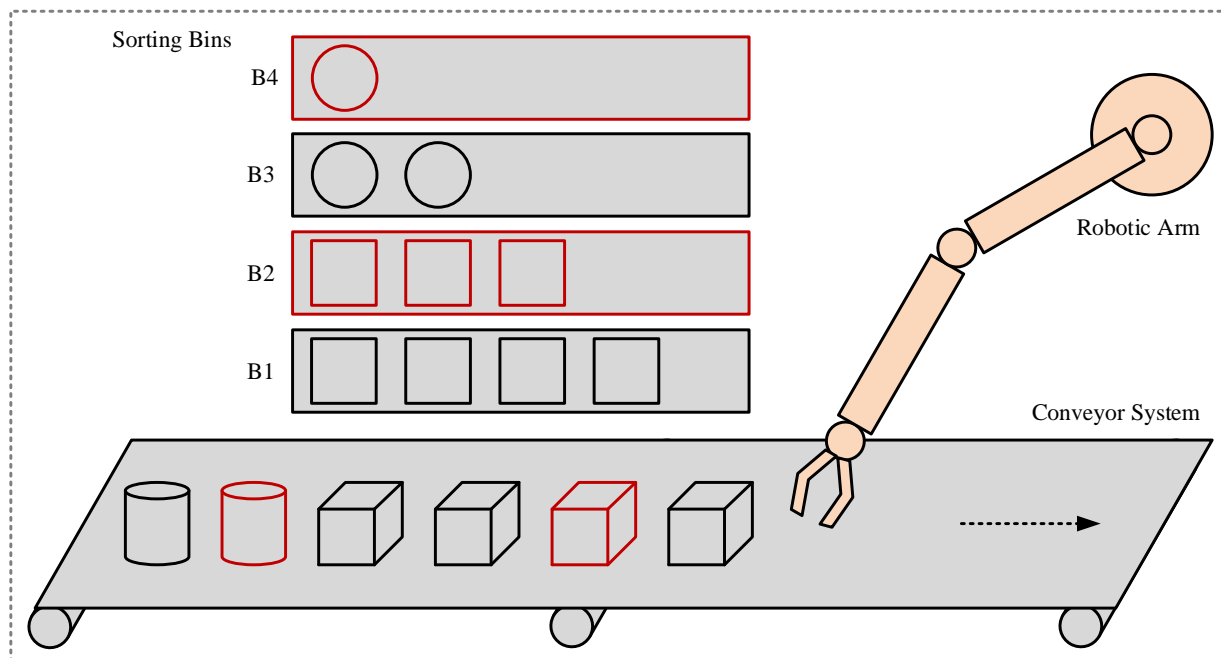


Figure 2-1: A simplified and typical industrial automation scenario

This type of scenario is common for many industrial applications, such as assembly lines in automotive manufacturing and sorting systems in distribution centres. They are considered as traditional industrial automation systems with enough automated capabilities to replace repetitive labour work or with some degree of flexibility for programmable controls. Typical features of these systems can be summarized as:

- static programming with determined procedures (i.e., pre-programmed procedures);
- structured environments (i.e., same blocks and bins on a specific workbench);
- a set of predefined tasks/behaviours (i.e., reach, grasp, and place); and
- centralized coordinating control (i.e., centralized pace control of the robotic arm and the conveyor).

For traditional industrial automation systems, control units or automation functions have been programmed in IEC 61131-3 and executed in programmable logic controllers (PLCs) for decades [3], [10]. PLCs originated in the late 1960s and were programmed in Boolean formats or relay-derived ladder logic, as automated machines at that time were controlled by complex relays and were hard for maintenance and reconfiguration. In the early 1980s, the introduction of IEC 61131-3 standardized the programming aspects (e.g., languages, data types, input/output) for PLCs. Especially, the International Electrotechnical Commission (IEC) defined five programming languages, including: two textual programming languages, instruction list (IL) and structured text (ST); two graphical programming languages, ladder diagram (LD) and function block diagram (FBD); and an additional set of graphical and equivalent textual elements, sequential function chart (SFC) [3], [10]. In the 1990s, PLCs welcomed programmable human-machine interfaces (HMI) to replace traditional pushbutton styles for better human machine interaction (e.g., troubleshooting). The recent two decades saw the emergence of advanced

software and hardware technologies, and IEC 61131-3 cannot sufficiently support the design of distributed intelligent industrial systems. The IEC 61499, a new and improved standard based on extended function block concept of IEC 61131-3 was developed in the 2000s for the implementation of distributed industrial process measurement and control systems (IPMCSs) to support flexibility, portability, interoperability, and reconfigurability [8]-[9].

Although IEC 61131-3 still dominates the design of legacy systems, current industrial environments have created many challenges for these legacy systems, especially when viewed in the context of Industry 4.0. Major issues for example are: a) current PLC technologies are not suitable for building distributed intelligent automation system architectures; b) automation programming languages of IEC 61131-3 used in current PLCs are implemented by each vendor and thus prevent interoperability; c) current IEC 61131-3 programmed PLC systems are difficult in adaption and in response to changes while maintaining predictable and stable operations during runtime. Consider the scenario in Figure 2-1 by adding additional elements:

- there is a speed change of the conveyor which delivers blocks;
- there is one red block mixed into those same type of black blocks;
- there is also a red bin for those red blocks right behind the black bin; and
- there is a different shape (e.g., cylinder) mixed into those blocks.

In traditional industrial systems, all interactions and functions are designed at the development stage by considering limited available requirements. Although some of above cases may be considered at the beginning (e.g., encoders to measure conveyor speed), such type of case may occur unexpectedly and thus beyond the design capability of the system. This could result in modifying affected functions offline according to changes of design specifications or new requirements. That means industrial systems designed under traditional engineering



techniques and running on legacy automation platforms are unable to automatically discover alternative solutions, flexibly coordinate reconfigurable modules, and actively deploy corresponding functions, to quickly respond to frequent changes and intelligently adapt to evolving requirements in dynamic environments. Consider the scenario in Figure 2-1 and see what will happen next:

- the robotic arm could miss blocks from the conveyor because of pace changes between them;
- the robotic arm could also pick up the wrong red block and place it into the black bin as usual, not the red one; and
- the robotic arm could not be able to pick up different shapes (e.g., cylinders).

With predetermined procedures in structured environments for predefined tasks, even when the system is operating normally, errors could still happen because of those frequent changes and evolving requirements that may not all be considered at the very beginning. Therefore, as envisioned in this research, a well-integrated design framework to model automation architectures is required for the development of next-generation industrial systems in the Industry 4.0 era, that are responsive to frequent changes and adaptive to evolving requirements in a distributed and intelligent way during runtime. That means under the new architecture modelling framework:

- the robotic arm can detect speed changes of the conveyor through communication to coordinate the pace;
- the robotic arm can distinguish between red and black blocks, and other shapes;
- the robotic arm can learn from past experience on black blocks to transfer sorting skills to the red block, or in more complex situations to pick up and put it into the red bin; and

- the robotic arm can learn from raw sensory data with black blocks to achieve skills to pick up different shapes (e.g., cylinders).

The challenges associated with this type of scenario are becoming critical requirements for developing next-generation industrial automation systems. In the Industry 4.0 era, such systems should be modelled with open architectures to support portability, interoperability, and reconfigurability, which is the vision of IEC 61499 compared to the traditional IEC 61131-3 standard. As reviewed in Chapter Three, research on IEC 61499 based industrial automation systems was mainly focused on function block execution semantics and transformation techniques for IEC 61131-3 based systems to IEC 61499 based ones. Until recently, there are studies on the integration with IEC 61499 enabling technologies of design and computing paradigms for modelling industrial automation systems. Recent advances in hardware (e.g., smart control devices) and software (e.g., mobile automation apps) provide new opportunities to develop such industrial automation systems. Furthermore, artificial intelligence/machine learning techniques have created a set of computational tools (e.g., deep learning and deep reinforcement learning) that can empower the system to be intelligent to a certain human level.

In summary, industrial communicating and computing techniques have evolved into a new era, in which service-oriented and event-based programming, machine learning and data analytics, etc. are widely applied in the system design. A new paradigm is required to bring all these together to realize distributed intelligent system architectures envisioned by Industry 4.0. In light of the recent work in the area of industrial automation systems, especially the research on:

- the standard of IEC 61499 for the application-based distributed architecture design that applies object-oriented function block modelling and event-driven control application execution; and

- the vision of Industry 4.0 for a smart and networked world that leverages the integration of industrial cyber and physical systems (iCPS) and the application of industrial internet of things and services (IIoTS).

This research will continue to enrich design methods and computing frameworks for building self-managing industrial systems and to integrate multi-agent modelling with IEC 61499 function block modelling for programming distributed automation solutions. The question that this research is trying to ask and help answer can be stated as follows:

*“How to achieve self-manageable industrial cyber-physical systems for IEC 61499 based distributed intelligent automation (i.e., to explore the design of multi-layer automation architectures to enable real-time adaptation at the device level and run-time intelligence throughout the whole system by integrating multi-agent modelling and IEC 61499 function block modelling)”*

## 2.2 Objectives and Methodologies

As stated in the research question, the long-term goal is to achieve *self-manageable industrial cyber-physical systems for IEC 61499 based distributed intelligent automation*. However, in this research the core objective is to explore *the design of multi-layer automation architectures to enable real-time adaptation at the device level and run-time intelligence throughout the whole system under a well-integrated modelling framework*.

Central to this goal is the research on the integration of multi-agent modelling and IEC 61499 function block modelling, together with other enabling techniques, to form a new automation infrastructure for iCPS. The major research methodologies focus on system architecture modelling including high-level cyber module and low-level physical module through deploying autonomic computing into architecture design [11]-[13]. In detail, the reference architecture is employed in the cyber module design and implemented as multi-agent systems, resulting in a new computing module with self-learning capabilities for high-level multi-agent-based automation architectures [11], [13]. This methodology uses autonomous and cooperative agents to achieve run-time intelligence in system design and module reconfiguration. The self-managing properties are employed in physical module design and implemented as agent-embedded IEC 61499 function blocks, resulting in a new design pattern with embedded agent intelligence for low-level function block modelled control solutions [12]-[13]. This methodology applies object-oriented and event-driven function blocks to realize real-time adaption of automation logic and control algorithms. The proposed design are evaluated through simulation model development and experimental testbed design to show expected system capabilities to respond to major challenges in Industry 4.0 [13]-[15].

## 2.3 Anticipated Contributions

As stated before, the core objective of this research is to explore the design of multi-layer automation architectures to enable real-time adaptation at the device level and run-time intelligence throughout the whole system under a well-integrated modelling framework. A multi-agent modelling framework of iCPS for IEC 61499 based distributed intelligent automation is proposed in this thesis, focusing on a two-layer self-manageable architecture modelling. This research is expected to result in the following contributions:

- 1) *High-Level iCPS Architecture*. This architecture will use a multi-agent modelling framework to support autonomic computing (i.e., *Monitoring*, *Analysis*, *Planning*, *Execution*, and *Knowledge*). Instead of modelling these elements as services, a multi-agent model is applied so that it can actively interact with each other, operating environments, and system modules to achieve real-time communication and computation.
- 2) *Self-Learning Agent*. The traditional autonomic computing framework will be enhanced with the introduction of a self-learning agent. Traditionally, only predefined rules, policies, and goals are provided by *Analysis* with limited situations. The system can work in some simple situations with predefined knowledge whereas in most cases it is far less capable of dealing with real-time dynamic situations. Furthermore, recent advances in artificial intelligence have created a set of computational tools (e.g., deep learning and deep reinforcement learning) that can empower the system to be intelligent to a certain human level. These artificial intelligence techniques are leveraged to enable system self-learning capabilities.
- 3) *Low-Level iCPS Architecture*. This architecture will be based on the IEC 61499 function block modelling framework for event-driven distributed execution. Instead of programming automation systems in IEC 61131-3 in a centralized way, IEC 61499 function block

modelling for distributed automation is applied, which allows more flexibility, portability, interoperability, and reconfigurability. Also, node intelligence is embedded into function blocks to reduce data transmitting bandwidth and processing load.

- 4) *Self-Management Agents*. A new agent-embedded design pattern for modelling IEC 61499 function block based control applications will be introduced that provides self-management capabilities for real-time adaptation. These self-manageable agents embedded in IEC 61499 function blocks are either initialized to be active for predefined tasks (i.e., self-configuration, self-optimization, self-healing, and self-protection) or deactivated in a sleep state. With this design pattern, the new agent-embedded function block types can be introduced to build self-manageable control applications.

*[This page intentionally left blank]*

## **Chapter Three: Literature Review**

### **3.1 Introduction**

Real-time control units or automation functions are mainly programmed under IEC 61131-3 or IEC 61499 and executed in PLCs or intelligent embedded devices. IEC 61131-3 dominates the design of traditional industrial automation systems and faces a lot of challenges, while IEC 61499 is developed for programming next-generation industrial automation systems to support portability, interoperability, and configurability. Table 3-1 provides a list of published review papers or keynotes on the IEC 61499 research. As stated before, the main theme of this thesis is to explore the design of architecture modelling frameworks for IEC 61499 based distributed intelligent automation. Therefore, this chapter will focus on two major topics [16]-[18]:

- how IEC 61499 has evolved as a standalone standard for programming next-generation industrial automation systems (Section 3.2, 3.3, and 3.4); and
- how IEC 61499 has interacted with enabling technologies to realize distributed intelligent automation (Section 3.4, 3.5, and 3.6).

Before the summary in Section 3.7, the two major topics are further detailed into five sub-topics. In these sub-topics, this research is based on the IEC 61499 reference architecture (Section 3.2), focuses on design and computing paradigms for modelling IEC 61499 based systems (Section 3.5), and applies one of the IEC 61499 engineering environments for implementation (Section 3.6). Discussions on IEC 61499 function block execution and system transition are for an overview purpose. These five sub-topics are as follows:

- how the IEC 61499 standard is envisioned for programming next-generation industrial automation systems (Section 3.2);



- how each IEC 61499 FB is activated through event scheduling for determined execution (Section 3.3);
- how existing systems programmed in IEC 61131-3 can be transitioned to IEC 61499 based systems (Section 3.4);
- how IEC 61499 has integrated with enabling technologies for distributed intelligent automation (Section 3.5); and
- how engineering environments for IEC 61499 have been implemented (Section 3.6).

Table 3-1: IEC 61499 review/keynote papers and their scopes

Reference	Scope
[Georg and Hussain 2006] [19]	Modelling techniques for distributed control systems based on IEC 61499.
[Zoitl <i>et al.</i> 2007] [20]	Execution, verification, reconfiguration, and industrial adoption of IEC 61499.
[Thramboulidis 2007] [21]	Analysis of inefficiencies of the IEC 61499 model in factory automation.
[Hall <i>et al.</i> 2007] [22]	Challenges to industry adoption of IEC 61499 event-based function blocks.
[Brennan <i>et al.</i> 2008] [23]	Dynamic and intelligent reconfiguration of IEC 61499 based industrial automation.
[Zoitl and Vyatkin 2009] [24]	Modelling of distribution and architecture-centric design issues in IEC 61499.
[Zoitl <i>et al.</i> 2009] [25]	Comparative study of IEC 61131-3 and IEC 61499 for distributed automation systems.
[Hanisch <i>et al.</i> 2009] [26]	Formal modelling and verification of IEC 61499 function blocks.
[Vyatkin 2009] [27]	The IEC 61499 standard and its semantics for distributed automation systems.
[Vyatkin 2011] [28]	IEC 61499 as enabler of distributed and intelligent automation.
[Strasser <i>et al.</i> 2011] [29]	Design and execution issues in IEC 61499 distributed automation systems.
[Strasser <i>et al.</i> 2012] [30]	Launch and takeoff of the IEC 61499 function block standard.
[Christensen <i>et al.</i> 2012a] [31]	Overview of the second edition of the IEC 61499 function block standard.
[Christensen <i>et al.</i> 2012b] [32]	Software tools and running platforms of the IEC 61499 function block standard.
[Thramboulidis 2015] [33]	Service-oriented architectures for IEC 61499 industrial automation systems.
[Sinha <i>et al.</i> 2019] [34]	Static formal methods for industrial automation systems in IEC 61499/IEC 61131-3.
[Prenzel <i>et al.</i> 2020] [35]	Comparative study of IEC 61499 runtime environments.
[Lyu and Brennan 2020] [17]	Transformation methods, modelling techniques, and implementation tools in IEC 61499.
[Sonnleithner <i>et al.</i> 2021] [36]	Summary of a catalog of suboptimal structures or patterns in IEC 61499 applications.

## **3.2 Initiation of IEC 61499 Standards**

### ***3.2.1 IEC 61499 Theoretical Fundamentals***

The theoretical fundamentals to develop IEC 61499 originated from its predecessor the IEC 61131-3 standard. IEC 61131-3 has been designed with a scan-based centralized and closed architecture for PLCs in industrial automation for decades [4]. With advanced modelling methodologies, complex engineering requirements, and emerging software and hardware technologies, the IEC 61131-3 standard cannot sufficiently support the design of intelligent distributed automation systems. However, the defined programming languages, especially the concept of function blocks, are well established and widely utilized in industrial automation practices. Therefore, the IEC 61499 standard based on extended function block concept of IEC 61131-3 is developed and continues to be improved.

An overview of the IEC 61499 standard publications is shown in Table 3-2. IEC 61499-1 defines a generic architecture in terms of implementable reference models, textual syntax and graphical representations, and their guidelines for the use of function blocks in industrial automation [6]. IEC 61499-2 defines software tool requirements to support engineering tasks of IEC 61499 based systems [37]. IEC 61499-4 defines rules for the development of compliance profiles to realize key requirements of IEC 61499 based systems, devices and software tools [38]. IEC 61499-3 is a technical report and has been withdrawn [39].

Table 3-2: Overview of the IEC 61499 standard publications

Title	Topic	Type	Publication	Status
IEC 61499-1	Function Blocks for IPMCSs - Part 1: Architecture	Publicly Available Specification	Edition 1.0, 2000	Replaced
	Function Blocks - Part 1: Architecture	International Standard	Edition 1.0, 2005 Edition 2.0, 2012	Revised Valid
IEC 61499-2	Function Blocks for IPMCSs - Part 2: Software Tools Requirements	Publicly Available Specification	Edition 1.0, 2001	Replaced
	Function Blocks - Part 2: Software Tools Requirements	International Standard	Edition 1.0, 2005 Edition 2.0, 2012	Revised Valid
IEC 61499-3	Function Blocks - Part 3: Tutorial Information	Technical Report	Edition 1.0, 2004	Withdrawn 2008
IEC 61499-4	Function Blocks for IPMCSs - Part 4: Rules for Compliance Profiles	Publicly Available Specification	Edition 1.0, 2002	Replaced
	Function Blocks - Part 4: Rules for Compliance Profiles	International Standard	Edition 1.0, 2005 Edition 2.0, 2013	Revised Valid

### 3.2.2 IEC 61499 Function Block Models

As stated before, IEC 61499 has been proposed for the development and implementation of distributed IPMCSs to support flexibility, portability, interoperability, and reconfigurability [40]-[43]. Compared to the traditional IEC 61131-3 standard, it provides an open reference architecture to design distributed IPMCSs with some key features, e.g., object-oriented modelling by using function blocks as basic elements and event-driven execution by using data/events as inputs/outputs.

The IEC 61499 standard [6] defines three types of function blocks (FBs): a) *basic function block* (BFB) defined as an event-driven state machine for execution of algorithms with inputs/outputs; b) *service interface function block* (SIFB) defined as a service sequence diagram for encapsulation of FB interaction with external services; and c) *composite function block* (CFB) defined as a network of FB instances through event and data connections. A FB model is first triggered by an input event with available input data, then executed through evaluating states (i.e., through the execution control chart (ECC)) and functioning algorithms (i.e., through the schedule function), and finally updated with data/event outputs (Figure 3-1). A typical *system* programmed under the IEC 61499 reference architecture is designed as: a) the control logic built

by *function blocks as applications*, and b) *physical devices* encapsulating required *resources* for implementation. The reference architecture will be further discussed in Section 6.2.

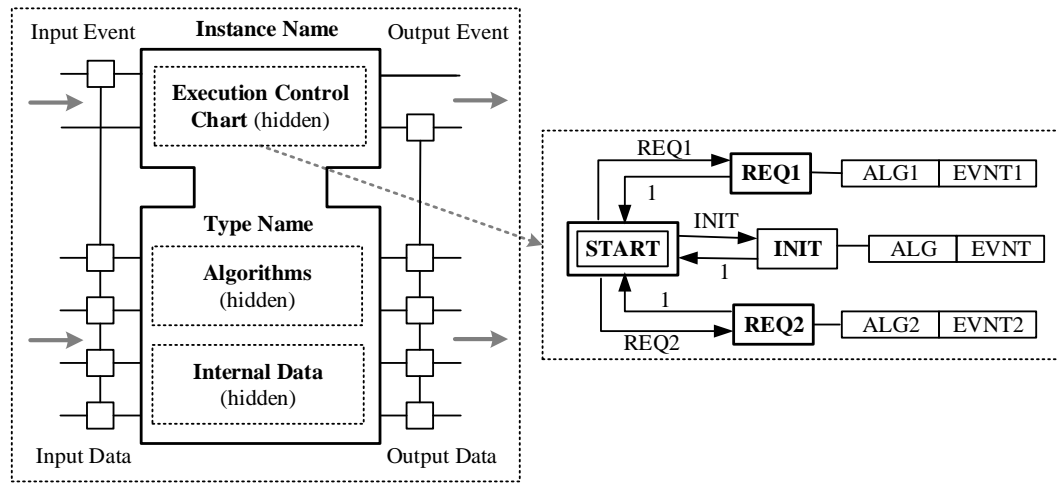


Figure 3-1: IEC 61499 function block model

In conclusion, both IEC 61131-3 and IEC 61499 have evolved with their own key characteristics and employment of emerging technologies to fulfill new industrial requirements. Table 3-3 provides a comparison of key aspects of IEC 61499 and IEC 61131-3.

Table 3-3: Comparison of key aspects of IEC 61499 and IEC 61131-3

	<b>IEC 61499</b>	<b>IEC 61131-3</b>
Modelling Paradigm	Object-Oriented	Object-Oriented (supported)
Modelling Component	Function Blocks (FBs)	Program Organization Units (POUs)
Input and Output	Events, Data	Data
Execution Mechanism	Event Driven	Scan Based (cyclic or periodic)
Data Type	Adopted from IEC 61131-3	Defined, e.g., integers, strings, etc.
Engineering Approach	Application Centered	Device Centered (in practice)
Architecture Characteristic	Open	Closed
Structure Style	Process Type	Subroutine Type
Communication Paradigm	Publish/Subscribe, Client/Server	Shared Memory, Communication Service
Communication Mechanism	Messages	Shared/Global Variables
Modelling Level	Programming of Complete System	Programming of Single Controller
Programming Language	No specific language defined, but IEC 61131-3 ones recommended	IL, ST, FBD, LD, SFC

### **3.3 Execution of IEC 61499 Function Blocks**

Design modelling and execution semantics of IEC 61499 applications have been researched and reviewed in numerous published papers [27]-[29], [44]-[45]. Research in this area addresses the question of how each FB in the FB network is activated through event scheduling to realize determined execution. In this section, some typical execution models are summarized and then recent research on semantic-correct mapping for IEC 61499 will be reviewed.

#### ***3.3.1 Execution Semantics of IEC 61499 Function Blocks***

Some typical execution models for IEC 61499 FBs are summarized in Table 3-4. The non-preemptive multi-threading resource (NPMTR) model implemented in FBRT/FBDK is event-triggered [46], whereas the cyclic execution model implemented in ISaGRAF and Cycle RT is PLC-like cyclic-scan [47]-[48]. Both FUBER and 4diac FORTE implement the sequential execution model with the main difference of the former using local event buffer, whereas the latter using global event buffer [49]-[51]. In the parallel execution model, aligning the FB execution speed with global instantaneous events distinguishes the synchronous and the asynchronous [51]-[53]. The hybrid model views a distributed IEC 61499 system as a collection of synchronous compositions of FBs communicating with each other over an asynchronous network [54].

Table 3-4: IEC 61499 FB execution semantics

References	Execution Model	Main Idea	Implementation
[Sünder <i>et al.</i> 2006b] [46]	Non-Preemptive Multi-Threading Resource (NPMTR)	Depth-first event scheduling for execution of a targeted FB with an emitted event immediately.	FBRT/FBDK
[Vyatkin and Chouinard 2008; Tata and Vyatkin 2009] [47]-[48]	Cyclic Execution	Scheduling events for periodical execution of each FB in the FBN in a cyclic way.	ISaGRAF Cyclic RT
[Cengic <i>et al.</i> 2006b; Vyatkin and Dubinin 2007; Vyatkin <i>et al.</i> 2007] [49]-[51]	Sequential Execution	Breadth-first event scheduling for execution of FBs with a sequence of emitted events in ways of local or global event buffer.	FUBER 4diac FORTE
[Vyatkin <i>et al.</i> 2007; Dubinin and Vyatkin 2008; Yoong <i>et al.</i> 2009; Yoong <i>et al.</i> 2015] [51]-[54]	Parallel Execution	Scheduling events for parallel execution of multiple FBs on multi-core processor architectures in synchronous, asynchronous, or hybrid ways.	Prototype Complier

### 3.3.2 Semantic-Correct Mapping for IEC 61499 Function Blocks

A refactoring approach to ECCs in BFBs by removing deadlock states was proposed and implemented through graph transformations with a set of defined rules [55]. The proposed method can be extended to FB networks and applied to semantic-correct transformation of control programs. Then semantics-robust design patterns for IEC 61499 to solve the portability problem were further proposed [56]. The general idea is to transform original FB applications executed in some source model to resulting FB applications in target models with the same behaviour. Dai *et al.* proposed a design recovery, semantic analysis, and code generation framework based on ontology models for IEC 61499 [57]-[58]. The key difference between two studies is whether the design process starts with a single IEC 61499 platform [57], e.g., either FBDK or nxtSTUDIO, or multiple IEC 61499 platforms [58], e.g., both FBDK and nxtSTUDIO.

Lindgren *et al.* proposed a mapping of IEC 61499 FBs to the real-time for the masses kernel (i.e., RTFM-kernel) for predictable real-time execution to address the execution problem of IEC 61499 FBs for light-weight controllers with limited resources [59]. Then a generic runtime system, i.e., RTFM-RT, to execute RTFM-core programs on threaded platforms for

predictable IEC 61499 execution was developed [60]. The proposed mapping from IEC 61499 FBs to RTFM task and resource models is used for the execution under RTFM-RT. RTFM-core, RTFM-kernel, and RTFM-RT are parts of the RTFM-lang framework [61]. The implementation difference is that the RTFM-kernel directly exploits the peripheral hardware, whereas the RTFM-RT utilizes available threading architectures for scheduling and representing tasks, resources, or baselines [62]. Furthermore, based on the mapping of IEC 61499 FBs to RTFM task and resource models, a real-time semantics for IEC 61499 to realize timing semantics [60] and a technique to assess the end-to-end response time of IEC 61499 distributed applications over switched Ethernet [63] were also proposed.

Yoong *et al.* proposed a synchronous approach for mapping of IEC 61499 FBs with the synchronous language *Esterel* primitives to support precise execution semantics and formal verification [53]. Then a tool was developed to translate IEC 61499 FBs to *Esterel* primitives for verification of both control and data properties in FB programs [64]. Two design patterns were further proposed and characterized as time-predictable, determinist, and reactive [65]. One is the order synchronous design pattern for intra-resource FBs and the other is the delayed synchronous design pattern for inter-resource FBs with parallel execution. Sinha *et al.* proposed a syntactic extension defined as hierarchical and concurrent execution control chart (HCECC) to IEC 61499 [66]. HCECCs introduce parallel and refined operators to allow explicit modelling of concurrency and hierarchy for ECCs in IEC 61499 BFBs and then are translated to IEC 61499 CFBs with synchronous execution semantics. Based on successful mapping between IEC 61499 FBs and synchronous primitives, an implementation scheme to define formal modelling and simulation verification for execution semantics of IEC 61499 FB networks via fixed point semantics [67] and a time-stamped discrete-event-based execution semantics for IEC 61499 FBs

with real-time constraints and deterministic execution behaviours [68] were proposed.

In conclusion, research in this area is concerned with the standard itself, i.e., FB execution to solve semantics ambiguities. The FB execution semantics define rules for behaviours of FB execution. Semantics ambiguities could lead to nondeterministic behaviours of the same application executing in different IEC 61499 implementations [49]. Research on semantic analysis for IEC 61499 has focused on formal modelling of IEC 61499 FBs, then semantic-correct mapping between IEC 61499 FBs and proposed models through defined transformation rules, and finally simulation verification of execution semantics.



### 3.4 Transition to IEC 61499 Based Systems

#### 3.4.1 Challenges with Transition to IEC 61499 Based Systems

IEC 61499 is promised to enable distributed architecture design for programming future industrial automation systems. Although, not as widely adopted by industry as IEC 61131-3, IEC 61499 is gaining more popularity than before in industry. For example, automation and control solution providers Schneider and Rockwell are leading in industrializing IEC 61499 with nxtControl [69] and ISaGRAF [70] kits, respectively. If the industrial adoption of IEC 61499 is viewed in the context of the three-phase S-shaped Logistic Curve [71], it was in the first “Launch” phase when promoted by innovators before 2012 (i.e., Ed. 2.0 published), and is now in the transition to the second “Takeoff” phase associated with early adopters. There is still a long way to go to reach the third “Maturity” phase until some key issues are fully solved. As well, some of the challenges with wide industrial adoption that were identified in the early years of the standard (e.g., [72]) remained during the publication of Ed. 2.0 (e.g., [30]). In conclusion of recent research on IEC 61499, three main types of challenges for industrial adoption are identified in Table 3-5: a) industrial concerns on business development, b) technical issues related to standard itself, and c) societal aspects of trained personnel.

Table 3-5: Main challenges for industrial adoption of IEC 61499

Main Challenge	Detailed Explanation
Industrial Concerns	Large amount of existing IEC 61131-3 based systems
	Little demand for a completely new design approach
	Huge cost incurred by introducing new technologies
Technical Issues	Few proved methods to redesign existing systems
	Same execution semantics but different system behaviours
	Better integration for efficient domain-specific design practice
Societal Aspects	New qualification requirements for control engineers
	New course design for teaching and learning IEC 61499
	New industrial training for applying and using IEC 61499

Considering these critical factors, much effort has been put into realizing successful and wide industrial adoption of IEC 61499. Some reasonable solutions are suggested as follows:

a) *To redesign existing IEC 61131-3 based systems for compliance with IEC 61499.*

Redesign is an intermediate step to transform existing systems programmed in IEC 61131-3 to IEC 61499 based systems to ensure they address industrial concerns on cost/benefit analysis and confidence/time for system transition. One example of work on this is Peltola *et al.*'s evaluation of IEC 61499 for the batch process industry [73]. The most recent research on the potential transitional path towards full adoption of IEC 61499 is the IEC 61499 CPS-izer proposed in Daedalus: a small-footprint controller capable of interacting with legacy systems through communication buses [74]. Daedalus is a pioneer European initiative for real-time distributed intelligence and cloud enabled CPS design modelling [75]. In Section 3.4.2, research on IEC 61499 transformation methods will be discussed in detail.

b) *To provide feasible methods, techniques, and guidelines for designing IEC 61499 based systems.* Design is concerned with system modelling from the perspective of technical issues. One of the most critical technical issues is execution semantics, which has been thoroughly researched before IEC 61499 Ed. 2.0 (e.g., [29]). This work led to significant technical changes in the second edition: e.g., concurrency issues in execution control for deterministic execution, data consistency in sampling, declaration of temporary variables, network and segment types [31]-[32]. Recent research is more focused on IEC 61499 design modelling by integrating enabling technologies (e.g., service-oriented architecture, autonomic computing, and cloud computing) for advanced system capabilities (e.g., solutions as services in the cloud in Daedalus Digital Marketplace [76], system self-managing features enabled by autonomic service management [77]) in Industry 4.0. In Section 3.5, research on this aspect will

be discussed in detail.

c) *To providing qualified courses and hands-on training programs for students and engineers to learn and use IEC 61499.* Teaching, learning and training are focused on societal aspects to support IEC 61499 applications based on available engineering environments. One great practice led by Zoitl *et al.* [78] is IEC 61499 workshops and events with hands-on courses and programs using Eclipse 4diac kit [79]. Another practice, initiated by Vyatkin *et al.* [80], is a hands-on training program to learn IEC 61499 using Schneider nxtControl kit [69]. An earlier hands-on tutorial [81] was presented to use Holobloc kit [82] to design distributed control applications. In Section 3.6, research on IEC 61499 system implementation will be discussed in detail.

### ***3.4.2 Methods of Transformation to IEC 61499 Based Systems***

Given the predominance of IEC 61131-3 based systems, there has been considerable interest in methods to transform IEC 61131-3 models to IEC 61499. For example, in this section, model-driven, object/class-oriented, and ontology-based approaches, and combination paradigms are identified and discussed. However, the effort of programming and complexity of implementation for the two standards for different types of applications are different [83]. Therefore, selection between the IEC 61499 event-driven execution model and the IEC 61131-3 cyclic execution model is application dependent [25].

#### ***A. Model-Driven Approach***

Sünder *et al.* provided concepts, rules, and methods for transformation of existing IEC 61131-3 automation projects into IEC 61499 control logic [84]-[85]. Transformation concepts include one based on equivalence of executing elements and another according to equivalence of resource elements. Transformation rules regarding aspects of configurations, resources, programs,

functions, and FBs are defined. Transformation methods are suggested as the one mapping POU with ECCs in BFBs and the other mapping POU with FB networks in CFBs. Based on this initial study, a model-driven automatic transformation approach was presented to test the second method for transformation of FBDs [86]. The proposed model to model transformation translates the input IEC 61131-3 source model through internal E-core models and finally into the output IEC 61499 target model. The implementation is realized through well matched libraries of both standards, input/output models as XML files, and XSD files as meta models. Further research focused on providing semantic correct transformations and two auxiliary transformations were studied to solve semantic issues [87]. One is the static transformation by converting IEC 61131-3 FBs into simple FBs to solve library differences between two standards, and the other is the project dependent transformation by extracting execution orders of IEC 61131-3 FBs to solve execution sequence problems. Wenger *et al.* also proposed an automatic reengineering approach to migrate IEC 61131-3 based control applications into IEC 61499 [88]. The process was tested as proof of concept for sorting stations programmed in CoDeSys V2.3 [89] and reengineered into Eclipse 4diac software tools.

### *B. Object/Class -Oriented Approach*

Dai and Vyatkin proposed two types of design patterns to redesign IEC 61131-3 PLCs using IEC 61499 FBs [90]-[91]. The object-oriented approach considers each device distributed in the system and the class-oriented approach considers each service provided by all devices. For both, FB represents a class of its objects (i.e., devices), encapsulates data and methods, and can be instantiated. The object-oriented approach creates each individual instance of one FB for each device, whereas the class-oriented approach creates a single instance of one FB to serve all devices of this class. As a result, the difference is whether to create only one instance or different

instances of one FB to serve all or each of its devices. The object-oriented approach also includes conversion of PLC code into an ECC and reuse of PLC code in an algorithm.

### *C. Ontology-Based Approach*

Semantic web technologies were used for automatic transformation of IEC 61131-3 based control systems to IEC 61499 based ones [92]-[94]. The whole process starts from importing IEC 61131-3 source code files into the IEC 61131-3 ontological knowledge base, then maps the ontology between IEC 61131-3 and IEC 61499 knowledge bases and ends by generating IEC 61149 target code files from the IEC 61149 ontological knowledge base. The key part is ontology mapping, which includes mapping IEC 61131-3 resources to IEC 61499 devices, mapping IEC 61131-3 tasks to IEC 61499 resources, mapping IEC 61131-3 programs with functions/FBs to IEC 61499 CFBs, mapping IEC 61131-3 programs without functions/FBs to IEC 61499 BFBs, mapping IEC 61131-3 SFC programs to ECCs inside IEC 61499 BFBs, and mapping IEC 61131-3 FBD programs to IEC 61499 CFBs. Formal IEC 61131-3 models are defined so that correct execution semantics is recreated in IEC 61499 models.

### *D. Communication Paradigms*

One of key elements to realize system transformation from IEC 61131-3 to IEC 61499 is the communication paradigm [95]. In IEC 61499, two paradigms, i.e., publish/subscribe for unidirectional communication and client/server for bidirectional communication, are defined. The publish/subscribe model is based on the n-to-n architecture in which one publisher can send messages to one or more subscribers and one subscriber can receive messages from one or more publishers. The client/server model is based on the n-to-1 architecture in which one or more clients communicate with one server in both sending and receiving messages. In IEC 61131-5, there are also some communication paradigms, e.g., programmed data acquisition and

interlocked control, for defined functions or FBs [96]. Campanelli *et al.* proposed an architecture model LowEffort-INTEgration (LE-INT) for coexistence of IEC 61131-3 and IEC 61499 in the same engineering environment [97]-[98]. The architecture model is based on SIFBs in IEC 61499 and specific FBs from programmed data acquisition (i.e., *USEND*, *URCV*, *BSEND*, and *BRCV*) and interlocked control (i.e., *SEND* and *RCV*). The architecture model is more effective in modular automation and control systems to realize two types of system integration: a) transformation of a centralized IEC 61131-3 system or several independent ones to a distributed system based on both standards; and b) insertion of one or more IEC 61131-3 systems in an existing distributed IEC 61499 system.

#### *E. Discussion*

In conclusion, recent research has focused on applying a variety of approaches (e.g., model-driven, object/class-oriented, and ontology-based approaches, and communication paradigms) to model system transformations from IEC 61131-3 to IEC 61499. From the research aspect, work in this area is concerned with redesigning components/systems, translating codes/models, and the syntax level using XML-based model-driven approaches to the semantic level using knowledge-based ontology-driven methods. The main differences are how and what they use to describe models and to what degree. From the industry aspect, research on system redesign provides alternative solutions and feasible guidelines for wide industry adoption of IEC 61499. Industrial partners can either reuse their accumulated knowledge (e.g., generic problem/solution templates, best practices and guidelines) or can have enough time/effort for a smooth transition preparing for new challenges in the Industry 4.0 era. Typical cases for example are CPS-izer research from Daedalus and nxtControl implementation from Schneider, both supporting hybrid system design (co-existence of IEC 61131-3 and IEC 61499).

### **3.5 Integration with IEC 61499 Enabling Technologies**

Industrial automation is envisioned to be realized through iCPS that are built from and depend on the integration and interaction of computational and physical components [2]. The iCPS paradigm together with its enabling technologies (e.g., service-oriented architecture [99], autonomic computing [100], and cloud computing [101]) is transforming the way human, machine, information, and environment interacting with each other, leading to Industry 4.0. This section will focus on how IEC 61499 has been integrated with its enabling technologies for distributed intelligent automation. Two perspectives are provided: a) design paradigms including object-oriented design, component-based design, and service-oriented architecture (Section 3.5.1); and b) computing paradigms including distributed intelligence, autonomic computing, and cloud computing (Section 3.5.2).

#### ***3.5.1 Design Paradigms for Modelling IEC 61499 Based Systems***

##### ***A. Object-Oriented Design***

The object-oriented design (OOD) applies object-oriented programming features (e.g., inheritance, instantiation, encapsulation, and polymorphism) to design industrial control programs and automation applications. In OOD, data structures are modelled based on interacting objects which may contain data fields (i.e., attributes) and code procedures (i.e., methods). Objects are instances of their classes which define data formats and available procedures.

IEC 61499 FBs have some object-oriented features, for example, mapping types with classes, instances with objects, events with methods, and data with parameters/variables. FBs encapsulate data structures and internal algorithms, and can be instantiated working copies by

type definitions. FBs are triggered by input events with available data, implemented through control algorithms in BFBs or service sequences in SIFBs, and finally updated with events/data outputs. Since CFB is a network of FB instances, data exchange among its composited FBs is through publish/subscribe SIFBs [102]. Polymorphism and inheritance are not often used in automation programming due to issues raised by computation cost and execution determinism [103], except that adapters provide a kind of inheritance for similar FBs to share common interfaces.

Vyatkin *et al.* proposed a conceptual OOD framework for modelling automation software based on IEC 61499 for potential benefits of intellectual property encapsulation and reuse [104]. Dai and Vyatkin proposed an object-oriented approach, including conversion of PLC code into an ECC and reuse of PLC code in an algorithm, to redesign distributed PLC control systems using IEC 61499 FBs [90]-[91]. Two cases of modern building management systems [105] and airport baggage handling systems [102] were studied by using OOD to model IEC 61499 based system architectures.

### *B. Component-Based Design*

The component-based design (CBD) utilizes coarse-grained and loose-coupled components with certain well-defined functions and pre-defined communication interfaces from a cohesive set of fine-grained objects. Compared with OOD, CBD models a system with functional components rather than physical objects; multiple functions share a single algorithm with one generic event input instead of using dedicated events and algorithms for each method call [102].

Proposed frameworks for modelling component-based distributed automation systems are mainly based on the automation component/object (AC/AO) concept and then toward intelligent control [106]-[109]. AC is an attempt to generalize the FB concept to represent a modelling unit



of hardware and software for the performance of automation and control functions [104], [110]-[112]. In general, BFBs can be considered as software components, whereas SIFBs and CFBs cannot if without well-described interfaces and behaviours [85]. For other elements (e.g., applications, resources, and devices) may not be considered as software components, either [85].

Black and Vyatkin proposed a component-based architecture of an embedded intelligent control implementation with IEC 61499 [113]. The key parts of the architecture are reusable intelligent software components encapsulated in IEC 61499 FBs, especially the introduction of simulation components for predictive behaviours. The proposed architecture is scalable, reconfigurable, and fault tolerant, and paves the way to self-configuration. Dai and Vyatkin proposed a multi-layer component-based design pattern for improved reusability of distributed automation programs, including low-level basic control and interface layer, service layer, and high-level intelligent control layer [114]. Zoitl and Prähofer proposed design guidelines and patterns for building hierarchical automation solutions with IEC 61499, in which two concepts were focused for hierarchical component architectures: adapters and SubApp [115]-[116]. The IEC 61499 adapters models (i.e., typed interface accepting adapters *plug* and typed interface providing adapters *socket*) are in the form of FBs functioning as interfaces to hub input/output events and data [115]-[116]. The IEC 61499 SubApp model is a means to group application components in the top-down/bottom-up manner and share their common public interfaces [115]-[116]. Compared to typed CFB models which new types are created during each adaptation of applications, the IEC 61499 SubApp model supports application adaptation on all hierarchic levels for reuse and configuration which is much faster for application development and structure modelling [115]-[116].

### *C. Service-Oriented Architecture*

The service-oriented architecture (SOA) paradigm approaches software system design as a network of loose-coupled and discoverable services with formal interfaces communicating through messages [99]. A systematic literature review of SOA research on IEC 61499 based industrial automation systems is provided in [33]. Therefore, Section 3.5.2 will focus on recent research on SOA with computing paradigms for IEC 61499 based iCPS.

### *D. Discussion*

From a broad perspective, OOD, CBD, and SOA are closely related. Designs are modelled through IEC 61499 FBs with mapping, creation, composition, and execution of FBs as objects/components/services on different modelling levels. By evaluating these studies, common features of proposed methods can be summarized as: a) multi-layer or service-oriented architecture is employed; b) communication or interface/adaptor design is focused; c) reconfiguration, reuse, and flexibility is aimed. Theoretically, IEC 61499 adopts object-oriented programming features in designing control programs and automation applications. For distributed system architecture modelling, component-based architectures with a higher level of abstraction are commonly applied in practice to incorporate system design, simulation, and validation. SOA is widely adopted in automation system design for advanced capabilities, e.g., autonomy and interoperability, due to rapid development of computing and networking technologies. Therefore, research on this aspect is tightly concerned with computing paradigms and iCPS in Industry 4.0. That, as a result, requires emphasis on interfaces or adapters design in IEC 61499.

### ***3.5.2 Computing Paradigms for Modelling IEC 61499 Based Systems***

#### ***A. Distributed Intelligence***

Distributed intelligence is a major step for distributed and intelligent automation, usually realized through multi-agent systems (MAS). With this computing paradigm, distributed and intelligent automation is commonly achieved through autonomous and cooperative agents that are capable of operating independently or in collaboration with others to respond to system requests or changes and to achieve individual or shared goals [117]-[118]. Multi-agent modelling plays a key role in the development of complex industrial automation systems, allowing a decentralized way to design distributed and intelligent systems [117]-[119]. This approach is being applied in several domains of industrial applications: e.g., factory and building automation, power and energy systems [120]-[121].

Integration of intelligent software agents with low-level control functions provides a promising way to design distributed automation systems; however, real-time adaptation is a great challenge at this level given real-time constraints [119], [122]. Incorporating with IEC 61499 FBs, research has focused on real-time distributed control for dynamic and intelligent reconfiguration, including reconfiguration models, implementation architectures, software platforms and evaluation methods [23], [118], [123]-[128]. For example, a three-layered FB and agent-based model for dynamic and intelligent reconfiguration of real-time distributed control systems was proposed, in which the architecture includes IEC 61499 FB models for low-level real-time control, mobile agents to model middle-level monitoring and activation systems, and software agents designed for high-level planning, scheduling, and configuration [123]. A reconfigurable concurrent FB model was also proposed to separate two control paths: IEC 61499 FB modelled control application execution and multi-agent modelled configuration control

operation [124]. One recent research in this area is applying wireless sensor networks (WSN) to model distributed intelligent sensing and control systems. For example, Cai *et al.* proposed an application-oriented middleware architecture (AoMA) for distributed intelligent sensing and control of industrial WSN through MAS and IEC 61499 FBs [129]-[130]. Three main agents are designed to facilitate upper-level management, including device management agent, service mapping agent, and task management agent. IEC 61499 FBs are used as modelling tools for lower-level implementation, including node intelligence in BFBs, data acquisition/hardware control in SIFBs, and modular tasks in CFBs.

Khalgui *et al.* proposed an architecture of reconfigurable multi-agent systems for IEC 61499 based distributed control systems [131]. Two types of agents implemented in extensive markup languages are provided: reconfiguration agents modelled by nested state machines for local automatic reconfiguration and coordination agents defined by coordination matrices and communication protocols for managing reconfiguration behaviours. Guellouz *et al.* proposed a reconfiguration FB approach which is a series of guidelines in the design, modelling, and verification of IEC 61499 FB based reconfiguration control systems [132]. The key idea is to propose a new design pattern reconfiguration FB defined as event-triggered software components to control and execute reconfiguration tasks. As a result, compared to IEC 61499 FB model with input/output control events and execution control charts, it adds reconfiguration events to the interface and the master-slave execution control chart to define reconfiguration functions.

Bonci *et al.* proposed a relational-model multi-agent system (RMAS) architecture that focused on multi-agent systems: the goal of this approach is to develop IEC 61499 FB based distributed intelligent applications to realise self-manageable iCPS in industry 4.0 [133]. The

RMAS architecture was first proposed to implement multi-agent systems for modelling, simulation, and control of iCPS. The architecture is designed on data-centric, event-based, and publish-subscribe paradigms and the core of the architecture is a generic agent skeleton structure that contains the genotype of RMAS units which in essence are relational active database management systems. Then the resulting architecture was proposed to serve as a middleware architecture for autonomic computing that will enable iCPS with self-management capabilities [134]. Recently, the RMAS architecture was further analysed to match the IEC 61499 reference models, focusing on integration of RMAS with IEC 61499 FB model, resource model, and device management model [135]-[136].

Furthermore, MAS and SOA are considered as key enabling technologies to model IEC 61499 based iCPS with cloud and autonomous computing capabilities [77], [120], [137].

### *B. Autonomic Computing*

In previous chapters the evolution of iCPS in Industry 4.0 characterized as distributed and intelligent to be able to self-manage were discussed. IEC 61499 based iCPS are not only designed with fundamental features (e.g., distributed to be flexible, configurable, portable, and interoperable) but also envisioned for advanced capabilities (e.g., learning abilities, self-managing capabilities). Learning abilities to perform intelligent behaviours require iCPS to be self-manageable with flexible architectures (e.g., hardware and software) and adaptable strategies (e.g., rules and knowledge). The goal is to support real-time self-configuration, self-healing, self-optimization, and self-protection for responsiveness to changes [23], [100], [118], [138]. It is summarized as: a) *self-configuration* of configuring and reconfiguring functions, structures, and processes to adapt to dynamical changes; b) *self-optimization* of improving and optimizing performances and operations with respect to predefined goals; c) *self-healing* of

detecting and recovering from disturbances and faults to maximize system availability; and d) *self-protection* of identifying and protecting against safety and security attacks to preserve system integrity [100].

Recent research has been focused on system autonomic service management modelling with consideration of IEC 61499. Mubarak and Göhner proposed an agent-based architecture for self-manageable industrial automation systems in which self-healing is illustrated by an example of passenger lift design [139]. The proposed architecture is developed with agents deployed on three levels: the control and supervision level, the self-management functionality level, and the automation system connection level. Lepuschitz *et al.* proposed an automation agent architecture for low-level control (LLC) self-reconfiguration of IEC 61499 based applications [140]. The approach is to use ontological representation of low-level functions on the high-level control (HLC) to enable HLC to reason and initiate reconfiguration processes for LLC. Strasser and Froschauer discussed a general concept for autonomous recovery of applications in IEC 61499 based intelligent automation and control systems [141]. The proposed framework facilitates the exchange of hardware components with no need for extra configuration. Kaindl *et al.* proposed an agent-based architecture containing self-representation for automation systems to support self-configuration and monitoring [142]. The approach is based on the concept of automation agents composed of hardware and software components. In software components, the real-time LLC is implemented using IEC 61499 FBs and the high-level control is for agents' configuration, monitoring, and communication.

Dai *et al.* proposed a service-oriented execution environment architecture, i.e., function block service runtime (FBSR), to support the SOA-based design of the IEC 61499 model [143]. For software services in iCPS, detailed definitions of common interfaces and feasible solutions

of dynamical discovery were further developed [144]. Inspired by autonomic computing, Dai *et al.* proposed a knowledge-driven autonomic service management architecture for self-optimization of resource utilization [77] and a cloud-based decision support system for self-healing in distributed automation systems using fault tree analysis [137]. FBSR was then extended to introduce concurrent models of computation for modelling distributed automation systems in the iCPS view [145] and used to test a new feature, i.e., real-time data acquisition support for IEC 61499 based iCPS, to monitor and optimize industrial processes with real-time feedback data [146].

### *C. Cloud Computing*

Recently cloud computing has emerged as a new computing paradigm for iCPS [101]. As defined by the National Institute of Standards and Technology (NIST), cloud computing enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction [147]. Cloud computing employs a multi-layer architecture including application, platform, infrastructure, and hardware layers, and is realized through different service models including software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS) [147]. For modelling iCPS, integration of the cyber (i.e., cloud) and the physical (i.e., devices) are enabled by encapsulating services in design entities (e.g., IEC 61499 FBs). Furthermore, to realize distributed and intelligent industrial automation, the capability of computing plays a critical role. Cloud computing provides a promising solution to model both system architectures and computing resources.

Karnouskos *et al.* proposed a SOA-based architecture for empowering future collaborative cloud-based industrial automation [148]. They envisioned that future industrial

automation systems would be virtualized as cloud-based composition of cyber-physical services for multi-system interaction and cross-layer collaboration in both architectures and behaviours. Dai *et al.* proposed a configurable cloud-based validation environment for interoperability tests between various distributed automation systems from the bottom protocol level to the top system level [149]. The testing framework is implemented in a multi-layer infrastructure in which testing models are designed as IEC 61499 FBs. Demin *et al.* proposed a cloud-based framework for designing an IEC 61499 based application as a web service in SOA [150]. SOA and cloud computing make it possible to convert FBs to services and then deploy them to the cloud for as needed use.

To support IEC 61499 with runtime monitoring, behavioral types as extensions to IEC 61499 were proposed in [151]. Compared with traditional types (e.g., strings and integers) which model interfaces on a syntactic level, behavioral types extend the expressiveness of interface specifications by adding regular expression-based, protocol-like usages of components [151]. This work was further developed into a cloud-based monitoring framework to check timed properties described as behavioral types of IEC 61499 based industrial automation systems [152]. More features (i.e., *Event*, *Var*, and *Watch*) were proposed to support both publish/subscribe and client/server models for IEC 61499 based iCPS [146].

#### *D. Discussion*

In conclusion, the key idea of the above research is to introduce distributed intelligence, cloud computing, and autonomic computing frameworks combined with SOA in the design modelling of iCPS to support flexibility and interoperability and to realize self-management capabilities. Research on design paradigms focuses on how to model IEC 61499 based systems while research on computing paradigms focuses on how intelligent IEC 61499 modelled systems will



be. More specifically, industrial multi-agent solutions (e.g., [153]) allow distributed intelligence which means decentralized architectures and inherent capabilities for system self-adaption to changes. Usually, multi-layered architectures are employed with high-level industrial agents and low-level automation devices for modelling iCPS. One issue is the communication between the high-level and the low-level. Cloud computing allows design entities encapsulated as services and deployed into the cloud. Most IEC 61499 research on this aspect is focused more on cloud-based system design and modelling; studies on sharing computing resources in the cloud (e.g., local intelligence dynamically linked to remote runtime functionalities in the cloud for sharing [75]) is as important towards distributed intelligent automation. Rather than accessing computing resources completely in the cloud, low-level intelligence can be achieved for real-time response and control through agent-embedded devices (e.g., node intelligence in WSN [130]). Autonomic computing enables system self-managing capabilities which are key components towards distributed intelligent systems. Current research mainly focuses on part of them theoretically, i.e., self-configuration, self-healing, and self-optimization. Design tests and implementation cases are still required for validation.

