FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

ROS-based Robot Manipulation Using the IEC 61499 Standard

Pedro Saraiva Carvalho



Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Rui Pedro Ferreira Pinto

March 31, 2026

Resumo

Um Sistema de Controlo Distribuído (SCD) é uma estrutura digital identificável pela sua abordagem altamente modular à gestão de sistemas, desviando-se dos controladores centralizados, para aumentar a fiabilidade e reduzir a carga de trabalho. Estas qualidades definem um ambiente muito desejável para sistemas robóticos avançados, permitindo-lhes escalar com maior complexidade, protegidos pela base modular. Sistemas Ciberfísicos (SCF) são um tipo de estrutura que surgiu como resultado dessa evolução, por isso são frequentemente integrados como SCD.

Atualmente, existem vários métodos que auxiliam a concepção, gestão e implementação desses sistemas, como o *Robot Operating System* (ROS) e as normas IEC 61499. ROS é uma estrutura de código aberto caracterizada por uma arquitetura baseada em nós, o que destaca a sua modularidade. IEC 61499, no entanto, é uma norma internacional que fornece uma estrutura baseada em modelos com um conjunto de normas que ditam o comportamento, a comunicação e a implantação de blocos de funções. Independentemente das suas vantagens, esses sistemas também apresentam alguns problemas, exigindo um alto nível de compatibilidade e devendo ser dimensionados de acordo com o aumento da sua magnitude. Esses problemas são acentuados ainda mais pelo fato de que configurações modulares complexas incorporam componentes de diferentes fabricantes, que podem seguir diretrizes conflitantes.

Com isso em mente, este trabalho apresenta *Kindred Architecture Integration Nucleus of DINASORE and ROS Environments* (KAINOSORE) como uma abordagem alternativa ao método habitual, com o objetivo de resolver as falhas do modus operandi modular. Esta solução baseia-se nas normas ROS e IEC 61499 já estabelecidas, sendo esta última implementada como parte do software Dynamic Intelligent Architecture for Software and Modular Reconfiguration (DINASORE). Essencialmente, o KAINOSORE utiliza blocos funcionais para a configuração do sistema, gerindo a comunicação entre eles e os elementos ROS.

Além disso, testamos a solução proposta e comparamo-la com o ROS 2 e o DINASORE utilizando a mesma configuração experimental. As métricas utilizadas focam-se no desempenho, complexidade e facilidade de manutenção. Posteriormente, discutimos os resultados obtidos.

Por último, analisamos a solução proposta com base no que foi discutido ao longo desta dissertação e tiramos conclusões objetivas, apoiadas por uma visão introspectiva sobre trabalhos futuros.

Abstract

A Distributed Control System (DCS) is a digital structure identifiable by its highly-modular approach to systems management, deviating from centralized controllers, to increase reliability and reduce workload. These qualities define a very desirable environment for advanced robotic systems, allowing them to scale with increased complexity, safeguarded by the modular fundamental. Cyber-Physical Systems (CPS) are a type of structure that has emerged as a result of this evolution, so they are often integrated as DCS.

Currently, there are several methods that help the design, management and implementation of these systems, such as the Robot Operating System (ROS) and the IEC 61499 Standard. ROS is an open-source framework characterized by a node-based architecture that underlines its modularity. The IEC 61499, however, is an international standard that provides a model-based structure with a set of norms that dictate the behaviour, communication and deployment of function blocks.

Regardless of its perks, these systems also incur some issues, requiring a high level of compatibility and must scale accordingly as its magnitude increases. These issues are accentuated further by the fact that complex modular configurations incorporate components from different manufacturers, that may follow conflicting guidelines.

With this in mind, this work presents Kindred Architecture Integration Nucleus of DINASORE and ROS Environments (KAINOSORE) as an alternative approach to the de facto methodologies, aiming to tackle the flaws of the modular *modus operandi*. This solution is built upon the already established ROS and IEC 61499 standard, with the latter being implemented as part of the Dynamic Intelligent Architecture for Software and Modular Reconfiguration (DINASORE) software. In essence, KAINOSORE uses function-blocks for the system's configuration, handling the communication between them and ROS elements.

Furthermore, we test the proposed solution and compare it to ROS 2 and DINASORE using the same experimental setup. The metrics used focus on performance, complexity and maintainability. Afterwards, we discuss the results obtained.