## 3.6 Implementation of IEC 61499 Engineering Environments

### 3.6.1 Development of IEC 61499 Engineering Environments

Since the publication of the IEC 61499 standard, academic activities and industrial practices on developing engineering environments to implement IEC 61499 FB models have been conducted.

Typical projects for IEC 61499 engineering environments development are listed in Table 3-6.

Table 3-6: Projects of developing IEC 61499 engineering environments

Developer	Product	Type <sup>1</sup>	Grade <sup>2</sup>	Capability <sup>3</sup>	Status	Technology	Comment
Schneider nxtControl [69]	nxtSTUDIO nxtIECRT nxtLIB nxtHMI	ST RP LB RP	IND	P; I; C.	Active	Microsoft .NET Framework; XML DTD; ST.	Comprehensive industrial solution packages; Hardware independent engineering.
Rockwell ISaGRAF [70]	Workbench Runtime ISaVIEW	ST RP RP	IND	C	Active	Microsoft Visual Studio Shell; Virtual Machine.	First commercial software environment.
Eclipse 4diac [79]	4diac IDE 4diac FORTE 4diac LIB	ST RP LB	OPS	P; I; C.	Active	Eclipse Framework; C++.	Open source solutions.
Holobloc [82]	FBDK FBRT	ST RP	IND	P; I; C.	Active	Oracle Java SE Platform; XML DTD.	First IEC 61499 feasibility demonstration.
Automation of Things [154]	FourZero™ Runtime FourZero™ Studio	RP ST	IND	C	Active	C++; 4diac FORTE	Distributed and task-oriented architecture; Hardware and topology independent program; Real and virtual application creation.
Yueyi Automation [155]	FBB FBSRT FBDL	ST RP RP	IND	C	Active	Microsoft .NET Framework; ST, LD; C++; HTML/JavaScript.	Service based runtime and dynamic reconfiguration; Real-time monitoring and data management.
NOJA Power [156]	SGA	ST	IND	P; I; C.	Active	Eclipse 4diac Framework; User Defined Analogue; Dynamic Data Types.	Interaction with IEC 61850 <i>et al.</i> ; System access to database; Query and control IEC 61499/SGA devices.
PRETZel [157]	BlokIDE	ST	ACA	C	Active	Microsoft VS 2010/2013	Synchronous execution; Formal verification; Static timing analysis; Highly efficient code.
O <sup>3</sup> neida [158]-[159]	Workbench FBench	ST	OPS	C	Inactive	Java; NetBeans; Eclipse.	Experimental use for Automation Objects.
Fuber [160]	FUBER	RP	OPS	C	Inactive	Java; BeanShell.	An IEC 61499 interpreter.
SEG [161]	CORFU Archimedes	RP	ACA	C	Inactive	Unified Modelling Language; Model Integrated Mechatronics.	An IEC 61499 runtime embedded tool.
UDESC [162]-[163]	ICARU_FB GASR-FBE	RP ST	OPS	P; I; C	Inactive	XML DTD.	Dynamic reconfiguration; Code simplicity.

<sup>1</sup> ST: Software Tool; RP: Runtime Platform; LB: Library of software components.

<sup>2</sup> ACA: Academic; IND: Industrial; OPS: Open Source.

<sup>3</sup> P: Portability; I: Interoperability; C: Configurability. Not formally tested because of license issues.

Engineering environments usually includes three components [28], [32]: a) software tools (ST), i.e., an integrated development environment to model designs; b) runtime platforms (RP), i.e., a runtime environment to execute programs; c) libraries of software components (LB), i.e., a

library to store elements. Furthermore, these implementations can be classified into three categories: a) IEC 61499 based, e.g., Holobloc FBDK/FBRT; b) IEC 61131 based but IEC 61499 supported, e.g., ISaGRAF Workbench/Runtime; and c) IEC 61499 and/or IEC 61131 based (i.e., hybrid), e.g., nxtControl nxtSTUDIO/nxtIECRT. As proposed in IEC 61499-4, three key features are expected in developing those IEC 61499 engineering environments [39]: a) configurability, i.e., multi-source devices can be manipulated by multi-source software tools; b) portability, i.e., multi-source libraries can be used among multi-source software tools; and c) interoperability, i.e., multi-source devices can be exchanged among multi-source runtime platforms.

A brief description of these engineering environments is as follows. The Schneider nxtControl includes a) nxtSTUDIO to integrate automation tasks, b) nxtLIB to offers prefabricated software objects, c) nxtIECRT to support hybrid control paradigms, and d) nxtHMI together with SCADA to enable multi-client/multi-server visualization [69]. The Rockwell ISaGRAF includes a) ISaGRAF Workbench to provide plug-in functions, b) ISaGRAF Runtime to execute target independent code generated by control applications, and c) ISaVIEW as a plug-in for HMI [70]. The Eclipse 4diac includes a) 4diac IDE based on the Eclipse framework, b) 4diac FORTE supporting online reconfiguration of applications and real-time execution of FB types, and c) 4diac LIB containing FBs, adapters, and sub-applications [79]. The Holobloc FBDK/FBRT were developed to support fundamental features of IEC 61499 based on design patterns (e.g., proxy, local multicast, tagged data, time-stamped messaging, model/view/controller/diagnostics [164]) [82]. The FourZero™ platform developed by Automation of Things includes a) FourZero™ Studio and b) FourZero™ Runtime with key features, for example, distributed and task-oriented architecture, hardware and topology independent programming, and real-time monitoring and management [154]. The platform

developed by Yueyi Automation is based on the core model FBSR [143]. It includes a) data link FBDL, b) execution environment FBSRT, and c) FB builder FBB for real-time monitoring, dynamic reconfiguration, and data management [155]. The IEC 61499 design toolset Smart Grid Automation (SGA) from NOJA Power is based on Eclipse 4diac framework and focused on distributed power system automation applications in smart grid [156]. The PRETzel BlokIDE is academic software tool based on several research results, e.g., research on hierarchical and concurrent execution control chart (HCECC) for IEC 61499 [66] and efficient C code generation from IEC 61499 FBs [54], [165]. It is a design environment for model-driven engineering of programmable electronics integrated with IEC 61499 to allow automatic code generation, synchronous execution, formal verification, and static timing analysis [157]. Other projects are not currently active now but still valuable reference implementation [158]-[163].

### ***3.6.2 Application of IEC 61499 Engineering Environments***

Applications of IEC 61499 engineering environments for industrial practices or academic experiments are common now (Table 3-7). In general, IEC 61499 has achieved successes in some typical domains, e.g., smart factory, smart building, smart grid, as envisioned in [30]. For smart factory, examples are design of IEC 61499 based control systems using ISaGRAF Workbench for shoe manufacturing plants [166] and using Eclipse 4diac IDE for Pick & Place stations [79]. Another typical example in current research is the design modelling of airport baggage handling systems using Holobloc FBDK/FBRT (e.g., [102], [113]). For smart building, nxtControl nxtSTUDIO is used to design IEC 61499 based building management systems for energy-efficient lighting system control (e.g., [105], [167]). For smart grid, solutions based on the combination of IEC 61499 FB implementation and IEC 61850 interoperable communication are researched by groups of Vyatkin *et al.* using Holobloc FBDK (e.g., [168]) or nxtControl

nxtSTUDIO (e.g., [169]), and groups of Strasser *et al.* using Eclipse 4diac IDE (e.g., [170]). An industrial application is NOJA Power’s Automatic Circuit Reclosers (ACR) running applications developed by its IEC 61499 design toolset SGA [156]. Recently, the standards working group of Open Process Automation (OPA) Forum is examining the IEC 61499 standard, and ExxonMobile, as a member of OPA Forum, is establishing the test bed to evaluate candidate components and standards including IEC 61499 for distributed FB applications [171].

Table 3-7: Applications of typical IEC 61499 engineering environments

No.	IEC 61499 Application	Software
1	Meat processing plant and fertilizer production plant [172].	Holobloc
2	Airport baggage handling systems [113].	
3	Smart grid automation through IEC 61850/IEC 61499 logical nodes [168].	
4	Design of the control system of the transport line in a shoe manufacturing plant [173].	
5	IEC 61499 based distributed control and IEC 61850 based automation for smart grids [79].	Eclipse 4diac
6	The Pick & Place station for the design of IEC 61499 compliant control applications [79].	
7	Control engineering for heating, ventilation and air-conditioning, lighting control [69].	Schneider nxtControl
8	Fertilizer production plant [69].	
9	Food processing embedded machine control [174].	Rockwell ISaGRAF
10	Research center data acquisition and control on a drying test bench [174].	
11	Control of hydraulic parameters of district heating region “Zemliane” in Sofia [175].	
12	High-speed train monitoring and control [176].	
13	Railway safety functions in the mining transport system [177].	
14	I-8000 wastewater treatment system [177].	
15	Adaptive automation control for customized shoes manufacturing [166].	

In conclusion, IEC 61499 has gained more popularity in academia and industry since the second edition published in 2012. Various IEC 61499 engineering environments have been developed to support IEC 61499 applications in a variety of domains. Capabilities of each IEC 61499 engineering environments may vary but each serves an important role in promoting IEC 61499 for distributed automation, especially in smart factory, smart grid, and smart building areas. These IEC 61499 engineering environments such as Eclipse 4diac kit are also a critical component in hands-on training programs in teaching and learning. One great thing to mention is that Schneider nxtControl includes a new module nxtServices in its kit to provide know-how services of IEC 61499 projects.

### 3.7 Summary

In this chapter, major topics of research on IEC 61499 were reviewed. First, an overview of the IEC 61499 standard focusing on its development background, proposed reference architecture models, and FB execution semantics was reviewed. Then, challenges and methods of transforming existing IEC 61131-3 programmed systems to IEC 61499 based systems were discussed. By analyzing recent research on integration with IEC 61499 enabling technologies, perspectives of design methods (i.e., object-oriented design, component-based design, and service-oriented architecture) and computing frameworks (i.e., distributed intelligence, autonomic computing, and cloud computing) for modelling IEC 61499 based systems were provided. This thesis is based on studies in this section and further developed to propose a framework to model self-manageable iCPS for IEC 61499 distributed intelligent automation. At the end of this chapter, several implemented IEC 61499 engineering environments were listed, in which Eclipse 4diac will be used for experiments in the research. The reason to choose Eclipse 4diac is that it is now the most popular, comprehensive, and capable engineering environment with active and open-source features (i.e., free for academic use), required elements (i.e., ST, RP, and LB) and capabilities (i.e., C, P, and I) defined by IEC 61499.

*[This page intentionally left blank]*

## **Chapter Four: Architecture Modelling Framework**

### **4.1 Introduction**

As stated in previous chapters, industrial cyber-physical systems (iCPS), in which cyber and physical components collaborate with each other and are empowered for intelligence by communicating and computing cores, appears to hold the most promise of achieving next-generation industrial automation systems to be distributed and intelligent in the Industry 4.0 era. In this thesis, the research question is how to achieve self-manageable iCPS for IEC 61499 based distributed intelligent automation. More specifically, how to model such type of systems that are responsive to frequent changes and adaptive to evolving requirements in a distributed and intelligent way through integration of multi-agent modelling and IEC 61499 FB modelling. Thus, the core objective of the research is to explore the design of multi-layer automation architectures to enable real-time adaptation at the device level and run-time intelligence throughout the whole system under a well-integrated modelling framework.

In this chapter, a multi-layer architecture modelling framework will be explored and the following chapters will focus on part of this modelling framework, that is the design of a two-layer self-manageable architecture modelling for distributed and intelligent automation.



## 4.2 Modelling Framework

The legacy industrial automation systems are typically designed as a 5-level architecture: a) low levels 0, 1, and 2 focus on industrial automation control and monitoring by applying, e.g., sensors and actuators (Field Devices, Level 0), programmable logical controllers and distributed control systems (PLC/DCS, Level 1), and supervisory control and data acquisition systems (SCADA, Level 2); b) high levels 3 and 4 focus on manufacturing and enterprise operations management and decision-making support by applying, e.g., manufacturing execution systems (MES, Level 3) and enterprise resource planning (ERP, Level 4) [178]. As envisioned in Industry 4.0, the development of next-generation industry systems will be leveraged by the integration of industrial cyber and physical systems (iCPS) and the application of industrial internet of things and services (IIoTS), both of that are enabled by industrial computing and communicating technologies and powered by artificial intelligence and data analytics. In this thesis, these industrial computing and communicating frameworks will be applied to form a feasible design of multi-layer automation architectures to enable real-time adaptation at the device level and run-time intelligence throughout the whole system under a well-integrated modelling framework.

The proposed multi-layer system architecture modelling framework is shown in Figure 4-1. The macro architecture (Figure 4-1a) is designed as a multi-layer model by deploying cloud computing [101], fog computing [179], and edge computing [180] into the framework. The micro architecture (Figure 4-1b) is designed as a multi-layer model by applying multi-agent based autonomic computing and agent-embedded IEC 61499 FB modelling to the framework [11]-[15]. In the following sections, a brief introduction to the proposed architecture modelling framework will be provided.

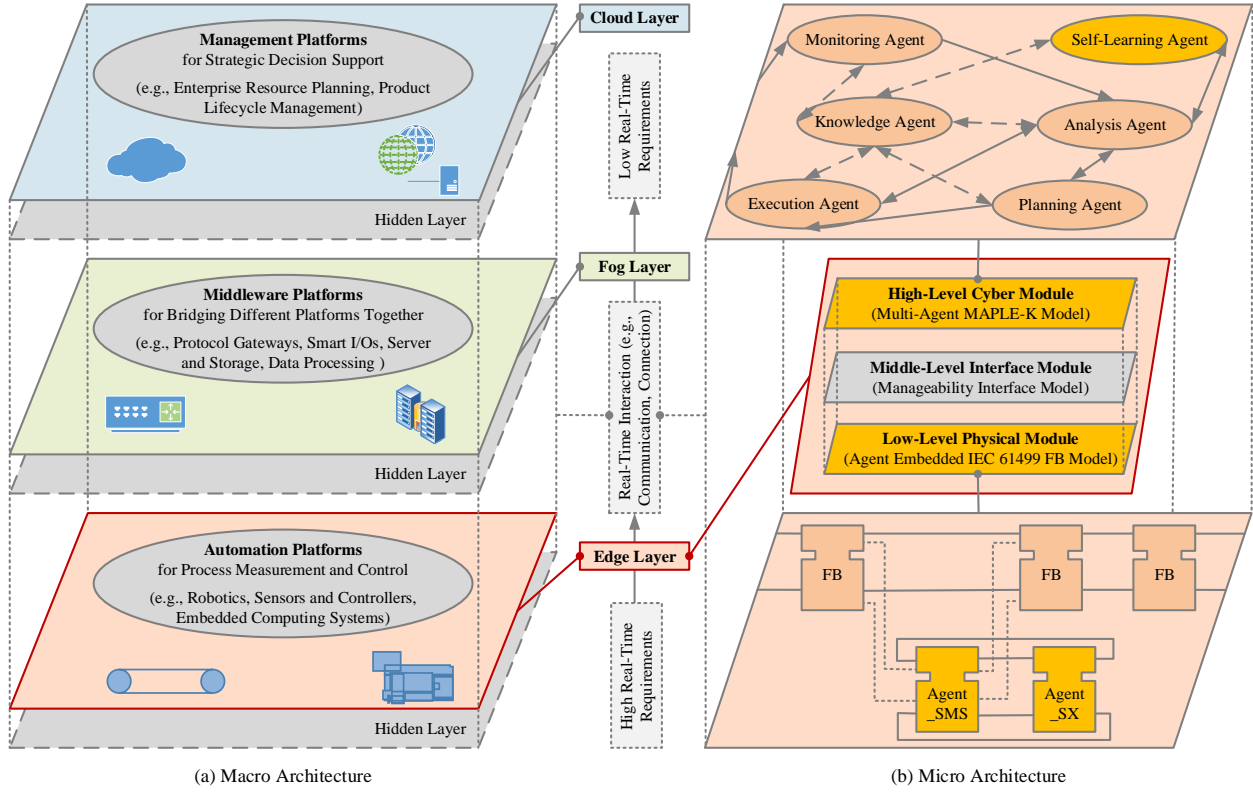


Figure 4-1: Multi-layer system architecture modelling framework

#### 4.2.1 Multi-Layer Macro Architecture

The multi-layer macro architecture is designed with three key layers, i.e., *Cloud Layer*, *Fog Layer*, and *Edge Layer*, in distributing intelligence from top to bottom across the whole system. As it is not the focus of the research and some key concepts have been reviewed before, this section will provide a general ideal of the multi-layer macro architecture.

##### A. Cloud Layer

*Cloud Layer* is a network of cloud computing enabled management platforms for strategic decision support. Legacy product lifecycle management platforms (e.g., ERP, MES) can either be migrated to industrial clouds (e.g., public cloud, cooperate cloud) as accessible services, or furthermore, be powered with advanced computing platforms (e.g., plug-in modules like

machine learning tools) for enterprise-wide management, planning, and optimization. This layer mainly relies on powerful data centers to deal with high-volume, historical, advanced data processing and analysis, aiming at complex pattern detection to provide optimal solutions for mid- to long- term decision-making support.

### *B. Fog Layer*

*Fog Layer* is a network of fog computing enabled middleware platforms for bridging different platforms together. Middleware platforms (e.g., protocol gateways, smart I/Os, servers and storage) play a crucial role in connecting different platforms and optimizing their communications. This layer mainly works on balancing local computing, communication, and storage resources to deal with mid-volume, lightweight, streaming data preprocessing and analysis, aiming at providing latency acceptable, solution reasonable, and near real-time responses.

### *C. Edge Layer*

*Edge Layer* is a network of edge computing enabled automation platforms for process measurement and control. Automation platforms are home to front-end devices (e.g., sensors, actuators, controllers) with embedded computing intelligence (e.g., single board computers like Jetson Nano and Raspberry Pi) and available communicating connectivity (e.g., WiFi and Zigbee). This layer mainly focuses on built-in capabilities to deal with low volume, raw streaming data preprocessing and analysis, aiming at providing real-time adaptation for self-management. This thesis focuses on *Edge Layer* and will explore a detailed design of this layer architecture modelling.

#### *D. Hidden Layers*

*Hidden Layers* designed in the proposed architecture modelling framework represent other possible layers that are either a new emerging system layer to envision a future system architecture, or more detailed layers of an existing system layer.

#### **4.2.2 Multi-Layer Micro Architecture**

The multi-layer micro architecture is a detailed layered system architecture of *Edge Layer* in the multi-layer macro architecture. The multi-layer micro architecture is designed with three layered modules, i.e., high-level cyber module, middle-level interface module, and low-level physical module. The multi-layer micro architecture design deploys autonomic computing in the architecture modelling framework, including: a) the reference architecture employed in the high-level cyber module and implemented as multi-agent systems, and b) the self-managing properties employed in the low-level physical module and implemented as agent-embedded IEC 61499 FBs. The design results in a new computing module for high-level multi-agent based automation architectures and a new design pattern for low-level function block modelled control solutions. The objective is to achieve multi-agent enabled, IEC 61499 FB based distributed intelligent automation and control.

##### *A. High-Level Cyber Module*

The high-level cyber module design deploys autonomic computing reference architecture into the modelling framework with the implementation of multi-agent modelling techniques. The core of the reference architecture for autonomic computing is structured by different modules, including autonomic managers of five architectural elements *Monitoring*, *Analysis*, *Planning*, *Execution*, and *Knowledge* (i.e., MAPE-K) as an intelligent control loop, managed resources of software or hardware entities (e.g., databases, networks, and applications), and touchpoints of

sensors and effectors for monitoring and controlling managed resources by autonomic managers [100]. Furthermore, the architectural element *Self-Learning* is introduced in this research to the traditional autonomic computing reference architecture. For the architecture implementation, multi-agent modelling techniques are applied by using autonomous and cooperative agents to achieve run-time distributed intelligence in system design and module reconfiguration.

Therefore, the high-level cyber module is designed as multi-agent computing model (Figure 4-1b, top) consisting of *Monitoring Agent*, *Analysis Agent*, *Self-Learning Agent*, *Planning Agent*, *Execution Agent*, and *Knowledge Agent*. An overview of these agents is shown as follows:

- *Monitoring Agent*: monitoring and collecting system operation data, engineering data, and operating environment data through sensors.
- *Analysis Agent*: pre-analyzing collected data for modeling complex situations to understand current system operations and to predict better future states.
- *Self-Learning Agent*: designed under open-source artificial intelligence frameworks to deploy various machine learning models and employ rich data analytics tools aiming at gaining insights of system operations.
- *Planning Agent*: selecting a series of action steps and generating an optimal action plan to respond to changes and to achieve goals.
- *Execution Agent*: implementing action plans and controlling execution processes through actuators.
- *Knowledge Agent*: maintaining data sets or knowledge repositories to provide support to and receive updates from other agents or entities.

### *B. Low-Level Physical Module*

The low-level physical module design deploys autonomic computing self-managing properties into the modelling framework with the implementation of IEC 61499 FB modelling techniques. The key system self-managing properties envisioned by autonomic computing are *self-configuration*, *self-healing*, *self-optimization*, and *self-protection* [100]. For the architecture implementation, IEC 61499 FB modelling techniques are applied by using object-oriented and event-driven function blocks to realize real-time adaption of automation logic and control algorithms. Furthermore, a new design pattern, i.e., agent-embedded IEC 61499 FB model, is proposed for self-manageable services with the separation of control application execution and self-manageable service agent execution.

Thus, the low-level physical module is designed as agent-embedded IEC 61499 FB model (Figure 4-1b, bottom) with *Self-Manageable Service Execution Agent (Agent\_SMS)*, *Self-Configuration Agent*, *Self-Healing Agent*, *Self-Optimization Agent*, and *Self-Protection Agent (Agent\_SX)*. An overview of these agents is shown as follows:

- *Self-Manageable Service Execution Agent*: monitoring system states and responding to changes by deciding the adequate behaviors to perform (i.e., activate one or more self-manageable agents and execute self-manageable services).
- *Self-Configuration Agent*: configuring/reconfiguring functions, structures, and process to adapt to dynamical changes.
- *Self-Healing Agent*: detecting and recovering from disturbances and faults to maximize system availability.
- *Self-Optimization Agent*: improving and optimizing performance and operations with respect to predefined goals.

- *Self-Protection Agent*: identifying and protecting against safety and security attacks to preserve system integrity.

### *C. Middle-Level Interface Module*

The middle-level interface module serves as middleware for communication and connection of the high-level cyber module and the low-level physical module. Depending on the implementation of the high-level cyber module and the low-level physical module, the design will vary [122]. Although not the focus of this research, communication and connection of agent-agent, agent-FB, FB-FB will be discussed in related chapters.

### 4.3 Summary

In this chapter, the architecture modelling framework was proposed. The proposed multi-layer system architecture modelling framework includes three key layers (i.e., *Cloud Layer*, *Fog Layer*, and *Edge Layer*) from the macro view. The modelling framework employs the three-level industrial computing framework with consideration of the traditional industrial system architecture. The thesis focuses on the multi-layer micro architecture modelling by detailing *Edge Layer* that is mainly responsible for industrial automation and control platforms. The micro architecture, under the vision of Industry 4.0 that leverages the integration of industrial cyber and physical systems (iCPS) and the application of industrial internet of things and services (IIoTS), is designed as a multi-layer model by applying multi-agent based autonomic computing and agent-embedded IEC 61499 FB modelling to the framework. The design results in a new computing module for high-level multi-agent based automation architectures and a new design pattern for low-level FB modelled control solutions. In the following chapters, details of the design of this two-layer architecture modelling will be discussed.



*[This page intentionally left blank]*

## **Chapter Five: High-Level Cyber Module Architecture Modelling**

### **5.1 Introduction**

In the proposed high-level architecture modelling framework (Figure 5-1), the cyber module is designed as multi-agent MAPLE-K model in which the intelligent control loop (i.e., the MAPLE-K loop) is performed. A detailed algorithm is shown in Table 5-1. Generally, the cyber module starts from current state monitoring of the operating environment properties, the system engineering properties, and the real-time operation behaviours of the physical module (e.g., through sensors); continues to data analysis by analyzing or self-learning intelligence by learning from the collected environmental, engineering, and operational data; and ends with action planning and execution for responses (e.g., through actuators); while the knowledge base provides support to and receives updates from the whole process. The cyber module can be proactive (i.e., actively collects data for analysis and builds models for prediction) or reactive (i.e., passively receives data for analysis and applies models for prediction). For example, sensor nodes in regular operating states, the cyber module could work in a reactive way to save energy or resources; whereas sensor nodes detect fluctuations in operation, the cyber module could work in a proactive way to obtain more information in order to respond to changes. In the following chapters, simple and complex situations will be used and they are different simply based on if regular system operation requires adaptation (e.g., change or new requirement request).

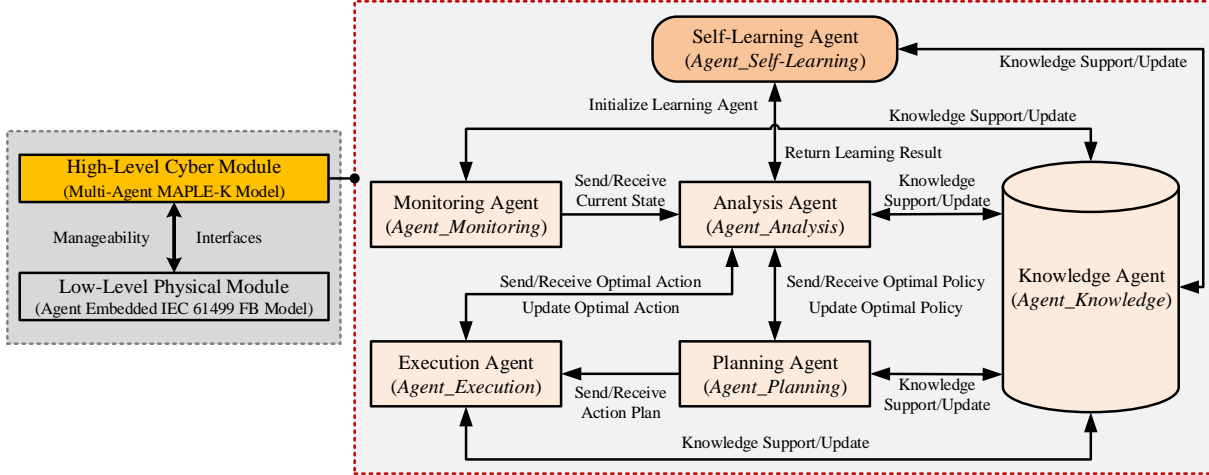


Figure 5-1: The proposed high-level architecture modelling framework

Table 5-1: Algorithm for the proposed multi-agent MAPLE-K model

Algorithm 1: Multi-Agent MAPLE-K model	
01	<b>Input:</b> Current State //system running status
02	<b>Output:</b> Action Plan //system updating plan
03	<b>Initialize</b> <i>Agent_Knowledge</i> //run upon request from other agents
04	<b>Invoke</b> <i>reasonEngine</i>
05	<b>Query</b> <i>knowledgeBase</i>
06	<b>Update</b> <i>knowledgeBase</i>
07	<b>Initialize</b> <i>Agent_Monitoring</i>
08	<b>Call</b> <i>SensorNode</i> for state perception
09	<b>Read</b> <i>currentState</i>
10	<b>Send</b> <i>currentState</i> to <i>Agent_Analysis</i>
11	<b>Initialize</b> <i>Agent_Analysis</i>
12	<b>Receive</b> <i>currentState</i> from <i>Agent_Monitoring</i>
13	<b>Call</b> <i>AnalysisDecision</i> for pre-analysis
14	<b>Compute</b> the comparison result of current and planned state/action
15	<b>Return</b> <i>decisionResult</i>
16	<b>If</b> decision result is positive <b>Then</b>
17	<b>Send</b> <i>optimalAction</i> in planned <i>optimalPolicy</i> to <i>Agent_Execution</i> directly
18	<b>Else</b>
19	<b>Call</b> <i>Agent_Self-Learning</i> for deep learning/reinforcement learning
20	<b>Return</b> <i>selflearningResult</i>
21	<b>Send</b> computed <i>optimalPolicy</i> to <i>Agent_Planning</i>
22	<b>End</b>
23	<b>Initialize</b> <i>Agent_Planning</i>
24	<b>Receive</b> computed <i>optimalPolicy</i> from <i>Agent_Analysis</i>
25	<b>Call</b> <i>PlanningDecision</i> for optimal action plan
26	<b>Update</b> computed <i>optimalPolicy</i>
27	<b>Send</b> <i>actionPlan</i> in updated <i>optimalPolicy</i> to <i>Agent_Execution</i>
28	<b>Initialize</b> <i>Agent_Execution</i>
29	<b>If</b> decision result in <i>AnalysisDecision</i> is positive <b>Then</b>
30	<b>Execute</b> <i>optimalAction</i> in planned <i>optimalPolicy</i> from <i>Agent_Analysis</i>
31	<b>Else</b>
32	<b>Execute</b> <i>actionPlan</i> in updated <i>optimalPolicy</i> from <i>Agent_Planning</i>
33	<b>End</b>

## 5.2 Monitoring Agent Design

The monitoring function provides the mechanisms that collect, aggregate, filter, and report data that represents the system's current state in either passive or active way [100]. In this design, *Agent\_Monitoring* collects data on the operating environment properties, the system engineering properties, and the real-time operation behaviours of the physical module through sensors, which is used by *Agent\_Analysis* for data analysis, model building, behaviour learning and prediction. Sensory data are classified into three types: the internal engineering properties (i.e., machine health like tear on parts), the external environment properties (i.e., environmental conditions like temperature), and the real-time operation behaviours (i.e., working states like efficiency), which represent three key sources from industrial systems.

Wireless sensor networks (WSNs) is one of the key enablers in the design of iCPS which are built of wireless networked sensor nodes (mainly composed of sensing, processing, transceiver, and power units) and are capable of distributed communication and intelligent control [181]. Sensor nodes are considered in the proposed architectural model as they can be deployed in the system as interfaces through which cyber and physical modules of iCPS can collaborate with each other to perceive system states, adapt to changes, and maintain its operation. For example, several autonomous mobile robots are added to the scenario described previously in Section 2.1 (Figure 2-1) to be responsible for carrying sorting bins to desired areas (Figure 7-1). Wireless sensors will be a good way to be attached for tracking these autonomous mobile robots and monitoring their working conditions. Sensor nodes are generally classified into regular sensor nodes and sink nodes [182]. For regular sensor nodes, those with fixed locations are referred to as anchor nodes whereas those without fixed locations are referred to as mobile nodes. Regular sensor nodes collect raw data and transmit to sink nodes before

transmitting to *Agent\_Analysis* for advanced data analytics. As raw data aggregators and processors, sink nodes have advantages over regular sensor nodes in aspects of the data processing capability, the data transmitting bandwidth, and the battery supplying life [182].

In the *Agent\_Monitoring* data model (Figure 5-2), the *MonitoringAgent* class perceives current states from the *SensorNode* class by implementing the method *perceiveCurrentState* and then sends to the *AnalysisAgent* class by implementing the method *sendCurrentState*. The *SensorNode* class is a part of the *MonitoringAgent* class, in which one *MonitoringAgent* have one or more *SensorNode* whereas one *SensorNode* belongs to one *MonitoringAgent*. Three subclasses *SinkNode*, *AnchorNode*, and *MobileNode* are inherited from the superclass *SensorNode*, and can be instantiated their own objects. The *SinkNode* class has an aggregation relationship with the *AnchorNode* class and the *MobileNode* class, in which one *AnchorNode* or *MobileNode* has only one *SinkNode* each time whereas one *SinkNode* can have zero or more of both. The *SensorNode* class is aggregated by the *Engineering* class, the *Environment* class, and the *Operation* class, which represent three different types of sensory data.

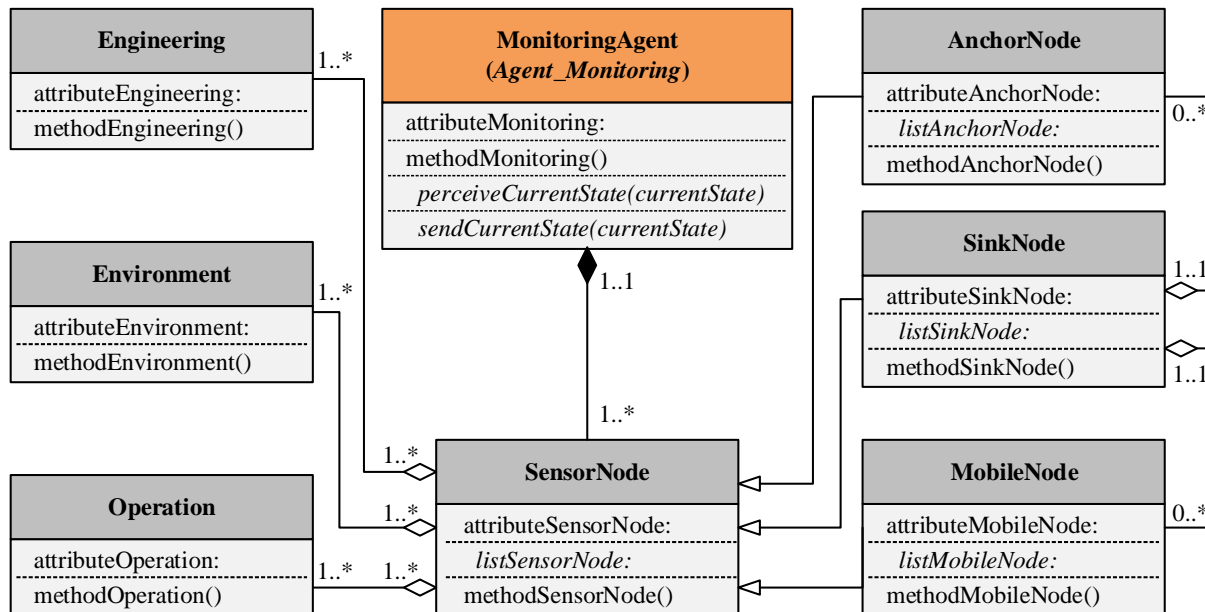


Figure 5-2: The *Agent\_Monitoring* data model

## 5.3 Analysis Agent Design

### 5.3.1 Analysis Agent Modelling

The analysis function provides the mechanisms that correlate and model complex situations to learn about system operations and predict future situations, thus it deals with the ability to understand the current context and to determine a better system state [100]. In this design, *Agent\_Analysis* is a model of data analysis, model building, behaviour learning and prediction according to previous and current states, executed and planned actions, and is aimed at providing an optimal policy with optimal actions for the current state. *Agent\_Analysis* receives the current state from *Agent\_Monitoring* and sends optimal policies to *Agent\_Planning* for action plans or sends optimal actions directly to *Agent\_Execution* for implementation.

Two situations are identified: a) simple situations with normal operations, *Agent\_Analysis* receives data from *Agent\_Monitoring* for analysis or sends desired results directly to *Agent\_Execution* for immediate implementation; b) complex situations with abnormal operations, *Agent\_Analysis* receives data from *Agent\_Monitoring* for analysis and sends desired results to *Agent\_Planning* for optimal action plans. Consider the scenario of a robotic arm sorting blocks into bins described previously in Section 2.1 (Figure 2-1):

a) *simple situations with normal operations*. For example, the default setting of the robotic arm is to grasp black blocks and place them into the black bin. If nothing monitored changed, there will be no change in *Agent\_Analysis* and *Agent\_Execution* will implement the regular action plan.

b) *complex situations with abnormal operations*. For example, mixed black and red blocks come for the robotic arm to sort into corresponding black and red bins. The system has to distinguish different colored blocks and then sort them into different bins. As color change

detected by *Agent\_Monitoring* and analyzed by *Agent\_Analysis*, the action plan has to be revised by *Agent\_Planning* before being implemented by *Agent\_Execution*. In this case, as color changed, the analysis and planning will require a little bit more effort than in simple situations.

In the *Agent\_Analysis* data model (Figure 5-3), the *AnalysisAgent* class implements the *AnalysisDecision* interface and the *SelfLearningAgent* interface. Typical attributes are *previousState* and *executedAction*, *currentState* and *plannedAction*, and *computedAction*. Typical methods are *receiveCurrentState* to communicate with the *MonitoringAgent* class, *sendOptimalPolicy* to communicate with the *PlanningAgent* or *ExecutionAgent* class, and *initializeSelfLearningAgent* to invoke the *SelfLearningAgent* interface. The *AnalysisDecision* interface is implemented for simple situations with optimal decisions available or complex situations as data pre-analysis before initializing *SelfLearningAgent*. Typical methods implemented in the *AnalysisDecision* interface are *selectComputedAction*, *compareStateAction*, and *returnDecisionResult*, in which the *plannedAction* and the *computedAction* are usually the same in simple situations while are different in the complex situations.

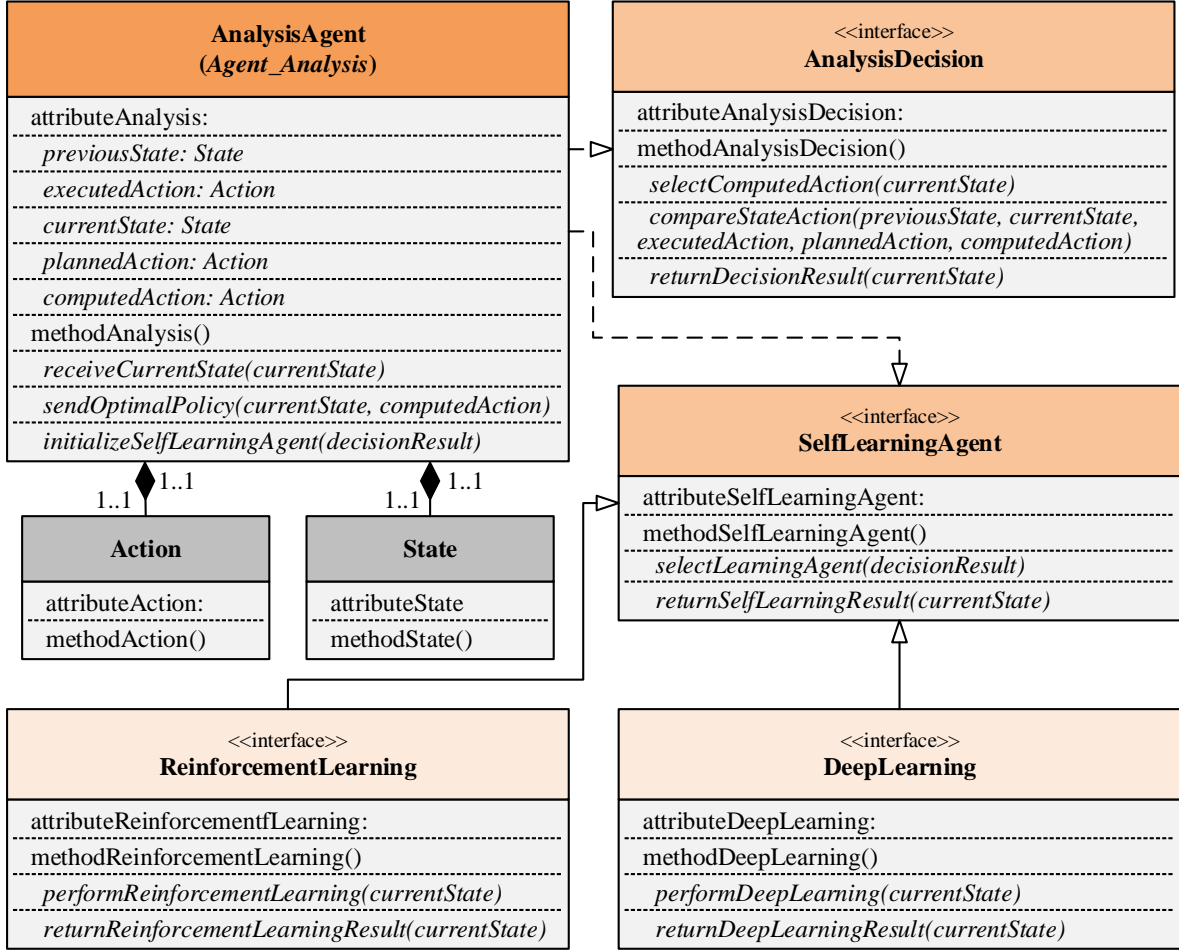


Figure 5-3: The *Agent\_Analysis* data model

### 5.3.2 Self-Learning Agent Modelling

*Agent\_Self-Learning* is a model to support *Agent\_Analysis* for artificial intelligence so that the system can be intelligent to a certain human level. *Agent\_Self-Learning* is designed under open-source machine learning frameworks to deploy various learning models. In this research, it will not focus on details of machine learning models and algorithms, but on different levels of capabilities expected for *Agent\_Self-Learning*. The *SelfLearningAgent* interface is realized through two sub-interfaces *DeepLearning* and *ReinforcementLearning* which are invoked by the



method *selectLearningAgent* (Figure 5-3). Consider the scenario of a robotic arm sorting blocks into bins described previously in Section 2.1 (Figure 2-1):

a) *To learn primitive skills from sensory data.* For example, the robotic arm learns to move to the black block, pick up the black block, and place the black block into the black bin from its perception of the operating environment. Primitive skills like reach, grasp, and place are acquired through learning models like deep reinforcement learning.

b) *To learn from past experience to cope with new tasks.* Tasks are reasonable planning of a collection of primitive skills/actions. Past experience of similar tasks can lead fast learning for new tasks. Therefore, this type of learning can be achieved through deep learning models like transfer learning. Considering the complexity of new tasks, two cases are further identified:

b1) *To deal with a simple new task.* For example, a bigger black block comes for the robotic arm. The system can use the same learning model with a change of some types of parameters (e.g., holding force and opening angle of the gripper) since engineering features (e.g., dimension and mass) of objects have changed. The whole task is still similar with reach, grasp, and place skills.

b2) *To deal with a complex new task.* For example, mixed black and red blocks come for the robotic arm to sort to black and red bins. The previous learning model cannot use because the system has to distinguish colored blocks and then sort them into different bins. Therefore, instead of model transfer, meta-model learning is required to add more primitive skills to form a new learning model for block sorting.

Traditionally, only predefined rules, policies, and goals are provided by *Agent\_Analysis* with limited situations. The cyber module can work in some simple situations with predefined knowledge whereas in most cases it is far less capable of dealing with real-time dynamic

situations. Therefore, *Agent\_Self-Learning* is proposed to perform two types of machine learning models: a) deep learning to autonomically use existing data to train algorithms to find patterns and then make predictions about new data; b) reinforcement learning to autonomically adjust actions in the environment to maximize cumulative rewards.

As proposed, the cyber module can be proactive or reactive in consideration of available data and models for analysis. One practical scenario for the previously described case of a robotic arm sorting blocks into bins (Figure 2-1) is as follows. The industrial system starts working with some predefined knowledge of normal operations (e.g., the robotic arm programmed to sorting blocks into bins), typical failures (e.g., the robotic arm could fail to catch the block on the fly), and regular maintenances (e.g., scheduled maintenance after thousands of picks). At the beginning of these simple situations, the cyber module could actively collect data for analysis and build models for prediction (e.g., rotation speed and angle, holding force and opening angle of the robotic arm). As enough data are available and robust models are built, the cyber module could passively receive data for analysis and apply models for prediction. However, for abnormal operations, untypical failures, and irregular maintenances, no previous experiences are available to the cyber module and it could actively adjust actions by trial-and-error to achieve the best result (e.g., the robotic arm adjust its holding force and opening angle to catch a bigger and heavier block). Then these complex situations become simple situations with available solutions. As shown, deep learning and reinforcement learning techniques are required whether it is with inputs and outputs to find mapping functions, with only inputs to find mapping functions and outputs, or learning directly from interactions with environments.

## 5.4 Planning Agent Design

The planning function provides the mechanisms that construct the actions needed to achieve goals and objectives [100]. In this design, *Agent\_Planning* selects optimal actions and determines the action plan according to the received optimal policy from *Agent\_Analysis*. An update of the optimal policy with the action plan is sent back to *Agent\_Analysis* for references and the action plan is sent to *Agent\_Execution* for implementation. Thus, the action plan represented by orchestrated steps is generated from *Agent\_Planning*, governed by optimal policies computed from *Agent\_Analysis* or *Agent\_Self-Learning* and described in *Agent\_Knowledge*, and finally executed through *Agent\_Execution*, in order to adapt the system from current state to desired state. Consider the scenario of a robotic arm sorting blocks into bins described previously in Section 2.1 (Figure 2-1). *Agent\_Planning* works in complex situations where the regular action plan needs to be adapted. For example, a bigger black block comes for the robotic arm. As monitored engineering features (e.g., dimension and mass) of objects have changed, *Agent\_Planning* has to provide an adapted action plan according to *Agent\_Analysis* by considering changes of some types of parameters (e.g., holding force and opening angle of the gripper). The other case is with large monitoring data of the same type of black block, *Agent\_Planning* will update the regular action plan according to *Agent\_Self-Learning*.

In the *Agent\_Planning* data model (Figure 5-4), the *PlanningAgent* class implements methods *receiveOptimalPolicy* and *updateOptimalPolicy* to communicate with the *AnalysisAgent* class, and the method *sendActionPlan* to communicate with the *ExecutionAgent* class. The *updateOptimalPolicy* method provides feedbacks to the *AnalysisAgent* class to recompute optimal policies or for references of the next same situation. The *PlanningAgent* class implements the *PlanningDecision* interface to decide if the computed action in the optimal

policy is really optimal for execution, then updates the optimal policy by replacing the computed action with the optimal action, and finally sends the action plan to the *ExecutionAgent* class. The *PlanningDecision* interface maintains optimal planning decisions with executable state-action pairs compared to computed optimal policies recommended by the *AnalysisAgent* class.

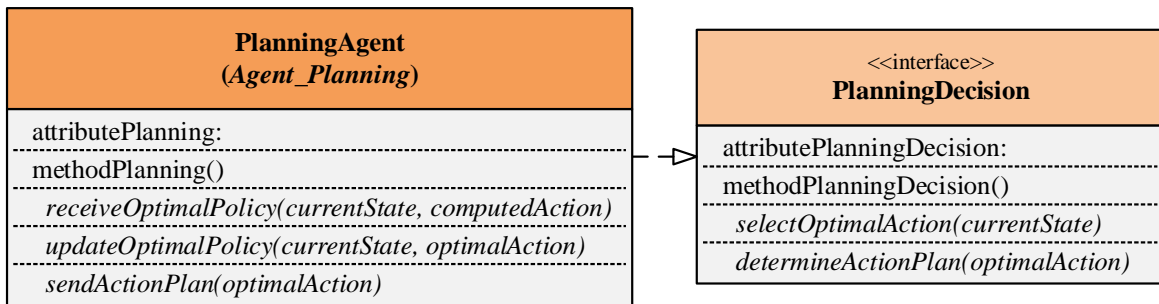


Figure 5-4: The *Agent\_Planning* data model

## 5.5 Execution Agent Design

The execution function provides the mechanisms that control the execution of a plan with considerations for dynamic updates [100]. In this design, *Agent\_Execution* directly interacts with actuators and carries out actions or action plans on systems according to results from *Agent\_Analysis* or *Agent\_Planning*. Different situations are identified at the system level. One is the simple situation with normal operations, *Agent\_Execution* receives optimal actions from and sends back updates to *Agent\_Analysis*. In normal operations, self-optimization happens for improving and optimizing system performance and operations. The other is the complex situation with abnormal operations, *Agent\_Execution* receives action plans from *Agent\_Planning*. In abnormal operations, if the situation is unrecoverable, *Agent\_Execution* performs action plans for self-protection; if the situation is recoverable, *Agent\_Execution* performs action plans for self-healing or self-optimization. Self-configuration can happen in any of the above situations and is required for system reconfiguration.

In the *Agent\_Execution* data model (Figure 5-5), the *ExecutionAgent* class implements the method *receiveActionPlan* to communicate with the *AnalysisAgent* class or the *PlanningAgent* class, and the method *performActionPlan* is realized through the *SelfManageableService* class. The *SelfManageableService* class is aggregated by classes that describe its service types to realize system level self-manageable behaviours. For example, classes *DeviceManagement* and *TaskManagement* are designed for management of devices and tasks to perform action plans on devices and tasks. These classes are realized or implemented through different interfaces. For example, interfaces *ParameterService* and *LifecycleService* are collections of desired behaviours or operations offered for classes *DeviceManagement* and

*TaskManagement* to use. Classes *DeviceManagement* and *TaskManagement* may have one or more types of devices and tasks, and in each type one or more objects may be instantiated.

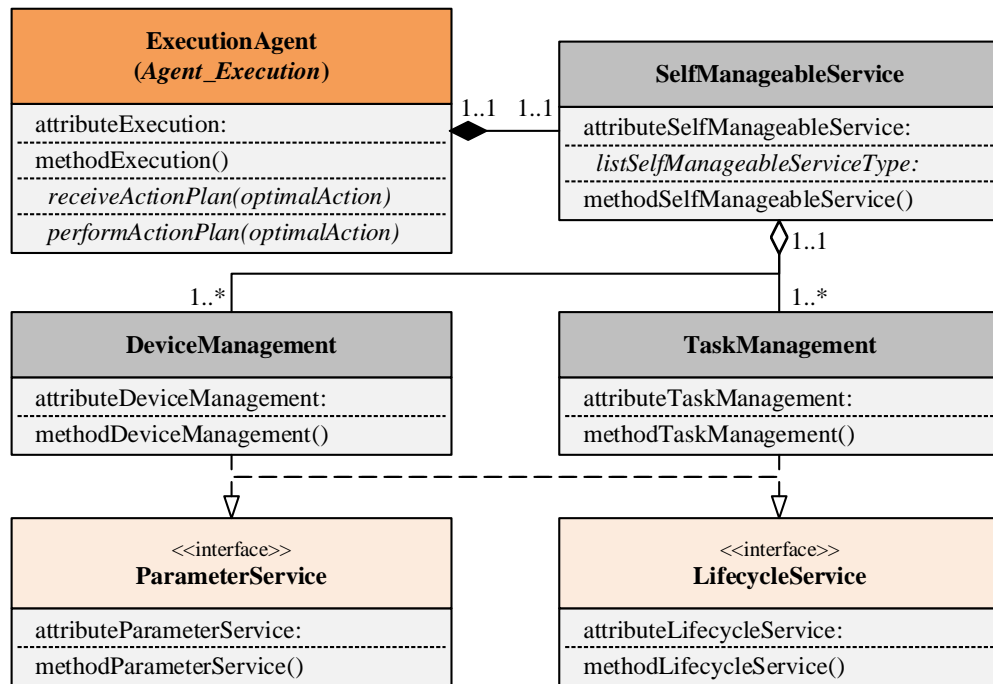


Figure 5-5: The *Agent\_Execution* data model

## 5.6 Knowledge Agent Design

The knowledge source is an implementation of a set of information required to realize system functionalities [100]. In this design, *Agent\_Knowledge* maintains data sets, information centers, or knowledge repositories, providing support to and receiving updates from other agents or entities. *Agent\_Knowledge* either directly manages all sources of desired data or indirectly cooperates with other databases to support the functioning of the multi-agent computing model.

In the *Agent\_Knowledge* data model (Figure 5-6), the *KnowledgeAgent* class has three types of knowledge including *SelfRelatedKnowledge*, *ProblemSolvingKnowledge*, and *ServiceProvidingKnowledge*. They all have abstract attributes (e.g., types of knowledge) and methods (e.g., provide support and update database of knowledge). The *SelfRelatedKnowledge* class mainly describes the operating environment and the system engineering properties, and the real-time operation behaviours. It's all about the system itself, its operations, and the operating environment, which is taken care of by *Agent\_Monitoring*. The *ProblemSolvingKnowledge* class provides goals, policies, rules, models, etc. which are directly in support of problem-solving tasks in *Agent\_Analysis* and *Agent\_Planning*. In general, goals describe what is desired, or the best to be envisioned; policies describe how to achieve it, or optimal actions that can be performed for state transition; rules describe actions to be taken under verified conditions for monitored events; models describe representations of more complex cases that can provide possible solutions to those cases [100], [138]. The *ServiceProvidingKnowledge* class serves the role in *Agent\_Execution* to provide desired services in response to request for system adaptation. Certain kinds of services include parameter services (e.g., adjust the frequency of a sensor node transceiver to transmit and receive sensory data), life-cycle services (e.g., start, stop, or remove a sensor node in the network), etc.

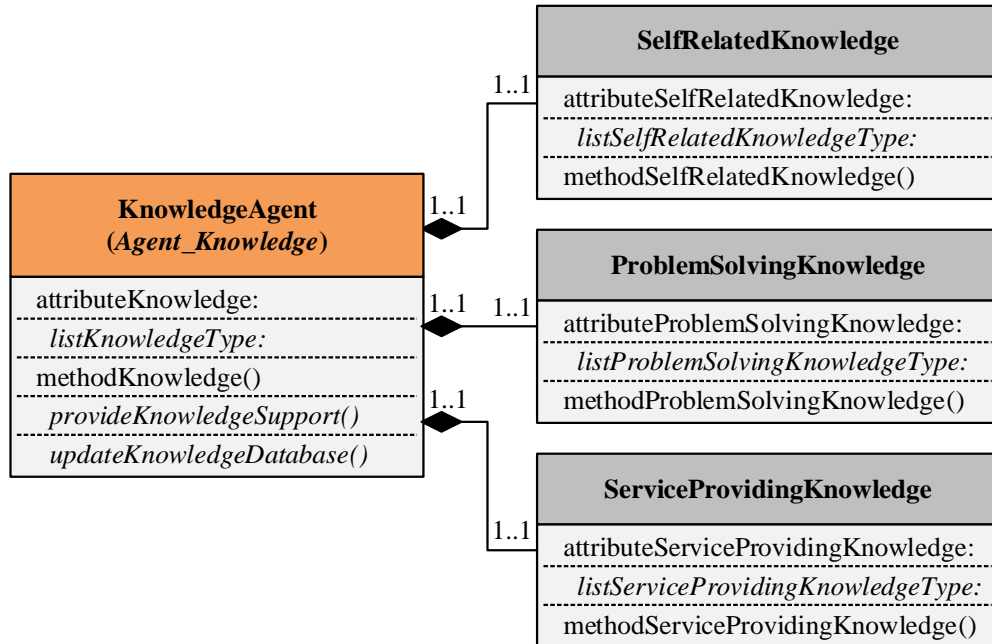


Figure 5-6: The *Agent\_Knowledge* data model



## 5.7 Summary

In this chapter, the architecture modelling framework for the high-level cyber module was proposed. The high-level cyber module design deploys autonomic computing reference architecture into the modelling framework with the implementation of multi-agent modelling techniques. It is designed as multi-agent computing model consisting of *Monitoring Agent*, *Analysis Agent*, *Self-Learning Agent*, *Planning Agent*, *Execution Agent*, and *Knowledge Agent*. The design results in a new computing module for high-level multi-agent based automation architectures, aiming at achieving run-time distributed intelligence in system design and module reconfiguration.

## Chapter Six: Low-Level Physical Module Architecture Modelling

### 6.1 Introduction

The low-level self-manageable architecture design shown in Figure 6-1 deploys autonomic computing self-managing properties into the modelling framework with the implementation of IEC 61499 FB modelling techniques. The key system self-managing properties envisioned by autonomic computing are *self-configuration*, *self-healing*, *self-optimization*, and *self-protection*. For the architecture implementation, the IEC 61499 FB modelling technique is applied by using object-oriented and event-driven function blocks to realize real-time adaption of automation logic and control algorithms (i.e., IEC 61499 Function Block Model in Figure 6-1) and the multi-agent modelling technique is used by embedding the multi-agent system into IEC 61499 FBs to support self-management capabilities of the low-level system architecture (i.e., Self-Manageable Service Model in Figure 6-1). The key feature of this proposed self-manageable architecture design is the separation of the self-manageable service execution from the control application execution. More specifically, one execution path is responsible for control applications (built as IEC 61499 FBs for control purposes) and a second execution path is responsible for self-manageable services (designed as embedded multi-agent models for system configuration, optimization, healing, and protection purposes). The design results in a new agent-embedded design pattern for modelling IEC 61499 FB based control solutions that are capable of self-management in real-time adaptation.

In this chapter, an overview of IEC 61499 reference architecture with exiting design patterns will be discussed, and then the proposed hybrid model with agent-embedded design

pattern is introduced. A self-manageable system programed in IEC 61499 is expected to be modelled by applying both existing and proposed design patterns.

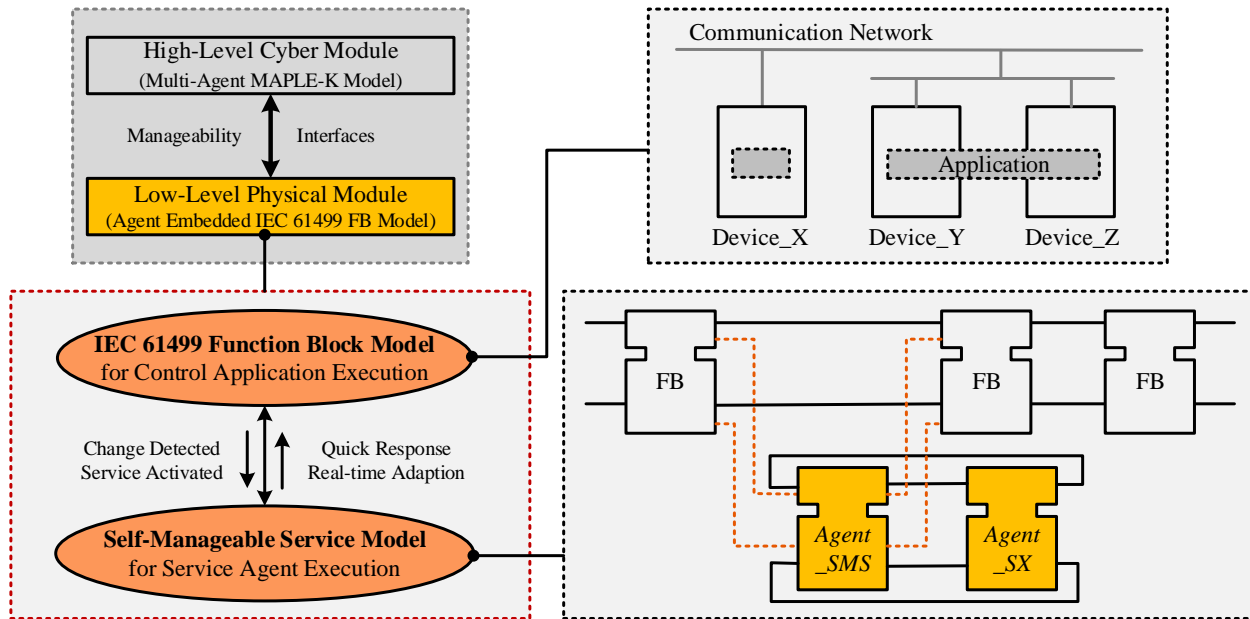


Figure 6-1: The proposed low-level architecture modelling framework

## 6.2 Developing Low-level Control Systems in IEC 61499 Function Blocks

### 6.2.1 IEC 61499 Reference Architecture

Figure 6-2 provides an overview of the IEC 61499 reference architecture [6]. A *function block* is an object-oriented modelling element with event-driven execution. An *application* model (Figure 6-2a) is defined as a network of interconnected FBs linked by event/data flows and distributed over resources and devices. A *resource* model (Figure 6-2b) is defined to support the execution of one or more application fragments. A *device* model (Figure 6-2c) is defined to support one or more resources to exchange data through interface services internally (i.e., the process interface to enable interaction via input/output points in local devices) and externally (i.e., the communication interface to enable interaction via networks with resources in remote devices). A *system* model (Figure 6-2d) is a collection of interconnected devices interacting with each other through communication networks.

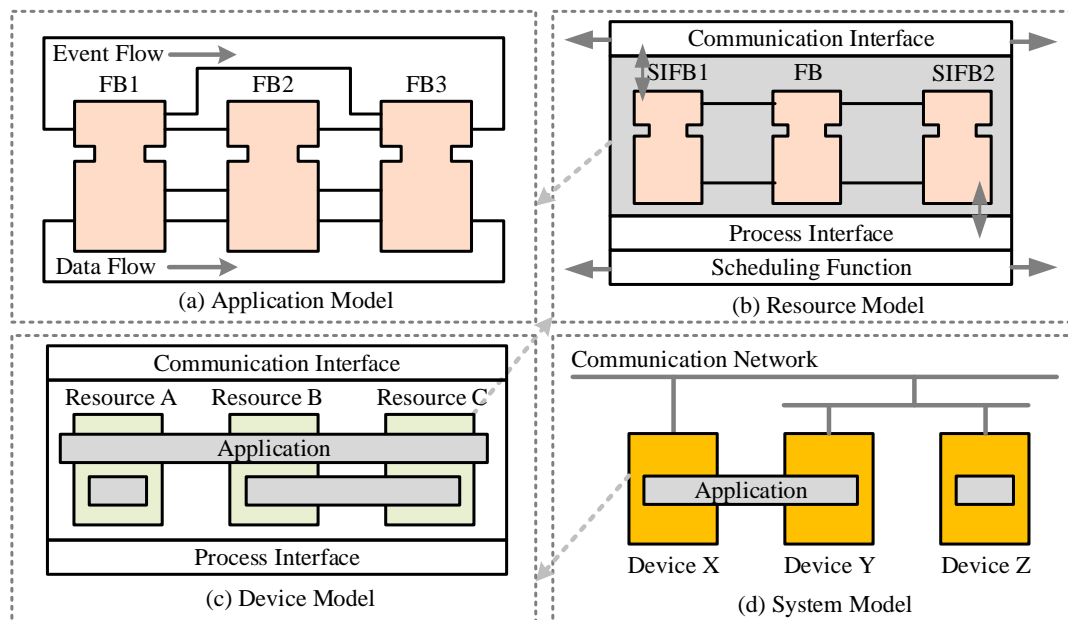


Figure 6-2: The IEC 61499 reference architecture

A typical *system* programmed under the IEC 61499 reference architecture is designed as:

a) the control logic built by *function blocks* as *applications*, and b) physical *devices* encapsulating required *resources* for implementation. In the IEC 61499 FB data model (Figure 6-3), classes *System*, *Device*, *Resource*, *Application* are main entities of the model. Component encapsulation is realized through CFBs (designed and implemented as a physical network of FB instances) and SubApps (designed as a logical network of FB types and then implemented by FB instantiation). Interface declaration is achieved by SIFBs (encapsulation of FB interaction with external services) and adapters (encapsulation of FB interaction with internal services). SIFBs are implemented as a pair of application-initiated requester remaining passive until receiving input events by the application, and resource-initiated responder sending output events to act on the device (e.g., design patterns *publish/subscribe*, *client/server*). Adapters are realized through a pair of *plug* to group required interfaces on the high-level FB side and *socket* to group provided interfaces on the low-level FB side.

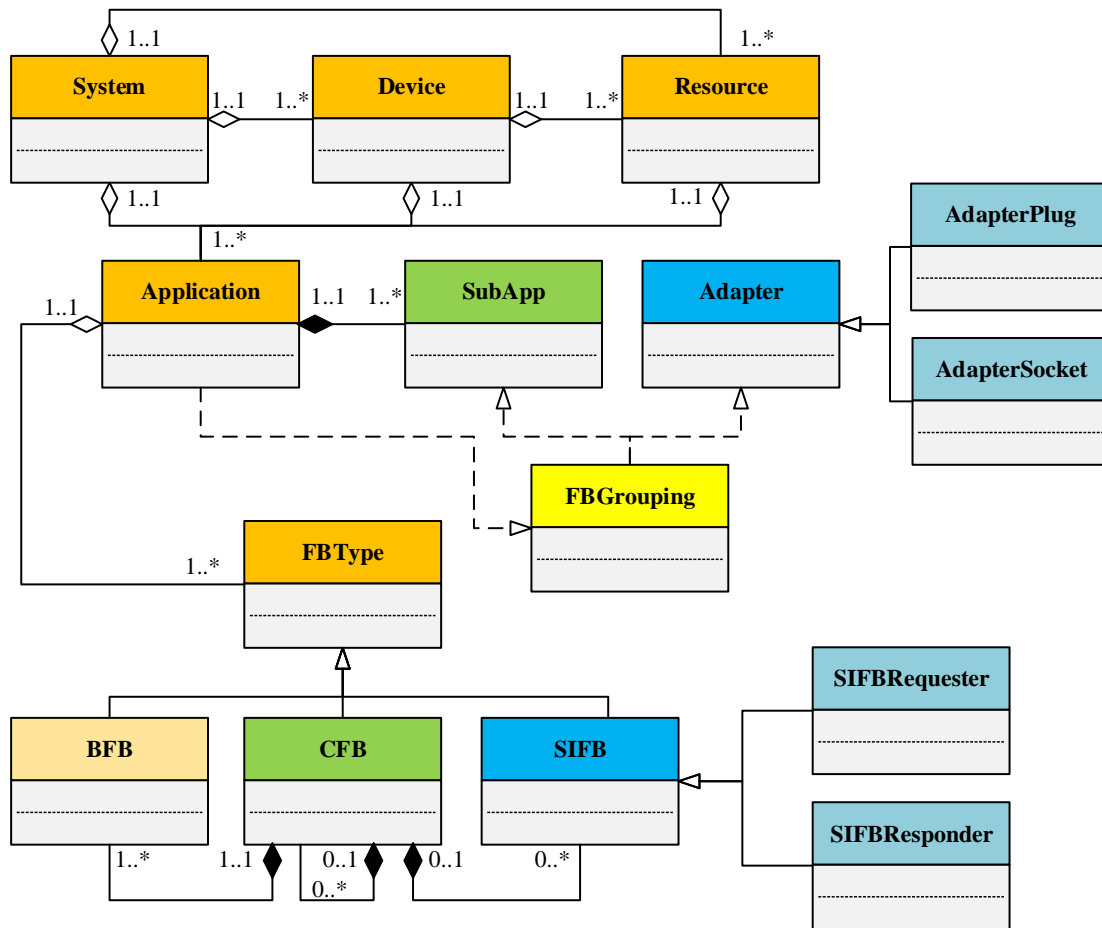


Figure 6-3: The IEC 61499 function block class diagram

## 6.2.2 Interface Declaration Model

### A. IEC 61499 SIFB Models

SIFBs represent the interfaces to services provided by managed components of low-level hardware systems so that the application deployed to several devices can get access to inputs/outputs of and communicate with managed components [6]. That means SIFBs are activated not only by the input events but also by the managed components. Two types of SIFBs are defined as a pair (Figure 6-4): a) SIFB *requester*, an application-initiated type which remains passive until receiving input events by the application; b) SIFB *responder*, a resource-initiated

type which sends output events to act on the device [6]. SIFBs are one type of FBs and their dynamic behaviours are defined as service sequence diagrams as shown in Figure 6-4. The difference of these two types is in the data transfer part, in which the type depends on either applications or resources trig the data transfer (the green part in Figure 6-4 and Figure 6-5).

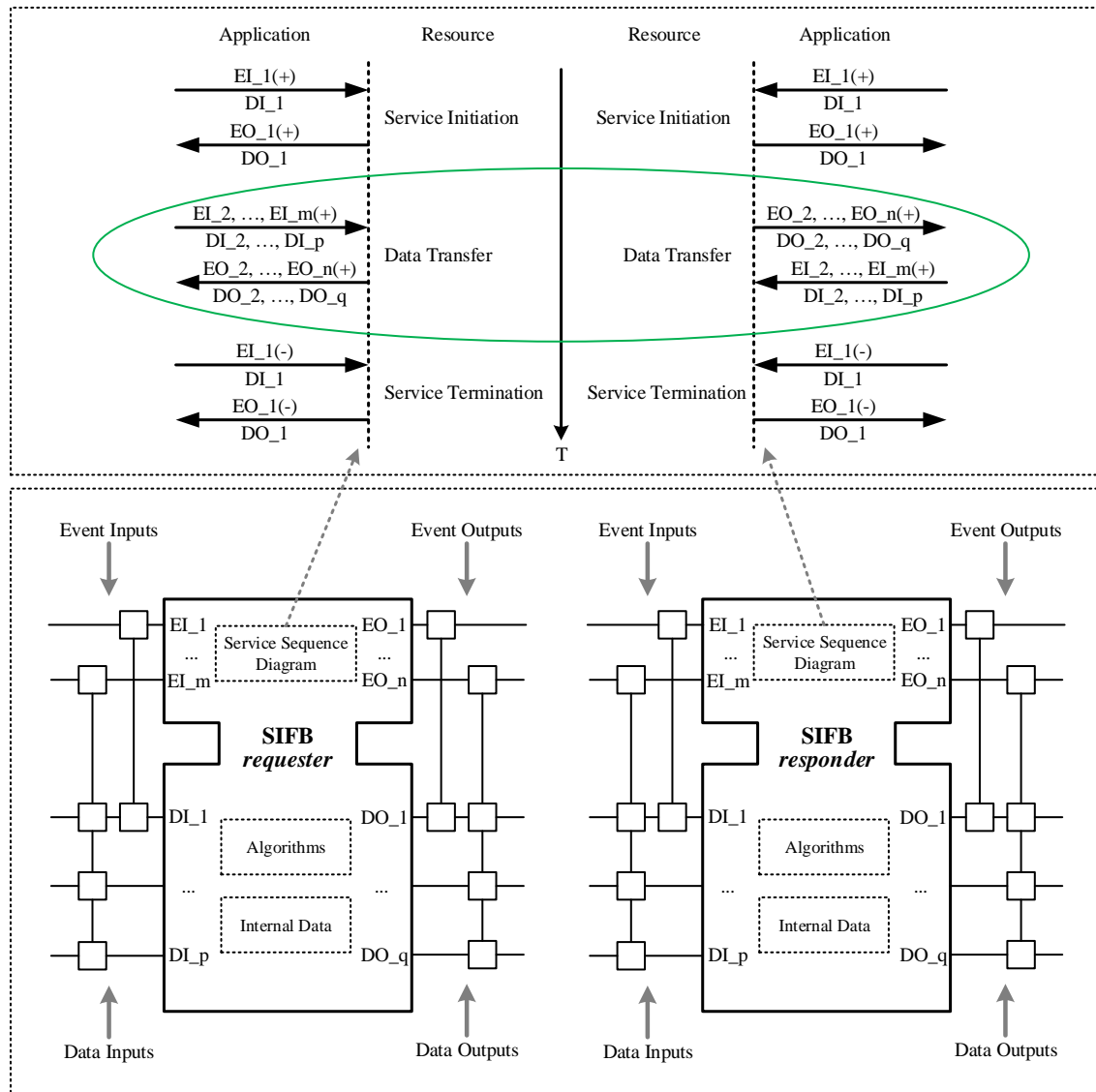


Figure 6-4: IEC 61499 SIFB *requester/responder* models

IEC 61499 SIFBs are used to access to hardware systems (e.g., devices, network segments) which BFBs and CFBs cannot. Two SIFB communication patterns are designed for low-level physical module architecture modelling (Figure 6-5): the publish/subscribe model for

unidirectional communication (the red part in Figure 6-5) and the client/server model for bidirectional communication (the blue part in Figure 6-5) [6]. The publish/subscribe model is based on the n-to-n architecture in which one publisher can send messages to one or more subscribers and one subscriber can receive messages from one or more publishers. The inputs of the publish SIFB match the outputs of the subscribe SIFB. The client/server model is based on the n-to-1 architecture in which one or more clients communicate with one server in both sending and receiving messages. The inputs of the client SIFB match the outputs of the server SIFB and the outputs of the client SIFB match the inputs of the server SIFB.

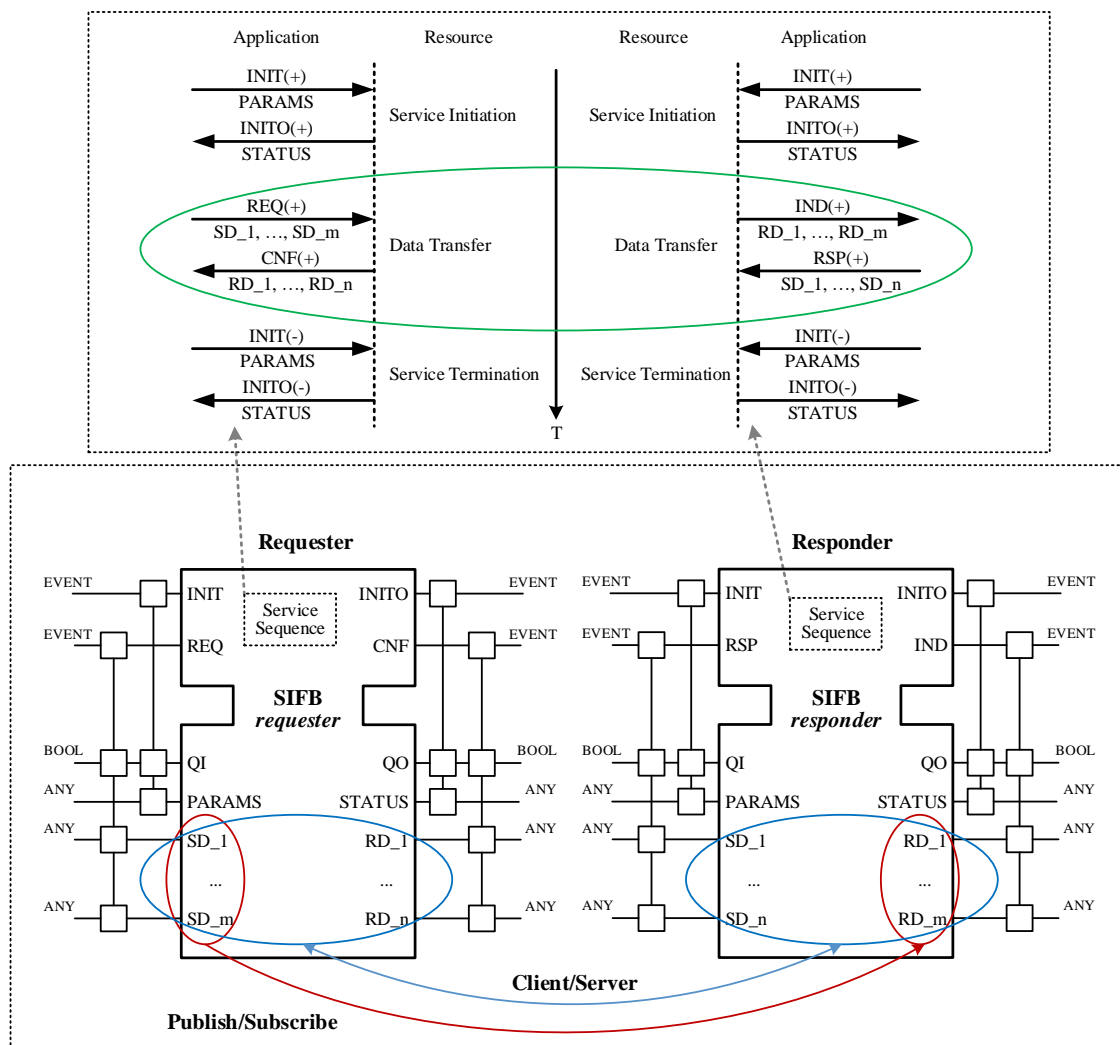


Figure 6-5: Communication patterns of IEC 61499 SIFB requester/responder models



For example, Figure 6-6 describes a simple control loop where the application is deployed into two devices. As sensors and actuators interact with managed components and/or operating environments, SIFBs are required to program the control application to communicate with those external services. This example can be seen as an abstraction of several scenarios. For example, the previously described scenario of a robotic arm sorting blocks into bins (Figure 2-1), the attached 3D sensor detects the object position, shapes, colors, etc. and transmits data to the robot control which controls the end gripper to perform desired actions.

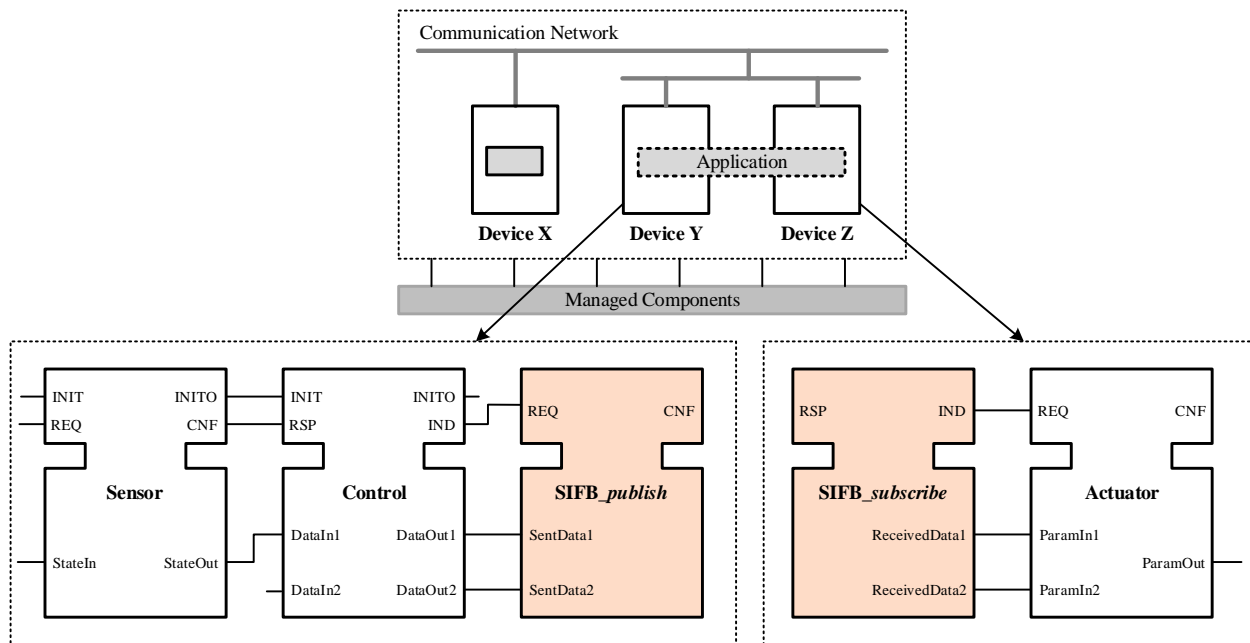


Figure 6-6: An example of IEC 61499 SIFB models

### B. IEC 61499 Adapter Models

The IEC 61499 adapter models are defined for encapsulation of FB interaction with internal services which is different from the SIFB models for external services [6]. Two types of adapters are defined as a pair (Figure 6-7): a) adapter *plug*, to group required interfaces on the high-level FB side; b) adapter *socket*, to group provided interfaces on the low-level FB side [6]. Input interfaces of the *plug* are output interfaces of the *socket* and output interfaces of the *plug* are

input interfaces of the *socket*. Figure 6-7 shows the static structure of the adapter in a form of FB and their dynamic behaviours are defined as service sequence diagrams as shown between them.

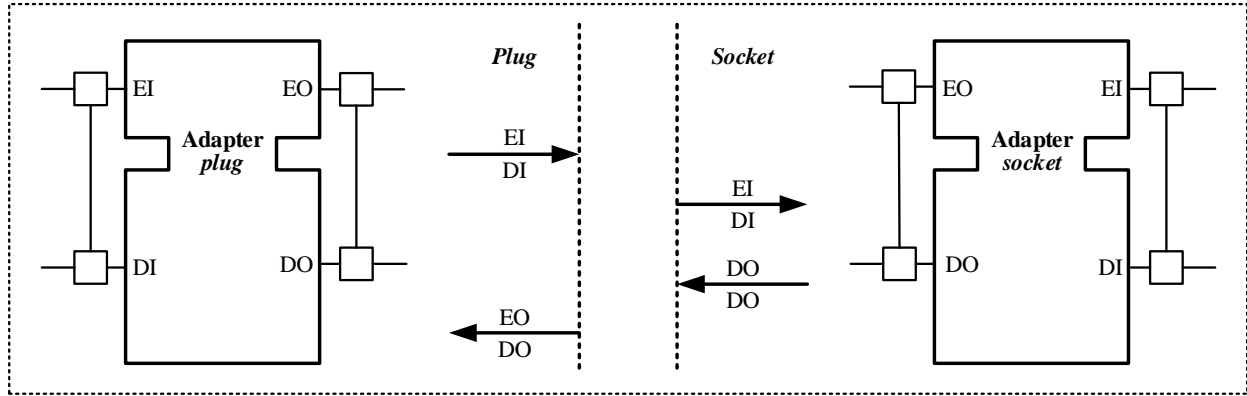


Figure 6-7: IEC 61499 adapter *plug/socket* models

The following example shows how IEC 61499 adapter models are applied in the low-level physical module architecture modelling. Three types of sensor nodes (i.e., sink node, anchor node, and mobile node) were designed for *Agent\_Monitoring* in Section 5.2. Figure 6-8 shows a generic form of IEC 61499 FB implementation without adapters for sensor nodes. Two clusters (i.e., *AnchorNodeCluster* and *MobileNodeCluster*) are designed to manage clustered anchor nodes and mobile nodes, respectively. Different types of sensor nodes (i.e., *AnchorNode1*, *AnchorNode2*, *MobileNode1*, *MobileNode2*) in each cluster are designed to communicate with the sink node (i.e., *SinkNode*). FBs are linked with event communications in solid blue lines and data communications in dash red lines. As a consequence, without applying adapters, the design of such a simple sensor network system could become very complicated with so many unclear communicating connections. Furthermore, if dynamically adding more sensor nodes, e.g., *MobileNode3*, the *MobileNodeCluster* has to be redesigned to provide available interfaces to the added sensor nodes. That is absolutely not acceptable for industrial systems to be self-manageable, especially these sensor nodes can be dynamically reconfigured. In Figure 6-9, two adapters for two clusters are designed as an example: *ACAdapter\_plug* and

*ACAdapter\_socket*, *MCAdapter\_plug* and *MCAdapter\_socket*. The IEC 61499 FB implementation with adapters is shown in Figure 6-10. The method to design adapters usually starts from the high-level FB side to group required interfaces as a *plug* and then the low-level FB side to group provided interfaces as a *socket*. Both are denoted with “>>” and its name. Compared with the same application in Figure 6-8, the system architecture is much easier to understand. More importantly, the control algorithms will be independent from any particular instances and therefore dynamically reconfiguring sensor nodes becomes possible if they share the same type of adapters. For example, several autonomous mobile robots are added to the scenario described previously in Section 2.1 (Figure 2-1) to be responsible for carrying sorting bins to desired areas (Figure 7-1), and these mobile robots or untethered vehicles can share the same mobile node adapters.

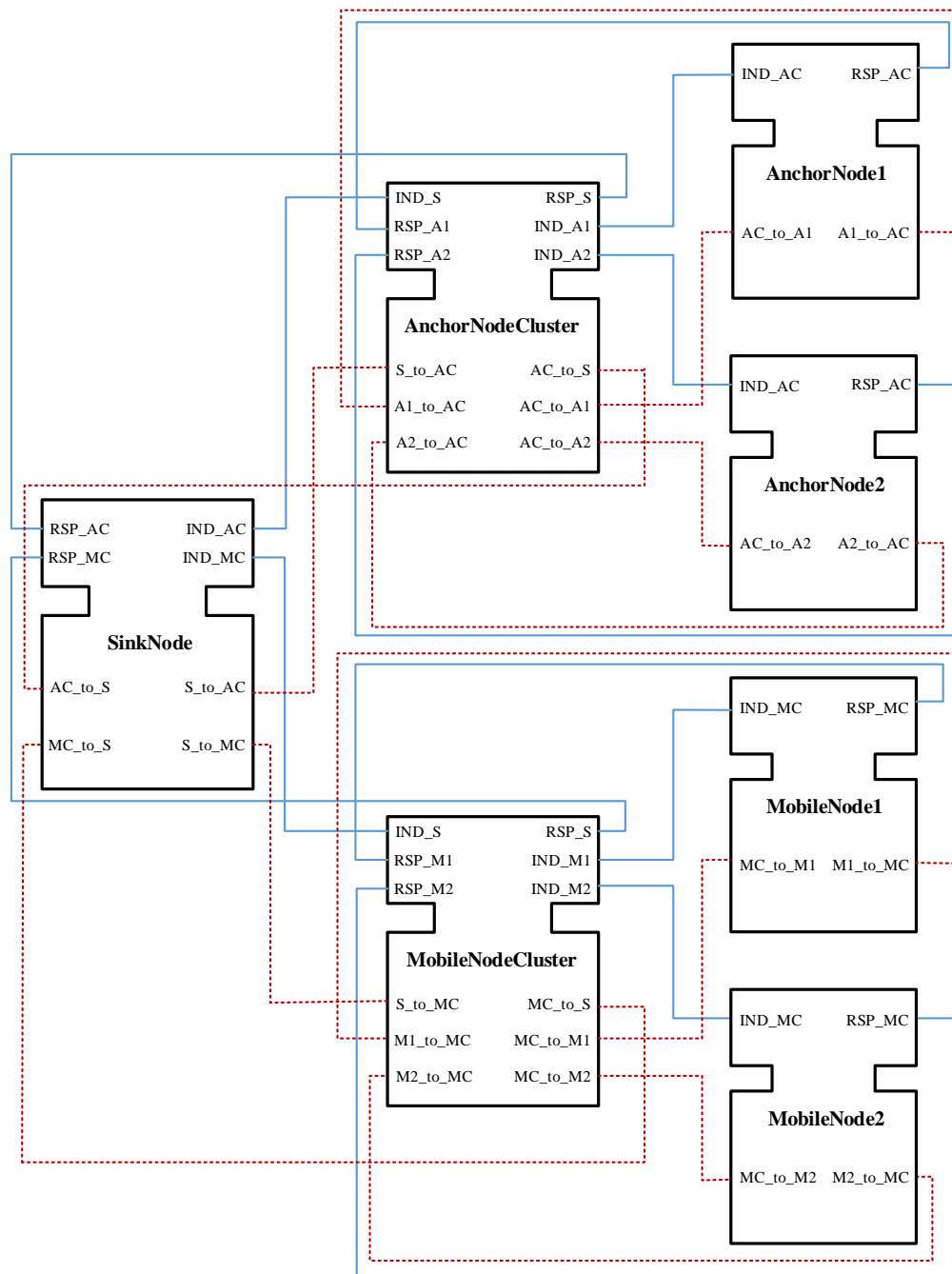


Figure 6-8: IEC 61499 FB application without adapters

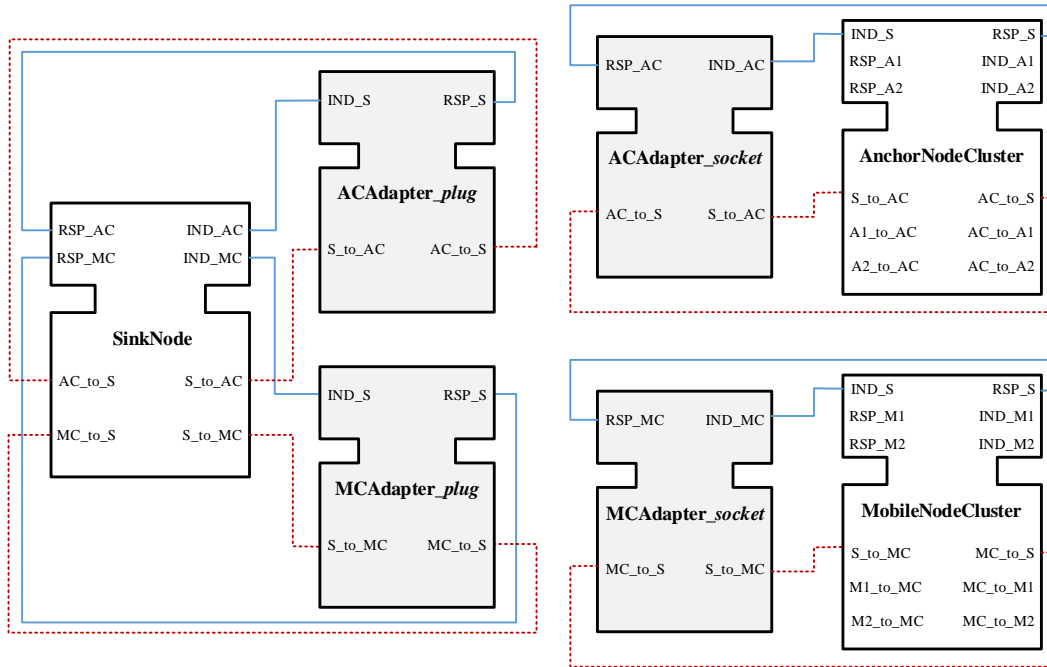


Figure 6-9: IEC 61499 adapter design

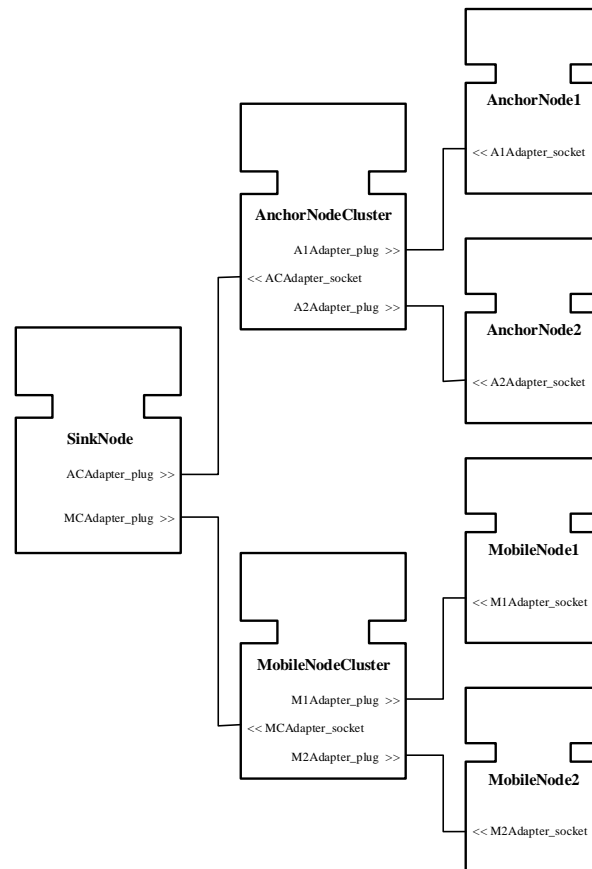


Figure 6-10: IEC 61499 FB application with adapters

### 6.2.3 Component Encapsulation Model

There are two ways to encapsulate FBs to build applications, either through CFBs or SubApps. IEC 61499 CFB is defined as a network of FB instances through event and data connections [6]. IEC 61499 SubApp is a means to group application components in the top-down/bottom-up manner and share their common public interfaces [6]. Compared to typed CFB models which new types are created during each adaptation of applications, the IEC 61499 SubApp model supports application adaptation on all hierarchic levels for reuse and reconfiguration which is much faster for application development and structure modelling.

To continue sensor node FB models discussed in Section 6.2.2 and go into some details about these two models applied in the control application programming. Figure 6-11 shows the example. For both, there are three interconnected component FB types (i.e., *SinkNode*, *AnchorNodeCluster*, and *MobileNodeCluster*) to be encapsulated as a CFB or SubApp for the sensor node, and these component FB types can be instantiated as multiple instances. The difference is: a) the *CFB\_SensorNode1* is physically a network of instances of three component FB types (i.e., *sinknode1*, *anchornodecluster1*, and *mobilenodecluster1*) which means the internal structure cannot be changed after the CFB is developed; b) the *SubApp\_SensorNode* is logically a network of desired FB types which means it can be an empty one in a top-down design or the same as the CFB in a bottom-up design. The advantage of SubApps over CFBs is in dynamical configuration of sensor nodes in which SubApps can be scalable, adaptable, and distributable.

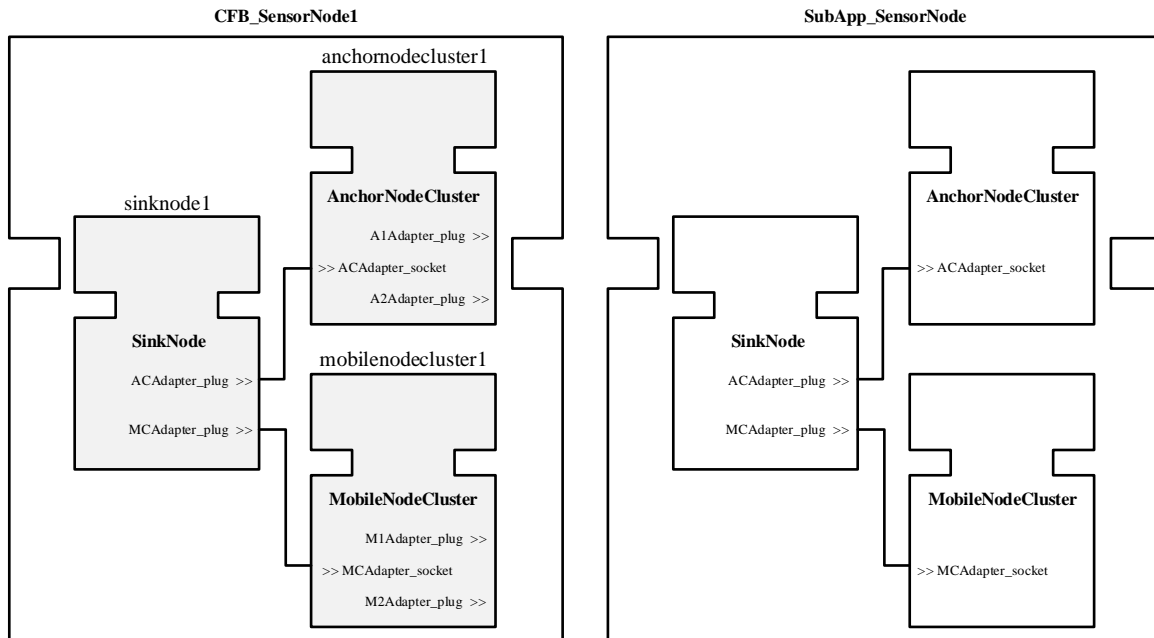


Figure 6-11: Comparison of IEC 61499 CFB and SubApp models

#### 6.2.4 IEC 61499 Application Model Design

IEC 61499 application model design is the core in system design. As stated before, the system is designed as two parts, the control logic built by function blocks as applications and the physical devices encapsulating required resources for application implementation. In this section, how various FB models mentioned in previous sections are used in the application model design will be discussed. For example, the sensor-control-actuator FB model previously discussed in Section 6.2.2. Figure 6-12 shows the example in a generic form. From a physical view, the sensor-actuator represents the hardware part (i.e., 3D sensor attached to the robotic arm) and the control represents the software part (i.e., control application). However, logically the data/events flow from the sensor to the control and then to the actuator (i.e., the attached 3D sensor detects the object position, shapes, colors, etc. and transmits data to the robot control which controls the end gripper to perform desired actions). Actually, a good application-oriented design is to separate each part into different devices, model each part as SubApps, and group common interfaces as

adapters. SIFBs are used since each part are mapped to networks for communication. Compared to the one in Figure 6-6 which no adapter is designed and the sensor and the control are deployed into one device, each part in the new design is loosely coupled, can be dynamically configured, and is able to share the same type of interfaces.

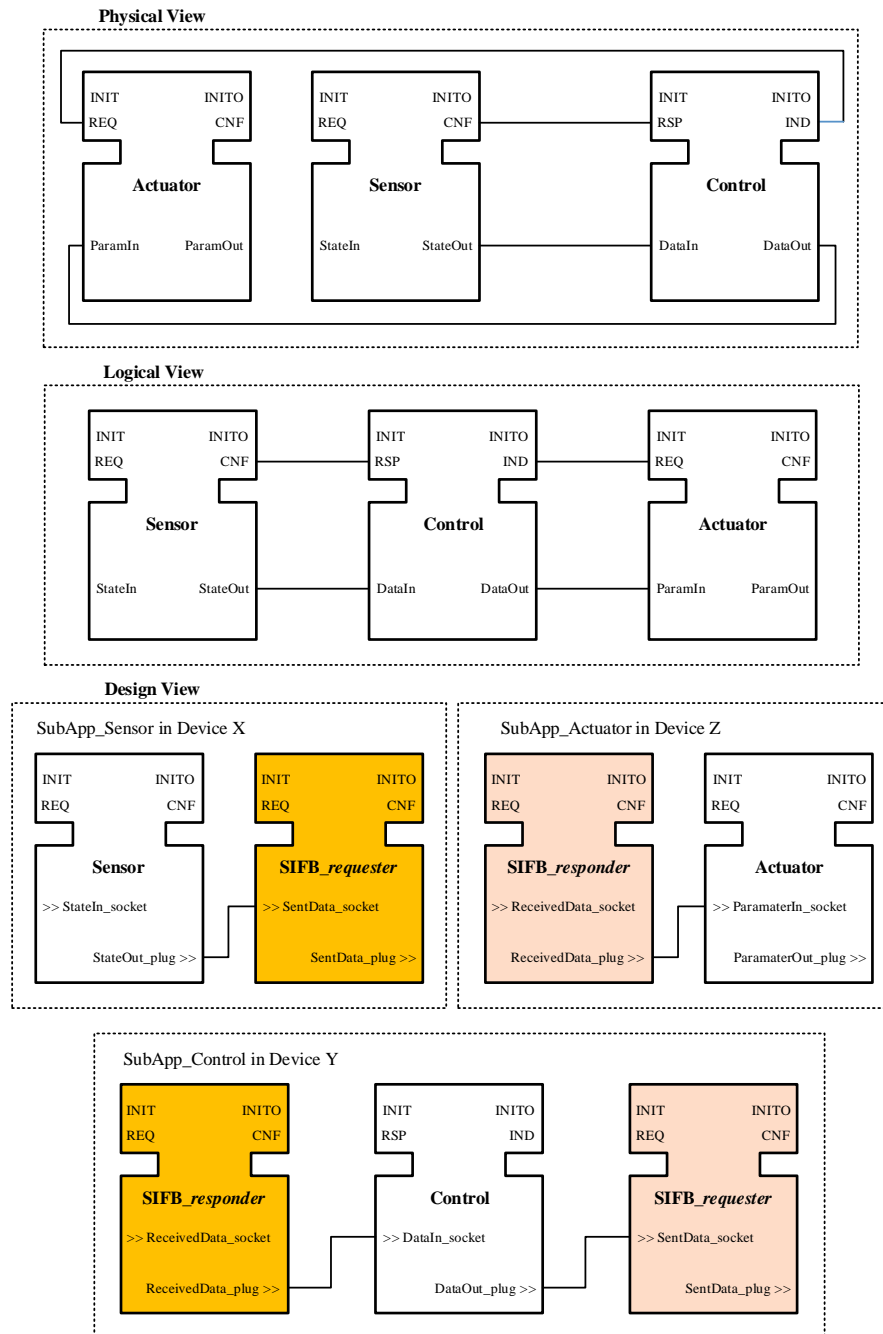


Figure 6-12: Design of an IEC 61499 application model for distributed automation



## 6.3 Self-Manageable Service Model for Architecture Design in IEC 61499

### 6.3.1 Self-Manageable Service Model

In the proposed low-level architecture modelling framework (Figure 6-13), the self-manageable service model is design as a multi-agent system embedded in IEC 61499 FBs. Two new agent types are designed: a) the self-manageable service execution agent *Agent\_SMS*; b) self-manageable agents including *Agent\_SC*, *Agent\_SO*, *Agent\_SH*, and *Agent\_SP*. In general, *Agent\_SMS* is mainly responsible for monitoring system states and responding to changes by deciding the adequate behaviours to perform (i.e., activate one or more self-manageable agents and execute self-manageable services). The second type of agent is primary concerned with generating self-manageable service action plans upon requests: a) *Agent\_SC* for configuring/reconfiguring functions, structures, and process to adapt to dynamical changes; b) *Agent\_SO* for improving and optimizing performance and operations with respect to predefined goals; c) *Agent\_SH* for detecting and recovering from disturbances and faults to maximize system availability; and d) *Agent\_SP* for identifying and protecting against safety and security attacks to preserve system integrity.

A detailed procedure is shown in Table 6-1 for the implementation of the self-manageable service model. An example of this process is illustrated in Figure 6-13. The self-configuration process begins with a single change request that results in the old *FB\_S* being replaced by two new *FB\_S1* and *FB\_S2* in the *Application* residing in *Device\_Y* and *Device\_Z*. The self-manageable service is activated as *Agent\_SMS* detects the change request and communicates to *Agent\_SC* to request the self-configuration service. *Agent\_SC* generates the action plan and sends it back to *Agent\_SMS* for execution on the application.

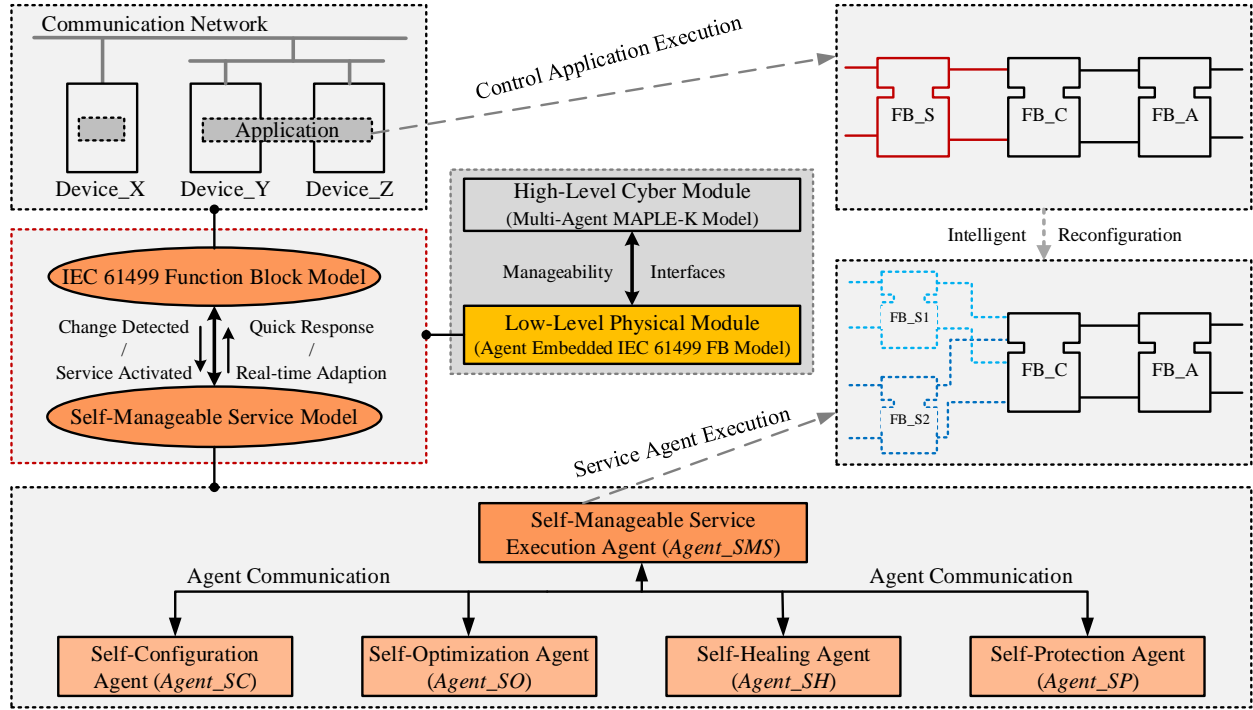


Figure 6-13: The low-level self-manageable architecture modelling framework

Table 6-1: Algorithm for the proposed self-manageable service model

**Algorithm 2:** the self-manageable service model

```

01 Input: Current State //control application real-time event/data
02 Output: Action Plan // control application self-manageable services
03 Initialize Agent_SMS
04   Read currentSate to detect changes
05   //rea-time control application execution event/data
06   Call SelfManageableAgents to request self-manageable agents for response
07   Case normal operations of
08     //no changes or changes in the reasonable range
09     Condition: Execute Agent_SO for optimization
10     Execute Agent_SC for reconfiguration
11   Case abnormal operations of
12     //changes affecting predefined system capabilities
13     Condition Recoverable: Execute Agent_SH for healing
14     Execute Agent_SC for reconfiguration
15     Condition Unrecoverable: Execute Agent_SP for protection
16     Execute Agent_SC for reconfiguration
17   Return actionPlan generated by self-manageable agents
18   Call IEC61499FunctionBlockSystem to execute self-manageable services for adaptation
19   Start/Stop/Update IEC 61499 FB System
20   Create/Modify/Delete IEC 61499 FB System Element
21   //including device, resource, application, function block, event/data

```

### 6.3.2 Self-Manageable Service Execution Agent Design

*Agent\_SMS* plays a key role in requesting one or more self-manageable agents to respond to changes and executing self-manageable services provided by self-manageable agents to adapt IEC 61499 FB based systems. In Figure 6-14, the *Agent\_SMS* class implements two interfaces (i.e., the *SelfManageableAgents* interface and the *IEC61499FunctionBlockSystem* interface) to realize its predefined functions. Typical attributes of the *Agent\_SMS* class include *previousState*, *executedAction*, *currentState*, *plannedAction*, and *computedAction*. Typical methods are *receiveCurrentState*, *initializeSMAgent*, and *executeAgentSMS*.