Lastly, we review the proposed solution based on the what's been discussed throughout this dissertation and draw objective conclusions, supported by an introspective look onto future work.

Keywords: *ROS • IEC 61499 • I4.0 • CPPS • Advanced Robotics*

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

In this dissertation, we direct most of our attention towards SDG 9 (Industry, Innovation and Infrastructure) and SDG 12 (Responsible Consumption and Production) by promoting an open-source and collaborative-focused framework, available for all users with an intuitive architecture supported by software abstraction.

The specific Sustainable Development Goals mentioned have the following names:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SDG 12 Ensure sustainable consumption and production patterns

SGD	**Target**	**Contribution**	**Performance Indicators and Metrics**
9	9.5	Providing free and public access to an easy-to-learn open-source framework encourages inclusivity and innovation for any user regardless of their technological background.	System's maintainability and components' reusability; Qualitative analysis and comparison (future work)
12	12.2	Focusing on tool-cooperation and scalable methodologies helps reduce the amount of technologies becoming obsolete, which also reduces the number of industrial robots produced specifically to replace/accommodate them.	Interoperability and scalability

Contents

1 Introduction **1**
- 1.1 Contextualization . . . 1
- 1.2 Motivation . . . 3
- 1.3 Problem Definition . . . 4
- 1.4 Objectives and Work Proposal . . . 4
  - 1.4.1 Research Questions . . . 5
- 1.5 Dissertation Structure . . . 5

2 Background and Fundamental Aspects **7**
- 2.1 Historical Background . . . 7
  - 2.1.1 Industry 4.0 . . . 8
- 2.2 Cyber-Physical Systems . . . 9
  - 2.2.1 Cyber-Physical Production Systems . . . 10
- 2.3 Robot Operating System . . . 11
  - 2.3.1 Architecture . . . 12
  - 2.3.2 Robot Operating System 2 . . . 12
  - 2.3.3 Pros and Cons . . . 13
- 2.4 IEC 61499 Standard . . . 14
  - 2.4.1 Architecture . . . 14
  - 2.4.2 4DIAC-IDE . . . 16
  - 2.4.3 DINASORE . . . 17

3 Literature Review **19**
- 3.1 Mapping Approach . . . 19
- 3.2 ReApp Workbench . . . 20
- 3.3 Additional Works . . . 21
  - 3.3.1 Sequential Integration . . . 22
  - 3.3.2 Specific Mapping . . . 22
- 3.4 Conclusions . . . 23

4 Proposed Solution **24**
- 4.1 Hypothesis . . . 24
- 4.2 KAINOSORE Approach . . . 24
- 4.3 Architecture . . . 26
- 4.4 Design Principles . . . 27
- 4.5 Design Objectives . . . 30

5 Implementation **31**
5.1 Requirements . . . 31
5.2 Structure . . . 32
5.2.1 Using 4DIAC . . . 32
5.2.2 Communication Interfaces . . . 33
5.2.3 Execution Model . . . 33
5.2.4 Programmable Function Block Behaviour . . . 34
5.3 ROS-based Integration . . . 34
5.4 *Turtlesim* Use Case . . . 37

6 Validation **40**
6.1 Setup . . . 40
6.1.1 Test Case Scenario . . . 41
6.2 Metrics . . . 43
6.2.1 Function Block Pipelines . . . 43
6.2.2 Performance . . . 44
6.3 Tests Description . . . 45
6.4 Results . . . 46
6.4.1 Function Block Pipeline Results . . . 47
6.4.2 Performance Results . . . 48

7 Discussion **50**
7.1 Findings . . . 50
7.1.1 Function Block Pipeline Discussion . . . 50
7.1.2 Resource Consumption Discussion . . . 51
7.1.3 Timers . . . 52
7.2 Answering Research Questions . . . 52
7.3 Main Advantages of KAINOSORE . . . 53
7.4 Main Limitations of KAINOSORE . . . 54