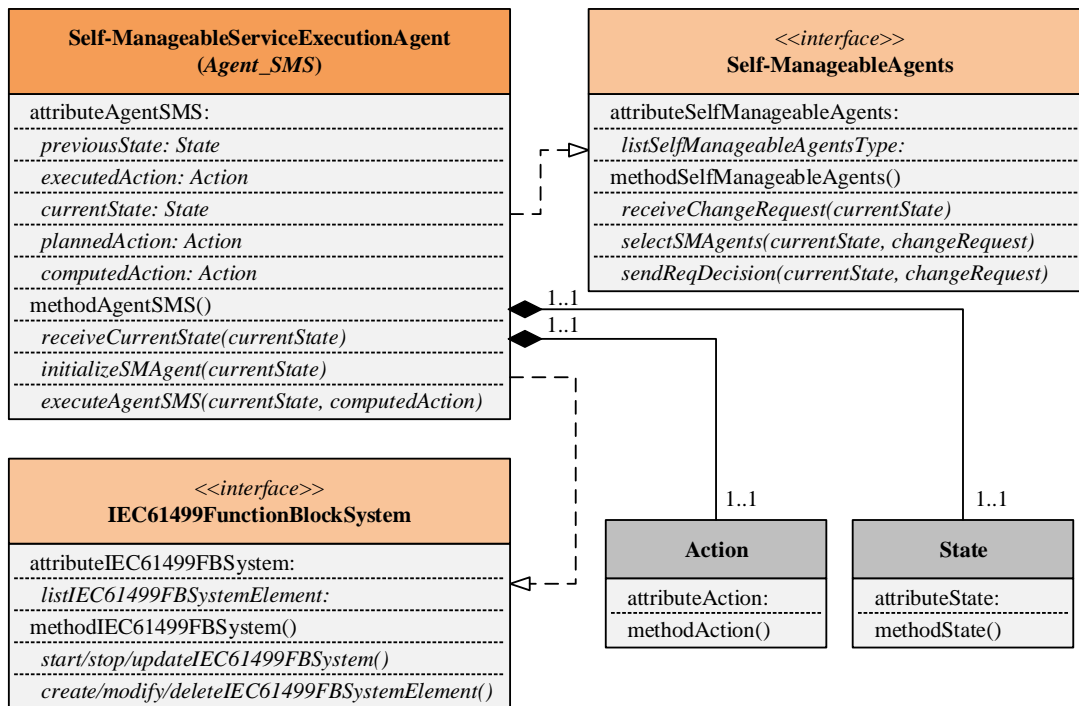


Figure 6-14: The *SelfManageableServiceExecutionAgent* data model

### 6.3.3 Self-Manageable Agents Interface Design

The *SelfManageableAgents* interface in Figure 6-15 provides access to communicate with self-manageable agents for self-manageable services in the low-level physical module. One typical attribute is the list of self-manageable agent types (i.e., *Agent\_SC*, *Agent\_SO*, *Agent\_SH*, and

*Agent\_SP*). Each agent is responsible for their own tasks. Typical methods are *receiveChangeRequest* to get the change request from *Agent\_SMS*, *selectSMAgents* to choose which agent to perform the self-manageable service according to *currentState* and *changeRequest*, and *sendReqDecision* to ask the selected agent to perform required tasks. For each self-manageable agent class, typical methods include receiving the request decision from the *SelfManageableAgents* interface, generating the self-manageable service action plan, and returning the plan to *Agent\_SMS* for execution.

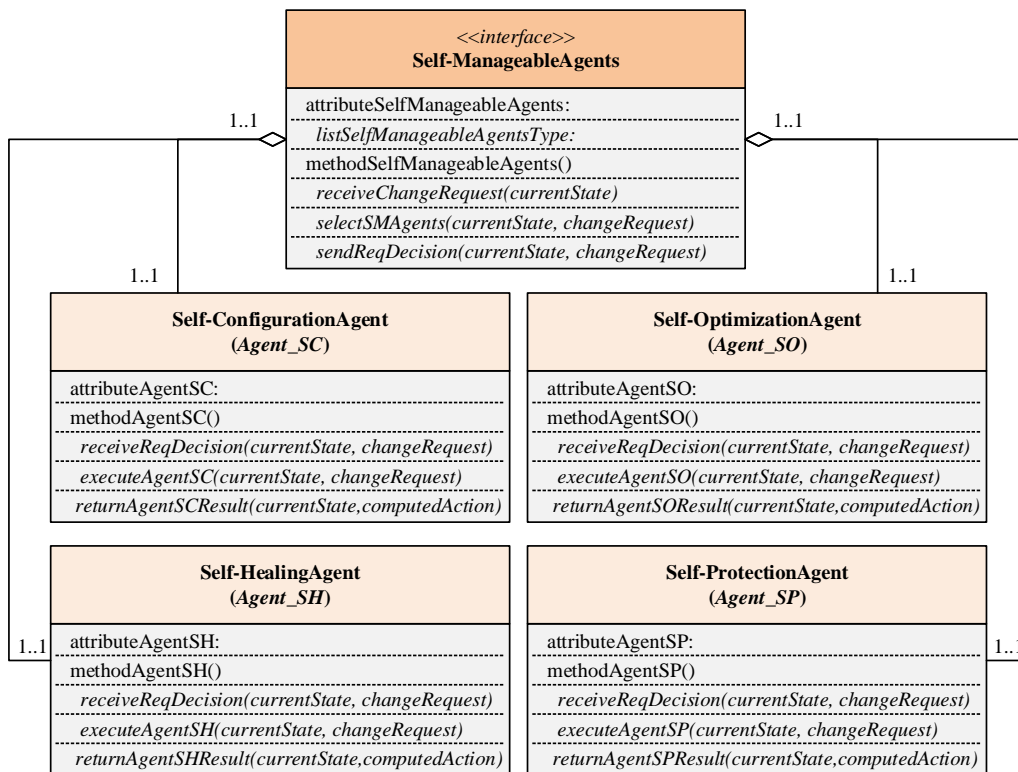


Figure 6-15: The *SelfManageableAgents* data model

### 6.3.4 IEC 61499 Function Block System Interface Design

The *IEC61499FunctionBlockSystem* interface in Figure 6-16 provides access to executing self-manageable services to adapt IEC 61499 FB based systems. The typical attribute is a list of system elements, e.g., devices, resources, applications, and FBs. Typical methods are

*start/stop/updateIEC61499FBSystem* and *create/modify/deleteIEC61499FBSystemElement*. The classes *Device*, *Resource*, *Application*, *FunctionBlock* with *Event/Data* are main entities that *Agent\_SMS* can execute the self-manageable service action plan on. In the *Device* class, typical methods include *add/remove/resetDevice* and *start/stop/killDevice*. In the *Resource* class, typical methods include *create/modify/deleteResource* and *start/stop/killResource*. For the *Application* class, typical methods are *create/modify/deleteApplication* and *start/stop/killApplication*. For the *EventData* class, typical methods are *read/write/resetEDValue* and *create/modify/deleteEDConnection*. For the *FucntionBlock* class, typical attributes are *FBType* (e.g., *BFB*, *CFB*, and *SIFB*), *FBInstance*, *FBConnection*, and *FBGrouping* (e.g., *plug* and *socket adapters*), and typical methods are to create, modify, and delete those attributes.

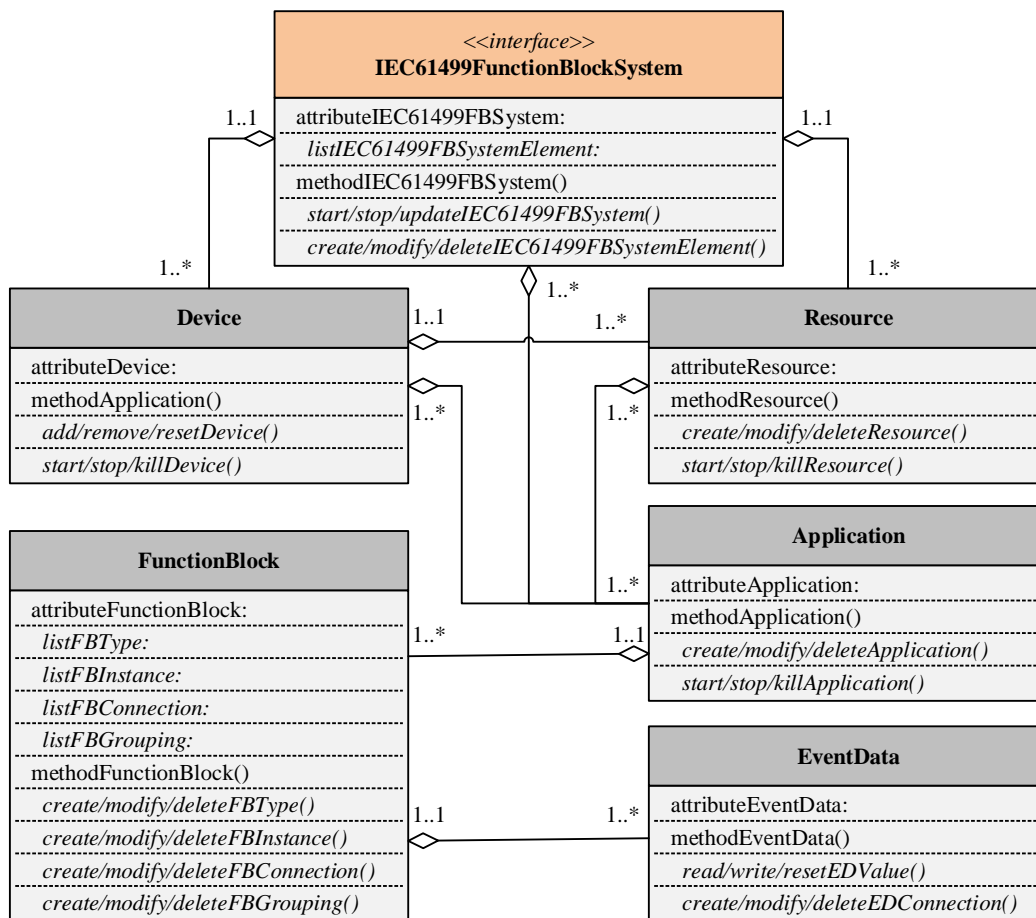


Figure 6-16: The *IEC61499FunctionBlockSystem* data model

### 6.3.5 Agent-Embedded Function Block Design Pattern

An agent-embedded IEC 61499 FB model is designed to support self-management capabilities of the low-level physical module. The key feature of this proposed model is the separation of the self-manageable service execution from the control application execution. More specifically, one execution path is responsible for control applications (built as IEC 61499 FBs for control purposes) and a second execution path is responsible for self-manageable services (designed as embedded multi-agent models for system configuration, optimization, healing, and protection purposes).

The agent-embedded function block (Figure 6-17a) is proposed as a new design pattern for IEC 61499 to build self-manageable control solutions. The basic FB type is used to create a new type called *Agent\_X FB*, including *Agent\_SMS FB*, *Agent\_SC FB*, *Agent\_SO FB*, *Agent\_SH FB*, and *Agent\_SP FB*. *Agent\_SMS FB* has at least two key state/action pairs in its execution control chart: a) *REQ* for requesting one or more self-manageable agents to respond to changes (implementing the *SelfManageableAgents* interface); b) *EXE* for executing self-manageable services provided by self-manageable agents to adapt FB based control systems (implementing the *IEC61499FunctionBlockSystem* interface). Self-manageable agents embedded in IEC 61499 FBs are either initialized by management events to be active for predefined tasks or deactivated in a sleep state. With this design pattern acting as meta-application for management purposes, the new agent-embedded FB types can be introduced to build self-manageable control applications (Figure 6-17b). In practice, in order for one IEC 61499 FB based application to be self-manageable, the *Agent\_SMS FB* type is required with one or more self-manageable agents for different tasks (i.e., configuration, optimization, healing, or protection).

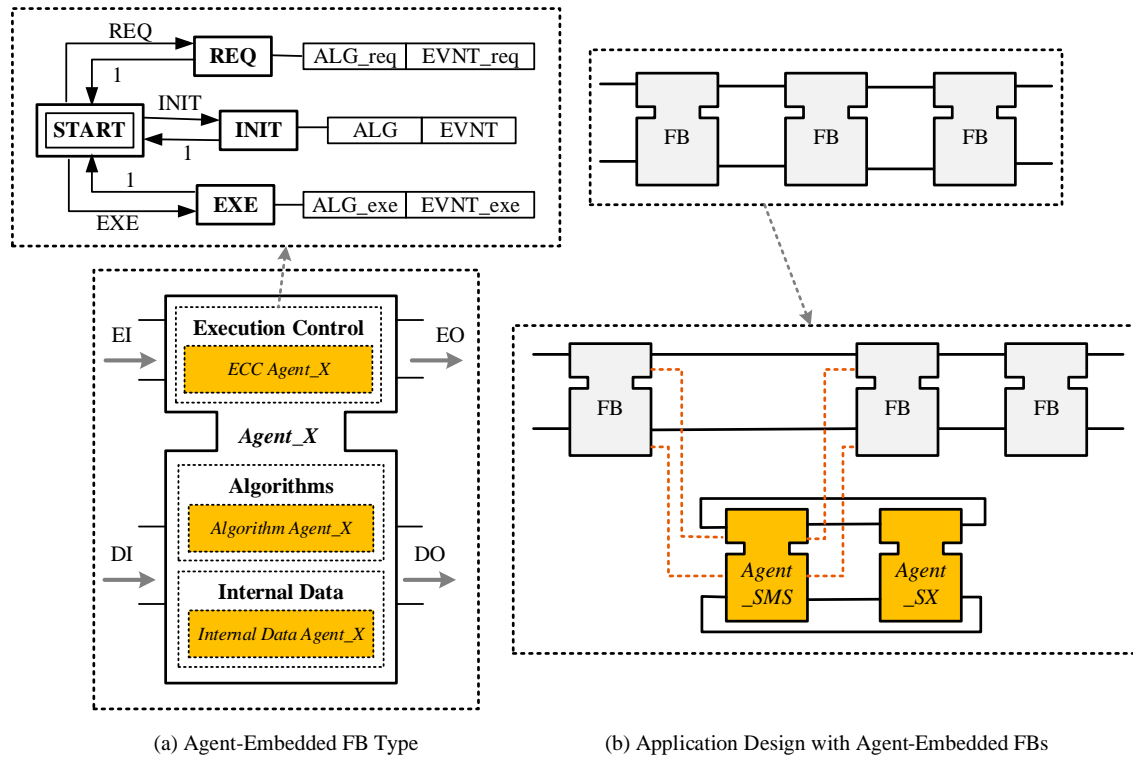


Figure 6-17: The agent-embedded IEC 61499 FB model

The design pattern of the agent-embedded FB module developed in IEC 61499 FB modelling tool Eclipse 4diac is shown in Figure 6-18. The module is designed as a template by using IEC 61499 sub-application type (i.e., SubApp) and the inside is designed as a network of agent-embedded FBs *Agent\_SMS*, *Agent\_SC*, *Agent\_SO*, *Agent\_SH*, and *Agent\_SP* by using IEC 61499 basic FB type (i.e., BFB).

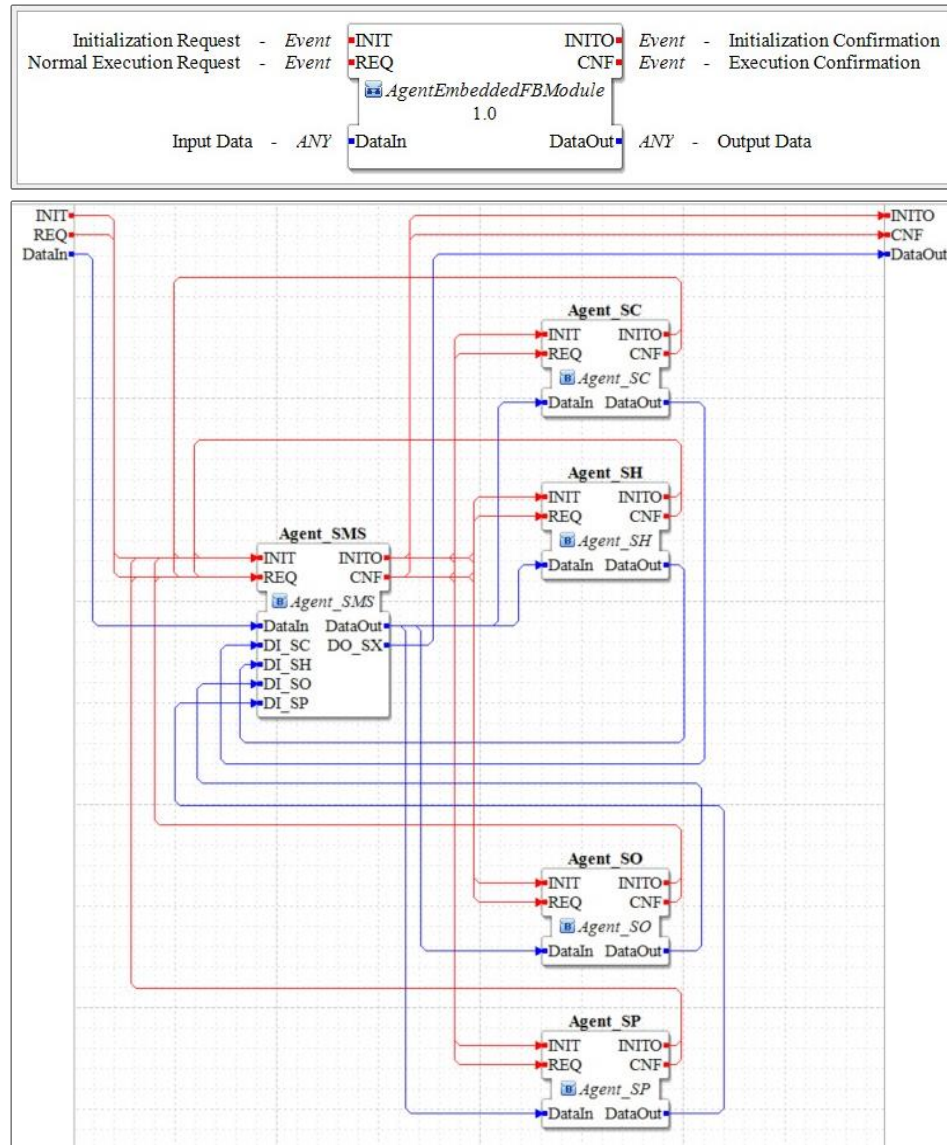


Figure 6-18: Interface and FB network design of the agent-embedded FB module

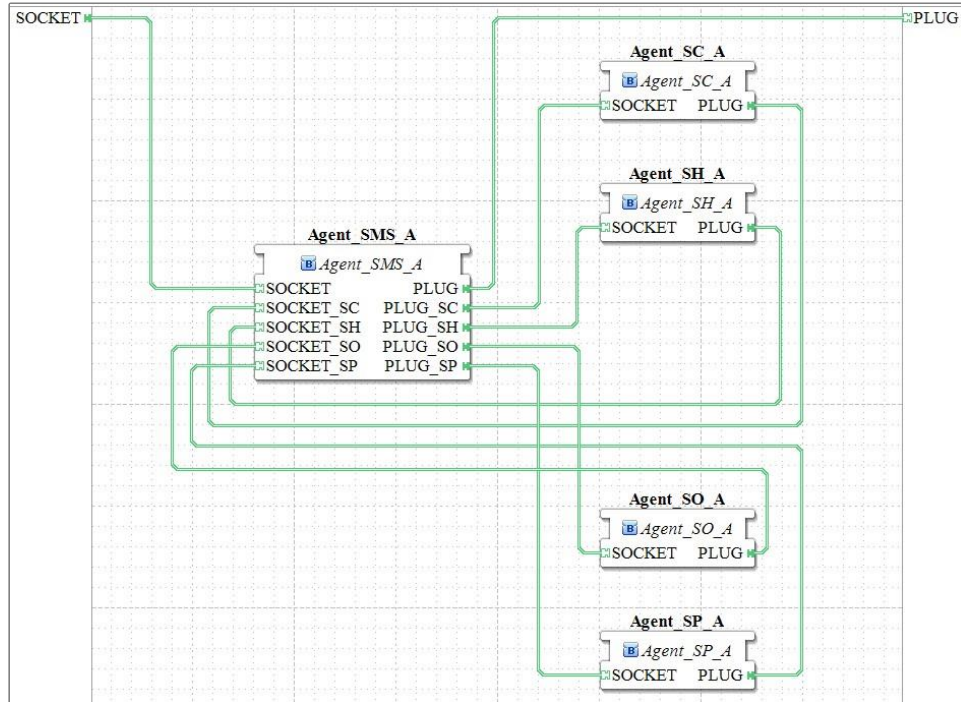
The interaction between the module and the external segments through the communication network (Figure 6-2d) are achieved by using the default IEC 61499 SIFB type (e.g., *publish/subscribe*, *client/server*). In Figure 6-19a, the internal agent-agent communication interface is designed as an *AgentInterface* adapter pair *plug* and *socket* for encapsulation of FB interaction with internal services by using IEC 61499 adapter type (i.e., *adapter*). Therefore, the



agent-embedded FB module in Figure 6-18 can also be designed with adapters in Figure 6-19b for the benefits discussed in Section 6.2.



(a) Agent Communication Interface Adapter Design



(b) FB Network of Agent-Embedded FB Module with Adapters

Figure 6-19: FB network design of the agent-embedded FB module with adapters

## 6.4 Summary

In this chapter, the architecture modelling framework for the low-level physical module was proposed. The low-level physical module design deploys autonomic computing self-managing properties into the modelling framework with the implementation of IEC 61499 FB modelling techniques. It is designed as agent-embedded IEC 61499 FB model with *Self-Manageable Service Execution Agent*, *Self-Configuration Agent*, *Self-Healing Agent*, *Self-Optimization Agent*, and *Self-Protection Agent*. The design results in a new design pattern to separate the execution of control applications and self-manageable services for low-level FB modelled automation solutions, aiming at realizing real-time adaption of automation logic and control algorithms.

*[This page intentionally left blank]*

## **Chapter Seven: Architecture Modelling Evaluation**

### **7.1 Introduction**

The major research focus in this thesis is on the design methods for system architecture modelling that enable industrial automation and control systems to be distributed and intelligent. Central to this work is a layered architecture design that focuses on the integration of multi-agent modelling and IEC 61499 FB modelling. In the proposed architecture modelling framework, a multi-agent computing model is designed for the high-level architecture with the aim of providing system intelligence by communicating and computing cores of cyber modules. An agent-embedded IEC 61499 FB model is developed for the low-level architecture in order to offer real-time adaptation by distributed and intelligent control of physical modules. It aims to enable systems to automatically discover alternative solutions, flexibly coordinate reconfigurable modules, and actively deploy corresponding functions, to quickly respond to frequent changes and intelligently adapt to evolving requirements in dynamic environments.

In this chapter, scenarios are first provided to illustrate the proposed modelling framework. Then a multi-agent simulation model based on the agent modelling tool NetLogo is developed and an experimental testbed on the Jetson Nano and Raspberry Pi platforms is designed for demonstration and evaluation. Finally, the performance is theoretically analyzed to evaluate the proposed architecture modelling framework.

## 7.2 Illustrative Example Demonstration

### 7.2.1 Typical Industrial Scenario

A simplified but typical scenario was described in Section 2.1 (Figure 2-1) and was enriched throughout the following chapters to explain the proposed architecture modelling framework. In this section, this typical industrial scenario will be used to wrap up the design of multi-layer automation architectures that aims to enable real-time adaptation at the device level and run-time intelligence throughout the whole system. Figure 7-1 shows the extended industrial automation scenario in which the system is designed to sort objects into corresponding bins with a group of autonomous mobile robots to deliver objects to the conveyor system and to carry bins back to storage areas. A detailed description is shown as follows:

- a group of autonomous mobile robots are designed to be able to deliver objects to the conveyor for sorting;
- the programmable robotic arm can rotate and translate to grasp and place objects from the conveyor into corresponding bins on the fly;
- the task for the robotic arm is to pick up one type of object from the conveyor and then place them into the corresponding type of the bin;
- the other group of autonomous mobile robots are designed to be able to carry the full bins back to storage areas according to the flashing LED lights;
- the group of autonomous mobile robots are attached with wireless sensor networks for understanding the surrounding environment (e.g., localization and navigation); and
- the system is designed and programmed to achieve its purposes with the help of all other necessary software/hardware that are not mentioned here.

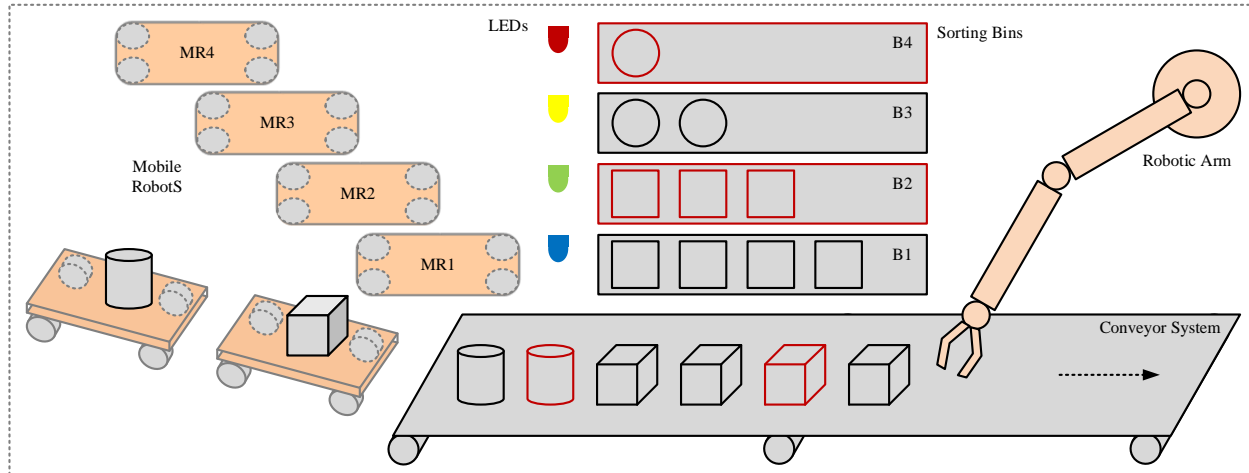


Figure 7-1: An extended industrial automation scenario

According to the proposed architecture modelling framework (Figure 5-1 and Figure 6-1), the system is generally designed as a two-layer architecture model to support real-time adaptation at the device level and run-time intelligence throughout the whole system. The high-level cyber module is designed as a multi-agent computing model consisting of *Monitoring Agent*, *Analysis Agent*, *Self-Learning Agent*, *Planning Agent*, *Execution Agent*, and *Knowledge Agent*. Multi-agent modelling uses autonomous and cooperative agents to achieve run-time intelligence in system design and module reconfiguration. The low-level physical module is designed as an agent-embedded IEC 61499 FB model with *Self-Manageable Service Execution Agent*, *Self-Configuration Agent*, *Self-Healing Agent*, *Self-Optimization Agent*, and *Self-Protection Agent*. IEC 61499 FB modelling applies object-oriented and event-driven FBs to realize real-time adaptation of automation logic and control algorithms.

### 7.2.2 High-Level Cyber Module Design

The high-level cyber module is expected to reside somewhere in the “*Cloud*” where the whole system is managed with enabling techniques for run-time intelligence.

*Agent\_Monitoring.* It is designed to collect data on the operating environment properties, the system engineering properties, and the real-time operation behaviours of each system module, typically through sensors or sensor networks. For the industrial automation scenario shown in Figure 7-1, these data can be environmental conditions (e.g., temperature/humidity that may affect autonomous mobile robots operating due to battery degradation, or unexpected obstacle on the ground that prevents autonomous mobile robots from moving around), engineering properties (e.g., moving speed and load capacity of autonomous mobile robots, tear or wear on most frequently used parts), and working states (e.g., idle time percentage of autonomous mobile robots or the robotic arm and cooperation efficiency between them, if best path found each time for autonomous mobile robots, sorting efficiency of the robotic arm).

*Agent\_Analysis.* It is designed to model system operating situations to understand current operation states and to predict future situations. In simple operations, if nothing monitored changed by comparing current states with pre-set values, the system will go directly for execution as planned. For example, the incoming stream of objects monitored as the same type (e.g., black block), the robotic arm will perform sorting as planned. However, in complex operations, as monitored states changed, the system has to go through new analysis and planning before execution. For example, a simple case is one red block mixed in the stream of black blocks and the robotic arm needs to distinguish between them and then sort each into the corresponding bins.

*Agent\_Self-Learning.* It is designed to employ artificial intelligence to support *Agent\_Analysis* as traditionally only predefined rules, policies, and goals are considered with limited situations. *Agent\_Self-Learning* can either learn primitive skills from sensory data or learn from past experience to cope with new tasks or to optimize existing performances. For

example, the robotic arm starts working with some predefined knowledge of normal operations through training sensory data (e.g., reach, grasp, and place objects). At the beginning of these simple situations, *Agent\_Self-Learning* could actively collect data for analysis and build models for prediction or optimization (e.g., rotation speed and angle, holding force and opening angle of the robotic arm). As enough data is available and robust models are built, *Agent\_Self-Learning* could passively receive data for analysis and apply models for prediction or optimization. However, for complex situations with no previous experiences available, *Agent\_Self-Learning* could actively adjust actions by trial-and-error to achieve the best result (e.g., the robotic arm adjusts its holding force and opening angle to catch a bigger and heavier block). Then these complex situations become simple situations with available solutions. *Agent\_Self-Learning* is key to the cyber module to enable the system to become intelligent to handle dynamic situations as not all situations can be considered at the beginning of the system design.

*Agent\_Planning*. It is designed to determine the optimal action plan with a series of actions to achieve goals, generally working in complex situations where the regular action plan needs to be adapted. For example, a bigger black block comes for the robotic arm. As monitored engineering features (e.g., dimension and mass) of objects have changed, *Agent\_Planning* has to provide an adapted action plan according to *Agent\_Analysis* by considering changes of some types of parameters (e.g., holding force and opening angle of the gripper). The other case is with large monitoring data of the same type of objects, *Agent\_Planning* will update the regular action plan according to *Agent\_Self-Learning*.

*Agent\_Execution*. It is designed to work with actuators to carry out action plans. For example, the autonomous mobile robot delivers a bigger and heavier object to the conveyor system and at the same time, the robotic arm increases the holding force and opening angle of the



gripper to try to catch that object. As monitored states of the object delivered from the autonomous mobile robots to the conveyor system changed, the action plan needs to be adapted accordingly for execution.

*Agent\_Knowledge*. It is designed to either directly manage all sources of desired data or indirectly cooperate with other databases to support the functioning of the multi-agent computing model. For example, the change detected and its corresponding solution generated by the cyber module will be stored and accessible to *Agent\_Knowledge* for the future same scenario (e.g., irregular objects are shared between the robotic arm and autonomous mobile robots, and the specific path for moving irregular objects will also be shared among autonomous mobile robots).

### **7.2.3 Low-Level Physical Module Design**

The low-level physical module is expected to be attached to the “*Edge*” where each automation module is programmed with control algorithms capable of real-time adaptation. Considering the industrial automation scenario shown in Figure 7-1, the system is developed based on the proposed agent-embedded IEC 61499 FB modelled control solutions to support self-management capabilities (Figure 6-1). The conveyor section shown in Figure 7-1 is the most critical one of the whole conveyor system where a backup motor (the same as the primary motor) is connected to this section. In abnormal operations (e.g., change of motors), as requested from the system, *Agent\_SMS* can initiate *Agent\_SH* for self-healing or *Agent\_SP* for self-protection. Generally, *Agent\_SH* activates the backup motor running plan in a recoverable situation where only the primary motor is not working, and *Agent\_SP* gives out warning signals in an unrecoverable situation (e.g., the backup motor stops working again because of system damage, or the two motor working together due to fake failure of the primary motor). In normal operations (e.g., regular operating states), as per the system’s request, *Agent\_SO* can be initiated to provide an

optimized service plan for *Agent\_SMS*. For example, the motor speed is reduced due to a variety of different objects coming as the robotic arm requires more time to ensure sorting precision, and the motor speed can increase a little bit to increase sorting efficiency if same objects coming in a period of time. *Agent\_SC* can be initialized in each situation for system reconfiguration.

### **7.3 Multi-Agent Simulation Model**

The simulation of the proposed design can help facilitate understanding of how the system designed under the proposed architecture modelling framework will perform when actually being implemented. In this section, the simulation model is developed for an automated conveyor system by using the multi-agent modelling tool NetLogo to demonstrate the proposed system design. NetLogo is a multi-agent programmable modelling environment for the simulation of multi-agent systems that involves a large number of agents [183-184].

#### ***7.3.1 Development of Agent-Based Model***

The automated conveyor system consists of a series of conveyor sections and part storage bins that are connected by part diverters (Figure 7-2). Conveyor sections of the same type (main sections and entrance/exit sections) are designed to be the same length and to operate at the same fixed speed. Part storage bins are designed as the destinations for incoming parts, which could also represent part workstations with corresponding re-routing to storage bins and/or conveyor sections for more complex simulations. The diverters are controlled by individual diverter controllers as highlighted in Figure 7-2. Each of these controllers includes a processor (i.e., an IEC 61499 device), an input sensor (e.g., an imaging device to detect the part type), and one or more diverter actuators. The objective of the automated conveyor system is to sort parts into storage bins based on part types (i.e., each part type is assigned to a unique storage bin). The automated conveyor system is also designed with extra part storage bins (i.e., 7 storage bins and at most 6 part types) and storage loops for the part types to circulate through the conveyor loop in case any disturbance happens (e.g., a new part type or a conveyor section failure). The automated conveyor system designed in the experiments could be regarded as part of typical

industrial systems, e.g., working together with robotic arms and autonomous mobile robots as described before in Figure 7-1.

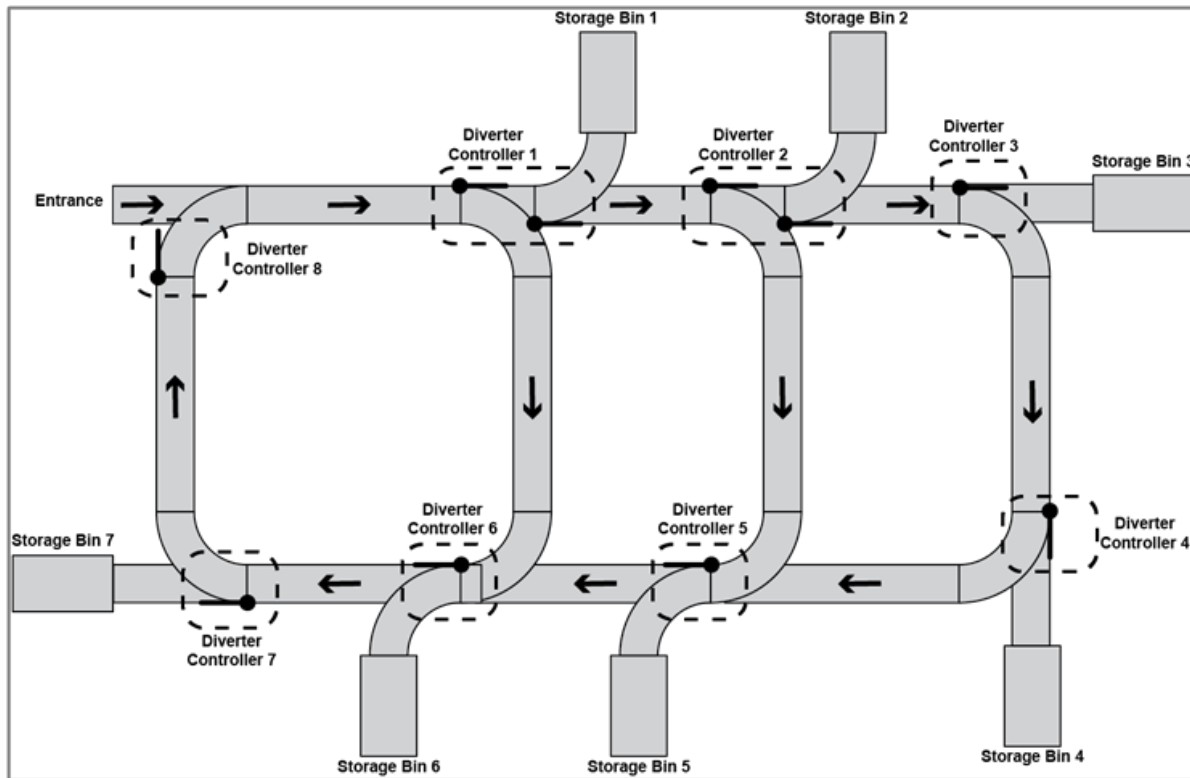


Figure 7-2: The automated conveyor system

To simulate reconfigurable automated conveyor systems of the type shown in Figure 7-2, the multi-agent simulation model is developed in NetLogo 6.3 as shown in Figure 7-3 (Appendix A). The automated conveyor system simulation model allows the user to specify the automated conveyor system layout, the number of part types, the initial diverter controller specifications, the system processing and agent response delays, the conveyor section failures, and the system mean time to failures. In the simulation model (Figure 7-3), conveyor sections are represented by white rectangular; diverter sections are represented by yellow squares with directional arrows showing the current position of the diverter; storage bins are represented by red squares; input

and output part sensors are represented by green squares; parts are represented by boxes with a letter and a unique colour assigned to each part type.

For the experiments in this research, main conveyor sections are designed as 7 part spaces long and entrance/exit conveyor sections are designed as 3 part spaces long. The system operates at a fixed speed with 1 part space per second or tick. Part arrival times are sampled from an exponential distribution based on a mean arrival time and each part is randomly assigned a part type that is sampled from a discrete uniform distribution with a range of 1 to  $n$ . Parts are transported through the system on the conveyor sections and routed to their assigned storage bins by the diverter controllers. The part routing policy is determined through two-levels as proposed in this thesis (Figure 5-1 and Figure 6-1): high-level cyber module for the overall system (e.g., part types [a, b, c, d] to storage bins [1, 2, 6, 3]) and low-level physical module at the execution (e.g., part type c re-routed to storage bin 5 due to conveyor section 4 failure or new part type e to storage bin 5). The system performance shown in Figure 7-3 is measured by average number of parts in the system ( $L$ ), average part arrival time to the system ( $1/\lambda$ ), and average wait time spent by parts in the system ( $W$ ), in which the simulation compares favourably with the theoretical model by Little's Law ( $L = \lambda W$ ) [185].

In addition to the automated conveyor system simulation, the agents designed in the self-manageable service model (Figure 6-13) and represented by "*Function Block*" (Figure 7-3) are simulated as individual Netlogo agents. Each of the service agents interacts directly with the automated conveyor system to monitor its operations and execute self-manageable services (i.e., self-configuration, self-optimization, self-healing, and self-protection). These self-manageable services are achieved by dynamically updating and changing the global *conveyor-sections* list (i.e., input file to configure conveyor system layouts), *part-types* list (i.e., part types introduced

to the system), *diverter-states* list (i.e., reconfiguring diverter states), and *diverter-connections* list (i.e., removing/establishing connections between diverters) in the program. The agents designed in the high-level MAPLE-K model (Figure 5-1) and represented by “Person” (Figure 7-3) are simulated as NetLogo agents to demonstrate the integrations among them. Currently, these agents are developed for the simulation of the automated conveyor system, and then are further required to be designed as external Python procedures.

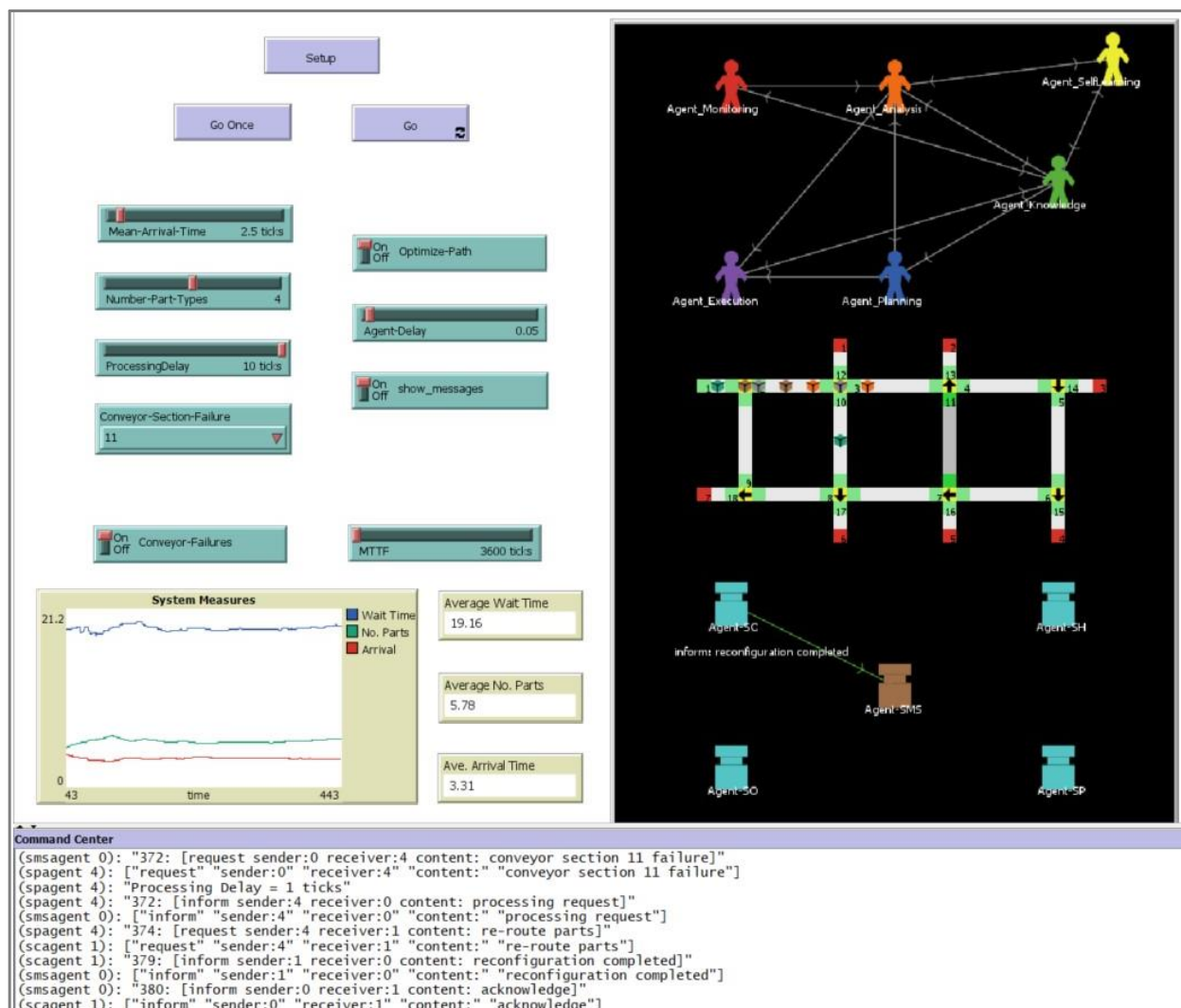


Figure 7-3: Agent-based simulation model for the automated conveyor system

To validate the proposed architecture modelling framework, especially the low-level self-manageable architecture model, a series of three experiments (7 typical tests in Figure 7-4) are

performed to test the system self-managing functionality by introducing random disturbances (one disturbance per time): a) introduction of a new part type (i.e., increase *number-part-types* from the simulation interface), b) failure of a conveyor section (i.e., select *conveyor-section-failure* from the simulation interface), and c) optimizing part routing (i.e., switch *optimization-path* on from the simulation interface). As discussed before, each system has its initial design with predefined running conditions and adaptation logics. The major advantage of the proposed design over the traditional design is that these programmed control solutions are able to be improved and enriched intelligently during runtime through self-learning from system running conditions and operation histories. It is the same with the simulation experiments, in which the system is running with initial conditions while random disturbances are introduced to simulate the stochastic nature of real operations. Details are discussed in the following sections.

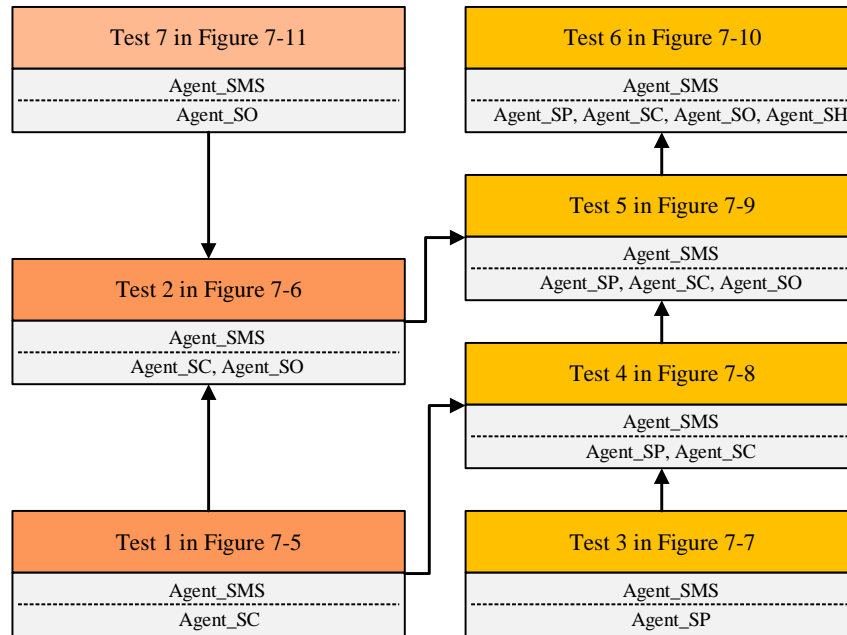


Figure 7-4: Typical tests in performed three experiments

### 7.3.2 Introduction of A New Part Type

The first experiment, the introduction of a new part type, involves tests of the self-configuration capability that is considered fundamental for system intelligent reconfiguration. *Agent\_SC* initiates the self-configuration process upon receiving a *<request: new part type e>* message from *Agent\_SMS* and maintains the present diverter controller configuration while it processes the request. In the case of a new part type introduced, this results in parts of the new type circulating around the conveyor loops until the reconfiguration is completed by *Agent\_SC* with an *<inform: reconfiguration completed>* message to *Agent\_SMS*.

A typical self-configuration scenario (Test 1) is shown in Figure 7-5: an interaction between *Agent\_SMS* and *Agent\_SC* in response to a new part type arriving at the automated conveyor system. In this test, *Agent\_SMS* is monitoring the operating state of the automated conveyor system. When it senses the arrival of a new part type (i.e., system reconfiguration required), *Agent\_SMS* sends a message to *Agent\_SC* to initialize the agent and request the change, indicating that a new part type *e* has been introduced to the system (system initially running with part types [a, b, c, d]). *Agent\_SC* responds to the request and works on a solution to the change. Once the new routing is determined, *Agent\_SC* informs *Agent\_SMS* with the reconfiguration plan and *Agent\_SMS* responds to *Agent\_SC* for acknowledgement and executes the reconfiguration to the diverter controllers.



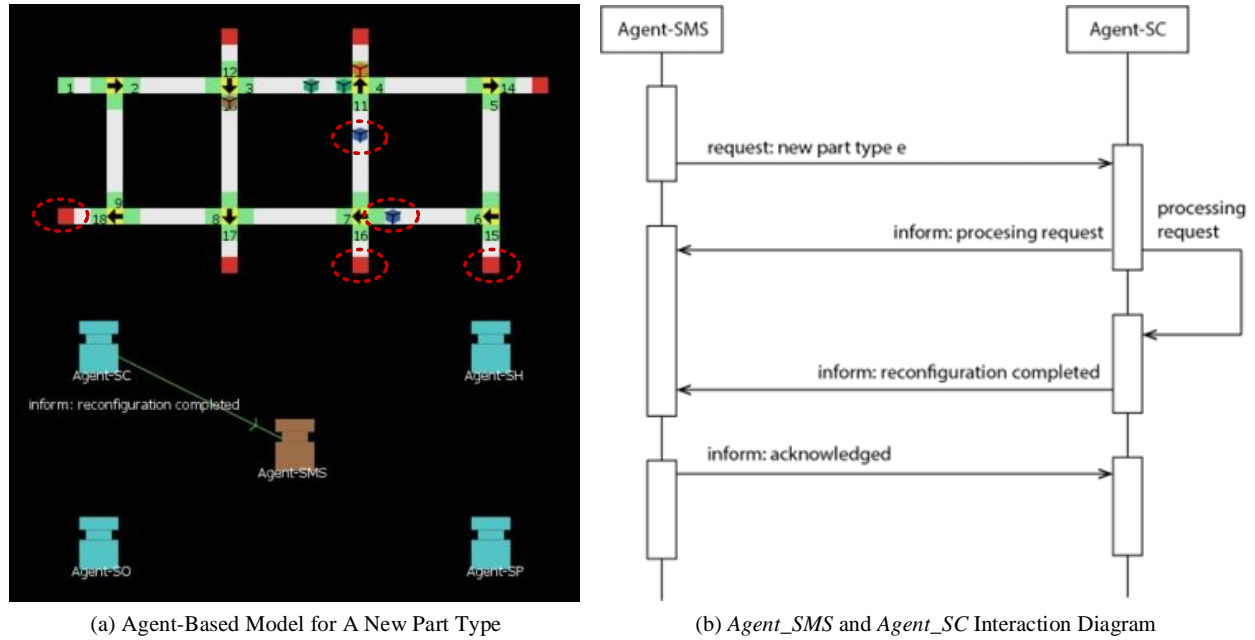


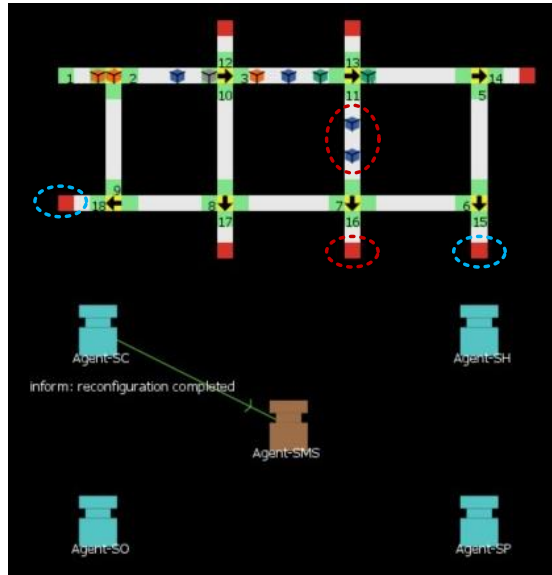
Figure 7-5: Introduction of a new part type: Test 1

In Test 1 (Figure 7-5), except storage bins [1, 2, 6, 3] for initial part types [a, b, c, d] occupied, several system configurations exist for the new part type e in blue (i.e., increase *number-part-types* from 4 to 5), which means in the simulation the new part type e circulates through conveyor loops to find extra storage bins (bins [4, 5, 7] in red circle) automatically governed by the proposed design. There are several loops to reach each extra storage bin, e.g.:

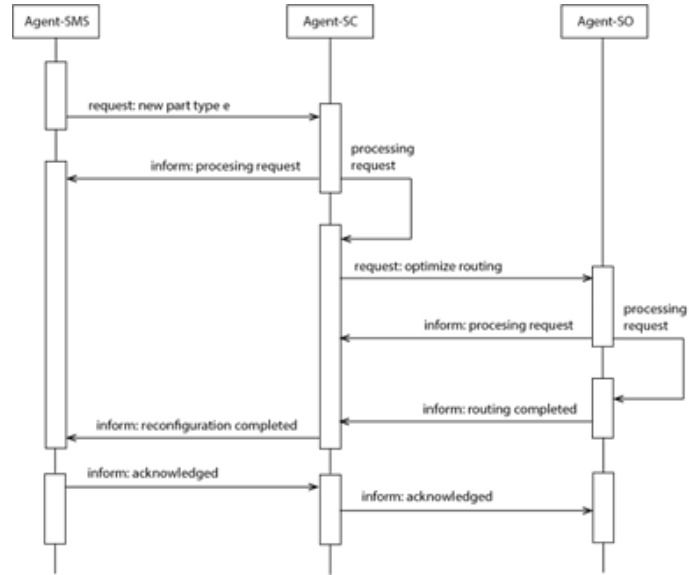
- route 1 conveyor sections [1->2->10->8->18] or else to storage bin 7,
- route 2 conveyor sections [1->2->3->11->16] or else to storage bin 5, and
- route 3 conveyor sections [1->2->3->4->5->15] or else to storage bin 4.

In most cases, the optimal system configuration is required for the new part type if possible. In Test 2 (Figure 7-6), once *Agent\_SC* has determined that several system configurations are possible, it sends a *<request: optimize routing>* message to *Agent\_SO* (Figure 7-6b) for self-optimization. In this case, self-optimization involves finding an optimum routing of the new part type to storage bins (e.g., Test 7 in Section 7.3.4 describes one possible self-

optimization approach). The resulting routing strategy (e.g., red circle route 2 conveyor sections [1->2->3->11->16] to storage bin 5) is then used by *Agent\_SC* to execute the diverter controller reconfiguration. For the automated conveyor system, this involves updating the execution control charts (ECC) of the diverter controller FBs to ensure that parts are diverted to the appropriate storage bin or conveyor section based on the upstream conveyor section's output sensor signal.



(a) Agent-Based Model for A New Part Type



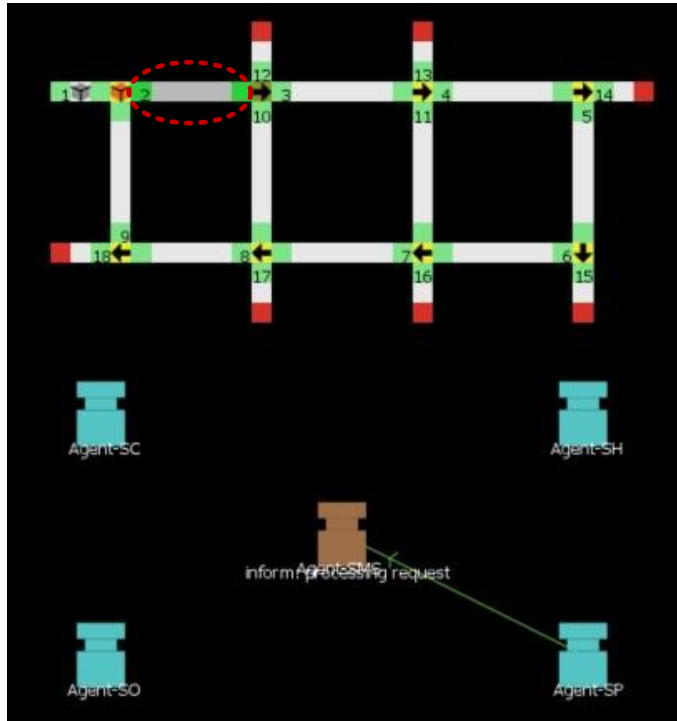
(b) *Agent\_SMS*, *Agent\_SC*, and *Agent\_SO* Interaction Diagram

Figure 7-6: Introduction of a new part type: Test 2

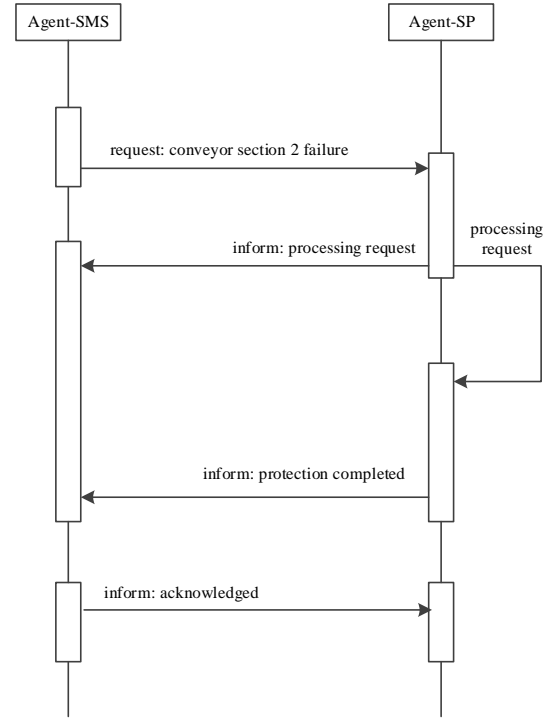
### 7.3.3 Responding to A Conveyor Section Failure

In this experiment, an equipment failure such as a conveyor section failure is introduced to the simulation to test one or more self-management capabilities. As shown in Figure 7-2, the importance of each conveyor section in the automated conveyor system is different. For example, conveyor sections 1 and 2 are more important than others (e.g., conveyor sections 3 and 4) as failures in these conveyor sections will cause the system to stop running. For this experiment, different simulations are run for different test cases.

A very simple case (Test 3) is that failures occur in entrance conveyor sections (e.g., grey section 2 in red circle) as shown in Figure 7-7a. In this case, the system reconfiguration is not possible (i.e., no extra route available), and *Agent\_SMS* will send a *<request: conveyor section 2 failure>* message to *Agent\_SP* directly for the self-protection plan (e.g., full system shutdown). *Agent\_SH* could also be triggered in parallel to respond to the conveyor section failure if the malfunction can be healed automatically (e.g., Test 6 in Figure 7-10).



(a) Agent-Based Model for A Conveyor Section Failure

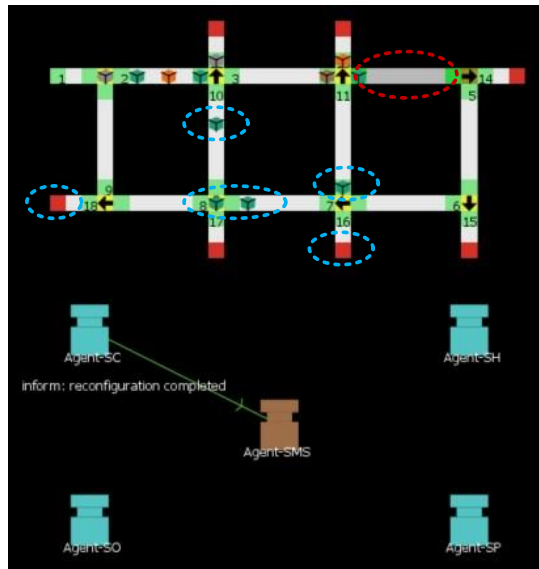


(b) *Agent\_SMS* and *Agent\_SP* Interaction Diagram

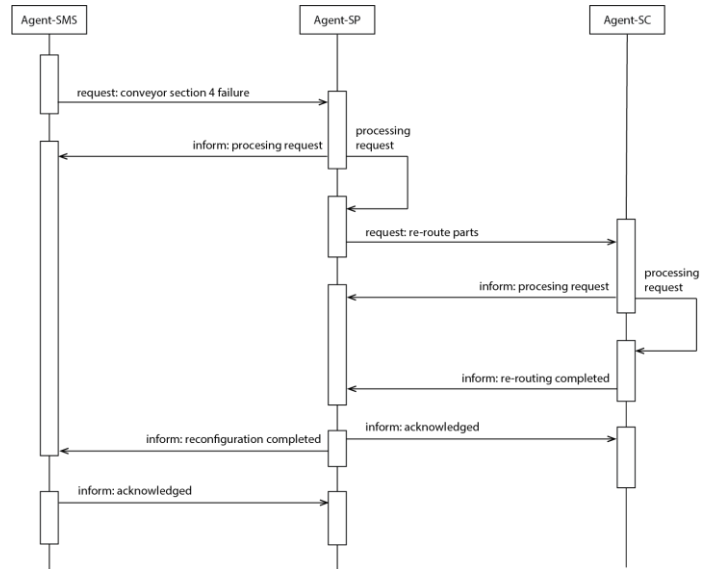
Figure 7-7: Responding to a conveyor section failure: Test 3

The simulation snapshot shown in Figure 7-8 shows another typical scenario (Test 4): an interaction between *Agent\_SMS*, *Agent\_SP*, and *Agent\_SC* in response to a non-important conveyor section failure that is in healing (grey section 4 in red circle). In this case, *Agent\_SMS* is monitoring the operating state of the automated conveyor system. When it senses the failure of a conveyor section that is not healed (i.e., operation protection and system reconfiguration

required), *Agent\_SMS* sends a *<request: conveyor section 4 failure>* message to *Agent\_SP* to initialize the agent and request the change (operation protection first), indicating that the incoming parts (green boxes in blue circle) using that failed conveyor section have to be re-routed. *Agent\_SP* responds to the request and then coordinates with *Agent\_SC* to re-route incoming parts (if system reconfiguration possible). Once the new routing is determined, *Agent\_SC* informs *Agent\_SP* that the system can be reconfigured with a new routing (Test 1 in Figure 7-5). Then *Agent\_SMS* works together with *Agent\_SP* and *Agent\_SC* to execute the reconfiguration plan to solve the problem (i.e., the incoming parts are re-routed from storage bin 3 to bin 5 or 7 in blue circle). In this process, if system reconfiguration not possible (i.e., no extra route available), *Agent\_SP* will inform *Agent\_SMS* to execute the self-protection plan (Test 3 in Figure 7-7). In this scenario, *Agent\_SH* could also be triggered in parallel to respond to the conveyor section failure if the malfunction can be healed automatically (e.g., Test 6 in Figure 7-10).



(a) Agent-Based Model for A Conveyor Section Failure



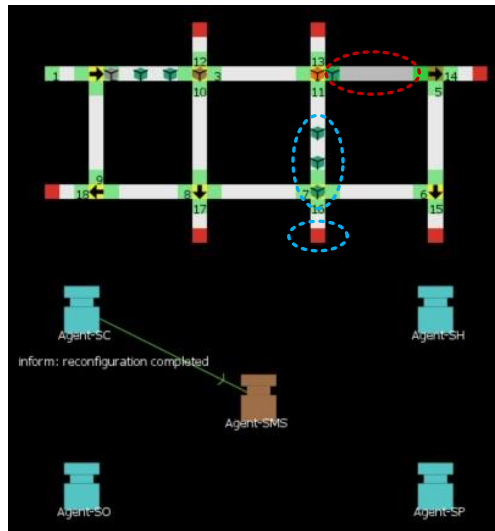
(b) *Agent\_SMS*, *Agent\_SP*, and *Agent\_SC* Interaction Diagram

Figure 7-8: Responding to a conveyor section failure: Test 4

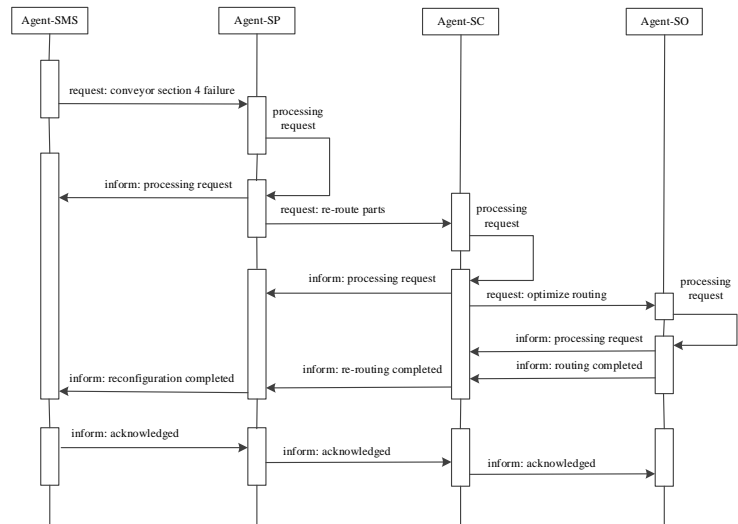
In Test 4 (Figure 7-8), the system is initially running with part types [a, b, c, d] to storage bins [1, 2, 6, 3]. Due to conveyor section 4 failure (i.e., set *conveyor-section-failure* as 4), the green part type d to storage bin 3 requires re-routing, which means in the simulation the part type d circulates through conveyor loops to find extra storage bins (bins [5, 7] in blue circle) automatically governed by the proposed design. Several system configurations exist, e.g.:

- route 1 conveyor sections [1->2->3->11->16] or else to storage bin 5,
- route 2 conveyor sections [1->2->10->8->18] or else to storage bin 7, and
- route 3 conveyor sections [1->2->3->11->7->8->18] or else to storage bin 7.

The remaining process after self-protection for conveyor section 4 will be the same case in Test 2 (Figure 7-6). One of the optimal re-routing strategies (e.g., route 1 conveyor sections [1->2->3->11->16] to storage bin 5) is selected for re-routing the incoming green part type from storage bin 3 to 5. In this case, self-optimization involves finding an optimum routing of the new part type to storage bins (e.g., Test 7 in Section 7.3.4 describes one possible self-optimization approach).



(a) Agent-Based Model for A Conveyor Section Failure



(b) *Agent\_SMS*, *Agent\_SP*, *Agent\_SC*, and *Agent\_SO* Interaction Diagram

Figure 7-9: Responding to a conveyor section failure: Test 5

In a complex scenario (Test 6) as shown in Figure 7-10, all agents are called upon to first protect the automated conveyor system from further damage, configure the system to operate in a degraded state, heal the failure conveyor section, and optimize the system configuration for normal operation. In this test, the failed section (grey section 11) may have occurred because of a conveyor malfunction (failed conveyor motor, broken belt, etc.), an input or output sensor failure, or a blockage.

As with the previous case, *Agent\_SMS* identifies the disturbance; however, given that this case involves equipment failure or malfunction, it prioritizes self-protection and send a *<request: conveyor section 11 failure>* message to *Agent\_SP*, as illustrated in Figure 7-10. Once this request is received, *Agent\_SP* first determines if a full system shutdown is required (e.g., if an entrance conveyor section such as conveyor section 1 or 2 is affected). In this case, it would shut down the conveyor system and then requests *Agent\_SH* to plan for a repair of the system (Test 3 in Figure 7-7). If *Agent\_SP* determines that it is safe to continue to operate, it sends *Agent\_SC* a request to re-route the parts and then requests *Agent\_SH* to plan for a repair of the system (Test 4 in Figure 7-8). Once *Agent\_SC* has determined that several system configurations are possible, it send a *<request: optimize routing>* message to *Agent\_SO* for an optimum routing of part types to storage bins (Test 5 in Figure 7-9). An example of one possible self-optimization approach that could be used by *Agent\_SO* is described in Section 7.3.4 (Test 7 in Figure 7-11).

To highlight this two-part process (system keeps running while operation protected and failure in healing), the agent interactions is shown in Figure 7-10. The whole process starts with self-protection in case any incoming part uses the conveyor section with failures and ends with self-healing to recover the conveyor section from failures, and the self-configuration and self-optimization process for available optimal system reconfigurations is repeated between them.

This repetitive re-routing process is initiated by *Agent\_SP* and *Agent\_SH*. First, *Agent\_SC* and *Agent\_SO* perform a reconfiguration of the diverter controllers that accounts for the new state of the automated conveyor system requested by *Agent\_SP*. Once the reconfiguration is completed, the automated conveyor system operates in a degraded state (i.e., parts are routed around the failed conveyor section and placed in available storage bins) until *Agent\_SH* has managed the repair. For example, this may involve sending a message to a maintenance person and waiting for the repair to be performed. Once the failed conveyor section is back on-line, *Agent\_SH* requests *Agent\_SC* and *Agent\_SO* to re-route the parts, and the same process that was initiated by *Agent\_SP* to place the system in a degraded state is repeated. However, at this point, the system will be placed back in its normal operating state.

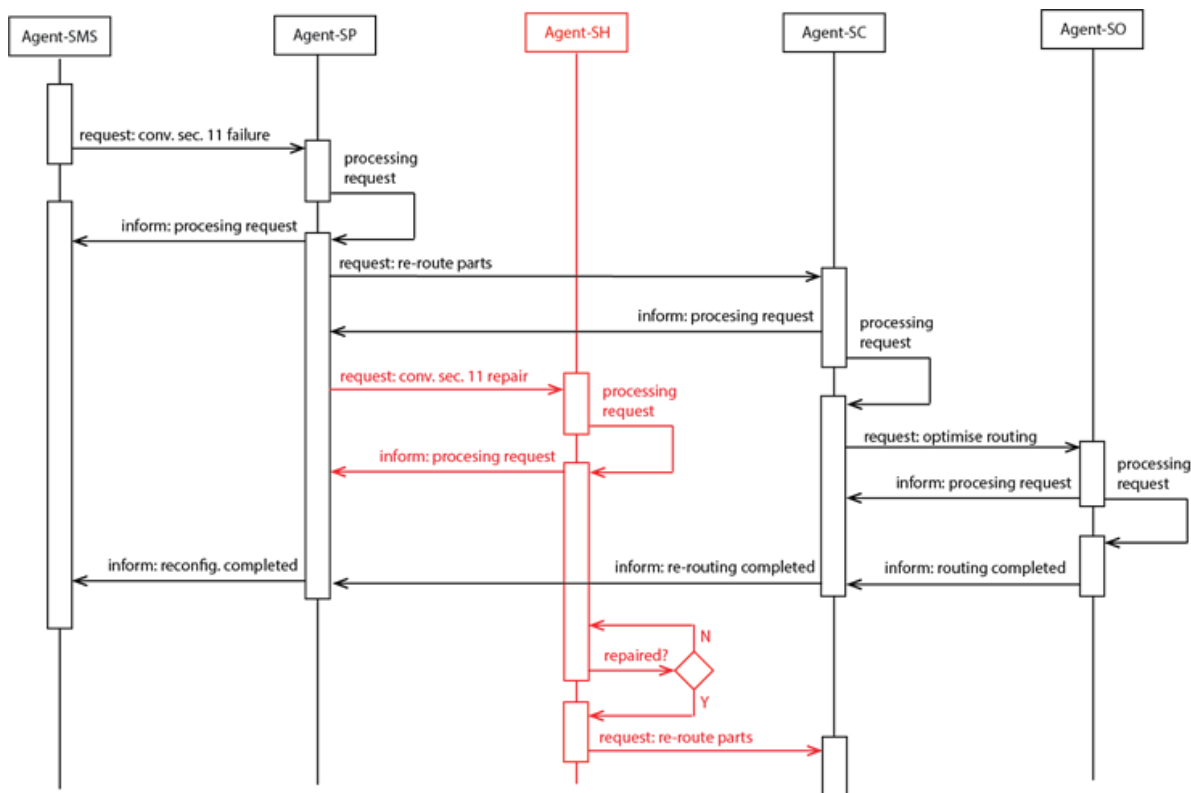


Figure 7-10: All agent interaction diagram in Test 6

### 7.3.4 Optimizing Part Routing

As described in the previous sections, the reconfiguration process involves a self-optimization step that is performed by the self-optimization agent, *Agent\_SO*. Depending on the nature of the optimization to be performed, a wide variety of optimization algorithms or heuristics can be used by *Agent\_SO*. For example, dynamic programming theory-based approaches such as Dijkstra's algorithm [186] can provide an optimal shortest path solution. However, given the computational complexity of optimal shortest path approaches, a variety of heuristic approaches have been proposed for real-time applications that include strategies such as limiting the area searched, decomposing the search problem, and limiting the links searched [187]. For this experiment, a heuristic shortest path approach is chosen and developed within the multi-agent simulation model given the real-time nature of the automated conveyor system application.

The goal of the part routing heuristic is to find the shortest routes to each part storage bin for any given automated conveyor system configuration. To accomplish this, the part routing agent-based model (Figure 7-11) uses two NetLogo agent types to encapsulate the shortest path functionality within *Agent\_SO*: a) a single fixed-position *entrance* agent (the blue circle in Figure 7-11) that represents the automated conveyor system entrance conveyor section, and b) mobile *diverter* agents (green triangles in Figure 7-11) that represent each of the conveyor system's diverters. The directed links point to each *diverter* agent's upstream *diverter* agent(s) or *entrance* agent.

The part routing heuristic is initialized by *Agent\_SC* providing *Agent\_SO* with a list of diverters and their corresponding upstream diverter sections: e.g., the initial state shown in Figure 7-11a corresponds to the normal operating state of the automated conveyor system (entrance conveyor + 8 diverters). On setup, the *diverter* agents first create directed links to all



directly preceding *diverter* and/or *entrance* agents (i.e., conveyor sections that feed parts to the diverter). As the model runs, each of the *diverter* agents perform a sequence of three behaviours during each simulation step: a) point to the *diverter* (or *entrance*) agent(s) feeding the parts; b) move one step forward; and c) maintain a minimum spacing between agents.

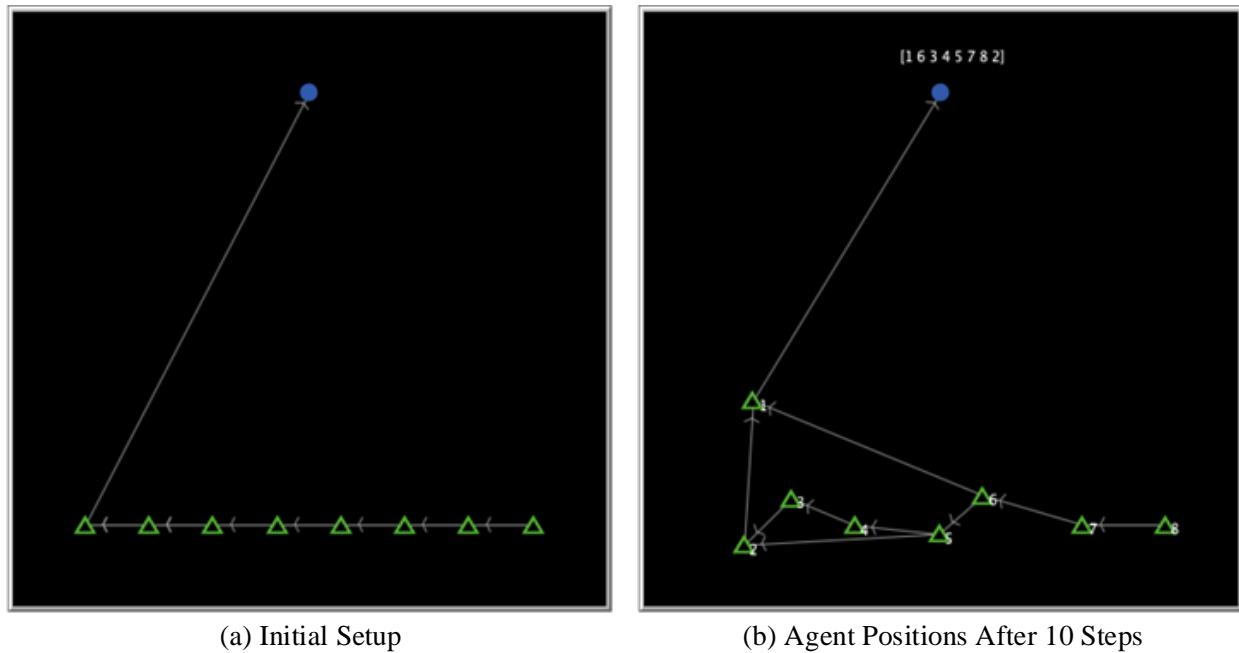


Figure 7-11: The part routing agent-based model: Test 7

Figure 7-11b shows the new positions of the *diverter* agents after 10 simulation steps. Once the simulation has stabilized (typically after 30-40 steps for this set of inputs), a pattern emerges that corresponds to the shortest distances from the entrance to each of the diverters. An example of this pattern for the normal operating state of the automated conveyor system is shown in Figure 7-12. At the top of the figure, a sorted list,  $P$ , is generated that shows the order of *diverter* agents by their distance to the *entrance* agent: i.e., a minimum distance routing order for the system configuration with the diverter order from closest to furthest is *diverter* 1, *diverters* 2 or 6, *diverters* 3 or 5 or 7, and *diverters* 4 or 8. Therefore, there are totally 24 possible

shortest routings ( $P_{\text{total}} = 1*(2*1)*(3*2*1)*(2*1)$ ) determined by the part routing agent-based model and the  $P = [1\ 2\ 6\ 3\ 5\ 7\ 4\ 8]$  routing order is one of the optimal part routing strategies.

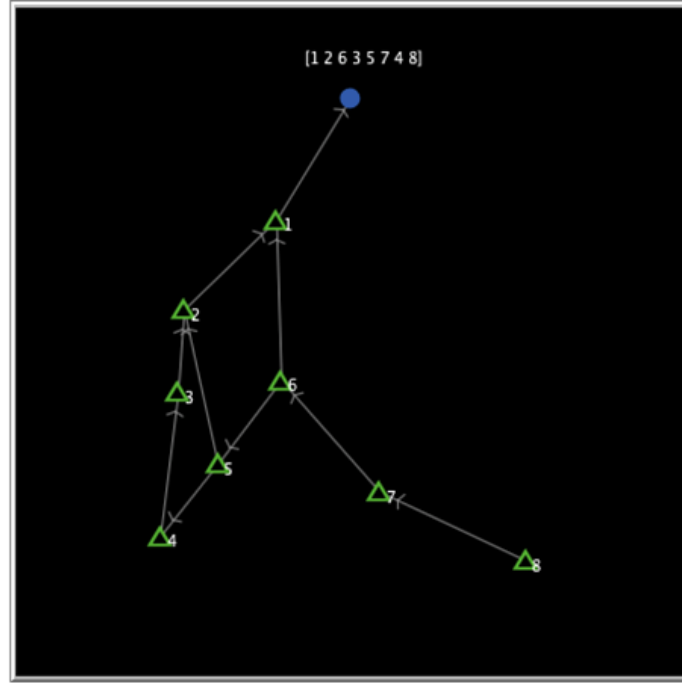


Figure 7-12: The final positions of the *diverter* agents in Test 7

To test the routing heuristic, the automated conveyor system simulation was run with and without optimized part routing. More specifically, two part routing outputs are compared: a) the optimized routing determined using the part routing heuristic, and b) the ordered part routing,  $R = [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8]$ . In each experiment, 100 replications of the simulation were run to account for the stochastic nature of the part arrivals.

The results of the experiments are summarized in Figure 7-13. As would be expected, the optimized routing shows improved performance (wait time decreased about 10-15%) over a strict ordered routing. However, the degree of improvement varies with the number of part types in the default system layout (7 storage bins for max 6 part types in the simulation). With two part types, the ordered routing,  $R = [1\ 2]$ , is a member of the set of possible optimized routings (i.e.,  $P_1 = [1\ 2]$  and  $P_2 = [1\ 6]$ ). Although the ordered routing,  $R = [1\ 2\ 3\ 4\ 5\ 6]$ , is not a member of the set of

possible optimized routings for six part types, it shares the same list of diverters as every member of the set of possible optimized routings. In other words, if  $P_i$  is the  $i^{\text{th}}$  member of the set of possible optimized routings for six part types,  $R$  is equivalent to all members of the set of optimized part routings  $P_i$ : i.e.,  $\forall i, R = P_i$ .

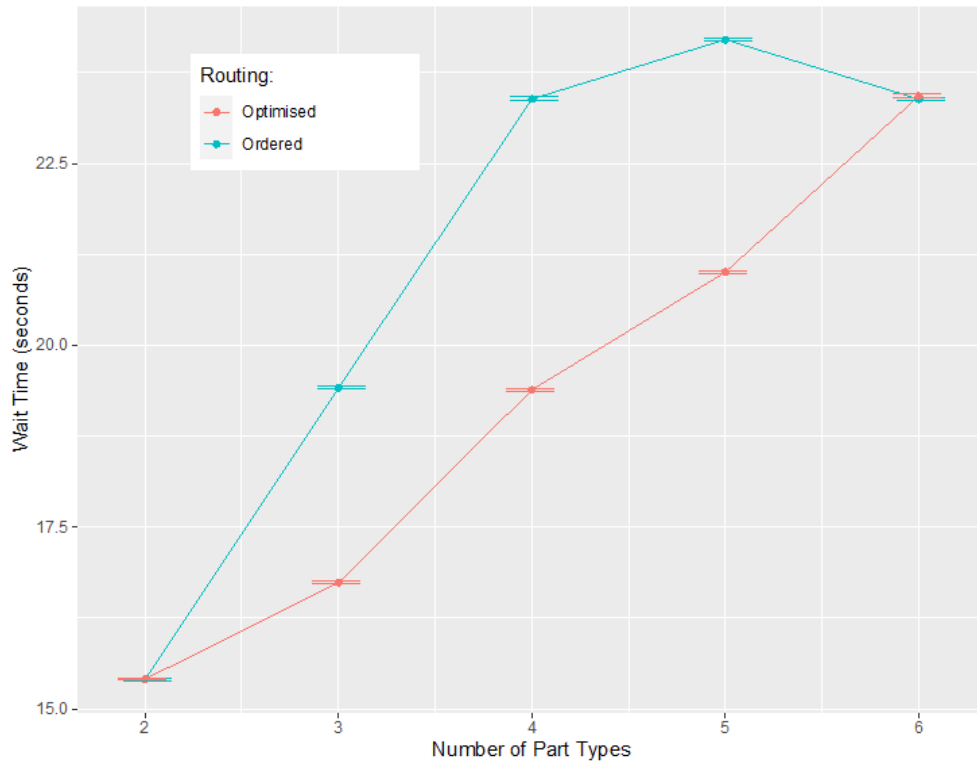


Figure 7-13: Wait time performance for the ordered and optimized routing options (mean and 95% confidence intervals) in Test 7

## 7.4 Experimental Testbed Design

The agent-based simulation model discussed in Section 7.3 demonstrated how the proposed design works and how better it works. In this section, an experimental testbed (Appendix B) is designed to illustrate how the IEC 61499 FB based systems are modelled through the proposed self-manageable architecture (e.g., programming controllers in Figure 7.2). The design tool SPADE is used to develop multi-agent models and the Eclipse 4diac design tool is used to develop IEC 61499 FB models.

Jetson Nano and Raspberry Pi are both small, powerful single-board computers designed to program applications and power devices, in which Jetson Nano is generally considered more powerful than Raspberry Pi in all aspects especially the capability of edge computing [188]-[189]. They are selected for the experiments as they meet the following requirements: a) availability (commercially accessible and relatively cheap) and b) extensibility (multi-using platforms with rich features and functions). The agent modelling tool SPADE (i.e., Smart Python Agent Development Environment) is a multi-agent systems platform written in Python and based on instant messaging [190]. The FB modelling tool Eclipse 4diac is an IEC 61499 engineering environment for developing FB based distributed automation and control applications [79]. They are selected for the experiments as they are open-sourced and powerful enough engineering solutions development environments.

### 7.4.1 Testbed Setup

The proposed testbed setup in Figure 7-14 includes: a) the signaling platform represented by Raspberry Pi (#3 in Figure 7-14) powered LEDs (#2 in Figure 7-14) in blue, green, red, and yellow; b) the sorting platform represented by Raspberry Pi (#3 in Figure 7-14) powered Motors

(#4 in Figure 7-14); and c) the carrying platform represented by Jetson Nano powered JetBot (#1 in Figure 7-14). In this experiment, JetBot powered by Jetson Nano is designed to represent the high-level cyber module and can move intelligently for a series of tasks (e.g., collision avoidance, road following, and object detection). LEDs and Motors powered by Raspberry Pi are designed to represent low-level physical modules and are controlled by agent-embedded IEC 61499 FB modelled applications.

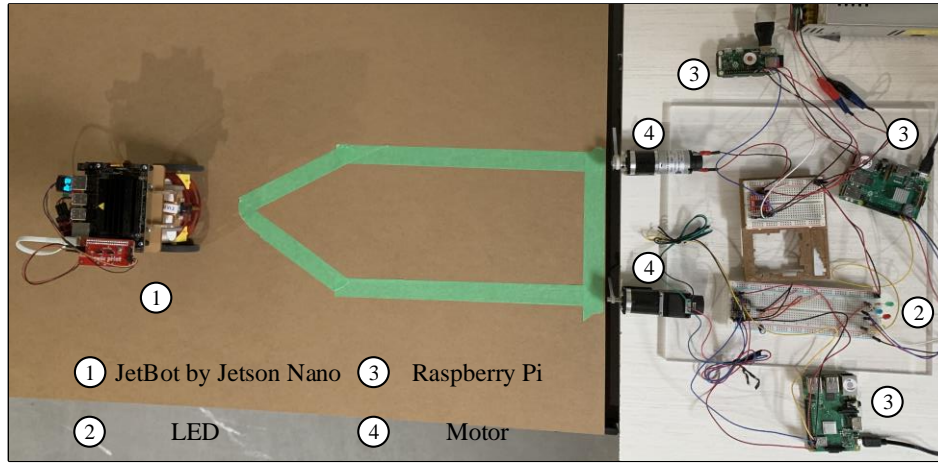


Figure 7-14: Experimental testbed design

The full system configuration under the proposed architecture modelling framework developed in Eclipse 4diac is shown in Figure 7-15.

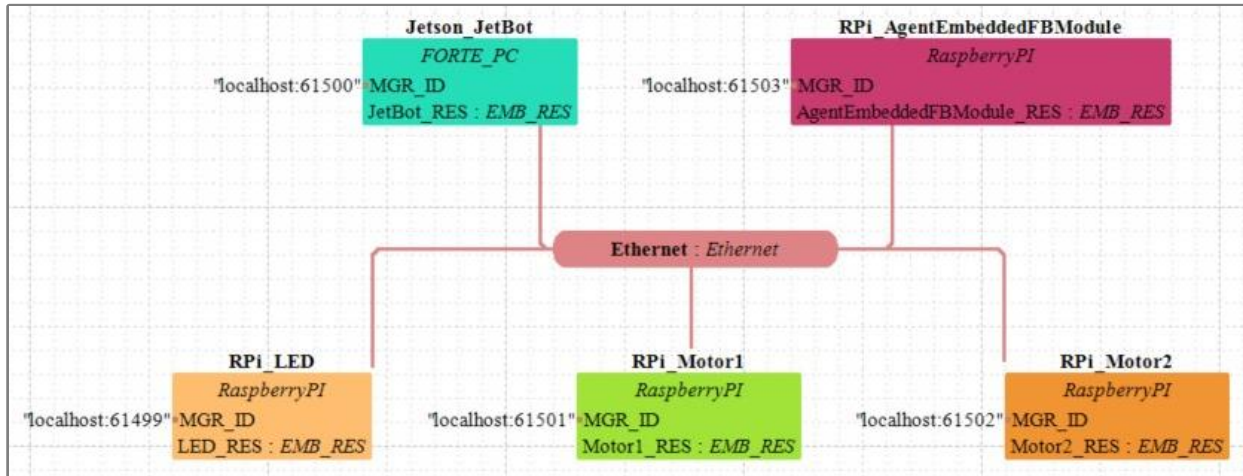


Figure 7-15: The full system configuration in Eclipse 4diac

In Figure 7-15, the FB networks for LEDs (Figure 7-16) and Motors (Figures 7-17 and 7-18) designed in Eclipse 4diac are shown as follows. In each application, FB IX and QX are used to read input and write output signals in the system and publish/subscribe SIFBs are used to communicate with each other through communication services (e.g., Ethernet).

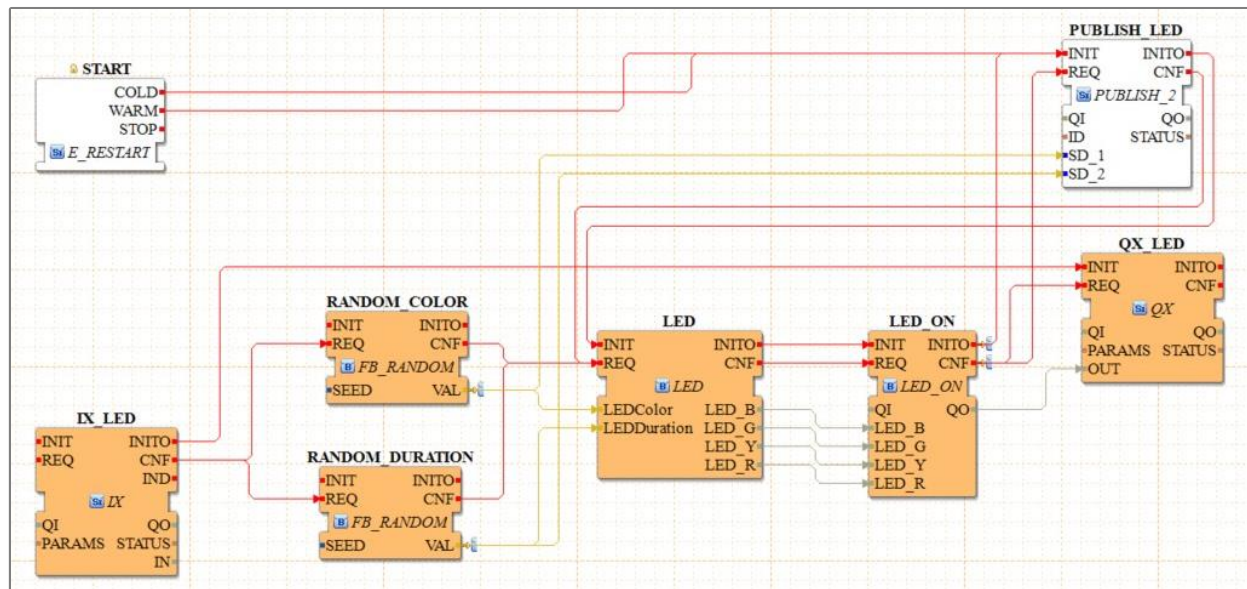


Figure 7-16: FB network design for LEDs in Eclipse 4diac

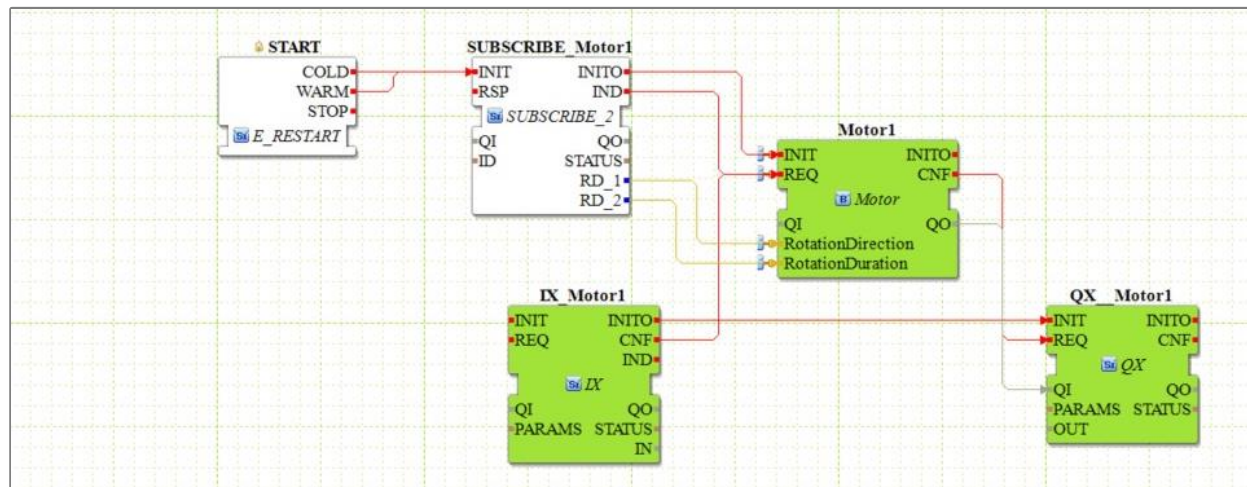


Figure 7-17: FB network design for Motor1 in Eclipse 4diac



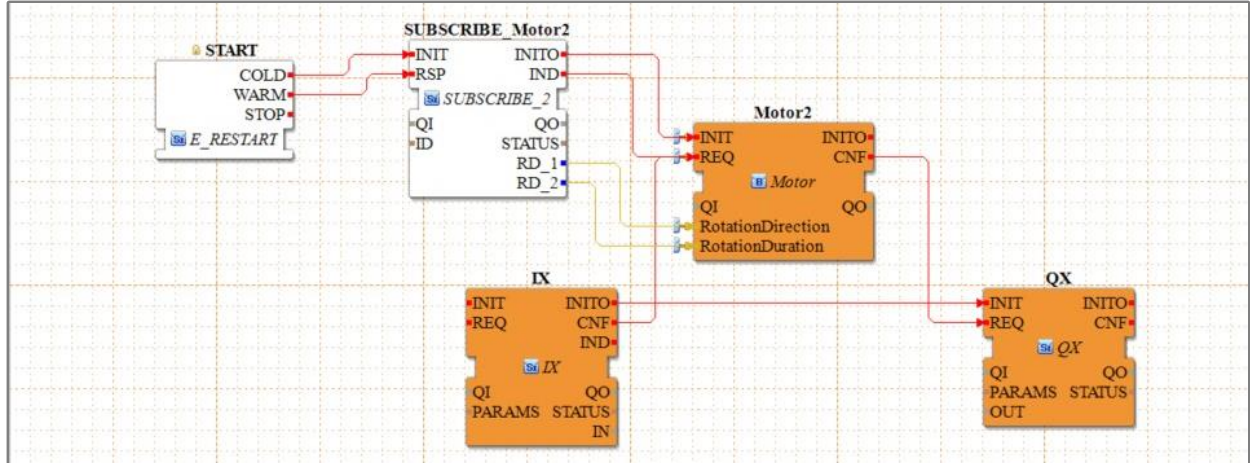


Figure 7-18: FB network design for Motor2 in Eclipse 4diac

JetBot powered by Jetson Nano is programmed in Python using JupyterLab for remote control from a PC. The communication with LEDs and Motors designed in Eclipse 4diac is shown in Figure 7-19. The agent-embedded FB module, as discussed in Section 6.3.5 (Figures 6-18 and 6-19), is designed in the above low-level control application for self-management capabilities and the agent interactions were simulated in the agent-based model in Section 7.3.

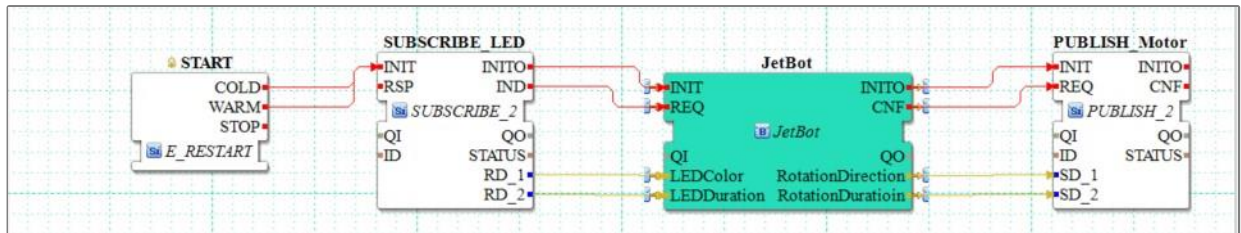


Figure 7-19: Communication network design for JetBot in Eclipse 4diac

#### 7.4.2 Test Scenarios

For the test process, the signaling platform randomly selects a colored LED and turns it on for a few seconds. The color and duration are communicated to the carrying platform. Next, JetBot moves to the sorting platform and sends a message to the Motor, specifying the rotation direction

and duration. The Motor executes the command and sends back a confirmation message to JetBot.

To test the system self-managing capabilities, a simple single direction of rotation scenario is introduced: only a blue LED randomly flashes to request clockwise rotation of Motor1 (Figure 7-20). This is extended to multi-direction rotation: a green LED is added for the counterclockwise rotation of Motor1 (Figure 7-20). Applying the proposed architecture modelling framework, new IEC 61499 FBs can be added to self-configure control applications, and resources can be re-distributed accordingly to support this change.

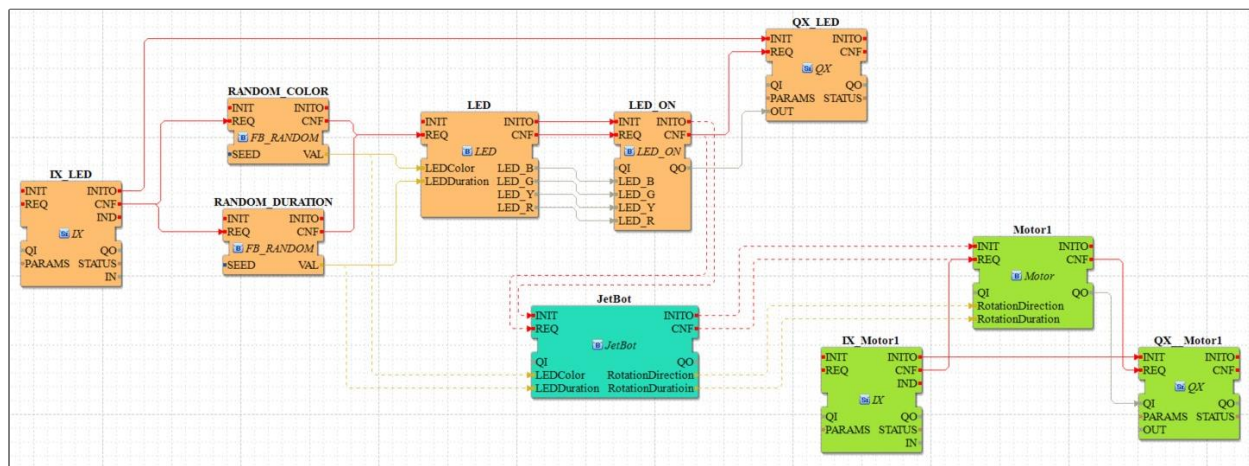


Figure 7-20: Test scenario with two LEDs and one motor in Eclipse 4diac

A more complicated scenario is to add another sorting line (i.e., Motor2) for low-speed rotation with high precision (Figure 7-21). The existing control system can be easily reconfigured and redeployed to the new line with the proposed architecture modelling framework. Self-optimization can also be achieved in both lines due to operation data collected from the old line. One self-healing case is the system can quickly update its IEC 61499 FB modelled application when detecting that the blue LED is broken and replaced with a yellow one. For the self-protection feature, for example, a much heavier bin blocks the line operation, and the system will automatically shut down for protection.



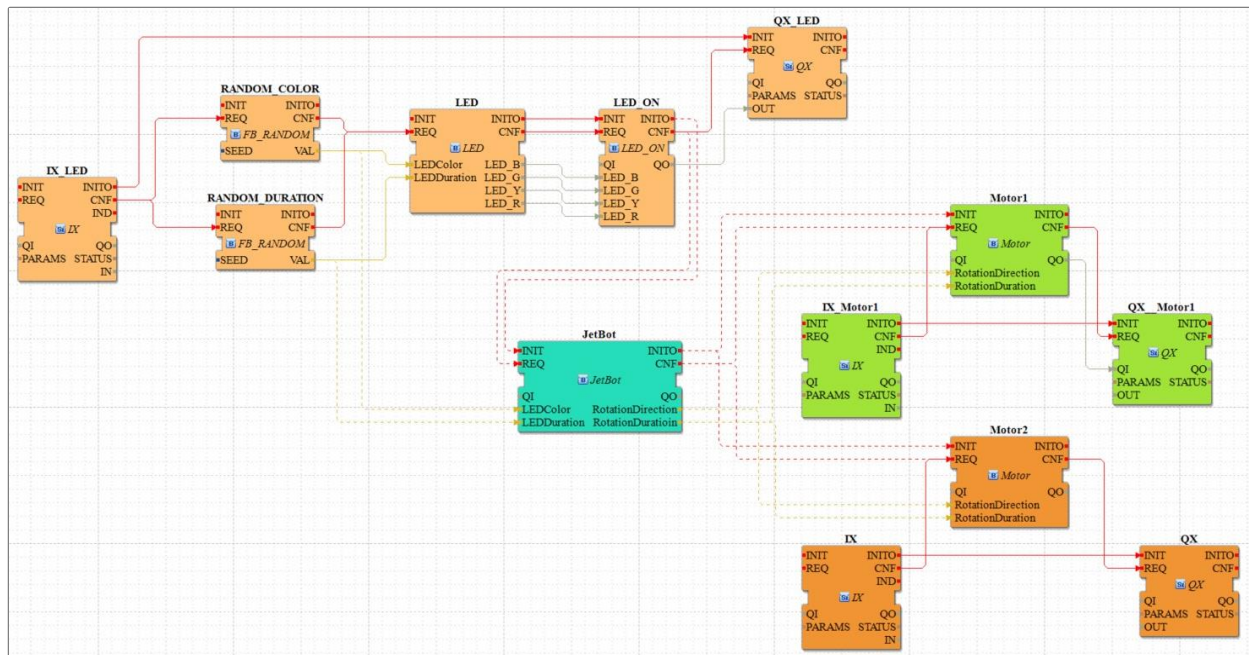


Figure 7-21: Test scenario with four LEDs and two motors in Eclipse 4diac

## 7.5 Performance Evaluation Analysis

In Section 7.3.4, Test 7 has shown that the system designed under the proposed architecture modelling framework has improved performances. In this section, a theoretical analysis is further performed. The time performance of control application programs is a very critical factor in evaluating the automation and control system design, especially the program execution time, scan time, and response time [10]. Traditionally, the program execution time is the time for control code execution and is under control by the watchdog timer. The scan time is the time to perform all the functions internal to the control structure, depending mainly on the program execution time, I/O channels, and the peripherals. The response time is the time that occurs between a variation from the regular system operating states and the corresponding reaction of the control application programs.

The time performance of control applications programs can be affected by several factors, e.g., performance of hardware modules (e.g., devices capability) and software modules (e.g., algorithms complexity). In this research, only the design of control application programs under different architectures is considered for the time performance evaluation, assuming all other aspects the same. Therefore, performance evaluation will be conducted through comparisons of execution time of control applications programmed in IEC 61131-3 function block diagrams (IEC 61131-3 FBD), IEC 61499 function blocks (IEC 61499 FB), and agent-embedded IEC 61499 function blocks (Agent-embedded IEC 61499 FB), under two different circumstances, i.e., regular running conditions and adaptation required conditions.

### 7.5.1 Regular Running Conditions

In regular running conditions, the execution time of automation and control systems (e.g., PLC and DCS) is the sum of working time caused by several steps in which the control application programs are executed. For the time performance evaluation in this research, the following categories of the time performance for the design of automation and control systems under different architectures are considered critical:

- *Device Sensor/Actuator Reaction Time ( $T_{Reaction}$ )*: the reaction time required for corresponding actions from the end devices (i.e., sensors and actuators), and the delay resulted from their responses.
- *Fieldbus Input/Output Refresh Time ( $T_{Refresh}$ )*: the refresh time required for fieldbus inputs/outputs through industrial communication networks.
- *Control Algorithm Execution Time ( $T_{Exection}$ )*: the execution time required for the control algorithms to deliver predefined functions.
- *Algorithm Input/Output Update Time ( $T_{Update}$ )*: the update time required for the inputs read to and the outputs wrote from the control algorithms.
- *Algorithm Execution Result Propagation Time ( $T_{Propagation}$ )*: the propagation time required for the algorithm execution result to be communicated over industrial communication networks.

The execution time of a typical IEC 61131-3 FBD programmed system (e.g., PLC) under regular running conditions ( $T_{RegularFBD}$ ) includes: 1) Device Sensor/Actuator Reaction Time ( $T_{Reaction}$ ), 2) Fieldbus Input/Output Refresh Time ( $T_{Refresh}$ ), 3) Control Algorithm Execution Time ( $T_{Exection}$ ), 4) Algorithm Input/Output Update Time ( $T_{Update}$ ), and 5) Algorithm Execution Result Propagation Time ( $T_{Propagation}$ ).  $T_{RegularFBD}$  is defined as:

$$T_{RegularFBD} = \sum_{i=1}^n (T_{Reaction(i)} + T_{Refresh(i)} + T_{Execution(i)} + T_{Update(i)} + T_{Propagation(i)}) \quad (7.1)$$

The execution time of the equivalent system programmed in IEC 61499 FB (e.g., DCS) under regular running conditions ( $T_{RegularFB}$ ) includes: 1) Device Sensor/Actuator Reaction Time ( $T_{Reaction}$ ), 2) Control Algorithm Execution Time ( $T_{Exection}$ ), and 3) Algorithm Execution Result Propagation Time ( $T_{Propagation}$ ).  $T_{RegularFB}$  is defined as:

$$T_{RegularFB} = \sum_{i=1}^n (T_{Reaction(i)} + T_{Execution(i)} + T_{Propagation(i)}) \quad (7.2)$$

The execution time of the equivalent system programmed in agent-embedded IEC 61499 FB (e.g., DICS) under regular running conditions ( $T_{RegularAgentFB}$ ) is the same as  $T_{RegularFB}$  as no change requests to trigger the execution of high-level and low-level multi-agent modules. More specifically, the self-manageable sub-application designed in low-level IEC 61499 FBs only works as the management events (e.g., change request) trigger the execution (Chapter 6), which will be considered as adaptation required conditions. The MAPLE-K model designed in the high-level cyber module can work during regular running conditions to collect and analyze data (Chapter 5). However, no extra time required as it is working in parallel with the main control applications and if an adaptation required, it falls in the consideration of adaptation required conditions ( $T_{SMSEvaluation}$ ).  $T_{RegularAgentFB}$  is defined as:

$$T_{RegularAgentFB} = \sum_{i=1}^n (T_{Reaction(i)} + T_{Execution(i)} + T_{Propagation(i)}) \quad (7.3)$$

In regular running conditions, the execution time of equivalent automation and control systems designed under different architectures are not the same (Table 7-1). Assuming that  $T_{Reaction}$ ,  $T_{Execution}$ , and  $T_{Propagation}$  are the same,  $T_{RegularFBD}$  (Eq. 7.1) with extra  $T_{Refresh}$  and  $T_{Update}$  is bigger than  $T_{RegularFB}$  (Eq. 7.2) and  $T_{RegularAgentFB}$  (Eq.

7.3). The main time performance gain in two IEC 61499 cases results from the elimination of traditional scan cycles in IEC 61131-3, especially in this case, no extra time-consuming execution for  $T_{Refresh}$  and  $T_{Update}$ . As previously discussed, IEC 61499 applies the event-driven execution mechanism, which means the execution of the FB based control application programs is assumed to start immediately whenever input events arrive and to end immediately whenever output events emit. However, in IEC 61131-3, continuous I/O refresh ( $T_{Refresh}$ ) over the industrial communication network and the delay to update inputs/outputs in control application programs require more time and resource. Another reason is IEC 61131-3 adopts global memory to exchange data in program organization units (i.e., POU includes programs, functions, and function blocks in IEC 61131-3), while IEC 61499 encapsulates global data directly and locally into function blocks. It is convenient to list variables in global memory in the development phase but is time-consuming during runtime and not easy for module reuse.

Table 7-1: Execution time comparison in regular running conditions

Design Architectures	IEC 61131-3 FBD	IEC 61499 FB	Agent-embedded IEC 61499 FB
Evaluation Metrics	$T_{RegularFBD}$	$T_{RegularFB}$	$T_{RegularAgentFB}$
$T_{Reaction}$	1	1	1
$T_{Refresh}$	1	0	0
$T_{Exection}$	1	1	1
$T_{Update}$	1	0	0
$T_{Propagation}$	1	1	1

### 7.5.2 Adaptation Required Conditions

Adaptation required conditions in this research can be understood as the conditions in which the system is required to adjust itself and respond to frequent changes and evolving requirements. The adaptation tasks occur in different formats (typically system reconfiguration, e.g., failure recovery) and are caused by various factors (e.g., the high-level request to update function

modules for system upgrade, low-level request to adjust parameter values due to state variations).