8 Conclusion and Future Work **55**
8.1 Main Contributions . . . 55
8.2 Future Work . . . 56

References **57**

List of Figures

1.1 Industrial Revolutions [33] 2

2.1 Design Principles of Industry 4.0. [43] 9
2.2 General structure of a Cyber-Physical System with Sensors and Actuators to connect the Digital and Physical planes. 10
2.3 ROS 2 node interfaces: Topics, services, and actions. [32] 13
2.4 Qualities of Open Automation and Control Systems (Source: [12]) 15
2.5 Overview of a generic BFB 16

3.1 Editors for the programming of nodes (blue) according to the IEC 61499 standard [49] 21

4.1 KAINOSORE layered architecture integrating ROS 2 communication with IEC 61499-based orchestration through ROS-enabled Function Blocks 27

5.1 Structure of the ROS2_PUB template 35
5.2 Examples of Monitor FBIRs templates: *ACTIONS_LIST* (top left), *PUBS_LIST* (top right), *SERVICES_LIST* (bottom left), *SCLIENTS_LIST* (bottom right) . . . 36
5.3 Structure of the *INTERFACE_FORMAT_FB* template 37
5.4 *Turtlesim* Pipeline Example 38
5.5 Original and final stages of the *turtlesim* simulator 39

6.1 *UFactory xArm 5* 41
6.2 Pick and stack laboratory architecture scheme (not to scale) 42
6.3 Deployed structures for single component sequential workflow 45
6.4 Deployed structure for pipeline parallel workflow (Example: *KAINOSORE (pipeline)*) 46
6.5 System configuration for the *DINASORE (single FB)*, *DINASORE(pipeline)*, *KAINOSORE (single FBIR)* and *KAINOSORE (pipeline)* scenarios 47

List of Tables

2.1 Experimental results with average grade [37] 18

6.1 Function Block Pipeline Metrics (N/A: Not Applicable) 47
6.2 Resource Consumption in **Six** Piece Pick and Stack Scenario 48
6.3 Resource Consumption in **Twelve** Piece Pick and Stack Scenario 48
6.4 Resource Consumption Comparison 48
6.5 Duration of **Six** Piece Pick and Stack Operation 49
6.6 Duration of **Twelve** Piece Pick and Stack Operation 49

List of Acronyms

AC	Alternating Current
AI	Artificial Intelligence
API	Application Programming Interface
BFB	Basic Function Block
CPPS	Cyber-Physical Production System
CPS	Cyber-Physical System
CPU	Central Processing Unit
DDS	Data Distribution Service
DINASORE	Dynamic Intelligent Architecture for Software and Modular Reconfiguration
DOF	Degrees of Freedom
ECC	Execution Control Chart
FB	Function Block
FBIR	Function Block Integrating ROS 2
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEC	International Electrotechinal Comission
IoT	Internet of Things
IP	Internet Protocol
IPC	Interprocess Communication
IT	Information Technology
I4.0	Industry 4.0
I5.0	Industry 5.0
KAINOSORE	Kindred Architecture Integration Nucleus of DINASORE and ROS Environments
LOC	Lines of Code
MI	Maintainability
ML	Machine Learning
M2M	Machine-to-Machine
OPC-UA	Open Platform Communications - Unified Architecture
P2P	Peer-to-peer
RAM	Random Access Memory
RMW	ROS Middleware
ROS	Robot Operating System
RTE	Runtime Environment
SDK	Software Development Kit
TCP	Transmission Control Protocol
XML	Extensible Markup Language
4IR	Fourth Industrial Revolution
5IR	Fifth Industrial Revolution

Chapter 1

Introduction

1.1 Contextualization

Throughout the years, human civilization has shown an increasingly rapid rate of growth and evolution. Consequently, the tools we use have also evolved, befittingly for the lifestyle changes that have occurred, most notably in the modern times.

This is, by and large, most evident in the worldwide spread of industrialization in our society, which has grown accustomed to a certain lifestyle of overconsumption, matched only by the standard mass production. As a result, there has been an incessant development of new technologies in an attempt to fulfill these needs, which, in turn, introduces new expectations, that often turn into demands, solely satisfied by even newer developments [51]. This socio-technological feedback cycle has become a foundational principle in systems engineering, with conceptual roots tracing back to the First Industrial Revolution.