As reviewed before, the automation and control system adaptation is commonly classified into the dimensions of simple, dynamic, and intelligent reconfiguration [23]. In general, simple reconfiguration (at least IEC 61131-3 required) aims at avoiding software-coupling issues during reconfiguration; the dynamic (at least IEC 61499 required) focuses on reconfiguration during runtime to satisfy the timing criteria; and the intelligent exploits distributed artificial intelligence to reconfigure automatically [23]. *Reconfiguration* can be understood as the automation and control system adapt itself (software/hardware) to respond to changes on the fly or to satisfy new requirements by adopting different techniques (i.e., simple, dynamic, and intelligent). Under the proposed multi-agent modelling framework towards IEC 61499 FB based distributed intelligent automation, *self-management* is used in this research to describe such system adaptation, including self-configuration, self-optimization, self-healing, and self-protection.

Under adaptation required conditions, the system is required to perform adaptation to respond to frequent changes and evolving requirements in dynamic environments. The following categories of the time performance for the design of automation and control systems under different architectures are considered critical:

- *Evaluation of Frequent Changes and Evolving Requirements ( $T_{Evaluation}$ )*: the time required for the evaluation of frequent changes and evolving requirements and thus the right decision is made for system adaptation. It is expected that this type of evaluation occurs in the higher management level with human interaction as these changes and requirements should be complex enough that the system cannot handle itself. For example, decision of

system overall function upgrade for advanced capabilities, in this case the system itself may not detect any change request from monitoring its operations.

- *Execution of System Reconfiguration or Self-Management Plans ( $T_{Reconfiguration}$  or  $T_{Selfmanagement}$ )*: the time required for the execution of system reconfiguration or self-management plans and thus the system adaptation is performed.
- *Validation of System Adaptation ( $T_{Validation}$ )*: the time required for the validation of that the system is adapted in time with desired and expected behaviours. This is the final step to ensure that the adaptation is validated.

There are three more time performance evaluation metrics considered for the proposed two-layer design architecture (Figure 4-1b). They are explained as follows:

- *Self-Manageable Service Evaluation Time ( $T_{SMSEvaluation}$ )*: the time required for MAPLE-K agents of the high-level cyber module to evaluate changes and requirements and thus to deliver system adaptation actions (Figure 5-1). It is expected that this type of evaluation occurs in the high-level cyber module which is directly related to automation and control systems and focuses on system software modules. That means this type of evaluation can be done by the multi-agent module automatically through monitoring and analyzing system states without human interactions. For example, bug fix in the system control applications, in this case the system detect and fix the bug to maintain its stable operations.
- *Self-Manageable Service Request Time ( $T_{SMSRequest}$ )*: the time required for the self-manageable service execution agent of the low-level physical module to achieve its predefined functions and satisfy the request (Figure 6-13). The request time includes implementation of two interface functions, i.e., requesting self-manageable agents for self-

manageable services in the low-level physical module and executing self-manageable services to adapt IEC 61499 FB based systems (Figure 6-14).

- *Self-Manageable Service Response Time ( $T_{SMSResponse}$ )*: the time required for the self-manageable agents of the low-level physical module to deliver their predefined self-manageable services to respond to the request from the self-manageable service execution agent (Figure 6-13). The response time includes one or more self-manageable agents to execute their own algorithms to respond to the request from self-manageable service execution agent (Figure 6-15).

The adaptation time of a typical IEC 61131-3 FBD programmed system under adaptation required conditions ( $T_{AdaptationFBD}$ ) is defined as:

$$T_{AdaptationFBD} = T_{Evaluation(FBD)} + T_{Validation(FBD)} + T_{Reconfiguration(FBD)} \quad (7.4)$$

The adaptation time of the equivalent system programmed in IEC 61499 FB under adaptation required conditions ( $T_{AdaptationFB}$ ) is defined as:

$$T_{AdaptationFB} = T_{Evaluation(FB)} + T_{Validation(FB)} + T_{Reconfiguration(FB)} \quad (7.5)$$

The adaptation time of the equivalent system programmed in agent-embedded IEC 61499 FB under adaptation required conditions ( $T_{AdaptationAgentFB}$ ) is defined as:

$$T_{AdaptationAgentFB} = T_{Evaluation(AgentFB)} + T_{Validation(AgentFB)} + T_{Selfmanagement(AgentFB)} + T_{SMSEvaluation(AgentFB)} + T_{SMSRequest(AgentFB)} + T_{SMSResponse(AgentFB)} \quad (7.6)$$

In adaptation required conditions (consider one adaptation in Eqs. 7.4, 7.5, and 7.6), the adaptation time of equivalent automation and control systems designed under different architectures are not the same (Table 7-2). Assume that in this research  $T_{Evaluation}$  and  $T_{Validation}$  are the same for the three system design architectures. For  $T_{Evaluation}$ , as described before, it usually occurs in a large time scale (e.g., day/week) and the capabilities of



management platforms (e.g., ERP/PLM) are expected to be the same. For  $T_{Validation}$ , generally speaking, it can be within a large time scale for complex tests (e.g., the initial system validation test) or be in real-time or at least in parallel during system runtime. These two metrics are not the focus of this research.

As summarized in Table 7-2, the system designed under the IEC 61131-3 architecture (more specifically programmed in FBD) are manually reconfigured using simple reconfiguration techniques offline while the system designed under the IEC 61499 architecture (programmed in FB) can apply dynamic reconfiguration techniques to reconfigure online semi-automatically. It should be expected to have some performance gain from IEC 61499 FB programmed systems compared to the IEC 61131-3 FBD programmed ones (e.g., the time required for adaptation  $T_{AdaptationFBD}$  in Eq. 7.4 greater than  $T_{AdaptationFB}$  in Eq. 7.5), as the IEC 61499 design helps reduce the impact and increase the predictability of system reconfiguration and maintain system consistency and stability [23].

Table 7-2: Execution time comparison in adaptation required conditions

<b>Design Architectures</b>	<b>IEC 61131-3 FBD</b>	<b>IEC 61499 FB</b>	<b>Agent-embedded IEC 61499 FB</b>
<b>Evaluation Metrics</b>	$T_{AdaptationFBD}$	$T_{AdaptationFB}$	$T_{AdaptationAgentFB}$
$T_{Evaluation}$	1	1	1
$T_{Validation}$	1	1	1
$T_{Reconfiguration}$	simple; offline; manually.	dynamic; online; semi-automatically.	n/a
$T_{Selfmanagement}$	n/a	n/a	intelligent; online; automatically.
$T_{SMSEvaluation}$	n/a	n/a	intelligent; real-time; automatically.
$T_{SMSRequest}$	n/a	n/a	intelligent; real-time; automatically.
$T_{SMSResponse}$	n/a	n/a	intelligent; real-time; automatically.

For the proposed IEC 61499 architecture (programmed in agent-embedded FB), it aims at intelligent, online, and automatic system self-management (or traditionally saying reconfiguration) by adopting a two-layer architecture with multi-agent cyber module modelling to achieve run-time intelligence and IEC 61499 function block physical module modelling to realize real-time adaption. Although, more items ( $T_{SMSEvaluation}$ ,  $T_{SMSRequest}$ , and  $T_{SMSResponse}$ ) in Eq. (7.6), they all occurs in software algorithm execution level and are expected to be within a small time scale. As described before for  $T_{SMSEvaluation}$  in Eq. 7.6, this type of evaluation occurs in the high-level cyber module and is done by multi-agent MAPLE-K module automatically through monitoring and analyzing system states regularly without human interactions. This proposed multi-agent cyber module applies machine learning techniques for system operation state analysis and helps reduce work in the higher management evaluation ( $T_{Evaluation}$ ). For the request and response in the low-level physical module ( $T_{SMSRequest}$  and  $T_{SMSResponse}$ ) performed by the agent-embedded IEC 61499 FB-based sub-application, it is designed to be able to handle simple and straightforward tasks with correct alternative solutions during runtime. That means this type of execution by the agent-embedded IEC 61499 FB-based sub-applications in the low-level physical module can be done automatically for real-time adaptation through monitoring system states without exchanging large amount of data with the high-level cyber module for analytics and learning (reducing real-time communication burden). For example, parameter change of the control application, in this case the system adjust the parameter to maintain its stable operations.

As analyzed before for the proposed IEC 61499 architecture (programmed in agent-embedded FB), the performance improvement should come not only from the reduced adaptation time (i.e.,  $T_{AdaptationAgentFB}$  in Eq. 7.6 less than  $T_{AdaptationFB}$  in Eq. 7.5 or

$T_{AdaptationFBD}$  in Eq. 7.4) but also from the increased solution quality (e.g., data analytics and machine learning from agent modules). As in this type of architecture design, the multi-agent cyber module improves a lot in evaluation of changes and new requirements, and the agent-embedded IEC 61499 FB physical module improves a lot in quick response to changes and new requirements.

### 7.5.3 Performance Evaluation Estimation

In previous sections, the execution time in regular running conditions (Eqs. 7.1, 7.2, and 7.3) and the adaptation time in adaptation required conditions (Eqs. 7.4, 7.5, and 7.6) are analyzed for the automation and control systems designed under three different architectures (IEC 61131-3 FBD, IEC 61499 FB, and Agent-embedded IEC 61499 FB). In this section, the total time performance is formulated as Eqs. 7.7, 7.8, and 7.9 by considering only key aspects and focusing on control application programs. The item  $[rand() \% 2]$  in these equations represents that not all nodes (a general and abstract representation of subsystems, devices, or applications) in the system need to be adapted.

$$T_{TotalFBD} = \sum_{i=1}^n (T_{Reaction(i)} + T_{Refresh(i)} + T_{Execution(i)} + T_{Update(i)} + T_{Propagation(i)}) + \sum_{j=1}^m (T_{Reconfiguration(j)}) \times [rand_j() \% 2] \quad (7.7)$$

$$T_{TotalFB} = \sum_{i=1}^n (T_{Reaction(i)} + T_{Execution(i)} + T_{Propagation(i)}) + \sum_{j=1}^m (T_{Reconfiguration(j)}) \times [rand_j() \% 2] \quad (7.8)$$

$$\begin{aligned}
T_{TotalAgentFB} = & \sum_{i=1}^n (T_{Reaction(i)} + T_{Execution(i)} + T_{Propagation(i)}) \\
& + \sum_{j=1}^m (T_{Selfmanagement(j)} + T_{SMSEvaluation(j)}) \times [rand_j() \% 2] \\
& + \sum_{k=1}^l (T_{SMSRequest(k)} + T_{SMSResponse(k)}) \times [rand_k() \% 2]
\end{aligned} \tag{7.9}$$

The total time performance of the system programmed in IEC 61131-3 FBD is defined in Eq. 7.7 as  $T_{TotalFBD}$  and the system programmed in IEC 61499 FB is defined in Eq. 7.8 as  $T_{TotalFB}$ . For the proposed design architecture in this research, the total performance  $T_{TotalAgentFB}$  is defined in Eq. 7.9, in which as described before the two-layer self-manageable system architecture design ( $T_{Selfmanagement}$  and  $T_{SMSEvaluation}$  from the cyber module and  $T_{SMSRequest}$  and  $T_{SMSResponse}$  from the physical module) is proposed to replace the traditional reconfiguration architecture design ( $T_{Reconfiguraiton}$  in Eq. 7.7 and 7.8). The estimated performance evaluation are shown in Figure 7-22. With increased system nodes (this matches the fact that the industrial automation and control systems are becoming more and more complex in Industry 4.0), the time performance gain of the proposed system architecture design tends to become larger and larger. Figure 7-22 presented here is for graphical display of previous discussions to show the performance gain trend and the real case is not supposed to be exactly the same.

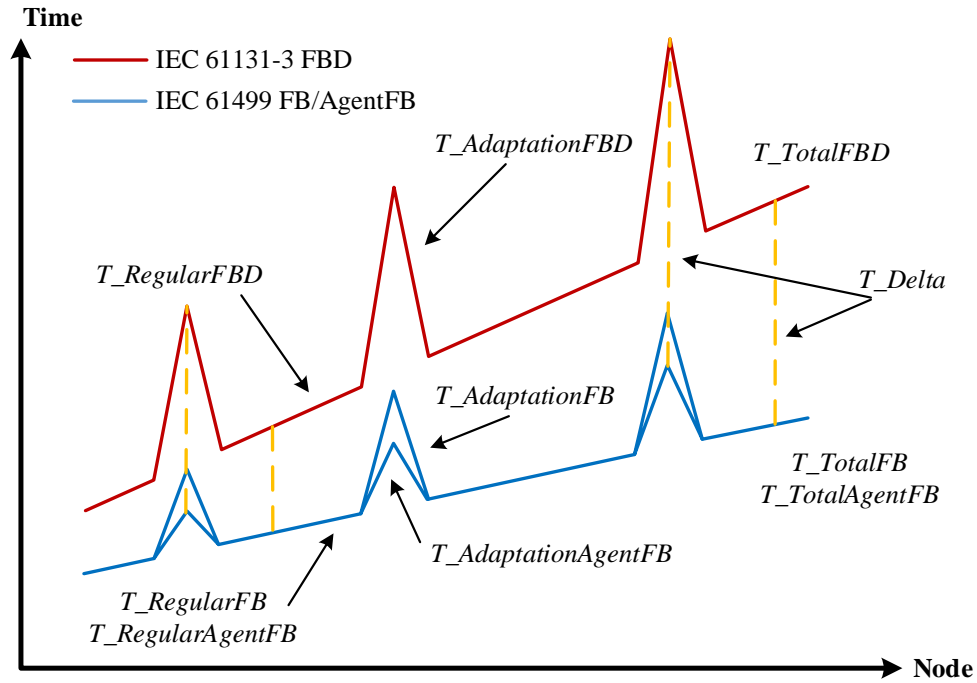


Figure 7-22: Estimated performance evaluation under different system design architectures

In this research, only the performance evaluated by system execution time and adaptation time during runtime is considered. One thing should be noted that the time and effort required in the system design phase programming in IEC 61499 FB compared to IEC 61131-3 (e.g., FBD) are considered intensive, especially for the proposed two-layer multi-agent and IEC 61499 FB hybrid design architecture. Besides challenges identified in the literature review, one major issue is that it is not easy to incorporate artificial intelligence frameworks and techniques into systems design and to apply advanced data analytics to enable learning capabilities and intelligent behaviours of next-generation automation and control systems.

## 7.6 Summary

In this chapter, demonstration of the typical industrial scenario, development of the multi-agent simulation model, design of the experimental testbed, and the performance evaluation analysis were discussed to show that the proposed architecture modelling framework is feasible and effective. The proposed design was tested through various experiments on the multi-agent simulation model based on the agent modelling tool NetLogo and the experimental testbed on the Jetson Nano and Raspberry Pi platforms. The proposed design was further evaluated theoretically through performance analysis of regular execution time and adaptation time in two typical conditions (i.e., regular running conditions and adaptation required conditions) for systems designed under three comparable architectures (i.e., IEC 61131-3 FBD, IEC 61499 FB, and Agent-embedded IEC 61499 FB). Through the above demonstration, simulation, experiment, and evaluation, it demonstrates the ability of the proposed architecture to respond to major challenges in Industry 4.0.

*[This page intentionally left blank]*

## Chapter Eight: Conclusions and Future Work

### 8.1 Conclusions

The research presented in this thesis explored the design of a two-layer self-manageable architecture to enable real-time adaptation at the device level and run-time intelligence throughout the whole system. To achieve this, multi-agent modelling techniques were applied by using autonomous and cooperative agents to achieve run-time intelligence in system design and module reconfiguration, and IEC 61499 function block modelling techniques were applied by using object-oriented and event-driven function blocks to realize real-time adaption of automation logic and control algorithms. The main reason behind that is the autonomous, cooperative, and distributed multi-agent modelling approach matches the object-oriented, event-driven, and application-based IEC 61499 function block architecture design, which holds the most promise of achieving industrial cyber-physical systems to be self-manageable in Industry 4.0. The major research work and contributions are summarized as follows:

#### 1) *High-Level Cyber Module Design*

One major contribution is the multi-agent MAPLE-K computing model for the high-level iCPS architecture design with the introduction of a self-learning agent for high-level cyber module learning capabilities. The design applied multi-agent modelling techniques to implement the autonomic computing reference architecture (i.e., traditional MAPE-K model) and empowered the architecture with self-learning capabilities (i.e., proposed MAPLE-K model). The traditional MAPE-K loop design is based on fixed, predefined rules, policies, goals, etc., which is knowledge-intensive in the early design by considering limited possible adaptation scenarios and is inflexible to be reconfigured during the system runtime. The proposed design leveraged the



industrial computing tools and models by introducing the self-learning agent to empower the system learning capabilities for adaptation at runtime.

## 2) *Low-Level Physical Module Design*

The other major contribution is the self-manageable IEC 61499 function block model for the low-level iCPS architecture design with the introduction of a new agent-embedded function block design pattern for low-level physical module self-managing capabilities. The design deployed autonomic computing self-managing properties into IEC 61499 function block modelling framework by forming a meta-application (i.e., proposed agent-embedded function block application) to self-manage control applications. Traditionally, system configuration is achieved simply (i.e., avoiding software-coupling issues) or dynamically (i.e., satisfying timing criteria) through device management SIFBs to access standardized management functions (e.g., *START* and *KILL* in application execution, *CREATE* and *DELETE* for function block instances). The proposed design incorporated the agent-embedded design pattern into traditional function block application design with the separation of typical control application execution and intelligent self-manageable service execution.

## 3) *Architecture Modelling Evaluation*

The proposed architecture modelling framework was demonstrated through various experiments on the multi-agent simulation model developed in the agent modelling environment NetLogo and the experimental testbed designed on the Jetson Nano and Raspberry Pi platforms. The multi-agent simulation model focused on low-level system self-managing capabilities. The simulation results showed that with the proposed design, the system is able to self-manage adaptation autonomously to respond to typical changes and requirements (e.g., new tasks, system failures, and operation optimization) with better performances. The experimental testbed design focused

on high-level system computing capabilities with self-learning features. The experiment showed that the proposed architecture modelling framework is feasible (i.e., from context monitoring, data analysis and machine learning, to action planning and execution) with available embedded devices (e.g., Jetson Nano). The performance evaluation of regular execution time and adaptation time in two typical conditions for systems designed under three different architectures were further theoretically analyzed.

To note that, it is possible in implementation that several physical modules (i.e., agent-embedded IEC 61499 FB model) can be attached to one cyber module (i.e., multi-agent MAPLE-K model), and several cyber modules are managed by the higher-level management platforms (e.g., ERP). It depends on system architectures and is similar to that each sub-system root node that cyber module works on is an aggregation of several distributed nodes that built as physical modules. Humans are not expected to interact too much with these two levels during system operation, but with higher-level management platforms. The typical data flow in the system is that the low-level physical module interacts directly with operating states for real-time operation and adaptation (higher real-time requirements) while the high-level cyber module manages several physical modules' operations for system run-time intelligence (high real-time requirements) and the management platform is responsible for strategic decision-making (low real-time requirements). All these matches the objective of designing self-manageable iCPS to enable real-time adaptation at the device level and run-time intelligence throughout the whole system with computing intelligence distributed over different system levels to satisfy different timing requirements.

## 8.2 Future Work

The research work in this thesis is an attempt to investigate the design of self-manageable system architecture modelling for IEC 61499 based distributed intelligent automation by employing multi-agent modelling and function block modelling techniques. The research result is a two-layer architecture model that is characterized of the high-level multi-agent modelled cyber module design and the low-level agent-embedded IEC 61499 FB modelled physical module design. The future work around this research is expected to continue from the following aspects:

### 1) *Interaction Interface Design and Implementation*

Interaction interface design and implementation is a key aspect in the system architecture design as communicating and computing flows from homogeneous and heterogeneous system modules (e.g., agent-agent, FB-FB, agent-FB) are connected together through interaction interfaces. Future work is required for the middle-level interface module sandwiched between the high-level cyber module and the low-level physical module. The work on hybrid interaction interfaces (i.e., agent-FB) needs special attention. Only by understanding the characteristics of this type of interface (e.g., interface design and implementation patterns) can the multi-layer system design architecture modelled by multi-agent systems and IEC 61499 FBs be achieved.

### 2) *Implementation of Autonomic Computing Framework*

Deploying autonomic computing reference architecture and self-managing properties into the modelling framework is crucial to achieve distributed intelligent iCPS. However, proper implementation of multi-agent modelling and IEC 61499 FB modelling (i.e., balancing powerful but not reliable agents and time-critical, predictable, and stable FB based controls) is not an easy task. Questions that need to be answered are: e.g., how to ensure agents in the high-level cyber module are programmed with enough self-learning capabilities and are designed in modules for

reuse; how to ensure agents embedded in IEC 61499 FBs are programmed with generic, lightweight, and robust algorithms as the low-level physical module has to guarantee system responsiveness, correctness, safety, reliability, etc.

### 3) *Development of Integrated Engineering Environments*

An integrated engineering environment is required for design and development, verification and validation, evaluation and implementation of hybrid multi-agent and IEC 61499 FB modelled systems. Such integrated engineering environment can be based on one major design modelling tool (e.g., Eclipse 4diac) with added function modules (e.g., design simulation and evaluation), open interfaces to the external (e.g., agent modelling software), and rich function block libraries (e.g., to include the proposed computing modules and self-manageable sub-applications as templates). Future work on enriching integrated engineering environments serves the key role in industrializing the IEC 61499 standard and promoting its related research.

*[This page intentionally left blank]*

## References

- [1] H. Kagermann, W. Wahlster, and J. Helbig, “Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Securing the future of German manufacturing industry,” Final Report of INDUSTRIE 4.0 Working Group, Frankfurt, Germany, Apr. 2013.
- [2] NSF, “Cyber-Physical Systems.” Accessed: Dec. 05, 2022. [Online]. Available: [https://www.nsf.gov/funding/pgm\\_summ.jsp?pims\\_id=503286](https://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503286).
- [3] R. Lewis, *Programming industrial control systems using IEC 1131-3*, 2nd ed., London, UK: The Institution of Engineering and Technology, 1998.
- [4] IEC 61131-3, *Programmable controllers - Part 3: Programing Languages*, International Standard, 3rd ed., Geneva, Switzerland: International Electrotechnical Commission, 2013.
- [5] R. Lewis, *Modelling control systems using IEC 61499: Applying function blocks to distributed systems*, 1st ed., London, UK: The Institution of Engineering and Technology, 2001.
- [6] IEC 61499-1, *Function blocks - Part 1: Architecture*, International Standard, 2nd ed., Geneva, Switzerland: International Electrotechnical Commission, 2012.
- [7] V. Vyatkin, “Software engineering in industrial automation: State-of-the-art review,” *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1234-1249, Aug. 2013.
- [8] R. N. Nagel and R. Dove, *21st century manufacturing enterprise strategy: An industry-led view*, Bethlehem, PA, USA: Iacocca Institute, 1991.
- [9] J. H. Christensen, “Holonc manufacturing systems: Initial architecture and standards directions,” In *Proceedings of 1st European Conference on Holonic Manufacturing Systems*, Hannover, Germany, Dec. 1994.
- [10] F. Bonfatti, P. Monari, and U. Sampieri, *IEC 61131-3 programming methodology: Software engineering methods for industrial automated systems*, ICS Triplex ISaGRAF Inc., 2003.
- [11] G. Lyu, A. Fazlirad, and R. W. Brennan, “Multi-agent modelling of cyber-physical systems for IEC 61499 based distributed automation,” In *Proceedings of 30th International Conference on Flexible Automation and Intelligent Manufacturing*, Athens, Greece, June 2021, *Procedia Manufacturing*, vol. 51, pp. 1200-1206.
- [12] G. Lyu and R. W. Brennan, “Multi-agent based IEC 61499 function block modelling for distributed intelligent automation,” In *Proceedings of 31st International Conference on Flexible Automation and Intelligent Manufacturing*, Detroit, USA, June 2022, *Lecture Notes in Mechanical Engineering*, vol. 2, pp. 395-407.
- [13] G. Lyu and R. W. Brennan, “Multi-agent modelling of cyber-physical systems for IEC 61499 based distributed intelligent automation,” *International Journal of Computer Integrated Manufacturing*. (Under Review).
- [14] G. Lyu and R. W. Brennan, “Evaluating a self-manageable architecture for industrial automation systems,” In *Proceedings of 32nd International Conference on Flexible Automation and Intelligent Manufacturing*, Porto, Portugal, June 2023. (In Publication).
- [15] G. Lyu and R. W. Brennan, “Evaluating a self-manageable architecture for industrial automation systems,” *Robotics and Computer-Integrated Manufacturing*. (In Publication).
- [16] G. Lyu and R. W. Brennan, “Towards IEC 61499 based distributed intelligent automation: design and computing perspectives,” In *Proceedings of 17th IEEE International Conference on Industrial Informatics*, Helsinki-Espoo, Finland, July 2019, pp. 160-163.
- [17] G. Lyu and R. W. Brennan, “Towards IEC 61499 based distributed intelligent automation: a literature review,” *IEEE Transactions on Industrial Informatics*, vol. 17, no. 4, pp. 2295-2306, Aug. 2020.

- [18] R. W. Brennan and G. Lyu, "IEC 61499 and the promise of holonic systems," In *Proceedings of 9th International Conference on Industrial Applications of Holonic and Multi-Agent Systems*, Linz, Austria, Aug. 2019, pp. 3-12.
- [19] F. Georg and T. Hussain, "Modeling techniques for distributed control systems based on the IEC 61499 standard - current approaches and open problems," In *Proceedings of IEEE 8th International Workshop on Discrete Event Systems*, Ann Arbor, MI, USA, July. 2006, pp. 176-181.
- [20] A. Zoitl, T. Strasser, K. Hall, R. Staron, C. Sünder, and B. Favre-Bulle, "The past, present, and future of IEC 61499," In *Holonic and Multi-Agent Systems for Manufacturing*, V. Mařík, V. Vyatkin, and A.W. Colombo, Eds., Berlin Heidelberg, German: Springer, 2007, pp. 15-28.
- [21] K. Thramboulidis, "IEC 61499 in factory automation," In *Advances in Computer, Information, and Systems Sciences, and Engineering*, K. Elleithy, T. Sobh, A. Mahmood, M. Iskander, and M. Karim, Eds., Dordrecht, Netherlands: Springer, 2007, pp. 115-124.
- [22] K. Hall, R.J. Staron, and A. Zoitl, "Challenges to industry adoption of the IEC 61499 standard on event-based function blocks," In *Proceedings of 5th IEEE International Conference on Industrial Informatics*, Vienna, Austria, July 2007, pp. 823-828.
- [23] R. W. Brennan, P. Vrba, P. Tichy, A. Zoitl, C. Sünder, T. Strasser, and V. Mařík, "Developments in dynamic and intelligent reconfiguration of industrial automation," *Computers in Industry*, vol. 59, no. 6, pp. 533-547, Aug. 2008.
- [24] A. Zoitl and V. Vyatkin, "IEC 61499 architecture for distributed automation: The 'glass half full' view," *IEEE Industrial Electronics Magazine*, vol.3, no. 4, pp. 7-23, Dec. 2009.
- [25] A. Zoitl, T. Strasser, C. Sünder, and T. Baier, "Is IEC 61499 in harmony with IEC 61131-3?" *IEEE Industrial Electronics Magazine*, vol.3, no. 4, pp. 49-55, Dec. 2009.
- [26] H. M. Hanisch, M. Hirsch, D. Missal, S. Preuß, and C. Gerber, "One decade of IEC 61499 modeling and verification - results and open issues," *IFAC Proceedings Volumes*, vol. 42, no. 4, pp. 211-216, Jan. 2009.
- [27] V. Vyatkin, "The IEC 61499 standard and its semantics," *IEEE Industrial Electronics Magazine*, vol.3, no. 4, pp. 40-48, Dec. 2009.
- [28] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768-781, Nov. 2011.
- [29] T. Strasser, A. Zoitl, J. H. Christensen, and C. Sünder, "Design and execution issues in IEC 61499 distributed automation and control systems," *IEEE Transactions on Systems, Man, and Cybernetics - Part C: Applications and Reviews*, vol. 41, no. 1, pp. 41-51, Jan. 2011.
- [30] T. Strasser, J. H. Christensen, A. Valente, J. Chouinard, E. Chapanzano, A. Valentini, H. Mayer, V. Vyatkin, and A. Zoitl, "The IEC 61499 function block standard: Launch and takeoff," *ISA Automation Week*, Orlando, FL, USA, Sept. 2012.
- [31] J. H. Christensen, T. Strasser, A. Valentini, V. Vyatkin, and A. Zoitl, "The IEC 61499 function block standard: Overview of the second edition," *ISA Automation Week*, Orlando, FL, USA, Sept. 2012a.
- [32] J. H. Christensen, T. Strasser, A. Valentini, V. Vyatkin, and A. Zoitl, "The IEC 61499 function block standard: Software tools and runtime platforms," *ISA Automation Week*, Orlando, FL, USA, Sept. 2012b.
- [33] K. Thramboulidis, "Service-oriented architecture in industrial automation systems - the case of IEC 61499: A review," *arXiv preprint*, arXiv:1506.04615, June 2015.
- [34] R. Sinha, S. Patil, L. Gomes, and V. Vyatkin, "A survey of static formal methods for building dependable industrial automation systems," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 7, pp. 3772-3783, July 2019.
- [35] L. Prenzel, A. Zoitl, and J. Provost, "IEC 61499 runtime environments: A state of the art comparison," In *Proceedings of 17th International Conference on Computer Aided Systems Theory*, Las Palmas de Gran Canaria, Spain, Feb. 2019, Lecture Notes in Computer Science, vol. 12014, pp. 453-460, Apr. 2020.

- [36] L. Sonnleithner, M. Oberlehner, E. Kutsia, A. Zoitl, and S. Bácsi, "Do you smell it too? Towards bad smells in IEC 61499 applications," In *Proceedings of 26th IEEE International Conference on Emerging Technologies and Factory Automation*, Vasteras, Sweden, Sept. 2021, pp. 1-4.
- [37] IEC 61499-2, *Function blocks - Part 2: Software tools requirements*, International Standard, 2nd ed., Geneva, Switzerland: International Electrotechnical Commission, 2012.
- [38] IEC 61499-4, *Function blocks - Part 4: Rules for compliance profiles*, International Standard, 2nd ed., Geneva, Switzerland: International Electrotechnical Commission, 2013.
- [39] IEC 61499-3, *Function blocks - Part 3: Tutorial information*, Technical Report, 1st ed., Geneva, Switzerland: International Electrotechnical Commission, 2004.
- [40] V. Vyatkin, *IEC 61499 function blocks for embedded and distributed control systems design*, 2nd ed., Durham, NC, USA: International Society of Automation, 2007.
- [41] A. Zoitl, *Real-time execution for IEC 61499*, Durham, NC, USA: International Society of Automation, 2008.
- [42] A. Zoitl and R. Lewis, *Modelling control systems using IEC 61499*, 2nd ed., London, UK: The Institution of Engineering and Technology, 2014.
- [43] A. Zoitl and T. Strasser, Eds., *Distributed control applications: Guidelines, design patterns, and application examples with the IEC 61499*, Boca Raton, FL, USA: CRC Press, 2017.
- [44] G. Cengic and K. Akesson, "On formal analysis of IEC 61499 applications, Part A: modeling," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 2, pp. 136-144, May 2010a.
- [45] G. Cengic and K. Akesson, "On formal analysis of IEC 61499 applications, Part B: execution semantics," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 2, pp. 145-154, May 2010b.
- [46] C. Sünder, A. Zoitl, J. H. Christensen, V. Vyatkin, R. W. Brennan, A. Valentini, L. Ferrarini, T. Strasser, J. L. M. Lastra, and F. Auinger, "Usability and interoperability of IEC 61499 based distributed automation systems," In *Proceedings of 4th IEEE International Conference on Industrial Informatics*, Singapore, Aug. 2006b, pp. 31-37.
- [47] V. Vyatkin and J. Chouinard, "On comparisons of the ISaGRAF implementation of IEC 61499 with FBDK and other implementations," In *Proceedings of 6th IEEE International Conference on Industrial Informatics*, Daejeon, Korea, July 2008, pp. 264-269.
- [48] P. Tata and V. Vyatkin, "Proposing a novel IEC 61499 runtime framework implementing the cyclic execution semantics," In *Proceedings of 7th IEEE International Conference on Industrial Informatics*, Cardiff, Wales, UK, June 2009, pp. 416-421.
- [49] G. Cengic, O. Ljungkrantz, and K. Akesson, "Formal modeling of function block applications running in IEC 61499 execution runtime," In *Proceedings of 11th IEEE International Conference on Emerging Technologies and Factory Automation*, Prague, Czech Republic, Sept. 2006b, pp. 1269-1276.
- [50] V. Vyatkin and V. Dubinin, "Sequential axiomatic model for execution of basic function blocks in IEC 61499," In *Proceedings of 5th IEEE International Conference on Industrial Informatics*, Vienna, Austria, June 2007, pp. 1183-1188.
- [51] V. Vyatkin, V. Dubinin, C. Veber, and L. Ferrarini, "Alternatives for execution semantics of IEC 61499," In *Proceedings of 5th IEEE International Conference on Industrial Informatics*, Vienna, Austria. June 2007, pp. 1151-1156.
- [52] V. Dubinin and V. Vyatkin, "On definition of a formal model for IEC 61499 function blocks," *EURASIP Journal on Embedded Systems*, vol. 2008, pp. 1-10, Apr. 2008.
- [53] L. H. Yoong, P. S. Roop, V. Vyatkin, and Z. Salcic, "A synchronous approach for IEC 61499 function block implementation," *IEEE Transactions on Computers*, vol. 58, no. 12, pp. 1599-1614, Dec. 2009.
- [54] L. H. Yoong, P. S. Roop, Z. E. Bhatti, and M. Kuo, *Model-driven design using IEC 61499: A synchronous approach for embedded and automation systems*, Switzerland: Springer, 2015.



- [55] V. Vyatkin and V. Dubinin, "Refactoring of execution control charts in basic function blocks of the IEC 61499 standard," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 2, pp. 155-165, May 2010.
- [56] V. Dubinin and V. Vyatkin, "Semantics-robust design patterns for IEC 61499," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 279-290, May 2012.
- [57] W. Dai, V. Dubinin, and V. Vyatkin, "IEC 61499 ontology model for semantic analysis and code generation," In *Proceedings of 9th IEEE International Conference on Industrial Informatics*, Caparica, Lisbon, Portugal, July 2011a, pp. 597-602.
- [58] W. Dai, V. Vyatkin and V. Dubinin, "Ontology-based design recovery and migration between IEC 61499-compliant tools," In *Proceedings of 37th Annual Conference of the IEEE Industrial Electronics Society*, Melbourne, Australia, Nov. 2011b, pp. 4332-4337.
- [59] P. Lindgren, M. Lindner, A. Lindner, J. Eriksson, and V. Vyatkin, "Real-time execution of function blocks for internet of things using the RTFM-kernel," In *Proceedings of 19th IEEE International Conference on Emerging Technology and Factory Automation*, Barcelona, Spain, Sept. 2014, pp. 1-6.
- [60] P. Lindgren, M. Lindner, A. Lindner, V. Vyatkin, D. Pereira, and L.M. Pinho, "A real-time semantics for the IEC 61499 standard," In *Proceedings of 20th IEEE International Conference on Emerging Technologies and Factory Automation*, Luxembourg, Sept. 2015, pp. 1-6.
- [61] J. Eriksson, F. Häggström, S. Aittamaa, A. Kruglyak, and P. Lindgren, "Real-time for the masses, step 1: Programming API and static priority SRP kernel primitives," In *Proceedings of 8th IEEE International Symposium on Industrial Embedded Systems*, Porto, Portugal, June 2013, pp. 110-113.
- [62] A. Lindner, M. Lindner, and P. Lindgren, "RTFM-RT: A threaded runtime for RTFM-core-towards execution of IEC 61499," In *Proceedings of 20th IEEE International Conference on Emerging Technologies and Factory Automation*, Luxembourg, Sept. 2015, pp. 1-8.
- [63] P. Lindgren, J. Eriksson, M. Lindner, A. Lindner, D. Pereira, L. M. Pinho, "End-to-end response time of IEC 61499 distributed applications over switched Ethernet," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 1, pp. 287-297, Feb. 2017.
- [64] L. H. Yoong and P. S. Roop, "Verifying IEC 61499 function blocks using Esterel," *IEEE Embedded Systems Letters*, vol. 2, no. 1, pp. 1-4, Mar. 2010.
- [65] M. Kuo and P. S. Roop. "New design patterns for time-predictable execution of function blocks," In *Distributed Control Applications: Guidelines, Design Patterns, and Application Examples with the IEC 61499*, A. Zoitl and T. Strasser, Eds., Boca Raton, FL, USA: CRC Press, 2017, pp. 69-95.
- [66] R. Sinha, P. S. Roop, G. Shaw, Z. Salcic, and M. Kuo, "Hierarchical and concurrent ECCs for IEC 61499 function blocks," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 59-68, Feb. 2016.
- [67] D. Li, Z. Zhai, Z. Pang, V. Vyatkin, and C. Liu, "Synchronous-reactive semantic modeling and verification for function block networks," *IEEE Transactions on Industrial Informatics*, vol 13, no. 6, pp. 3389-3398, Dec. 2017.
- [68] W. Dai, C. Pang, V. Vyatkin, J. H. Christensen, and X. Guan, "Discrete-event-based deterministic execution semantics with timestamps for industrial cyber-physical systems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 16, no. 99, pp. 1-12, Aug. 2017b.
- [69] *Schneider Electric nxtControl*. Accessed: Dec. 05, 2022. [Online]. Available: <https://www.nxtcontrol.com/en/>.
- [70] *Rockwell Automation ISaGRAF*. Accessed: Dec. 05, 2022. [Online]. Available: <http://www.isagraf.com/>.
- [71] C. Shapiro and H. R. Varian, *Information rules: A strategic guide to the network economy*, Boston, MA, USA: Harvard Business Press, 1999.
- [72] S. Sierla, J. H. Christensen, K. Koskinen, and J. Peltola, "Educational approaches for the industrial acceptance of IEC 61499," In *Proceedings of 12th IEEE International Conference on Emerging Technologies and Factory Automation*, Patras, Greece, Sept. 2007, pp. 482-489.

- [73] J. Peltola, J. H. Christensen, S. Sierla, and K. Koskinen, "A migration path to IEC 61499 for the batch process industry," In *Proceedings of 5th IEEE International Conference on Industrial Informatics*, Vienna, Austria, June 2007, pp. 811-816.
- [74] F. A. Cabadini, G. Montalbano, G. Kollegger, H. Mayer, and V. Vytakin, "IEC-61499 distributed automation for the next generation of manufacturing systems," In *the Digital Shopfloor: Industrial Automation in the Industry 4.0 Era - Performance Analysis and Applications*, J. Soldatos, O. Lazaro, and F. Cavadini, Eds., Denmark: River Publishers, 2019, pp. 103-127.
- [75] *Daedalus*. Accessed: Dec. 05, 2022. [Online]. Available: <http://www.daedalus.iec61499.eu/>.
- [76] A. Barni, A. Brusaferrri, F. A. Cavadini, G. Landolfi, S. Patil, D. Piga, S. Spinelli, and V. Vyatkin, "Fostering the creation of a digital ecosystem by a distributed IEC-61499 based automation platform," In *Proceedings of 17th IEEE International Conference on Industrial Informatics*, Helsinki-Espoo, Finland, July 2019, pp. 635-640.
- [77] W. Dai, V. Dubinin, J. H. Christensen, V. Vyatkin, and X. Guan, "Toward self-manageable and adaptive industrial cyber-physical systems with knowledge-driven autonomic service management," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 2, pp. 725-736, Apr. 2017a.
- [78] *Geeking IEC 61499*. Accessed: Dec. 05, 2022. [Online]. Available: <https://2019.ieee-indin.org/workshops/ws1-geeking-iec61499/>.
- [79] *Eclipse 4diac*. Accessed: Dec. 05, 2022. [Online]. Available: <https://www.eclipse.org/4diac/>.
- [80] *Flexbridge*. Accessed: Dec. 05, 2022. [Online]. Available: <https://training.flexbridge.se/>.
- [81] J. Fischer and T. O. Boucher, "Workbook for designing distributed control applications using Rockwell Automation's HOLOBLOC prototyping software." Accessed: Dec. 05, 2022. [Online]. Available: [http://www.diiit.unict.it/users/scava/dispense/II\\_270/TutorialFDBK.pdf](http://www.diiit.unict.it/users/scava/dispense/II_270/TutorialFDBK.pdf).
- [82] *Holobloc*. Accessed: Dec. 05, 2022. [Online]. Available: <https://www.holobloc.com/>.
- [83] P. Gsellmann, M. Melik-Merkumians, and G. Schitter, "Comparison of code measures of IEC 61131-3 and 61499 standards for typical automation applications," In *Proceedings of 23rd IEEE International Conference on Emerging Technologies and Factory Automation*, Torino, Italy, Sept. 2018, pp. 1047-1050.
- [84] C. Sünder, M. Wenger, C. Hanni, I. Gosetti, H. Steininger, and J. Fritsche, "Transformation of existing IEC 61131-3 automation projects into control logic according to IEC 61499," In *Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation*, Hamburg, Germany, Sept. 2008a, pp. 369-376.
- [85] C. Sünder, A. Zoitl, J. H. Christensen, H. Steininger, and J. Fritsche, "Considering IEC 61131-3 and IEC 61499 in the context of component frameworks," In *Proceedings of 6th IEEE International Conference on Industrial Informatics*, Daejeon, Korea, July 2008b, pp. 277-282.
- [86] M. Wenger, A. Zoitl, C. Sünder, and H. Steininger, "Transformation of IEC 61131-3 to IEC 61499 based on a model driven development approach," In *Proceedings of 7th IEEE International Conference on Industrial Informatics*, Cardiff, Wales, UK, June 2009a, pp. 715-720.
- [87] M. Wenger, A. Zoitl, C. Sünder, and H. Steininger, "Semantic correct transformation of IEC 61131-3 models into the IEC 61499 standard," In *Proceedings of 14th IEEE International Conference on Emerging Technologies and Factory Automation*, Mallorca, Spain, Sept. 2009b, pp. 1-7.
- [88] M. Wenger, A. Zoitl, and G. Schitter, "Automatic reengineering of IEC 61131-based control applications into IEC 61499. In *Distributed Control Applications: Guidelines, Design Patterns, and Application Examples with the IEC 61499*, A. Zoitl and T. Strasser, Eds., Boca Raton, FL, USA: CRC Press, 2017, pp. 97-121.
- [89] *CoDeSys*. Accessed: Dec. 05, 2022. [Online]. Available: <https://www.codesys.com/>.
- [90] W. Dai and V. Vyatkin, "Redesign distributed IEC 61131-3 PLC system in IEC 61499 function blocks," In *Proceedings of 15th IEEE International Conference on Emerging Technologies and Factory Automation*, Bilbao, Spain, Sept. 2010, pp. 1-8.

- [91] W. Dai and V. Vyatkin, "Redesign distributed PLC control systems using IEC 61499 function blocks," *IEEE Transactions on Automation Science and Engineering*, vol. 9, no. 2, pp. 390-401, Apr. 2012a.
- [92] W. Dai and V. Vyatkin, "Ontology model for migration from IEC 61131-3 PLC to IEC 61499 function block," In *Proceedings of 6th IEEE International Symposium on Electronic Design, Test and Application*, Queenstown, New Zealand, Jan. 2011, pp. 172-175.
- [93] W. Dai and V. Vyatkin, "Transformation from PLC to distributed control using ontology mapping," In *Proceedings of 10th IEEE International Conference on Industrial Informatics*, Beijing, China, Sept. 2012b, pp. 436-441.
- [94] W. Dai, V. Dubinin, and V. Vyatkin, "Migration from PLC to IEC 61499 using semantic web technologies," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 3, pp. 277-291, Mar. 2014a.
- [95] W. Dai, V. Vyatkin, and J. H. Christensen, "Essential elements for programming of distributed automation and control systems," In *Proceedings of 18th IEEE International Conference on Emerging Technologies and Factory Automation*, Cagliari, Italy, Sept. 2013, pp. 1-8.
- [96] IEC 61131-5, *Programmable controllers - Part 5: Communications*, International Standard, 1st ed., Geneva, Switzerland: International Electrotechnical Commission, 2000.
- [97] S. Campanelli, P. Foglia, and C. A. Prete, "Integration of existing IEC 61131-3 systems in an IEC 61499 distributed solution," In *Proceedings of 17th IEEE International Conference on Emerging Technologies and Factory Automation*, Krakow, Poland, Sept. 2012, pp. 1-8.
- [98] S. Campanelli, P. Foglia, and C. A. Prete, "An architecture to integrate IEC 61131-3 systems in an IEC 61499 distributed solution," *Computers in Industry*, vol. 72, pp. 47-67, Sept. 2015.
- [99] F. Jammes and H. Smit, "Service-oriented paradigms in industrial automation," *IEEE Transactions on Industrial Informatics*, vol. 1, no. 1, pp. 62-70, Feb. 2005.
- [100] IBM, *An architectural blueprint for autonomic computing*, IBM White Paper Autonomic Computing, 4th ed., June 2006.
- [101] O. Givchchi, H. Trsek, and J. Jasperneite, "Cloud computing for industrial automation systems - a comprehensive overview," In *Proceedings of 18th IEEE International Conference on Emerging Technologies and Factory Automation*, Cagliari, Italy, Sept. 2013, pp. 1-4.
- [102] W. Dai, V. Vyatkin, and J. H. Christensen, "Applying IEC 61499 design paradigms: Object-oriented programming, component-based design, and service-oriented architecture," In *Distributed Control Applications: Guidelines, Design Patterns, and Application Examples with the IEC 61499*, A. Zoitl and T. Strasser, Eds., Boca Raton, FL, USA: CRC Press, 2017c, pp. 39-68.
- [103] W. Dai, J. Peltola, V. Vyatkin, and C. Pang, "Service-oriented distributed control software design for process automation systems," In *Proceedings of 2014 IEEE International Conference on Systems, Man and Cybernetics*, San Diego, CA, USA, Oct. 2014b, pp. 3637-3642.
- [104] V. Vyatkin, J. H. Christensen, and J. L. M. Lastra, "OOONEIDA: An open, object-oriented knowledge economy for intelligent industrial automation," *IEEE Transactions on Industrial Informatics*, vol. 1, no. 1, pp. 4-17, Feb. 2005.
- [105] V. Vyatkin, C. Pang, Y. Deng, M. Sorouri, and H. Mayer, "System-level architecture for building automation systems: Object-orientated design and simulation," In *Proceedings of 39th IEEE Annual Conference of Industrial Electronics Society*, Vienna, Austria, Nov. 2013, pp. 5334-5339.
- [106] G. Cengic, O. Ljungkrantz, and K. Akesson, "A framework for component based distributed control software development using IEC 61499," In *Proceedings of 11th IEEE International Conference on Emerging Technologies and Factory Automation*, Prague, Czech Republic, Sept. 2006a, pp. 782-789.
- [107] C. Sünder, A. Zoitl, M. Rainbauer, and B. Favre-Bulle, "Hierarchical control modelling architecture for modular distributed automation systems," In *Proceedings of 4th IEEE International Conference on Industrial Informatics*, Singapore, Aug. 2006a, pp. 12-17.

- [108] W. Lepuschitz and A. Zoitl, "An engineering method for batch process automation using a component-oriented design based on IEC 61499," In *Proceedings of 13th IEEE International Conference Emerging Technologies and Factory Automation*, Hamburg, Germany, Sept. 2008, pp. 207-214.
- [109] R. Hametner, A. Zoitl, and M. Semo, "Automation component architecture for the efficient development of industrial automation systems," In *Proceedings of 6th IEEE International Conference on Automation Science and Engineering*, Toronto, Canada, Aug. 2010, pp. 156-161.
- [110] V. Vyatkin, "Intelligent mechatronic components: Control system engineering using an open distributed architecture," In *Proceedings of 8th IEEE International Conference on Emerging Technologies and Factory Automation*, Lisbon, Portugal, Sept. 2003, vol. 2, pp. 277-284.
- [111] O. J. L. Orozco and J. L. M. Lastra, "Adding function blocks of IEC 61499 semantic description to automation objects," In *Proceedings of 11th IEEE International Conference on Emerging Technologies and Factory Automation*, Prague, Czech Republic, Sept. 2006, pp. 537-544.
- [112] R. W. Brennan, L. Ferrarini, J. Martinez, and V. Vyatkin, "Automation objects: Enabling embedded intelligence in real-time mechatronic systems," *International Journal of Manufacturing Research*, vol. 1, no. 4, pp. 379-381, 2006.
- [113] G. Black and V. Vyatkin, "Intelligent component-based automation of baggage handling systems with IEC 61499," *IEEE Transactions on Automation Science and Engineering*, vol. 7, no. 2, pp. 337-351, Apr. 2010.
- [114] W. Dai and V. Vyatkin, "A component-based design pattern for improving reusability of automation programs," In *Proceedings of 39th Annual Conference of the IEEE Industrial Electronics Society*, Vienna, Austria, Nov. 2013, pp. 4328-4333.
- [115] A. Zoitl and H. Prähofer, "Building hierarchical automation solutions in the IEC 61499 modeling language," In *Proceedings of 9th IEEE International Conference on Industrial Informatics*, Caparica, Lisbon, Portugal, July 2011, pp. 557-564.
- [116] A. Zoitl and H. Prähofer, "Guidelines and patterns for building hierarchical automation solutions in the IEC 61499 modeling language," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 4, pp. 2387-2396, Nov. 2013.
- [117] V. Mařík and D. McFarlane, "Industrial adoption of agent-based technologies," *IEEE Intelligent Systems*, vol. 20, no. 1, pp. 27-35, Jan. 2005.
- [118] R. W. Brennan, "Toward real-time distributed intelligent control: A survey of research themes and applications," *IEEE Transactions on Systems, Man, and Cybernetics - Part C: Applications and Reviews*, vol. 37, no. 5, pp. 744-765, Sept. 2007.
- [119] P. Leitão, V. Mařík, and P. Vrba, "Past, present, and future of industrial agent applications," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 4, pp. 2360-2372, Nov. 2013.
- [120] P. Leitão, A. W. Colombo, and S. Karnouskos, "Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges," *Computers in Industry*, vol. 81, pp. 11-25, Sept. 2016a.
- [121] P. Leitão, S. Karnouskos, L. Ribeiro, J. Lee, T. Strasser, and A. W. Colombo, "Smart agents in industrial cyber-physical systems," *Proceedings of the IEEE*, vol. 104, no. 5, pp. 1086-1101, May 2016b.
- [122] IEEE Std 2660.1-2020, *IEEE recommended practice for industrial agents: Integration of software agents and low-level automation functions*, International Standard, Piscataway, NJ: IEEE Industrial Electronics Society, 2021.
- [123] R. W. Brennan, M. Fletcher, and D. H. Norrie, "An agent-based approach to reconfiguration of real-time distributed control systems," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 4, pp. 444-451, Aug. 2002a.
- [124] R. W. Brennan, X. Zhang, Y. Xu, and D. H. Norrie, "A reconfigurable concurrent function block model and its implementation in real-time Java," *Integrated Computer-Aided Engineering*, vol. 9, no. 3, pp. 263-279, Jan. 2002b.

- [125] S. Olsen, J. Wang, A. Ramirez-Serrano, and R. W. Brennan, "Contingencies-based reconfiguration of distributed factory automation," *Robotics and Computer-Integrated Manufacturing*, vol. 21, no. 4-5, pp. 379-390, Aug. 2005.
- [126] J. Chouinard and R. W. Brennan, "Software for next generation automation and control," In *Proceedings of 4th IEEE International Conference on Industrial Informatics*, Singapore, Aug. 2006, pp. 886-891.
- [127] J. J. Scarlett and R. W. Brennan, "A new evaluation method of communication for distributed control," In *Proceedings of IEEE Workshop on Distributed Intelligent Systems: Collective Intelligence and Its Applications*, Prague, Czech Republic, June 2006, pp. 97-102.
- [128] J. J. Scarlett and R. W. Brennan, "Evaluating a new communication protocol for real-time distributed control," *Robotics and Computer-Integrated Manufacturing*, vol. 27, no. 3, pp. 627-635, June 2011.
- [129] N. Cai and R. W. Brennan, "Distributed sensing and control architecture for automotive factory automation," In *Proceedings of 4th International Conference on Industrial Applications of Holonic and Multi-Agent Systems*, Linz, Austria, Aug. 2009, pp. 165-174.
- [130] N. Cai, M. Gholami, L. Yang, and R. W. Brennan, "Application-oriented intelligent middleware for distributed sensing and control," *IEEE Transactions on Systems, Man, and Cybernetics - Part C: Applications and Reviews*, vol. 42, no. 6, pp. 947-956, Nov. 2012.
- [131] M. Khalgui, O. Mosbahi, Z. Li, and HM Hanisch, "Reconfiguration of distributed embedded-control systems," *IEEE/ASME Transactions on Mechatronics*, vol. 16, no. 4, pp. 684-694, July 2010.
- [132] S. Guellouz, A. Benzina, M. Khalgui, G. Frey, Z. Li, and V. Vyatkin, "Designing efficient reconfigurable control systems using IEC61499 and symbolic model checking," *IEEE Transactions on Automation Science and Engineering*, vol. 16, no. 3 pp. 1110-1124, Nov. 2018.
- [133] A. Bonci, M. Pirani, and S. Longhi, "A database-centric framework for the modeling, simulation, and control of cyber-physical systems in the factory of the future," *Journal of Intelligent Systems*, vol. 27, no. 4, pp. 659-679, Oct. 2018.
- [134] A. Bonci, S. Longhi, and M. Pirani, "RMAS architecture for autonomic computing in cyber-physical systems," In *Proceedings of IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*, Lisbon, Portugal, Oct. 2019, pp. 2996-3003.
- [135] A. Bonci, S. Longhi, E. Lorenzoni, and M. Pirani, "RMAS architecture for industrial agents in IEC 61499," *Procedia Manufacturing*, vol. 42, pp. 84-90, Jan. 2020.
- [136] A. Bonci, S. Longhi, and M. Pirani, "IEC 61499 device management model through the lenses of RMAS," *Procedia Computer Science*, vol. 180, pp. 656-665, Jan. 2021.
- [137] W. Dai, L. Riliskis, P. Wang, V. Vyatkin, and X. Guan, "A cloud-based decision support system for self-healing in distributed automation systems using fault tree analysis," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 3, pp. 989-1000, Mar. 2018.
- [138] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing - degrees, models, and applications," *ACM Computing Surveys (CSUR)*, vol. 40, no. 3, pp. 1-28, Aug. 2008.
- [139] H. Mubarak and P. Göhner, "An agent-oriented approach for self-management of industrial automation systems," In *Proceedings of 8th IEEE International Conference on Industrial Informatics*, Osaka, Japan, July 2010, pp. 721-726.
- [140] W. Lepuschitz, A. Zoitl, M. Vallée, and M. Merdan, "Toward self-reconfiguration of manufacturing systems using automation agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 41, no. 1, pp. 52-69, Jan. 2011.
- [141] T. Strasser and R. Froschauer, "Autonomous application recovery in distributed intelligent automation and control systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 42, no. 6, pp. 1054-1070, Nov. 2012.

- [142] H. Kaindl, M. Vallée, and E. Arnautovic, "Self-representation for self-configuration and monitoring in agent-based flexible automation systems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 43, no. 1, pp. 164-175, Jan. 2013.
- [143] W. Dai, V. Vyatkin, J. H. Christensen, and V. Dubinin, "Bridging service-oriented architecture and IEC 61499 for flexibility and interoperability," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 3, pp. 771-781, June 2015a.
- [144] W. Dai, W. Huang, and V. Vyatkin, "Enabling plug-and-play software components in industrial cyber-physical systems by adopting service-oriented architecture paradigm," In *Proceedings of 42nd Annual Conference of the IEEE Industrial Electronics Society*, Florence, Italy, Oct. 2016, pp. 5253-5258.
- [145] W. Dai, V. Vyatkin, C. Chen, and X. Guan, "Modeling distributed automation systems in cyber-physical view," In *Proceedings of 10th IEEE International Conference on Industrial Electronics and Applications*, Auckland, New Zealand, June 2015b, pp. 984-989.
- [146] W. Huang, W. Dai, P. Wang, and V. Vyatkin, "Real-time data acquisition support for IEC 61499 based industrial cyber-physical systems," In *Proceedings of 43rd Annual Conference of the IEEE Industrial Electronics Society*, Beijing, China, Oct. 2017, pp. 6689-6694.
- [147] P. Mell and T. Grance, *The NIST definition of cloud computing*, The National Institute of Standards and Technology, Gaithersburg, MD, USA, Special Publication 800-145, Sept. 2011.
- [148] S. Karnouskos, A. W. Colombo, T. Bangemann, K. Manninen, R. Camp, M. Tilly, P. Stluka, F. Jammes, J. Delsing, and J. Eliasson, "A SOA-based architecture for empowering future collaborative cloud-based industrial automation," In *Proceedings of 38th Annual Conference of the IEEE Industrial Electronics Society*, Montreal, Canada, Oct. 2012, pp. 5766-5772.
- [149] W. Dai, L. Riliskis, V. Vyatkin, E. Osipov, and J. Delsing, "A configurable cloud-based testing infrastructure for interoperable distributed automation systems," In *Proceedings of 40th Annual Conference of the IEEE Industrial Electronics Society*, Dallas, TX, USA, Oct. 2014c, pp. 2492-2498.
- [150] E. Demin, S. Patil, V. Dubinin, and V. Vyatkin, "IEC 61499 distributed control enhanced with cloud-based web-services," In *Proceedings of 10th IEEE International Conference on Industrial Electronics and Applications*, Auckland, New Zealand, June 2015, pp. 972-977.
- [151] M. Wenger, A. Zoitl, and J. O. Blech, "Behavioral type-based monitoring for IEC 61499," In *Proceedings of 20th IEEE International Conference on Emerging Technologies and Factory Automation*, Luxembourg, Sept. 2015a, pp. 1-8.
- [152] M. Wenger, A. Zoitl, J. O. Blech, I. Peake, and L. Fernando, "Cloud based monitoring of timed events for industrial automation," In *Proceedings of 21st IEEE International Conference on Parallel and Distributed Systems*, Melbourne, Australia, Dec. 2015b, pp. 827-830.
- [153] A. D. Rocha, J. Tripa, D. Alemao, R. S. Peres, and J. Barata, "Agent-based plug and produce cyber-physical production system - test case," In *Proceedings of 17th IEEE International Conference on Industrial Informatics*, Helsinki-Espoo, Finland, July 2019, pp. 1545-1551.
- [154] *Automation of Things*. Accessed: Dec. 05, 2022. [Online]. Available: <https://aot-me.com/>.
- [155] *Yueyi Automation*. Accessed: Dec. 05, 2022. [Online]. Available: <http://www.iec61499.cn/>.
- [156] *NOJA Power*. Accessed: Dec. 05, 2022. [Online]. Available: <https://www.nojapower.com/product/software>.
- [157] *PRETzel BlokIDE*. Accessed: Dec. 05, 2022. [Online]. Available: <http://pretzel.ece.auckland.ac.nz/>.
- [158] *O3neida Workbench*. Accessed: Dec. 05, 2022. [Online]. Available: <http://ooneida-wb.sourceforge.net/>.
- [159] *O3neida FBench*. Accessed: Dec. 05, 2022. [Online]. Available: <http://ooneida-fbench.sourceforge.net/>.
- [160] *Fuber*. Accessed: Dec. 05, 2022. [Online]. Available: <https://sourceforge.net/projects/fuber/>.
- [161] *SEG*. Accessed: Aug. 20, 2019. [Online]. Available: <http://seg.ece.upatras.gr/>.
- [162] *UDESC ICARU\_FB*. Accessed: Dec. 05, 2022. [Online]. Available: <https://sourceforge.net/projects/icarufb/>.

- [163] *UDESC GASR-FBE*. Accessed: Dec. 05, 2022. [Online]. Available: <https://sourceforge.net/projects/gasrfbe/>.
- [164] J. H. Christensen, "Design patterns, frameworks, and methodologies," In *Distributed Control Applications: Guidelines, Design Patterns, and Application Examples with the IEC 61499*, A. Zoitl and T. Strasser, Eds., Boca Raton, FL, USA: CRC Press, 2017, pp. 27-37.
- [165] L. H. Yoong, P. S. Roop, and Z. Salcic, "Implementing constrained cyber-physical systems with IEC 61499," *ACM Transactions on Embedded Computing Systems*, vol. 11, no. 4, Dec. 2012, Article 78.
- [166] A. Brusafferri, A. Ballarino, and E. Carpanzano, "Reconfigurable knowledge-based control solutions for responsive manufacturing systems," *Studies in Informatics and Control*, vol. 20, no. 1, pp. 31-42, Mar. 2011.
- [167] R. Baniya, M. Maksimainen, S. Sierla, C. Pang, C. W. Yang, and V. Vyatkin, "Smart indoor lighting control: Power, illuminance, and color quality," In *Proceedings of 23rd IEEE International Symposium on Industrial Electronics*, Istanbul, Turkey, June 2014, pp. 1745-1750.
- [168] G. Zhabelova and V. Vyatkin, "Multiagent smart grid automation architecture based on IEC 61850/61499 intelligent logical nodes," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 5, pp. 2351-2362, May 2012.
- [169] G. Zhabelova, C. W. Yang, S. Patil, C. Pang, J. Yan, A. Shalyto, and V. Vyatkin, "Cyber-physical components for heterogeneous modelling, validation and implementation of smart grid intelligence," In *Proceedings of 12th IEEE International Conference on Industrial Informatics*, Porto Alegre, Brazil, July 2014, pp. 411-417.
- [170] F. Andr n, R. Br ndlinger, T. Strasser, "IEC 61850/61499 control of distributed energy resources: Concept, guidelines, and implementation," *IEEE Transactions on Energy Conversion*, vol. 29, no. 4, pp. 1008-1017, Dec. 2014.
- [171] *OPA Forum*. Accessed: Dec. 05, 2022. [Online]. Available: <https://www.controlglobal.com/articles/2018/opa-forum-expands-open-interoperable-standard-for-process-control-device-interfaces-part-2/>.
- [172] P. Tait, "A path to industrial adoption of distributed control technology," In *Proceedings of 3rd IEEE International Conference on Industrial Informatics*, Perth, Australia. Aug. 2005, pp. 86-91.
- [173] M. Colla, A. Brusafferri, and E. Carpanzano, "Applying the IEC-61499 model to the shoe manufacturing sector," In *Proceedings of 11th IEEE International Conference on Emerging Technologies and Factory Automation*, Prague, Czech Republic, Sept. 2006, pp. 1301-1308.
- [174] *Centris Technologies*. Accessed: Dec. 05, 2022. [Online]. Available: <https://www.centristech.com/en/>.
- [175] *Kibernetika*. Accessed: Dec. 05, 2022. [Online]. Available: <http://www.kibernetika-bg.com/>.
- [176] *EKE Electronics*. Accessed: Dec. 05, 2022. [Online]. Available: <https://www.eke-electronics.com/>.
- [177] *ICP DAS USA*. Accessed: Dec. 05, 2022. [Online]. Available: <https://www.icpdas-usa.com/>.
- [178] B. Scholten, *The road to integration: A guide to applying the ISA-95 standard in manufacturing*, Paris, France: International Society of Automation, 2007.
- [179] K. Tange, M. De Donno, X. Fafoutis, and N. Dragoni, "A systematic survey of industrial Internet of Things security: requirements and fog computing opportunities," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 4, pp. 2489-2520, July 2020.
- [180] W. Dai, H. Nishi, V. Vyatkin, V. Huang, Y. Shi, and X. Guan, "Industrial edge computing: enabling embedded intelligence," *IEEE Industrial Electronics Magazine*, vol. 13, no. 4, pp. 48-56, Dec. 2019.
- [181] C. Y. Lin, S. Zeadally, T. S. Chen, and C. Y. Chang, "Enabling cyber physical systems with wireless sensor networking technologies," *International Journal of Distributed Sensor Networks*, vol. 8, no. 5, pp. 1-21, May 2012.
- [182] M. S. Taboun and R. W. Brennan, "An embedded multi-agent systems based industrial wireless sensor network," *Sensors*, vol. 17, no. 9, pp. 2112, Sept. 2017.

- [183] U. Wilensky and W. Rand, *An introduction to agent-based modeling: Modeling natural, social, and engineered complex systems with NetLogo*, Cambridge, MA, USA: The MIT Press, 2015.
- [184] I. Sakellariou, P. Kefalas, and I. Stamatopoulou, "Enhancing NetLogo to simulate BDI communicating agents," *Lecture Notes in Artificial Intelligence*, vol. 5138, pp. 263-275, Oct. 2008.
- [185] J. D. C. Little, "A proof for the queuing formula:  $L = \lambda W$ ," *Operations Research*, vol. 9, no.3, pp. 383-387, June 1961.
- [186] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269-271, Jan. 1959.
- [187] L. Fu, D. Sun, and R. Rilett, "Heuristic shortest path algorithms for transportation applications: State of the art," *Computers & Operations Research*, vol. 33, no. 11, pp. 3324-3343, Nov. 2006.
- [188] *Jetson Nano*. Accessed: Dec. 05, 2022. [Online]. Available: <https://jetbot.org/master/>.
- [189] *Raspberry Pi*. Accessed: Dec. 05, 2022. [Online]. Available: <https://www.raspberrypi.org/>.
- [190] *SPADE*. Accessed: Dec. 05, 2022. [Online]. Available: <https://pypi.org/project/spade/>.



*[This page intentionally left blank]*

# Appendices

## Appendix A Multi-Agent Simulation Model

### *Appendix A.1 NetLogo Agent Based Simulation Model*

```
__includes ["communication.nls" "agents.nls"]

breed [ diverters diverter ]
breed [ parts part ]
breed [ SMSAgents SMSAgent ] ;; Self-Manageable Service Execution Agent
breed [ SCAgents SCAgent ] ;; Self-Configuration Agent
breed [ SOAgents SOAgent ] ;; Self-Optimization Agent
breed [ SHAgents SHAgent ] ;; Self-Healing Agent
breed [ SPAgents SPAgent ] ;; Self-Protection Agent

globals
[
  no-messages      ;; tally of the number of messages sent (used by communication.nls)
  part-types       ;; a word containing a list of the part types in the system
  no-conveyors      ;; number of conveyor sections
  diverter-states   ;; nested list of diverter states [[<L> <C> <R>][<L> <C> <R>]...]
  diverter-order    ;; a list of the diverters, sorted by shortest path
  ave-wait-time     ;; average time (in ticks) that parts spend in the system
  parts-completed   ;; number of parts that have made it to a storage bin
  ave-no-parts      ;; average number of parts in the system
  next-arrival      ;; arrival time of the next part
  conv-failure      ;; time (in ticks) when conveyor failure occurs
]

patches-own
[
  conveyor-number  ;; conveyor number (this is also the diverter number for Type 2)
  diverter?        ;; TRUE if a diverter patch
  in-sensor?       ;; TRUE if an input sensor patch
  out-sensor?      ;; TRUE if an output sensor patch
  storage-bin?     ;; TRUE if a storage bin patch
  bin-number       ;; storage bin number
  div-number       ;; diverter number
  inbound          ;; inbound diverter list
  direction        ;; direction (heading) for patch
  operating?       ;; TRUE if operating, FALSE if failed
  connections      ;; a list of conveyor link connections for input sensor patches
  part-at-bin      ;; used to check part types exiting storage bin
]

diverters-own
[
  diverter-number  ;; diverter number
]
```

```

parts-own
[
  part-type      ;; part type identifier (currently we are using lower case letters)
  time-in        ;; time (in ticks) when part entered the system
  wait-time      ;; time (in ticks) that the part spent in the system
]

SMSAgents-own
[
  incoming-queue  ;; incoming message queue
  proc-delay      ;; processing delay time (currently used to simulate a processing delay)
]

SCAgents-own
[
  incoming-queue  ;; incoming message queue
  proc-delay      ;; processing delay time (currently used to simulate a processing delay)
]

SOAgents-own
[
  incoming-queue  ;; incoming message queue
  proc-delay      ;; processing delay time (currently used to simulate a processing delay)
]

SHAgents-own
[
  incoming-queue  ;; incoming message queue
  proc-delay      ;; processing delay time (currently used to simulate a processing delay)
]

SPAgents-own
[
  incoming-queue  ;; incoming message queue
  proc-delay      ;; processing delay time (currently used to simulate a processing delay)
]

to setup
  clear-all
  reset-timer
  set Conveyor-Section-Failure 1
  setup-lagents
  setup-hagents
  setup-hconnections
  set no-messages 0
  set ave-wait-time 0
  set parts-completed 0
  set next-arrival 0
  ask patches
  [
    set diverter? FALSE
    set in-sensor? FALSE
    set out-sensor? FALSE
    set storage-bin? FALSE
  ]
  reset-ticks

```

```

create-conveyor "input.txt"
;setup-diverters "connections.txt"
;configure-diverters Number-Part-Types
;set diverter-order [1 2 3 4 5 6] ;; ordered list (and section 8 failure)
ifelse Optimize-Path = TRUE
[ set diverter-order [1 2 6 3 5 7 4 8] ] ;; normal
[ set diverter-order [1 2 3 4 5 6 7 8] ] ;; ordered list
diverter-config
set conv-failure random-exponential MTTF
end

to go
;; Create a new part with probability = Part-Arrival
if ticks >= next-arrival
[
  new-part Number-Part-Types
  set next-arrival ( random-exponential Mean-Arrival-Time ) + ticks
]
ask parts
[
  set heading direction
  if out-sensor? = TRUE [ diverter-control ]
  if storage-bin? = TRUE [ store-part ]
  if operating? = TRUE
  [
    if not any? parts-on patch-ahead 1 [ forward 1 ]
    ;forward 1
  ]
]
ask SMSAgents
[
  monitor-part-types
  monitor-conveyor-status Conveyor-Section-Failure
]
calculate-parts-in-system
manage-messages ;; might want to add an update period here (with interface slider)
;; Conveyor section failure
if ticks > conv-failure and Conveyor-Section-Failure = 1 and Conveyor-Failures = TRUE [ set Conveyor-Section-
Failure 4 ]
tick
end

to create-conveyor [ file-name ]
;; This procedure is used to setup conveyor sections by reading the input file "file-name". The format
;; of the input is described in the Info tab and is listed below. This procedure is called on startup
;; and can also be called to add additional sections at run time.
let x-in 0      ;; entrance xcor
let y-in 0      ;; entrance ycor
let c-length 0  ;; length of conveyor
let c-number 0  ;; conveyor number
let pathway 0   ;; direction (0 up, 90 right, 180 down, 270 left)
let conveyor-type 0 ;; 1 - basic, 2 - with diverter, 3 - with storage bin
set diverter-states [] ;; nested list of diverter states [[<L> <C> <R>][<L> <C> <R>]...]
let diverter-temp [] ;; temporary list of diverter states for one conveyor section
set no-conveyors 0
let i 0

```

```

file-open file-name
while [ not file-at-end? ]
[
  set x-in file-read
  set y-in file-read
  set c-length file-read
  set c-number file-read
  set pathway file-read
  set conveyor-type file-read
  set diverter-temp lput file-read diverter-temp      ;; Left entry
  set diverter-temp lput file-read diverter-temp      ;; Centre entry
  set diverter-temp lput file-read diverter-temp      ;; Right entry
  set diverter-states lput diverter-temp diverter-states ;; nested diverter states entry
  set diverter-temp []                                ;; reset the temp variable
  set i 0
  set no-conveyors no-conveyors + 1
  while [ i < c-length]
  [
    ask patch x-in y-in
    [
      (ifelse
        i = 0 and conveyor-type < 3      ;; input sensor (conveyor belt)
        [
          set pcolor 67
          set in-sensor? TRUE
          set plabel c-number
          set plabel-color black
        ]
        i = 0 and conveyor-type = 3      ;; input sensor (storage bin)
        [
          set pcolor 67
          set in-sensor? TRUE
          set storage-bin? TRUE
          set inbound []
          set plabel c-number
          set plabel-color black
        ]
      )
      i = c-length - 2 and conveyor-type = 2 ;; output sensor
      [ set pcolor 67 set out-sensor? TRUE ]
      i = c-length - 1 and conveyor-type = 2 ;; diverter
      [ set pcolor 45 set diverter? TRUE ]
      i = c-length - 1 and conveyor-type = 3 ;; storage bin
      [ set pcolor 15
        set storage-bin? TRUE
        set plabel c-number - 11
        set plabel-color black
      ]
      i = c-length - 1 and conveyor-type = 1 ;; output sensor
      [ set pcolor 67 set out-sensor? TRUE ]
      [ set pcolor 9 ]                ;; conveyor belt
    )
    set conveyor-number c-number
    set direction pathway
    set operating? TRUE
  ]
  (ifelse

```

```

    pathway = 0 [ set y-in y-in + 1 ]
    pathway = 90 [ set x-in x-in + 1 ]
    pathway = 180 [ set y-in y-in - 1 ]
    pathway = 270 [ set x-in x-in - 1 ]
  )
  set i i + 1
]
;; Diverter
if conveyor-type = 2
[
  create-diverters 1
  [
    (ifelse
      pathway = 0 [ setxy x-in (y-in - 1) ]
      pathway = 90 [ setxy (x-in - 1) y-in ]
      pathway = 180 [ setxy x-in (y-in + 1) ]
      pathway = 270 [ setxy (x-in + 1) y-in ]
    )
    set shape "arrow"
    set color black
    set diverter-number c-number
    set heading direction
  ]
]
set i 1
let div-no 1
let no-div 0
while [ i <= no-conveyors ]
[
  ;; Number the diverters
  ask patches with [ conveyor-number = i and diverter? = TRUE ]
  [
    set div-number div-no
    ask neighbors4 with [ out-sensor? = TRUE ] [ set div-number div-no ]
    set no-div div-no
    set div-no div-no + 1
  ]
  set i i + 1
]
file-close
end

```

to new-part [ no-types ]

;; This procedure is used to introduce new parts into the system. Parts are placed at the input sensor location of conveyor 1. Currently, three part types are possible ("a", "b", or "c") - the part types are selected randomly.

```

let x-parts 0
let y-parts 0
let p-type random no-types
let part-here? FALSE
ask patches with [ in-sensor? = TRUE and conveyor-number = 1 ]
[
  set x-parts pxcor
  set y-parts pycor
  if count parts-here > 0 [ set part-here? TRUE ]
]

```

```

if part-here? = FALSE
[

create-parts 1
[
  set xcor x-parts
  set ycor y-parts
  set shape "box"
  set time-in ticks
  (ifelse
    p-type = 0 [set color grey set part-type "a"]
    p-type = 1 [set color orange set part-type "b"]
    p-type = 2 [set color brown set part-type "c"]
    p-type = 3 [set color turquoise set part-type "d"]
    p-type = 4 [set color blue set part-type "e"]
    p-type = 5 [set color magenta set part-type "f"]
  )
]
]
end

to setup-lagents
;; This procedure is used to setup the self-management service agents
create-SMSAgents 1
[
  set incoming-queue []
  set proc-delay 0
  set shape "fb"
  set size 4
  set xcor 0
  set ycor -12
  set color brown
  set label "Agent-SMS"
]
create-SCAgents 1
[
  set incoming-queue []
  set proc-delay 0
  set shape "fb"
  set size 4
  set xcor -12
  set ycor -6
  set color cyan
  set label "Agent-SC"
]
create-SOAgents 1
[
  set incoming-queue []
  set proc-delay 0
  set shape "fb"
  set size 4
  set xcor -12
  set ycor -18
  set color cyan
  set label "Agent-SO"
]

```

```

create-SHAgents 1
[
  set incoming-queue []
  set proc-delay 0
  set shape "fb"
  set size 4
  set xcor 12
  set ycor -6
  set color cyan
  set label "Agent-SH"
]
create-SPAgents 1
[
  set incoming-queue []
  set proc-delay 0
  set shape "fb"
  set size 4
  set xcor 12
  set ycor -18
  set color cyan
  set label "Agent-SP"
]
end

to setup-hagents
;;to create multiple agents
;;turtle 5 Agent_Monitoring
create-turtles 1
[
  set shape "person"
  set color red
  set size 4
  setxy -12 32
  set label "Agent_Monitoring"
  set label-color white
]
;;turtle 6 Agent_Analysis
create-turtles 1
[
  set shape "person"
  set color orange
  set size 4
  setxy 0 32
  set label "Agent_Analysis"
  set label-color white
]
;;turtle 7 Agent_SelfLearning
create-turtles 1
[
  set shape "person"
  set color yellow
  set size 4
  setxy 16 34
  set label "Agent_SelfLearning"
  set label-color white
]

```



```

;;turtle 8 Agent_Planning
create-turtles 1
[
  set shape "person"
  set color blue
  set size 4
  setxy 0 18
  set label "Agent_Planning"
  set label-color white
]
;;turtle 9 Agent_Execution
create-turtles 1
[
  set shape "person"
  set color violet
  set size 4
  setxy -12 18
  set label "Agent_Execution"
  set label-color white
]
;;turtle 10 Agent_Knowledge
create-turtles 1
[
  set shape "person"
  set color green
  set size 4
  setxy 12 25
  set label "Agent_Knowledge"
  set label-color white
]
end

```

to setup-hconnections

```

;;to create workflow links
ask turtle 5 [ create-link-to turtle 6 ]
ask turtle 6 [ create-link-to turtle 7 ]
ask turtle 7 [ create-link-to turtle 6 ]
ask turtle 6 [ create-link-to turtle 8 ]
ask turtle 8 [ create-link-to turtle 6 ]
ask turtle 6 [ create-link-to turtle 9 ]
ask turtle 9 [ create-link-to turtle 6 ]
ask turtle 8 [ create-link-to turtle 9 ]

```

;;to create database links

```

ask turtle 10 [ create-link-to turtle 5 ]
ask turtle 10 [ create-link-to turtle 6 ]
ask turtle 10 [ create-link-to turtle 7 ]
ask turtle 10 [ create-link-to turtle 8 ]
ask turtle 10 [ create-link-to turtle 9 ]
ask turtle 10 [ create-link-from turtle 5 ]
ask turtle 10 [ create-link-from turtle 6 ]
ask turtle 10 [ create-link-from turtle 7 ]
ask turtle 10 [ create-link-from turtle 8 ]
ask turtle 10 [ create-link-from turtle 9 ]

```

end

to diverter-control

```
;; This procedure checks actuates the exit diverter based on the part type.
;; First, a check is performed to see if there is an exact match (indicating that the
;; direction corresponds to the storage bin location). If a direct match is not found,
;; a check is performed to see if the part type is included in one of the remaining paths.
;; Note: This code will need to be modified so that the part will continue along the
;; conveyor if no matches are found.
```

```
let diverted? FALSE
```

```
let next-diverter 0
```

```
ask patch-ahead 1 [set next-diverter conveyor-number]
```

```
(ifelse
```

```
  ;; Exact matches (i.e., arrived at storage bin)
```

```
  part-type = item 0 connections and diverted? = FALSE
```

```
  [ actuate-diverter next-diverter ((heading - 90) mod 360) set diverted? TRUE ]
```

```
  part-type = item 1 connections and diverted? = FALSE
```

```
  [ actuate-diverter next-diverter (heading) set diverted? TRUE ]
```

```
  part-type = item 2 connections and diverted? = FALSE
```

```
  [ actuate-diverter next-diverter ((heading + 90) mod 360) set diverted? TRUE ]
```

```
  ;; Part type included in one of the remaining paths
```

```
  ;member? part-type item 0 connections and diverted? = FALSE
```

```
  part-type = first item 0 connections and diverted? = FALSE
```

```
  [ actuate-diverter next-diverter ((heading - 90) mod 360) set diverted? TRUE ]
```

```
  ;member? part-type item 1 connections and diverted? = FALSE
```

```
  part-type = first item 1 connections and diverted? = FALSE
```

```
  [ actuate-diverter next-diverter (heading) set diverted? TRUE ]
```

```
  ;member? part-type item 2 connections and diverted? = FALSE
```

```
  part-type = first item 2 connections and diverted? = FALSE
```

```
  [ actuate-diverter next-diverter ((heading + 90) mod 360) set diverted? TRUE ]
```

```
  ;; Part type is not included (follow the path with the most options)
```

```
  [
```

```
    let part-path find-longest connections
```

```
    (ifelse
```

```
      part-path = 0 [ actuate-diverter next-diverter ((heading - 90) mod 360) ]
```

```
      part-path = 1 [ actuate-diverter next-diverter (heading) ]
```

```
      part-path = 2 [ actuate-diverter next-diverter ((heading + 90) mod 360) ]
```

```
    )
```

```
  ]
```

```
)
```

end

to actuate-diverter [ d-number d-position ]

```
;; This procedure actuates the direction of the diverter <d-number> to position <d-position>. The direction of the diverter
```

```
;; patch is the key parameter for control here. However, a diverter turtle is also used to show the diverter direction.
```

```
ask patches with [ diverter? = TRUE and conveyor-number = d-number ] [ set direction d-position ]
```

```
ask diverters with [ diverter-number = d-number ] [ set heading direction ]
```

end

to store-part

```
;; This procedure is used to collect parts in the storage bin
```

```
;; Currently, parts are just removed; however, eventually statistics can be collected by this procedure.
```

```
ifelse part-at-bin = 0
```

```
  [ set part-at-bin part-type ]
```

```
  [
```

```
    if part-type != part-at-bin [ show (word "B(" conveyor-number ")": " part-at-bin " exited: " part-type) ]
```

```

]
set wait-time ticks - time-in
calculate-wait-time wait-time
die
end

```

```

to-report find-longest [ connect-list ]
;; This procedure is used to check the "connections" list for the path with the most options.
let i 0
let longest-path 0
let longest-index 0
while [ i < 3 ]
[
  if length item i connect-list > longest-path
  [
    set longest-index i
    set longest-path length item i connect-list
  ]
  set i i + 1
]
report longest-index
end

```

```

to send-message [ sender receiver performative content ]
;; This procedure is used to send agent-to-agent FIPA messages.
;; The sender and receiver variables are the "who" for the self-management agents:
;; 0 = Agent-SMS
;; 1 = Agent-SC
;; 2 = Agent-SO
;; 3 = Agent-SH
;; 4 = Agent-SP
ask turtle sender
[
  let somemsg create-message performative
  set somemsg add-receiver receiver somemsg
  set somemsg add-content content somemsg
  if show_messages [ show (word ticks ": " somemsg) ]
  send somemsg
]
end

```

```

to manage-messages
;; This procedure is used by the self-management to manage incoming messages.
ask turtle 0 ;; Agent-SMS
[
  (ifelse
    length incoming-queue > 0 and member? "completed" get-content get-message-no-remove
    [
      send-message who 1 "inform" "acknowledge"
      remove-msg
    ]
    length incoming-queue > 0 and member? "inform" get-performative get-message-no-remove
    [ remove-msg ]
  )
]
ask turtle 1 ;; Agent-SC

```

```

[
  (ifelse
    length incoming-queue > 0 and member? "new part type" get-content get-message-no-remove and proc-delay =
0
    ;length incoming-queue > 0 and member? "new part type" get-content get-message-no-remove
    [
      processing-delay TRUE
      send-message who 0 "inform" "processing request"
      ask my-links [die]
      create-link-to turtle 0
      ask my-links
      [
        set color green
        set label "inform: processing request"
      ]
    ]
    ;length incoming-queue > 0 and member? "new part type" get-content get-message-no-remove and timer > proc-
delay
    length incoming-queue > 0 and member? "new part type" get-content get-message-no-remove and bernoulli
Agent-Delay
    [ add-storage-bin ]
    ;length incoming-queue > 0 and member? "re-route parts" get-content get-message-no-remove and timer > proc-
delay
    length incoming-queue > 0 and member? "re-route parts" get-content get-message-no-remove and bernoulli
Agent-Delay
    [ add-storage-bin ]
    length incoming-queue > 0 and member? "inform" get-performative get-message-no-remove
    [ remove-msg ]
  )
]
ask turtle 2 ;; Agent-SO
[
  if length incoming-queue > 0
  [
    ;show (word "Agent-SO received message from " identify-sender get-sender get-message-no-remove)
    ;show (word get-performative get-message-no-remove ": " get-content get-message)
  ]
]
ask turtle 3 ;; Agent-SH
[
  if length incoming-queue > 0
  [
    ;show (word "Agent-SH received message from " identify-sender get-sender get-message-no-remove)
    ;show (word get-performative get-message-no-remove ": " get-content get-message)
  ]
]
ask turtle 4 ;; Agent-SP
[
  (ifelse
    length incoming-queue > 0 and member? "failure" get-content get-message-no-remove and proc-delay = 0
    [
      processing-delay TRUE
      send-message who 0 "inform" "processing request"
      ask my-links [die]
      create-link-to turtle 0
      ask my-links
    ]
  )
]

```

```

[
  set color green
  set label "inform: processing request"
]
]
;length incoming-queue > 0 and member? "failure" get-content get-message-no-remove and timer > proc-delay
length incoming-queue > 0 and member? "failure" get-content get-message-no-remove and ticks > proc-delay
[ request-re-route ]
length incoming-queue > 0 and member? "inform" get-performative get-message-no-remove
[ remove-msg ]
)
]
end

```

```

to-report identify-sender [ sender ]
  (ifelse
    sender = "0" [ report "Agent-SMS" ]
    sender = "1" [ report "Agent-SC" ]
    sender = "2" [ report "Agent-SO" ]
    sender = "3" [ report "Agent-SH" ]
    sender = "4" [ report "Agent-SP" ]
  )
end

```

```

to conveyor-failure [ c-number ]
  ;; This procedure is used to change the state of a conveyor section to "failed":
  ;; c-number: conveyor section number
  ;show (word "Diverter States (pre): " diverter-states)
  ask patches with [ conveyor-number = c-number ]
  [
    ;update-diverter-state c-number direction
    set operating? FALSE
    set pcolor pcolor - 2
  ]
  ask patches with [ conveyor-number = c-number and out-sensor? = TRUE]
  [
    update-diverter-state c-number direction
  ]
  ;show (word "Diverter States (post): " diverter-states)
end

```

```

to processing-delay [ start? ]
  ;; This procedure is used to simulate processing delay
  ifelse start? = TRUE
  [
    ; set proc-delay timer + random-float ProcessingDelay
    ; if show_messages = TRUE [ show (word "Processing Delay = " (proc-delay - timer) " seconds") ]
    set proc-delay ticks + random ProcessingDelay
    if show_messages = TRUE [ show (word "Processing Delay = " (proc-delay - ticks) " ticks") ]
  ]
  [ set proc-delay 0 ]
end

```

```

to-report downstream-conveyor [ orientation ]
  ;; This procedure is used to identify the conveyor section downstream from the calling
  ;; conveyor section in diverter direction "orientation". This procedure is called

```

```

;; from the diverter patch of the conveyor section.
;; **** this procedure has an error: if there are two conveyor sections with the same direction
;; (e.g., at a T intersection), it cannot determine which is the downstream. To correct this
;; it is assumed that the true downstream conveyor section has a higher conveyor number.
let my-number conveyor-number
let downstream 0
let last-downstream 0
ask neighbors4
[
  if conveyor-number > 0 ;; blank patches have conveyor-number = 0
  [
    if direction = orientation and conveyor-number != my-number
    [
      if conveyor-number > last-downstream [ set downstream conveyor-number ]
    ]
  ]
]
report downstream
end

to update-diverter-state [ failed orientation ]
;; This procedure is used to update the diverter-states list when a conveyor section failure occurs.
;; - failed section: diverter set to [0 0 0]
;; - upstream section: diverter direction leading to failed conveyor section set to 0
;; - downstream section: diverter set to [0 0 0]
let new-state []
let difference 0
;; 1. set failed conveyor section's diverter to [0 0 0]
set diverter-states remove-item ( failed - 1 ) diverter-states
set diverter-states insert-item ( failed - 1 ) diverter-states [0 0 0]
;; 2. find upstream conveyor section and set its diverter to block entrance to the failed section
ask patches with [ diverter? = TRUE ]
[
  if downstream-conveyor orientation = failed
  [
    ;show conveyor-number
    ;show direction
    set new-state item ( conveyor-number - 1 ) diverter-states
    ;show new-state
    let my-number conveyor-number
    let my-direction 0
    ask patches with [conveyor-number = my-number and out-sensor? = TRUE] [set my-direction direction]
    set difference orientation - my-direction
    ;show difference
    (ifelse
      difference = -90 or difference = 270
      [
        ;show "L"
        set new-state remove-item 0 new-state
        set new-state insert-item 0 new-state 0
        set diverter-states remove-item ( conveyor-number - 1 ) diverter-states
        set diverter-states insert-item ( conveyor-number - 1 ) diverter-states new-state
        ;show new-state
      ]
      difference = 0
    )
  ]
]

```

```

;show "C"
set new-state remove-item 1 new-state
set new-state insert-item 1 new-state 0
set diverter-states remove-item ( conveyor-number - 1 ) diverter-states
set diverter-states insert-item ( conveyor-number - 1 ) diverter-states new-state
;show new-state
]
difference = 90 or difference = -270
[
;show "R"
set new-state remove-item 2 new-state
set new-state insert-item 2 new-state 0
set diverter-states remove-item ( conveyor-number - 1 ) diverter-states
set diverter-states insert-item ( conveyor-number - 1 ) diverter-states new-state
;show new-state
]
)
]
]
;; 3. set downstream conveyor section's diverter to [0 0 0]
;; - for now, just go with the next index (ideally the downstream-conveyor procedure should be used)
set diverter-states remove-item failed diverter-states
set diverter-states insert-item failed diverter-states [0 0 0]

end

to config-diverters [ no-part-types ]
;; This procedure is used to setup the diverter controllers.
;; - The output sensor, out-sensor?, of the conveyor preceding the diverter checks the part type
;; - The diverter is setup based on its outputs (i.e., do not enter, storage bin, next conveyor)
;; This procedure takes the number of part types as input and assigns a unique "a-z" letter to each type.
;; The possible diverter states are defined by the diverter-states nested list:
;; [[<L><C><R>][<L><C><R>] ... [<L><C><R>]]
;; where L=left, C=centre, R=right for each diverter and 0 = do not enter, 1 = part bin, and 2 = next conveyor
;; This procedure sets each diverter's controller by assigning parts to the <L>, <C>, and <R> slots of the diverter's
;; connections variable: i.e., "-" do not enter, "<single part type ... e.g., 'a'>" for storage bin, <all-types> for next
conveyor
let all-types "abcdefghijklmnopqrstuvwxyz"
set part-types substring all-types 0 no-part-types
let assigned 0
let i 1
;; Reset all of the diverter controls (i.e., the connections variables)
ask patches with [ out-sensor? = TRUE ] [ set connections ["?" "?" "?" ] ]
;; 1. Read diverter-states "0" values and set corresponding connections to "-"
;ask patches with [ out-sensor? = TRUE ]
while [ i <= length diverter-states ]
[
ask patches with [ out-sensor? = TRUE and conveyor-number = i ]
[
if item 0 item ( conveyor-number - 1 ) diverter-states = 0
[
set connections remove-item 0 connections
set connections insert-item 0 connections "-"
]
]
if item 1 item ( conveyor-number - 1 ) diverter-states = 0
[

```

```

    set connections remove-item 1 connections
    set connections insert-item 1 connections "-"
  ]
  if item 2 item ( conveyor-number - 1) diverter-states = 0
  [
    set connections remove-item 2 connections
    set connections insert-item 2 connections "-"
  ]
  ;show (word "Diverter " conveyor-number ": " connections)
]
set i i + 1
]
;; 2. Read diverter-states "1" values and set corresponding connections to each part type
;ask patches with [ out-sensor? = TRUE ]
(ifelse
  Optimize-Path = FALSE
  [
    set i 1
    while [ i <= length diverter-states ]
    [
      ask patches with [ out-sensor? = TRUE and conveyor-number = i ]
      [
        (ifelse
          item 0 item ( conveyor-number - 1) diverter-states = 1 and assigned < no-part-types
          [
            set connections remove-item 0 connections
            set connections insert-item 0 connections (item assigned part-types)
            set assigned assigned + 1
          ]
          item 0 item ( conveyor-number - 1) diverter-states = 1 and assigned = no-part-types
          [
            set connections remove-item 0 connections
            set connections insert-item 0 connections "-"
          ]
        )
        (ifelse
          item 1 item ( conveyor-number - 1) diverter-states = 1 and assigned < no-part-types
          [
            set connections remove-item 1 connections
            set connections insert-item 1 connections (item assigned part-types)
            set assigned assigned + 1
          ]
          item 1 item ( conveyor-number - 1) diverter-states = 1 and assigned = no-part-types
          [
            set connections remove-item 1 connections
            set connections insert-item 1 connections "-"
          ]
        )
        (ifelse
          item 2 item ( conveyor-number - 1) diverter-states = 1 and assigned < no-part-types
          [
            set connections remove-item 2 connections
            set connections insert-item 2 connections (item assigned part-types)
            set assigned assigned + 1
          ]
          item 2 item ( conveyor-number - 1) diverter-states = 1 and assigned = no-part-types

```



```

[
  set connections remove-item 2 connections
  set connections insert-item 2 connections "-"
]
)
;show (word "Diverter " conveyor-number ": " connections)
]
set i i + 1
]
]
;; **** is there a way of assigning to more than one out-sensor? (without exceeding assigned)
Optimize-Path = TRUE
[
  set i 1
  while [ i <= length diverter-order ]
  [
    ask patches with [ out-sensor? = TRUE and div-number = i ]
    [
      if item 0 item ( conveyor-number - 1) diverter-states = 1 and assigned < no-part-types
      [
        set connections remove-item 0 connections
        set connections insert-item 0 connections (item assigned part-types)
        ;set assigned assigned + 1
      ]
      if item 1 item ( conveyor-number - 1) diverter-states = 1 and assigned < no-part-types
      [
        set connections remove-item 1 connections
        set connections insert-item 1 connections (item assigned part-types)
        ;set assigned assigned + 1
      ]
      if item 2 item ( conveyor-number - 1) diverter-states = 1 and assigned < no-part-types
      [
        set connections remove-item 2 connections
        set connections insert-item 2 connections (item assigned part-types)
        ;set assigned assigned + 1
      ]
    ]
    set assigned assigned + 1
    set i i + 1
  ]
  set i 1
  while [ i <= length diverter-order ]
  [
    ask patches with [ out-sensor? = TRUE and div-number = i ]
    [
      if item 0 item ( conveyor-number - 1) diverter-states = 1 and item 0 connections = "?"
      [
        set connections remove-item 0 connections
        set connections insert-item 0 connections "-"
      ]
      if item 1 item ( conveyor-number - 1) diverter-states = 1 and item 1 connections = "?"
      [
        set connections remove-item 1 connections
        set connections insert-item 1 connections "-"
      ]
      if item 2 item ( conveyor-number - 1) diverter-states = 1 and item 2 connections = "?"

```

```

[
  set connections remove-item 2 connections
  set connections insert-item 2 connections "-"
]
]
set i i + 1
]
]
)
;; 3. Read diverter-states "2" values
;; **** possible update - if more than one "2", look to see what is down each and prioritize: e.g.,
;; - "a" on left, "b" on centre: "abc" left, "bac" centre
;; **** this section needs to be updated ****
;; - check downstream conveyor section output sensor
;; - e.g., ask patches with [ conveyor-number = 2 and diverter? = TRUE ] [ show downstream-conveyor 90 ]
;; - is there a "2" (pass through) variable?
;; - YES: set to all remaining part-types
;; - NO: set to all "1" part types of the downstream section
set i 1
let c-number 0
let c-direction 0
let index 0
let downstream []
let through-path? TRUE
let stored-parts ""
while [ i <= length diverter-states ]
[
  ask patches with [ out-sensor? = TRUE and conveyor-number = i ]
  [
    set c-number conveyor-number
    set c-direction direction
    if item 0 item ( conveyor-number - 1 ) diverter-states = 2
    [
      ;; This code is used to get the downstream diverter states list
      ask patches with [ conveyor-number = i and diverter? = TRUE ]
      [
        set index downstream-conveyor (diverter-direction c-direction 0)
        ask patches with [ conveyor-number = index and out-sensor? = TRUE ]
        [
          set downstream connections
          set stored-parts diverted-parts
        ]
      ]
      ;show (word "D " c-number " Left " index " " item (index - 1) diverter-states " " downstream " " stored-parts)
      ifelse member? 2 item (index - 1) diverter-states
      [set through-path? TRUE]
      [set through-path? FALSE]
    ]
  ]
  (ifelse
    through-path?
    [
      set connections remove-item 0 connections
      set connections insert-item 0 connections (remove diverted-parts part-types)
    ]
    [
      set connections remove-item 0 connections
      set connections insert-item 0 connections stored-parts
    ]
  )
]
]

```

```

    ]
  )
]
if item 1 item ( conveyor-number - 1) diverter-states = 2
[
  ask patches with [ conveyor-number = i and diverter? = TRUE ]
  [
    set index downstream-conveyor (diverter-direction c-direction 1)
    ask patches with [ conveyor-number = index and out-sensor? = TRUE ]
    [
      set downstream connections
      set stored-parts diverted-parts
    ]
  ]
  ;show (word "D " c-number " Centre " index " " item (index - 1) diverter-states " " downstream " " stored-
parts)
  ifelse member? 2 item (index - 1) diverter-states
  [set through-path? TRUE]
  [set through-path? FALSE]
]
(ifelse
  through-path?
  [
    set connections remove-item 1 connections
    set connections insert-item 1 connections (remove diverted-parts part-types)
  ]
  [
    set connections remove-item 1 connections
    set connections insert-item 1 connections stored-parts
  ]
)
]
if item 2 item ( conveyor-number - 1) diverter-states = 2
[
  ask patches with [ conveyor-number = i and diverter? = TRUE ]
  [
    set index downstream-conveyor (diverter-direction c-direction 2)
    ask patches with [ conveyor-number = index and out-sensor? = TRUE ]
    [
      set downstream connections
      set stored-parts diverted-parts
    ]
  ]
  ;show (word "D " c-number " Right " index " " item (index - 1) diverter-states " " downstream " " stored-parts)
  ifelse member? 2 item (index - 1) diverter-states
  [set through-path? TRUE]
  [set through-path? FALSE]
]
(ifelse
  through-path?
  [
    set connections remove-item 2 connections
    set connections insert-item 2 connections (remove diverted-parts part-types)
  ]
  [
    set connections remove-item 2 connections
    set connections insert-item 2 connections stored-parts
  ]
)
]

```

```

    )
  ]
  ;show (word "Diverter " conveyor-number ": " connections)
]
set i i + 1
]
end

```

```

to-report bernoulli [probability]
;; This procedure enables the Bernoulli distribution: i.e., a discrete probability distribution
;; with two outcomes (heads/tails, success/failure, true/false).
report ifelse-value (random-float 1 < probability) [true] [false]
end ;; end of bernoulli

```

```

to-report diverter-direction [ c-direction d-position ]
;; This procedure is used to return the "Left", "Centre", or "Right" diverter direction based on
;; the conveyor direction (c-direction) and the diverter position (d-position)
(ifelse
  d-position = 0 [ set c-direction c-direction - 90 ] ;; Left
  d-position = 2 [ set c-direction c-direction + 90 ] ;; Right
)
if c-direction = -90 [ set c-direction 270 ]
if c-direction = 360 [ set c-direction 0 ]
report c-direction
end

```

```

to-report diverted-parts
;; This procedure returns the list of part types that are diverted.
;; It is a patch context procedure that is called at a conveyor's output sensor.
;show (word "D: " conveyor-number " States: " item (conveyor-number - 1) diverter-states " Connections: "
connections)
let i 0
let diverted ""
while [ i < 3 ]
[
  if item i item (conveyor-number - 1) diverter-states = 1
  [
    set diverted word diverted item i connections
  ]
  set i i + 1
]
report diverted
end

```

```

to identify-upstream
;; This procedure is used to identify each diverter's upstream diverters
;; It appears to work in most cases. There are still some errors when a conveyor section fails.
let i 1
let no-div count patches with [ div-number > 0 ]
let conv-no 0
let temp-inbound []
set i 1
while [ i <= no-div ]
[
  ask patches with [ div-number = i ]
  [

```

```

set temp-inbound []
ask neighbors4 with [ out-sensor? = TRUE ]
[
  ;show conveyor-number
  set conv-no conveyor-number
  ask patches with [ conveyor-number = conv-no and in-sensor? = TRUE and operating? = TRUE ]
  [
    ask neighbors4 with [ diverter? = TRUE and operating? = TRUE]
    [
      set temp-inbound lput div-number temp-inbound
    ]
  ]
]
set inbound temp-inbound
show (word "Diverter " i ": " inbound)
]
set i i + 1
]
end

```

```

to-report through-list [ c-list ]
;; This procedure determines the part list that can pass through the "through parts" position of a diverter.
;; **** sort in order of downstream?
let i 0
let t-list part-types
let temp ""
while [ i < length part-types ]
[
  if member? item i part-types c-list
  [
    set temp item i part-types
    set t-list remove temp t-list
    set t-list (word temp t-list)
  ]
  set i i + 1
]
report t-list
end

```

```

to diverter-config
;; This is a new diverter configuration procedure. The procedure cycles through each of the diverters to setup their
;; connections list (that is used by the diverter-control procedure to control the diverter position when a part
arrives).
;; The diverter-states list is used to setup each of the diverter connections lists: (1) "no entry" positions, (2) "storage
;; bin" positions, and (3) "through parts" positions.
let all-types "abcdefghijklmnopqrstuvwxyz"
set part-types substring all-types 0 Number-Part-Types
let assigned 0
let i 1
let j 0
let assigned? FALSE
;; Reset all of the diverter controls (i.e., the connections variables)
ask patches with [ out-sensor? = TRUE ] [ set connections ["?" "?" "?"] ]
;; 1. "No Entry" diverter positions
while [ i <= length diverter-order ]
[

```

```

ask patches with [ out-sensor? = TRUE and div-number = i ]
[
  if item 0 item ( conveyor-number - 1 ) diverter-states = 0
  [
    set connections remove-item 0 connections
    set connections insert-item 0 connections "-"
  ]
  if item 1 item ( conveyor-number - 1 ) diverter-states = 0
  [
    set connections remove-item 1 connections
    set connections insert-item 1 connections "-"
  ]
  if item 2 item ( conveyor-number - 1 ) diverter-states = 0
  [
    set connections remove-item 2 connections
    set connections insert-item 2 connections "-"
  ]
]
set i i + 1
]
;; 2. "Storage Bin" diverter positions
let no-divs 0
let divs-assigned 0
while [ j < length part-types ]
[
  set i 1
  set assigned? FALSE
  ;show (word "Part: " item j part-types)
  while [ i <= length diverter-order ]
  [
    set no-divs count patches with [ out-sensor? = TRUE and div-number = item ( i - 1 ) diverter-order ]
    set divs-assigned 0
    ask patches with [ out-sensor? = TRUE and div-number = item ( i - 1 ) diverter-order ]
    [
      if item 0 item ( conveyor-number - 1 ) diverter-states = 1 and item 0 connections = "?" and assigned? = FALSE
      [
        set connections remove-item 0 connections
        set connections insert-item 0 connections item j part-types
        set divs-assigned divs-assigned + 1
        if divs-assigned = no-divs [ set assigned? TRUE ]
      ]
      if item 1 item ( conveyor-number - 1 ) diverter-states = 1 and item 1 connections = "?" and assigned? = FALSE
      [
        set connections remove-item 1 connections
        set connections insert-item 1 connections item j part-types
        set divs-assigned divs-assigned + 1
        if divs-assigned = no-divs [ set assigned? TRUE ]
      ]
      if item 2 item ( conveyor-number - 1 ) diverter-states = 1 and item 2 connections = "?" and assigned? = FALSE
      [
        set connections remove-item 2 connections
        set connections insert-item 2 connections item j part-types
        set divs-assigned divs-assigned + 1
        if divs-assigned = no-divs [ set assigned? TRUE ]
      ]
    ]
    ;show (word "D(" div-number ")": " connections)
  ]
]

```

```

]
  set i i + 1
]
set j j + 1
]
;; 3. "Through Parts" diverter positions
let downstream 0
let conv-direction 0
let t-list ""
;set i 1
;while [ i <= length diverter-order ]
set i length diverter-order
while [ i >= 0 ]
[
  ask patches with [ out-sensor? = TRUE and div-number = i ]
  [
    if item 0 item ( conveyor-number - 1) diverter-states = 2 or ( item 0 item ( conveyor-number - 1) diverter-states
= 1 and item 0 connections = "?" )
    [
      set conv-direction diverter-direction direction 0
      ask patches with [ diverter? = TRUE and div-number = i ]
      [
        set downstream downstream-conveyor conv-direction
        ask patches with [ out-sensor? = TRUE and conveyor-number = downstream ] [ set t-list through-list
connections ]
      ]
      set connections remove-item 0 connections
      set connections insert-item 0 connections t-list
      ;show (word "D(" div-number ")": " connections)
      ;show (word "Diverter: " div-number " Downstream: " downstream " Through List: " t-list)
    ]
    if item 1 item ( conveyor-number - 1) diverter-states = 2 or ( item 1 item ( conveyor-number - 1) diverter-states
= 1 and item 1 connections = "?" )
    [
      set conv-direction diverter-direction direction 1
      ask patches with [ diverter? = TRUE and div-number = i ]
      [
        set downstream downstream-conveyor conv-direction
        ask patches with [ out-sensor? = TRUE and conveyor-number = downstream ] [ set t-list through-list
connections ]
      ]
      set connections remove-item 1 connections
      set connections insert-item 1 connections t-list
      ;show (word "D(" div-number ")": " connections)
      ;show (word "Diverter: " div-number " Downstream: " downstream " Through List: " t-list)
    ]
    if item 2 item ( conveyor-number - 1) diverter-states = 2 or ( item 2 item ( conveyor-number - 1) diverter-states
= 1 and item 2 connections = "?" )
    [
      set conv-direction diverter-direction direction 2
      ask patches with [ diverter? = TRUE and div-number = i ]
      [
        set downstream downstream-conveyor conv-direction
        ask patches with [ out-sensor? = TRUE and conveyor-number = downstream ] [ set t-list through-list
connections ]
      ]
    ]
  ]
]

```

```

    set connections remove-item 2 connections
    set connections insert-item 2 connections t-list
    ;show (word "D(" div-number ")": " connections)
    ;show (word "Diverter: " div-number " Downstream: " downstream " Through List: " t-list)
  ]
  ;set connections clean-connections connections
  set connections clean-up connections
]
;set i i + 1
set i i - 1
]
end

to-report clean-connections [ c-list ]
;; This procedure is used by diverter-config to order the "through parts" list so that the downstream diverters
;; are prioritized.
let i 0
let j 0
let temp-first ""
let temp-item ""
while [ i < length c-list ]
[
  set temp-first first item i c-list
  ;show (word "(" i ")": temp-first)
  set j 0
  while [ j < length c-list ]
  [
    ;if j != i and temp-first != "-" and length item j c-list > 2
    if j != i and temp-first != "-"
    [
      set temp-item item j c-list
      set temp-item remove temp-first temp-item
      set c-list remove-item j c-list
      set c-list insert-item j c-list temp-item
    ]
    set j j + 1
  ]
  set i i + 1
]
report c-list
end

to-report clean-up [ c-list ]
let i 0
let j 0
let temp ""
let temp-item ""
while [ i < length c-list ]
[
  set temp item i c-list
  if length temp = 1 and member? temp part-types
  [
    ;show (word "single at (" i ")": temp)
    set j 0
    while [ j < 3 ]
    [

```



```

    if j != i and member? temp item j c-list
    [
        set temp-item item j c-list
        set temp-item remove temp temp-item
        set c-list remove-item j c-list
        set c-list insert-item j c-list temp-item
    ]
    set j j + 1
  ]
  set i i + 1
]
;; **** next sort in order of downstream? ****
report c-list
end

to calculate-wait-time [ time-in-system ]
  let sum-of-times ave-wait-time * parts-completed
  set sum-of-times sum-of-times + time-in-system
  set parts-completed parts-completed + 1
  set ave-wait-time sum-of-times / parts-completed
end

to calculate-parts-in-system
  let parts-in-system count parts
  let sum-of-parts 0
  if ticks > 0
  [
    set sum-of-parts ave-no-parts * ( ticks - 1 )
    set ave-no-parts ( sum-of-parts + parts-in-system ) / ticks
  ]
End

```

## ***Appendix A.2 NetLogo Routing Optimization Model***

```

;; this separate model for routing optimization by Agent_SO
;; a diverters.txt file below is read in running the model
;; 8
;; 1 0
;; 1 1
;; 1 2
;; 1 3
;; 2 2 4
;; 2 1 5
;; 1 6
;; 1 7

breed [ entrances entrance ]
breed [ diverters diverter ]

globals
[
  no-diverters
]

```

```

entrances-own
[
  to-entrance
]

diverters-own
[
  inbound
  to-entrance
]

to setup
  clear-all
  reset-ticks
  create-entrances 1
  [
    set color blue
    set shape "circle"
    set size 1
    setxy 0 (max-pycor - Spacing / 2)
    set to-entrance 0
  ]
  setup-diverters
end

to go
  ask diverters with [ length inbound > 0 ]
  [
    space-out
    set to-entrance distance entrance 0
    ;set label (word who "(" precision to-entrance 1 ")")
    set label who
    ;let new-heading get-heading
    let new-heading get-direction
    if (item 0 new-heading) > Spacing
    [
      set heading item 1 new-heading
      fd 1
    ]
  ]
  ;ask entrances [ set label closest-diverters ]
  ask patch 3 14 [ set plabel closest-diverters ]
  tick
end

to setup-diverters
  file-open "diverters.txt"
  let i 1 ;; diverter index
  let j 1 ;; inbound connection index
  let n 0 ;; number of inbound connections
  set no-diverters file-read
  create-diverters no-diverters
  [
    set color green
    set shape "triangle 2"
    set size 1.5
  ]

```

```

    set to-entrance distance entrance 0
    ;set label (word who "(" precision to-entrance 1 ")")
    set inbound []
    set to-entrance distance entrance 0
  ]
  ;layout-circle diverters 15
  let d-spacing max-pxcor * 2 / (no-diverters + 1)
  set i 1
  while [ i <= no-diverters ]
  [
    ask diverter i [ setxy (i * d-spacing + -1 * max-pxcor) (Spacing / 2 - max-pycor) ]
    set i i + 1
  ]
  set i 1
  while [ not file-at-end? ]
  [
    set n file-read
    set j 0
    while [ j < n ]
    [
      ask diverter i [ set inbound lput file-read inbound ]
      set j j + 1
    ]
    set i i + 1
  ]
  set i 1
  while [ i <= no-diverters ]
  [
    ask diverter i
    [
      set j 0
      while [ j < length inbound ]
      [
        create-link-to turtle (item j inbound)
        set j j + 1
      ]
    ]
    set i i + 1
  ]
  file-close
end

to space-out
  let i 0
  while [ i <= no-diverters ]
  [
    if i != who
    [
      if distance turtle i <= ( Spacing / 2 )
      [
        face turtle i
        rt 180
        fd 1
      ]
    ]
    set i i + 1
  ]

```

```

]
end

to-report get-heading
  let i 0
  let headings []
  let x-cor 0
  let y-cor 0
  while [ i < length inbound ]
  [
    ask turtle (item i inbound)
    [
      set x-cor x-cor + xcor
      set y-cor y-cor + ycor
    ]
    set i i + 1
  ]
  set x-cor x-cor / length inbound
  set y-cor y-cor / length inbound
  ;show (word "x_mean = " x-cor " y_mean = " y-cor)
  set headings lput ( distancexy x-cor y-cor ) headings
  set headings lput ( atan (x-cor - xcor) (y-cor - ycor) ) headings
  ;show (word "Heading: " headings)
  report headings
end

```

```

to-report get-direction
  let i 0
  let min-distance 1000
  let headings []
  let x-cor 0
  let y-cor 0
  while [ i < length inbound ]
  [
    ask turtle (item i inbound)
    [
      if to-entrance < min-distance
      [
        set min-distance to-entrance
        set x-cor xcor
        set y-cor ycor
      ]
    ]
    set i i + 1
  ]
  set headings lput ( distancexy x-cor y-cor ) headings
  set headings lput ( atan (x-cor - xcor) (y-cor - ycor) ) headings
  report headings
end

```

```

to-report closest-diverters
  let distances []
  let positions []
  let i 0
  let div-no 0
  ask diverters with [ length inbound > 0 ] [ set distances lput distance turtle 0 distances ]

```

```

set distances sort distances
while [ i < length distances ]
[
  ask diverters with [ distance turtle 0 = item i distances ]
  [
    set positions lput who positions
  ]
  set i i + 1
]
report positions
end

```

### Appendix A.3 Input for Agent-Based Simulation Model

The *Setup* button initialises the conveyor system by reading the “input.txt” file. Each line of the file specifies a conveyor section:

*<entrance xcor> <entrance ycor> <length> <number> <direction> ... <type> <left> <centre> <right>*

The *length* parameter (length of the conveyor section) is specified in number of patches. The *number* parameter assigns a unique conveyor section number to each conveyor section: this number should be an integer that is greater than 0 (all “blank” patches are assigned conveyor-number = 0 as a default). The *direction* parameter specifies the conveyor section direction: 0 = up; 90 = right; 180 = down; 270 = left.