As new advancements are integrated into the fundamental aspects of industrial procedures, every so often, they change the contemporary workflow of production so radically, that it signals the beginning of a new era. Such cases are referred to as "Industrial Revolutions" and, these events, are regarded as historical milestones, but their occurrence has become more frequent over time. The current age is generally accepted to be the Industry 4.0 (I4.0), which is commonly characterized by the integration of robotics, gene editing and artificial intelligence (AI), among other technologies, merging the concepts of the physical, biological and digital realities [42]. Even so, we are also undergoing a transition into the Fifth Industrial Revolution (5IR) [4], [31], defined as an intersection of human-centric, resilient and sustainable strategies, heavily relying on efficient AI and collaborative robots applications [13], [2].

The Figure 1.1 helps visualize the different Industrial Revolutions along with some of the technologies they are most often associated with.

The coeval of I4.0 and Industry 5.0 (I5.0), taking place, acts as a testament to the accelerated pace of the manufacturing progress. When comparing the time spans in-between the previous revolutions, this acceleration is even more evident, as they have become progressively shorter. Therefore, recent technological growth aims to take this into account, by pursuing scalable and

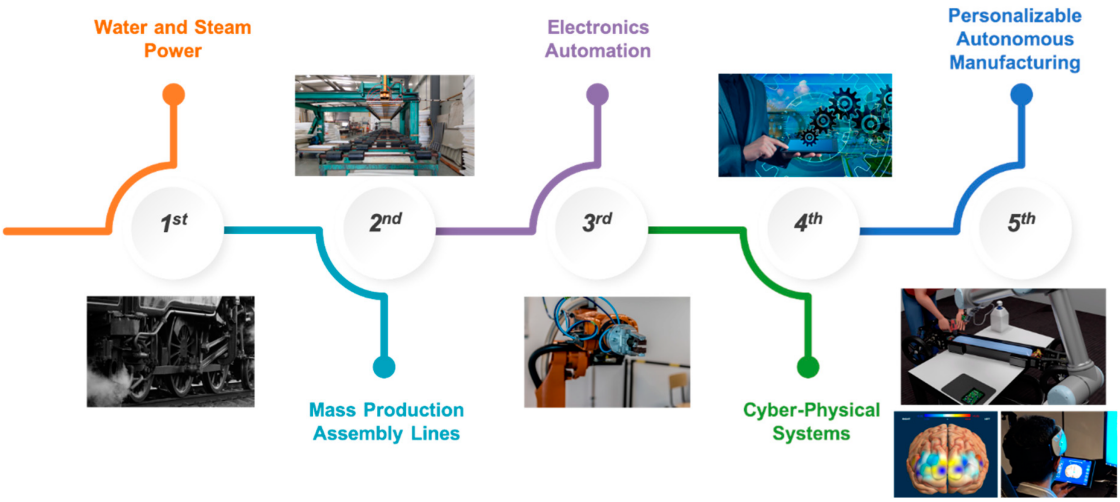


Figure 1.1: Industrial Revolutions [33]

future-proof methods. Regardless, even though scalability is a key motif throughout this investigation, the focus will remain on the I4.0.

Advanced robotics, one of the major themes explored, is characteristic of the Fourth Industrial Revolution (4IR) [6]. However, this idea, which is often associated with the 21st century by the general public, was first introduced during the 1960s, in its precursor form of industrial robotics. Ever since then, they were normalized and have been evolving to keep up with, and support, other technologies, as they surface in the mechanical world. As such, industrial robots grew into the norm and a necessity in factories all around the globe.

Another significant subject investigated, that was standardized during this era, is the Cyber-Physical System (CPS). This type of system is usually designed with a decentralized and event-driven architecture, intended to aid the creation and management of control systems [28]. However, in this work, we are prioritizing a section of the CPS, also known as Cyber-Physical Production System (CPPS). Furthermore, advanced robotics, as described above, is a component of the CPPS, which will be discussed with more detail in future chapters.

Various mechanisms have been developed, along the years, to help support these systems. Among these tools, the ones used for the purpose of this dissertation were, primarily, the Robot Operating System (ROS)[1][2] [39] and the IEC 61499[12] [11] standard.