The *left*, *centre*, and *right* parameters specify the options for the three possible directions of the conveyor section’s diverter (i.e., for conveyor sections with a diverter): 0 = “do not enter”, 1 = “storage bin exit”, 2 = “exit to another conveyor section”.

input.txt file:

```

-14 10 3 1 90 1 0 2 0
-10 10 7 2 90 2 1 2 2
-3 10 8 3 90 2 1 2 2
5 10 8 4 90 2 0 1 2
12 9 8 5 180 2 0 1 2
11 2 8 6 270 2 1 2 0
3 2 8 7 270 2 1 2 0
-5 2 7 8 270 2 0 1 2
-11 3 8 9 0 2 0 0 2
-4 9 7 10 180 1 0 1 2
4 9 7 11 180 1 0 1 2
-4 11 3 12 0 3 0 0 0
4 11 3 13 0 3 0 0 0
13 10 3 14 90 3 0 0 0
12 1 3 15 180 3 0 0 0
4 1 3 16 180 3 0 0 0
-4 1 3 17 180 3 0 0 0
-12 2 3 18 270 3 0 0 0

```

The user specifies the number of part types using the *Number-Part-Types* slider. The part types are identified by letters: e.g., three part types would result in an “a”, a “b”, and a “c” part type. When the simulation is initialised by the *Setup* button, the diverter parameters noted above are used in combination with the list of part types to configure the diverters. The diverters (specified with the “connections” list) use the same format; however, the left, centre, right fields are specified in terms of part types. For example, “-” signifies “do not enter”, and the part identifier specifies the path for the part type (e.g., “a” for part type “a”, “abc” for part types “a”, “b”, or “c”).

connections.txt file:

```
"abc"
"- " "abc" "- "
"a" "abc" "abc"
"b" "- " "abc"
"- " "c" "abc"
"- " "abc" "- "
"- " "- " "abc"
"- " "- " "abc"
"- " "- " "abc"
```

#### ***Appendix A.4 Procedures for Low-Level Self-Manageable Agents***

;; This file contains the procedures for the self-management agents

```
;;;;;;;;;;;;;
;;;;;;;;; Agent-SMS (Self-Manageable Service Execution Agent)
;;;;;;;;;;;;;
```

to monitor-part-types

;; This procedure is used to determine if any new part types are introduced to the conveyor system  
;; If a new part is introduced, a message is sent to Agent-SC. As well, the new part type is added  
;; to the list of part types "part-types".

```
let new-part? FALSE
let new-type ""
ask parts
[
  if not member? part-type part-types
  [
    set new-part? TRUE
    set new-type part-type
    set part-types word part-types part-type
  ]
]
if new-part?
[
  send-message who 1 "request" (word "new part type " new-type)
  ask my-links [die]
  create-link-to turtle 1
  ask my-links
  [
    set color yellow
    set label (word "request: new part type " new-type)
  ]
]
end
```

to monitor-conveyor-status [ c-number ]

;; This procedure is used to determine if a conveyor section has failed.  
;; It is assumed that conveyor-number = 1 is the entrance conveyor section, and as such,  
;; is not included as a possible failure mode (otherwise, there would be no possible diversion).

```
let failure? FALSE
if c-number > 1
[
```

```

ask patches with [ conveyor-number = c-number ]
[
  if operating?
  [
    conveyor-failure c-number
    ifelse Optimize-Path = TRUE
    [ set diverter-order [1 4 2 5 6 3 7 8] ] ;; just for diverter section 4 for now
    [ set diverter-order [1 2 3 4 5 6 7 8] ] ;; ordered list
    set failure? TRUE
  ]
]
]
if failure?
[
  send-message who 4 "request" (word "conveyor section " c-number " failure")
  ask my-links [die]
  create-link-to turtle 4
  ask my-links
  [
    set color yellow
    set label (word "request: conveyor section " c-number " failure")
  ]
]
end

```

```

.....
;;;;;; Agent-SC (Self-Configuration Agent)
.....

```

```

to add-storage-bin
;; This procedure is used to add a new storage bin when a new part is introduced.
remove-msg
processing-delay FALSE
;config-diverters Number-Part-Types
diverter-config
send-message who 0 "inform" "reconfiguration completed"
ask my-links [die]
create-link-to turtle 0
ask my-links
[
  set color green
  set label "inform: reconfiguration completed"
]
end

```

```

.....
;;;;;; Agent-SO (Self-Optimization Agent)
.....

```

```

;; This procedure is used to optimize routing when several system configurations available.
;; A separate simulation model is developed for routing optimization.

```

```

.....
;;;;;; Agent-SH (Self-Healing Agent)
.....

```

```

to divert-parts
  ;; This procedure is used to divert parts when a conveyor section has failed.
  ;; For now, a "contingency based approach is used.
  ;; Only a failure of conveyor section 4 is considered for now.
end

```

```

.....
;;;;; Agent-SP (Self-Protection Agent)
.....

```

```

to request-re-route
  ;; This procedure is used to request the SC agent to re-route parts when a conveyor section
  ;; failure occurs.
  remove-msg
  processing-delay FALSE
  send-message who 1 "request" "re-route parts"
  ask my-links [die]
  create-link-to turtle 1
  ask my-links
  [
    set color yellow
    set label (word "request: re-route-parts")
  ]
end

```

## ***Appendix A.5 Communication for Agent-Based Model***

```

;;; File to be included in NetLogo Mutliagent Models
;;; Communication for NetLogo Multiagent models
;;; Includes primitives for message creation and handling in NetLogo
;;; Adapted version for NetLogo 4 (2008) I. Sakellariou

```

```

.....
;;;;; COMMUNICATION
.....
;;; MESSAGE PROCESSING .....

```

```

.....
;; Sending messages
;; (One man's send is another man's receive..)
;; The second commented out line presents an alternative send implementation.
;; The commented out line represents an alternative method.
;; Problem: What if the agent I am sending the message is "killed"
;; Solution: Nothing Happens. Could yield an error message. Alternative: create a safe send.
to send [msg]
  let recipients get-receivers msg
  let rcv 0
  set no-messages no-messages + 1 ;add +1 to the total message count
  foreach recipients [
    ;set rcv turtle (read-from-string ?)
    i -> set rcv turtle (read-from-string i)
    if rcv != nobody [without-interruption [ask rcv [receive msg]]] ;; read-from-string is required to convert the
    string to number
  ]
  ;;interaction-plot-msg msg 0

```



```

end

.....
;; Message reception deals with updating incoming-queue
to receive [msg]
  if show_messages [show msg]
    set incoming-queue lput msg incoming-queue
  end

;; This reporter returns the next message in the list and removes it from the queue
to-report get-message
  if empty? incoming-queue [report "no_message"]
  let nextmsg first incoming-queue
  remove-msg
  report nextmsg
end

;; This reporter returns the next message in the list WITHOUT removing it from the queue
to-report get-message-no-remove
  if empty? incoming-queue [report "no_message"]
  report first incoming-queue
end

;; Explicit remove-msg
;; This is needed since reporters *cannot* change a variable's values (apparently).
to remove-msg
  set incoming-queue but-first incoming-queue
end

;; broadcasting to all agents of breed t-breed
to broadcast-to [t-breed msg]
  foreach [who] of t-breed [
    ;send add-receiver ? msg
    ;send [x -> add-receiver x msg]
    x -> send add-receiver x msg
  ]
end

;; Creating messages and adding the sender
to-report create-message [performative]
  report (list performative (word "sender:" who) )
end

to-report create-reply [performative msg]
  let msgOut 0

  set msgOut create-message performative
  set msgOut add-receiver (get-sender msg) msgOut
  report msgOut
end

;; Accessing information on messages
;; Reports the sender of a message
to-report get-sender [msg]
  ;report remove "sender:" first (filter [not is-number? ? and member? "sender:" ?] msg)
  report remove "sender:" first (filter [x -> not is-number? x and member? "sender:" x] msg)

```

```

;;report item ((position "sender:" msg) + 1) msg
end

;; Reports (returns) the content of a message
to-report get-content [msg]
  report item (position "content:" msg + 1) msg
end

;; Reports the list of receivers of a message
to-report get-receivers [msg]
  ;report map [remove "receiver:" ?] filter [not is-number? ? and member? "receiver:" ?] msg
  report map [? -> remove "receiver:" ?] filter [? -> not is-number? ? and member? "receiver:" ?] msg
end

;; reports the message performative
to-report get-performative [msg]
  report first msg
end

;;; Adding fields to a message
;; Adding a sender to a message
to-report add-sender [sender msg]
  report add msg "sender:" sender
end

;; add a receiver
to-report add-receiver [receiver msg]
  report add msg "receiver:" receiver
end

;; adding multiple recipients
to-report add-multiple-receivers [receivers msg]
  foreach receivers
  [
    ;set msg add-receiver ? msg
    set msg [x -> add-receiver x msg]
  ]
  report msg
end

;; Adding content to a message
to-report add-content [content msg]
  report add msg "content:" content
end

;; Primitive Add command
to-report add [msg field value]
  ifelse field = "content:"
  [report lput value lput field msg]
  [report lput (word field value) msg]
end

```

## Appendix B Experimental Testbed Setup

### *Appendix B.1 General Steps for Testbed Setup*

The general steps to set up the testbed are provided below:

*Step 1* to set up Raspberry Pi: a) flash a microSD card with the Raspbian operating system for Raspberry Pi; b) connect Raspberry Pi to a monitor with a keyboard and a mouse; c) plug Raspberry Pi to a power supply; d) connect Raspberry Pi to the Internet through wifi for remote access (SSH); e) install Python and related machine learning packages.

*Step 2* to set up SPADE: a) create instant messaging service accounts for agents' real-time communication through XMPP in which free XMPP/Jabber instant messaging service Jabber.de is used; b) program agents and agent behaviours.

*Step 3* to set up low-level devices: a) connect LEDs to Raspberry Pi; b) connect DC Motor to a motor driver and then Raspberry Pi; c) connect Stepper Motor to a motor driver and then Raspberry Pi; d) plug DC Motor and Stepper Motor to a power supply.

*Step 4* to set up high-level JetBot: a) set up hardware including Jetson Nano, body, motor, camera, power, wifi, display etc. modules; b) set up software including flashing JetBot image, booting Jetson Nano, connecting to wifi, remote programming through JupyterLab.

*Step 5* to develop low-level device control applications: a) design agent-embedded function block applications through Eclipse 4diac for motors and LEDs; b) compile and deploy function block applications to Raspberry Pi.

*Step 6* to develop high-level JetBot control applications: a) design required multi-agent functions for JetBot; b) develop programs through JupyterLab and deploy to Jetson Nano to run JetBot.

*Step 7* to run designed experiments to demonstrate the feasibility of the proposed architecture modelling framework.

### *Appendix B.2 High-Level JetBot Modelling Test*

#Tests included in this appendix are based on and referred to  
#SparkFun (<https://learn.sparkfun.com/tutorials/assembly-guide-for-sparkfun-jetbot-ai-kit/all>) and  
#Nvidia (<https://jetbot.org/>) tutorials.  
#to assemble and program JetBot for basic motion, collision avoidance, road following and object detection, etc.

#Key programs are shown as follows. For full applications, see references.

#### **a) Basi Motion**

```
from jetbot import Robot
import time
#initialize a robot instance from Robot class
robot = Robot()

#program JetBot to move forward and backward, and turn left and right
#call forward, backward, left, right, and stop methods
robot.forward(speed=0.3)
time.sleep(1.0)
robot.stop()
```

```

robot.right(speed=0.3)
time.sleep(1.0)
robot.stop()

robot.backward(speed=0.3)
time.sleep(1.0)
robot.stop()

robot.left(speed=0.3)
time.sleep(1.0)
robot.stop()

#control left and right motors separately
#call the set_motor method
robot.set_motors(0.3, 0.6)
time.sleep(1.0)
robot.stop()

```

## b) Collision Avoidance

#Step 1 Data Collection  
#collect sample data to teach JetBot two scenarios: blocked scenarios representing dangerous areas to go ahead  
#and free scenarios representing safe areas to move into.

```

import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display
from jetbot import Camera, bgr8_to_jpeg

#display live camera feed
#set image size as 224px by 224px for an appropriate scale dataset
camera = Camera.instance(width=224, height=224)
image = widgets.Image(format='jpeg', width=224, height=224)
#link camera with image
camera_link = traitlets.dlink((camera, 'value'), (image, 'value'), transform=bgr8_to_jpeg)
#display images
display(image)

import os
#create directories to store images
blocked_directory = 'dataset/blocked'
free_directory = 'dataset/free'
#error message if wrong directories
try:
    os.makedirs(free_directory)
    os.makedirs(blocked_directory)
except FileExistsError:
    print('Directories not created because they already exist')

```

#Step 2 Model Training  
#neural network model of image classifier is trained under open source deep learning framework PyTorch

```

#PyTorch is an optimized tensor library for deep learning using GPUs and CPUs
#import Python packages from PyTorch
#torch package includes data structures for multi-dimensional tensors and mathematical operations
import torch

```

```

import torch.optim as optim
import torch.nn.functional as F

#torchvision package includes datasets, model architectures, and image transformations.
import torchvision
import torchvision.datasets as datasets
import torchvision.models as models
import torchvision.transforms as transforms

#create dataset instance for training
dataset = datasets.ImageFolder(
    'dataset',
    transforms.Compose([
        transforms.ColorJitter(0.1, 0.1, 0.1, 0.1),
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
)

#split dataset into train and test sets.
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [len(dataset) - 50, 50])
#create data loaders to load data in batches
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=8, shuffle=True, num_workers=0)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=8, shuffle=True, num_workers=0)

#define neural network for image classification
#torchvision package provides a collection of pre-trained models can be repurposed and reused for new tasks

#Deep Residual Learning for Image Recognition (ResNet models)
#resnet18, resnet34, resnet50, resnet101, resnet152
model = models.resnet18(pretrained=True)

#repurpose and reuse model to 2 classes
model.fc = torch.nn.Linear(512, 2)
#execute model on GPU
device = torch.device('cuda')
model = model.to(device)

#train neural network model
NUM_EPOCHS = 30
BEST_MODEL_PATH = 'best_model.pth'
best_accuracy = 0.0

optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

for epoch in range(NUM_EPOCHS):

    for images, labels in iter(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = F.cross_entropy(outputs, labels)
        loss.backward()
        optimizer.step()

```

```

test_error_count = 0.0

for images, labels in iter(test_loader):
    images = images.to(device)
    labels = labels.to(device)
    outputs = model(images)
    test_error_count += float(torch.sum(torch.abs(labels - outputs.argmax(1))))

test_accuracy = 1.0 - float(test_error_count) / float(len(test_dataset))
print('%d: %f' % (epoch, test_accuracy))

if test_accuracy > best_accuracy:
    torch.save(model.state_dict(), BEST_MODEL_PATH)
    best_accuracy = test_accuracy

#Step 3 Model Deployment
#TensorRT is an optimized neural network model built on Nvidia's parallel programming model CUDA
#for reduced precision but high accuracy, and easily deployable to embedded platforms

#initialize PyTorch model (resnet18 model)
import torch
import torchvision

model = torchvision.models.resnet18(pretrained=False)
model.fc = torch.nn.Linear(512, 2)
model = model.cuda().eval().half()

#load model weights into CPU and then transfer to GPU
model.load_state_dict(torch.load('best_model.pth'))
device = torch.device('cuda')

#build TensorRT model
from torch2trt import torch2trt
data = torch.zeros((1, 3, 224, 224)).cuda().half()
model_trt = torch2trt(model, [data], fp16_mode=True)

#save optimized model
torch.save(model_trt.state_dict(), 'best_model_trt.pth')

```

### c) Road Following

```

#Step 1 Data Collection
#collect sample data to teach JetBot to follow a path using regression instead of classification.

IPython Libraries for display and widgets
import ipywidgets
import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display

#camera and motor interfaces for JetBot
from jetbot import Robot, Camera, bgr8_to_jpeg

#basic Python packages for image annotation
from uuid import uuid1

```

```

import os
import json
import glob
import datetime
import numpy as np
import cv2
import time

from jupyter_clickable_image_widget import ClickableImageWidget

DATASET_DIR = 'dataset_xy'

#error message if wrong directories
try:
    os.makedirs(DATASET_DIR)
except FileExistsError:
    print('Directories not created because they already exist')

camera = Camera()

#create image preview
camera_widget = ClickableImageWidget(width=camera.width, height=camera.height)
snapshot_widget = ipywidgets.Image(width=camera.width, height=camera.height)
traitlets.dlink((camera, 'value'), (camera_widget, 'value'), transform=bgr8_to_jpeg)

#create widgets
count_widget = ipywidgets.IntText(description='count')

#update counts at initialization
count_widget.value = len(glob.glob(os.path.join(DATASET_DIR, '*.jpg')))

def save_snapshot(_, content, msg):
    if content['event'] == 'click':
        data = content['eventData']
        x = data['offsetX']
        y = data['offsetY']

        #save to disk
        dataset.save_entry(category_widget.value, camera.value, x, y)
        uuid = 'xy_%03d_%03d_%s' % (x, y, uuid1())
        image_path = os.path.join(DATASET_DIR, uuid + '.jpg')
        with open(image_path, 'wb') as f:
            f.write(camera_widget.value)

        #display saved snapshot
        snapshot = camera.value.copy()
        snapshot = cv2.circle(snapshot, (x, y), 8, (0, 255, 0), 3)
        snapshot_widget.value = bgr8_to_jpeg(snapshot)
        count_widget.value = len(glob.glob(os.path.join(DATASET_DIR, '*.jpg')))

camera_widget.on_msg(save_snapshot)

data_collection_widget = ipywidgets.VBox([ipywidgets.HBox([camera_widget, snapshot_widget]), count_widget])

display(data_collection_widget)

```

```

#disconnect camera
camera.stop()

#Step 2 Model Training
#use PyTorch deep learning framework to train ResNet18 neural network model for JetBot road following

import torch
import torch.optim as optim
import torch.nn.functional as F
import torchvision
import torchvision.datasets as datasets
import torchvision.models as models
import torchvision.transforms as transforms
import glob
import PIL.Image
import os
import numpy as np

#create dataset instance
#create torch.utils.data.Dataset to implement __len__ and __getitem__ functions
def get_x(path, width):
    """Gets the x value from the image filename"""
    return (float(int(path.split("_")[1])) - width/2) / (width/2)

def get_y(path, height):
    """Gets the y value from the image filename"""
    return (float(int(path.split("_")[2])) - height/2) / (height/2)

class XYDataset(torch.utils.data.Dataset):

    def __init__(self, directory, random_hflips=False):
        self.directory = directory
        self.random_hflips = random_hflips
        self.image_paths = glob.glob(os.path.join(self.directory, '*.jpg'))
        self.color_jitter = transforms.ColorJitter(0.3, 0.3, 0.3, 0.3)

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]

        image = PIL.Image.open(image_path)
        width, height = image.size
        x = float(get_x(os.path.basename(image_path), width))
        y = float(get_y(os.path.basename(image_path), height))

        if float(np.random.rand(1)) > 0.5:
            image = transforms.functional.hflip(image)
            x = -x

        image = self.color_jitter(image)
        image = transforms.functional.resize(image, (224, 224))
        image = transforms.functional.to_tensor(image)
        image = image.numpy()[::-1].copy()
        image = torch.from_numpy(image)

```



```

image = transforms.functional.normalize(image, [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

return image, torch.tensor([x, y]).float()

dataset = XYDataset('dataset_xy', random_hflips = False)

#split dataset into train and test sets (90% vs 10%)
test_percent = 0.1
num_test = int(test_percent * len(dataset))
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [len(dataset) - num_test, num_test])

#create data loaders to load data in batches
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = 8, shuffle = True, num_workers = 0)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size = 8, shuffle = True, num_workers = 0)

#define neural network model
#ResNet18 in PyTorch TorchVision
model = models.resnet18(pretrained=True)

#transfer model for execution on GPU
model.fc = torch.nn.Linear(512, 2)
device = torch.device('cuda')
model = model.to(device)

#train regression to get the best model
NUM_EPOCHS = 70
BEST_MODEL_PATH = 'best_steering_model_xy.pth'
best_loss = 1e9

optimizer = optim.Adam(model.parameters())

for epoch in range(NUM_EPOCHS):

    model.train()
    train_loss = 0.0
    for images, labels in iter(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = F.mse_loss(outputs, labels)
        train_loss += float(loss)
        loss.backward()
        optimizer.step()
    train_loss /= len(train_loader)

    model.eval()
    test_loss = 0.0
    for images, labels in iter(test_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        loss = F.mse_loss(outputs, labels)
        test_loss += float(loss)

```

```

test_loss /= len(test_loader)

print('%f, %f' % (train_loss, test_loss))
if test_loss < best_loss:
    torch.save(model.state_dict(), BEST_MODEL_PATH)
    best_loss = test_loss

#Step 3 Model Deployment
#upload neural network model file to JetBot notebook directory
#initialize the PyTorch model
import torchvision
import torch

model = torchvision.models.resnet18(pretrained=False)
model.fc = torch.nn.Linear(512, 2)

#load model weights into CPU and then transfer to GPU
model.load_state_dict(torch.load('best_steering_model_xy.pth'))
device = torch.device('cuda')
model = model.to(device)
model = model.eval().half()

#create pre-processing function to match image format between model and camera
import torchvision.transforms as transforms
import torch.nn.functional as F
import cv2
import PIL.Image
import numpy as np

mean = torch.Tensor([0.485, 0.456, 0.406]).cuda().half()
std = torch.Tensor([0.229, 0.224, 0.225]).cuda().half()

def preprocess(image):
    image = PIL.Image.fromarray(image)
    image = transforms.functional.to_tensor(image).to(device).half()
    image.sub_(mean[:, None, None]).div_(std[:, None, None])
    return image[None, ...]

#create robot instance to drive motor
from jetbot import Robot
robot = Robot()

#define neural network execution function to process camera value changes
#pre-process the camera image; execute the neural network;
#compute the approximate steering value; control the motors.
angle = 0.0
angle_last = 0.0

def execute(change):
    global angle, angle_last
    image = change['new']
    xy = model(preprocess(image)).detach().float().cpu().numpy().flatten()
    x = xy[0]
    y = (0.5 - xy[1]) / 2.0

```

```

x_slider.value = x
y_slider.value = y

speed_slider.value = speed_gain_slider.value

angle = np.arctan2(x, y)
pid = angle * steering_gain_slider.value + (angle - angle_last) * steering_dgain_slider.value
angle_last = angle

steering_slider.value = pid + steering_bias_slider.value

robot.left_motor.value = max(min(speed_slider.value + steering_slider.value, 1.0), 0.0)
robot.right_motor.value = max(min(speed_slider.value - steering_slider.value, 1.0), 0.0)

execute({'new': camera.value})

#attach execution function to camera for processing
camera.observe(execute, names='value')

#unattach execution function to stop robot
import time
camera.unobserve(execute, names='value')
time.sleep(0.1)
robot.stop()

```

#### **d) Object Detection**

```

#the model is sourced from the TensorFlow Object Detection API
#and optimized through Nvidia TensorRT on Jetson Nano

#compute detections on single camera image
#import ObjectDetector class
from jetbot import ObjectDetector
model = ObjectDetector('ssd_mobilenet_v2_coco.engine')

#initialize JetBot camera
from jetbot import Camera
camera = Camera.instance(width=300, height=300)

#execute neural network model
detections = model(camera.value)
print(detections)

#display detections in text areas
#create detection widgets
from IPython.display import display
import ipywidgets.widgets as widgets

detections_widget = widgets.Textarea()
detections_widget.value = str(detections)
display(detections_widget)

#print out the first image
image_number = 0
object_number = 0
print(detections[image_number][object_number])

```

```

#control JetBot to follow central object
#Step1: detect object matching specified class
#Step2: select target object in the center of camera's field of vision
#Step3: steer JetBot towards target object, otherwise wander
#Step4: turn left if object blocked

#load pre-trained collision detection model
import torch
import torchvision
import torch.nn.functional as F
import cv2
import numpy as np

collision_model = torchvision.models.alexnet(pretrained=False)
collision_model.classifier[6] = torch.nn.Linear(collision_model.classifier[6].in_features, 2)
collision_model.load_state_dict(torch.load('../collision_avoidance/best_model.pth'))
device = torch.device('cuda')
collision_model = collision_model.to(device)

mean = 255.0 * np.array([0.485, 0.456, 0.406])
stdev = 255.0 * np.array([0.229, 0.224, 0.225])
normalize = torchvision.transforms.Normalize(mean, stdev)

def preprocess(camera_value):
    global device, normalize
    x = camera_value
    x = cv2.resize(x, (224, 224))
    x = cv2.cvtColor(x, cv2.COLOR_BGR2RGB)
    x = x.transpose((2, 0, 1))
    x = torch.from_numpy(x).float()
    x = normalize(x)
    x = x.to(device)
    x = x[None, ...]
    return x

#create robot instance to drive motor
from jetbot import Robot
robot = Robot()

#display control widgets and connect model execution function to camera updates
from jetbot import bgr8_to_jpeg

blocked_widget = widgets.FloatSlider(min=0.0, max=1.0, value=0.0, description='blocked')
image_widget = widgets.Image(format='jpeg', width=300, height=300)
label_widget = widgets.IntText(value=1, description='tracked label')
speed_widget = widgets.FloatSlider(value=0.4, min=0.0, max=1.0, description='speed')
turn_gain_widget = widgets.FloatSlider(value=0.8, min=0.0, max=2.0, description='turn gain')

display(widgets.VBox([widgets.HBox([image_widget, blocked_widget]), label_widget, speed_widget,
turn_gain_widget]))

width = int(image_widget.width)
height = int(image_widget.height)

def detection_center(detection):

```

```

    """Computes the center x, y coordinates of the object"""
    bbox = detection['bbox']
    center_x = (bbox[0] + bbox[2]) / 2.0 - 0.5
    center_y = (bbox[1] + bbox[3]) / 2.0 - 0.5
    return (center_x, center_y)

def norm(vec):
    """Computes the length of the 2D vector"""
    return np.sqrt(vec[0]**2 + vec[1]**2)

def closest_detection(detections):
    """Finds the detection closest to the image center"""
    closest_detection = None
    for det in detections:
        center = detection_center(det)
        if closest_detection is None:
            closest_detection = det
        elif norm(detection_center(det)) < norm(detection_center(closest_detection)):
            closest_detection = det
    return closest_detection

def execute(change):
    image = change['new']

    #execute collision model to determine if blocked
    collision_output = collision_model(preprocess(image)).detach().cpu()
    prob_blocked = float(F.softmax(collision_output.flatten(), dim=0)[0])
    blocked_widget.value = prob_blocked

    #turn left if blocked
    if prob_blocked > 0.5:
        robot.right(0.3)
        image_widget.value = bgr8_to_jpeg(image)
        return

    #compute all detected objects
    detections = model(image)

    #draw all detections on image
    for det in detections[0]:
        bbox = det['bbox']
        cv2.rectangle(image, (int(width * bbox[0]), int(height * bbox[1])), (int(width * bbox[2]), int(height *
bbox[3])), (255, 0, 0), 2)

    #select detections that match selected class label
    matching_detections = [d for d in detections[0] if d['label'] == int(label_widget.value)]

    #get detection closest to center of field of view and draw it
    det = closest_detection(matching_detections)
    if det is not None:
        bbox = det['bbox']
        cv2.rectangle(image, (int(width * bbox[0]), int(height * bbox[1])), (int(width * bbox[2]), int(height *
bbox[3])), (0, 255, 0), 5)

    #otherwise go forward if no target detected
    if det is None:

```

```

robot.forward(float(speed_widget.value))

#otherwise steer towards target
else:
    # move robot forward and steer proportional target's x-distance from center
    center = detection_center(det)
    robot.set_motors(
        float(speed_widget.value + turn_gain_widget.value * center[0]),
        float(speed_widget.value - turn_gain_widget.value * center[0])
    )

#update image widget
image_widget.value = bgr8_to_jpeg(image)

execute({'new': camera.value})

#connect execution function to each camera frame update
camera.unobserve_all()
camera.observe(execute, names='value')

#unattach execution function to stop robot
import time
camera.unobserve(execute, names='value')
time.sleep(0.1)
robot.stop()

```

#### e) Management Agent Test

```

# SPADE program for the management agent
# This agent receives messages from LEDs and send messages to Motors.

from spade.agent import Agent
from spade.behaviour import CyclicBehaviour
from spade.template import Template
from spade.message import Message
from spade import quit_spade
import asyncio

color_target = { "blue": ("DCAgent CW", "testagent@jabber.de"),
                  "green": ("DCAgent CCW", "testagent@jabber.de"),
                  "red": ("StepperAgent CW", "receiveragent@jabber.de"),
                  "yellow": ("StepperAgent CCW", "receivergent@jabber.de")
                }

class ManagerAgent(Agent): # extends the Agent class from Python
    # define a behaviour class that extends CyclicBehaviour from Spade.
    class ManageBehaviour(CyclicBehaviour):

        async def on_start(self): # on_start() function executes once when a behaviour instance is added to an agent
            template = Template() # set the template to match incoming messages. This ensures receive() works.
            template.set_metadata("performative", "inform")

        async def run(self): # in CyclicBehaviour run() runs indefinitely until the behaviour is stopped.
            message_from_led = await self.receive(20)
            if not message_from_led:
                # If no message is received from LED agent, quit Spade
                print("No message from LED agent")

```

```

await self.agent.stop()
quit_spade()

else:
    # extracting color from message, converting to target agent and direction
    color = ''.join(letter for letter in message_from_led.body if letter.isalpha())
    target_direction = color_target[color]
    target_agent = target_direction[1]
    target_name = target_direction[0].split()[0]
    direction = target_direction[0].split()[1]
    # extracting on-time from message from LED agent
    run_time_string = ''.join(letter for letter in message_from_led.body if not letter.isalpha())
    # creating a message to motor based on color, direction, and on-time
    message_to_motor = Message(to=target_agent)
    message_to_motor.set_metadata("performative", "inform")
    message_to_motor.body = direction+" "+run_time_string
    print("Message {} received from LED agent. Sending message {} to {}".
          format(message_from_led.body, message_to_motor.body, target_name))
    await self.send(message_to_motor) # sends message to appropriate motor
    # wait 30 seconds for motor to complete operation and send a confirmation message to manager
    message_from_motor = await self.receive(30)
    # if no message is received from motor agent, quit spade
    if not message_from_motor:
        print("No confirmation from motor, quitting Spade ...")
        await self.agent.stop()
        quit_spade()

    # if message from motor agent does not contain the phrase "Done", quit Spade
    elif "Done" not in message_from_motor.body:
        print("Keyword 'Done' not in message from motor, quitting Spade ...")
        await self.agent.stop()
        quit_spade()

    # if confirmation received from motor agent, send a "continue" message to LED agent
    else:
        print("Confirmation received from {}, resuming operation ...".format(target_name))
        await asyncio.sleep(2)
        message_to_led = Message(to="senderagent@jabber.de")
        message_to_led.set_metadata("performative", "inform")
        message_to_led.body = "continue"
        await self.send(message_to_led)

async def setup(self):
    manage_behaviour = self.ManageBehaviour()
    self.add_behaviour(manage_behaviour)

agent = ManagerAgent("manageragent@jabber.de", "managerlyu") # initiate the manager agent
future = agent.start() # runs manager agent behaviour
future.result()

```

### ***Appendix B.3 Low-level Devices Modelling Test***

# Tests in this appendix include low-level device setup and programming of LEDs and Motors in Raspberry Pi.

### a) LED Test

```
# LED Test with Raspberry Pi 3
# Blue --> GPIO 14, Green --> GPIO 15, Red --> GPIO 18, Yellow --> GPIO 23

import RPi.GPIO as GPIO
import time

# Set up GPIO pins
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(14, GPIO.OUT) # Blue GPIO 14
GPIO.setup(15, GPIO.OUT) # Green GPIO 15
GPIO.setup(18, GPIO.OUT) # Red GPIO 18
GPIO.setup(23, GPIO.OUT) # Yellow GPIO 23

# Turn on LED lights
print("LED On")
GPIO.output(14, GPIO.HIGH)
time.sleep(1)
GPIO.output(15, GPIO.HIGH)
time.sleep(1)
GPIO.output(18, GPIO.HIGH)
time.sleep(1)
GPIO.output(23, GPIO.HIGH)

time.sleep(3)

# Turn off LED lights
print("LED Off")
GPIO.output(23, GPIO.LOW)
time.sleep(1)
GPIO.output(18, GPIO.LOW)
time.sleep(1)
GPIO.output(15, GPIO.LOW)
time.sleep(1)
GPIO.output(14, GPIO.LOW)

# Reset GPIO pins
GPIO.cleanup()

# SPADE program to control LEDs
# This agent lights up LEDs with random colors (motor rotation direction) and durations (motor rotation time)

import RPi.GPIO as GPIO
import time
import random
import asyncio
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour
from spade.message import Message
from spade.template import Template
from spade import quit_spade

class LedAgent(Agent):
```



```

async def setup(self):
    self.lightbehaviour = self.LightBehaviour()
    self.add_behaviour(self.lightbehaviour)

class LightBehaviour(CyclicBehaviour):
    async def on_start(self):
        template = Template()
        template.set_metadata("performative", "inform")
        message_to_manager = Message(to="manageragent@jabber.de")
        message_to_manager.set_metadata("performative", "inform")
        # creating lists of colors and pins
        color_list = ["blue", "green", "red", "yellow"]
        GPIO.setmode(GPIO.BCM)
        pin_list = [14, 15, 18, 23]
        color_pin = dict(zip(color_list, pin_list))
        color = random.choice(color_list)
        # use integers for LED on-times
        on_time = random.randint(1, 4)
        pin = color_pin[color]
        print("Turning led on for first time: {}, {}".format(color, str(on_time)))
        # initiating the GPIO pins on the pi
        GPIO.setup(pin_list, GPIO.OUT)
        # turning the pin high
        GPIO.output(pin, GPIO.HIGH)
        # keeping the pin high and LED lit for on_time seconds
        await asyncio.sleep(on_time)
        # turning pin LOW turns off LED
        GPIO.output(pin, GPIO.LOW)
        # cleaning up prepares for next loop
        GPIO.cleanup(pin_list)
        mylist = [color, on_time]
        # message to manager consists of color and on_time both strings
        message_to_manager.body = mylist[0]+" "+str(mylist[1])
        self.message_to_manager = message_to_manager

    async def run(self):
        await self.send(self.message_to_manager) # start by sending first message to manager
        print("Message to manager agent: {}".format(self.message_to_manager.body))
        color_list = ["blue", "green", "red", "yellow"]
        GPIO.setmode(GPIO.BCM)
        pin_list = [14, 15, 18, 23]
        color_pin = dict(zip(color_list, pin_list))
        color = random.choice(color_list)
        on_time = random.randint(1, 10)
        pin = color_pin[color]
        GPIO.setup(pin_list, GPIO.OUT)
        message_from_manager = await self.receive(20)

        if not message_from_manager:
            # if no message is received from manager, quit Spade
            print("No message from manager, quitting Spade ...")
            await self.agent.stop()
            quit_spade()

        elif "continue" not in message_from_manager.body:

```

```

        # if message from manager does not say "continue" quit Spade
        print("Keyword 'continue' not in message from manager, quitting Spade ...")
        await self.agent.stop()
        quit_spade()

    else:
        await asyncio.sleep(2) # pause for two seconds to manage timing issues
        print("Confirmation received from manager, turning {} on for {} seconds ...".format(color, str(on_time)))
        GPIO.output(pin, GPIO.HIGH)
        await asyncio.sleep(on_time)
        GPIO.output(pin, GPIO.LOW)
        GPIO.cleanup(pin_list)
        mylist = [color, on_time]
        self.message_to_manager.body = mylist[0]+" "+str(mylist[1])

agent = LedAgent("senderagent@jabber.de", "senderlyu") # initiate LED agent
future = agent.start() # run LED agent behaviour
future.result()

```

## b) Stepper Motor Test

```

# Stepper Motor Test with Raspberry Pi 3
# STEP --> GPIO 14 (Blue), DIR --> GPIO 15 (Yellow)
# Phidgets Bipolar 12V Stepper Motor: https://www.phidgets.com/?tier=3&catid=24&pcid=21&prodid=340.
# Texas Instruments DRV8825 Stepper Motor Driver: https://www.pololu.com/product/2133.
.
# Set up connections
# VMOT and GND to 12V Power Supply
# A1, A2, B1, B2 to Step Motor
# SLEEP and RESET to RPi 5V, GND to RPi GND
# STEP to RPi GPIO 14, DIR to RPi GPIO 15

import RPi.GPIO as GPIO
import time

# Set up GPIO pins
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(14, GPIO.OUT) # GPIO 14 to STEP
GPIO.setup(15, GPIO.OUT) # GPIO 15 to DIR

# Drive the motor clockwise
print("ClockWise Rotation")
GPIO.output(14, GPIO.HIGH) # Set STEP
GPIO.output(15, GPIO.HIGH) # Set DIR

for x in range(10000):
    GPIO.output(14, GPIO.HIGH)
    time.sleep(0.0002)
    GPIO.output(14, GPIO.LOW)
    time.sleep(0.0002)

time.sleep(2)

# Drive the motor counterclockwise
print("CounterClockWise Rotation")

```

```

GPIO.output(14, GPIO.LOW) # Set STEP
GPIO.output(15, GPIO.LOW) # Set DIR

for x in range(10000):
    GPIO.output(14, GPIO.HIGH)
    time.sleep(0.0002)
    GPIO.output(14, GPIO.LOW)
    time.sleep(0.0002)

# Reset GPIO pins
GPIO.cleanup()

# SPADE program to control the stepper motor
# This agent runs the stepper in the direction and for the duration sent by the manager agent

import RPi.GPIO as GPIO
import time
import asyncio
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour
from spade.message import Message
from spade import quit_spade
from spade.template import Template

class StepperAgent(Agent):
    async def setup(self):
        template = Template()
        template.set_metadata("performative", "inform")
        self.stepbehaviour = self.StepBehaviour()
        self.add_behaviour(self.stepbehaviour, template)

class StepBehaviour(CyclicBehaviour):
    async def on_start(self):
        message_to_manager = Message(to="manageragent@jabber.de")
        message_to_manager.set_metadata("performative", "inform")
        message_to_manager.body = "Stepper Done"
        # create a message that is sent to manager when motor has completed operation
        self.message_to_manager = message_to_manager

    async def run(self):
        for attempt in range(10): # agent attempts to receive message from manager 10 times.
            message_from_manager = await self.receive(10)
            if message_from_manager:
                break
            if not message_from_manager: # If no message is received quit spade with a message
                print("No message from manager, quitting Spade ...")
                await self.agent.stop()
                quit_spade()

        # if message received, extract on time as an integer and direction as a string
        on_time = int(''.join(letter for letter in message_from_manager.body if letter.isdigit()))
        direction = ''.join(letter for letter in message_from_manager.body if letter.isalpha())

        if direction == "CW":
            time_start = time.perf_counter()

```

```

time_elapsed = 0
sleep_time = 0.0002 # time between pulses to motor based on experiments
print("Message received from manager. Running Stepper {} for {} seconds ...".format(direction, on_time))
while time_elapsed <= on_time:
    time_elapsed = time.perf_counter() - time_start
    # Initiating GPIO pins on the raspberry pi
    GPIO.setmode(GPIO.BCM)
    pin_list = [14, 15]
    GPIO.setup(pin_list, GPIO.OUT)
    GPIO.output(pin_list, GPIO.LOW)
    GPIO.output(15, GPIO.HIGH) # sets direction HIGH
    # full stepping: Enable LOW, M1, M2, M3 LOW
    GPIO.output(14, GPIO.HIGH)
    time.sleep(sleep_time)
    GPIO.output(14, GPIO.LOW)
    time.sleep(sleep_time)
GPIO.cleanup() # After the motor is run, clean up GPIO pins for the next loop
await self.send(self.message_to_manager)
# sends a message to manager, indicating the motor agent has finished operating

elif direction == "CCW":
    time_start = time.perf_counter()
    time_elapsed = 0
    sleep_time = 0.0002
    print("Message received from manager. Running Stepper {} for {} seconds ...".format(direction, on_time))
    while time_elapsed <= on_time:
        time_elapsed = time.perf_counter() - time_start
        GPIO.setmode(GPIO.BCM)
        pin_list = [14, 15]
        GPIO.setup(pin_list, GPIO.OUT)
        GPIO.output(15, GPIO.LOW) # sets direction LOW
        # full stepping: Enable LOW, M1, M2, M3 LOW
        GPIO.output(14, GPIO.HIGH)
        time.sleep(sleep_time)
        GPIO.output(14, GPIO.LOW)
        time.sleep(sleep_time)

    GPIO.cleanup()
    await self.send(self.message_to_manager)
    # sends confirmation message to manager, indicating motor has completed operation

agent = StepperAgent("receiveragent@jabber.de", "receiverlyu")
future = agent.start() # runs the agent behaviours
future.result()

```

### c) DC Motor Test

```

# DC Motor Test with Raspberry Pi Zero
# PWMA--> GPIO 14 (Blue), STBY--> GPIO 15 (White), AI1--> GPIO 18 (Red), AI2--> GPIO 23 (Black)
# Phidgets 12V DC Motor: https://www.phidgets.com/?tier=3&catid=20&pcid=17&prodid=1139.
# TB6612FNG DC Motor Driver on a SparkFun Breakout Board: https://www.sparkfun.com/products/14450.

# Set up Connections
# VM and GND to 12V Power Supply
# A01 and A02 to DC Motor
# VCC to RPi 5V, GND to RPi GND,

```

```
# PWMA to RPi GPIO 14, STBY to RPi GPIO 15, AIN1 to RPi GPIO 18, AIN2 to RPi GPIO 23
```

```
import time
import RPi.GPIO as GPIO
```

```
# Set up GPIO pins
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
```

```
GPIO.setup(14, GPIO.OUT) # GPIO 14 to PWMA
GPIO.setup(23, GPIO.OUT) # GPIO 23 to AIN2
GPIO.setup(18, GPIO.OUT) # GPIO 18 to AIN1
GPIO.setup(15, GPIO.OUT) # GPIO 15 to STBY
```

```
# Drive the motor clockwise
print("ClockWise Rotation")
GPIO.output(18, GPIO.HIGH) # Set AIN1
GPIO.output(23, GPIO.LOW) # Set AIN2
```

```
# Set the motor speed
GPIO.output(14, GPIO.HIGH) # Set PWMA
```

```
# Disable STBY
GPIO.output(15, GPIO.HIGH)
```

```
time.sleep(5)
```

```
# Drive the motor counterclockwise
print("CounterClockWise Rotation")
GPIO.output(18, GPIO.LOW) # Set AIN1
GPIO.output(23, GPIO.HIGH) # Set AIN2
```

```
# Set the motor speed
GPIO.output(14, GPIO.HIGH) # Set PWMA
```

```
# Disable STBY
GPIO.output(15, GPIO.HIGH)
```

```
time.sleep(5)
```

```
# Reset all the GPIO pins
GPIO.cleanup()
```

```
# SPADE program to control the DC motor
# This agent runs the DC motor in the direction and for the duration sent by the manager agent
```

```
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour
from spade.template import Template
from spade.message import Message
from spade import quit_spade
import asyncio
import RPi.GPIO as GPIO
import time
```

```
class DCAgent(Agent): # extends the Agent class from Python
```

```
    async def setup(self):
```

```
        template = Template() # set the template to match incoming messages. This ensures receive() works.
```

```
        template.set_metadata("performative", "inform")
```

```
        self.dc_behaviour = self.DC_Behaviour()
```

```
        self.add_behaviour(self.dc_behaviour, template)
```

```
class DC_Behaviour(CyclicBehaviour):
```

```
    async def on_start(self): # on_start() function executes once when a behaviour instance is added to an agent
```

```
        message_to_manager = Message(to="manageragent@jabber.de")
```

```
        message_to_manager.set_metadata("performative", "inform")
```

```
        message_to_manager.body = "DC Done"
```

```
        # create a message that indicates DC motor has completed operation
```

```
        self.message_to_manager = message_to_manager
```

```
    async def run(self): # in CyclicBehaviour run() runs indefinitely until the behaviour is stopped.
```

```
        # attempt to receive message from manager 10 times. This helps with the timing of operations
```

```
        for attempt in range(10):
```

```
            message_from_manager = await self.receive(10) # Waiting 10 seconds for message from manager
```

```
            if message_from_manager:
```

```
                break
```

```
            # if no message is received from manager, quit Spade
```

```
            if not message_from_manager:
```

```
                print("No message from manager agent, quitting Spade ...")
```

```
                await self.agent.stop()
```

```
                quit_spade()
```

```
        # extract on_time (as integer) and direction (as a string) from message
```

```
        on_time = int(''.join(letter for letter in message_from_manager.body if letter.isdigit()))
```

```
        direction = ''.join(letter for letter in message_from_manager.body if letter.isalpha())
```

```
        if direction == "CW":
```

```
            print("Message received from manager. Running DC motor {} for {} seconds ...".format(direction, on_time))
```

```
            # initiate GPIO pins
```

```
            GPIO.setmode(GPIO.BCM)
```

```
            pin_list = [14, 15, 18, 23]
```

```
            GPIO.setup(pin_list, GPIO.OUT)
```

```
            GPIO.output(pin_list, GPIO.LOW)
```

```
            GPIO.output(14, GPIO.HIGH) # PWMA HIGH
```

```
            GPIO.output(15, GPIO.HIGH) # SYTB HIGH
```

```
            # CW; AI1 HIGH, AI2 LOW
```

```
            GPIO.output(18, GPIO.HIGH)
```

```
            GPIO.output(23, GPIO.LOW)
```

```
            time.sleep(on_time)
```

```
            GPIO.output(14, GPIO.LOW) #PWMA LOW for short brake
```

```
            GPIO.cleanup() # clean up GPIO pins for next operation
```

```
            await self.send(self.message_to_manager) # send completion confirmation to manager agent
```

```
        elif direction == "CCW":
```

```
            print("Message received from manager. Running DC motor {} for {} seconds ...".format(direction, on_time))
```

```
            GPIO.setmode(GPIO.BCM)
```

```
            pin_list = [14, 15, 18, 23]
```

```
            GPIO.setup(pin_list, GPIO.OUT)
```

```

GPIO.output(pin_list, GPIO.LOW)
GPIO.output(14, GPIO.HIGH) # PWMA HIGH
GPIO.output(15, GPIO.HIGH) # SYTBV HIGH
# CCW; AI1 LOW, AI2 HIGH
GPIO.output(18, GPIO.LOW)
GPIO.output(23, GPIO.HIGH)
time.sleep(on_time)
GPIO.output(14, GPIO.LOW) #PWMA LOW for short brake
GPIO.cleanup()
await self.send(self.message_to_manager)

agent = DCAgent("testagent@jabber.de", "testlyu") # initiate the DC motor agent
future = agent.start() # Run the DC motor agent behaviour
future.result()

```