The aforementioned ROS is a versatile open-source framework, built to simplify the design, development and control of complex robotic systems. It carries many features, among which, hardware abstraction, low-level device control, communication between processes and package management, are of particular importance to this scientific venture. These capacities are supported by a broad collection of tools, libraries and conventions, made easily available to its users [27], [35], [39]. Which, as a result of being a freely accessible software, these include independent and industrial workers alike. In the current context, ROS2 became an indispensable component, as it is

[1] https://docs.ros.org/
[2] https://index.ros.org/

the primary middleware used for the execution of automated tasks, be it in simulation or laboratory environments. As such, the methodologies we will be describing in future chapters, constitute an integration with the ROS2 system and not a replacement for it. Fundamentally, ensuring that our technique maintains the capabilities already present in the former, we are working directly on specific segments that we wish to improve.

Furthermore, IEC 61499 is an international standard adopted as a development practice in distributed control systems. With its model-based paradigm, for designing and implementing these structures, it has become substantially relevant in modern automation. This configuration is employed through function blocks, containing certain processes, that can be interconnected to create complex applications. Essentially, it operates as a path, for other technologies to take, with established rules that facilitate the interpretation and development of a system and its components safeguarding a consistent, and easy to follow, form of communication between them [11], [1].

1.2 Motivation

The motivation behind this work's proposal can be epitomized as the search for a technique capable of reacting promptly to the growth of industrial processes, while unifying different software under a single framework.

In a critical sense, even though the current state of the I4.0 is the most advanced it has ever been, it still has several underlying issues raising some concerns that are addressed in this work. These problems, mostly stem from the industrial evolution itself, as a direct consequence of its stages and the accelerated transitions between them. This becomes more evident when comparing the progress of technologies versus that of humans.

As the industrial and societal worlds were rapidly evolving, as individuals, we weren't, which means that while the core of a mechanical system is getting more complex and its restrictions more abstract, we are forced to become more technologically aware and our own limitations more apparent. In contrast to the first advancements in robotics, which were created as a means to complement our civilization's growth, modern advancements have relegated us to adapt to their own improvements. Since computers process information faster, we introduce extra equipment to handle data exchange and communications, distancing ourselves from the main processor and decreasing the value of the human aspect in the human-machine interaction. Moreover, the arrival of new developments will often call for a revamp of operating workflows, which represents a liability and a professional's competence risks becoming obsolete. This is proportional to the adaptability of a given method.

Additionally, this results in an increasingly more complex manufacturing system, difficult to grasp with its heaps of minutiae, composed of several smaller technical structures, each incorporating multiple equipments that must be compatible with one another. Furthermore, currently employed tools for the design, management and implementation of these systems inadequately handle the balance between simplification and functionality.

Taking this into account, the presented alternative must be scalable with the expansion of CPPS and robotic structures. Besides that, we also intend to focus on the interoperability aspect of these systems, by providing a reliable method that can be implemented without additional modifications to the already equipped hardware. This can be achieved through the consolidation of well-established technologies, which are currently supported by the aforementioned industrial systems, on a comprehensive software nucleus. Effectively, this would fortify their expandability and compatibility with an user-friendly interface, minimizing oversimplification and overcomplication drawbacks.

1.3 Problem Definition

Considering that the root cause of the shortcomings mentioned, in modern manufacturing settings, seems to be an outcome of its own evolutionary path, it's unfeasible to tackle these issues at their source. So, when searching for a solution, we should look at them individually and detach them from the common causality, previously established. This way, we can introduce countermeasures without unsettling the overall balance of the systems, while avoiding the pitfall of oversimplification.

With this in mind, the problem undertaken in this research can be defined as the absence of a tool that promotes scalability, interoperability and adaptability, in CPPSs. The seemingly exponential growth of a system undermines its scalability due to its sheer size and task complexity. Meanwhile, the expansion to accommodate additional equipment from different manufacturers requires universal interoperability. Moreover, the introduction of new and updated machinery limits the adaptability. Lastly, we believe modern tools have been undermined by a trade-off between simplification and functionality, which don't have to be mutally exclusive.

1.4 Objectives and Work Proposal

Previously, we have contextualized the presence of robotics and CPPS within the industrial background, helping us better understand the common genesis of the issues enumerated above. We have also singled these issues out, underlining the essence of what we plan to solve. As such, this research isn't meant to reinvent the wheel, but to provide an alternative that unifies popular methodologies towards the same directives. Thusly, promoting tool-cooperation in lieu of tool replacement.

In light of what has been discussed so far, it has been proposed the idea of combining two prevalent tools used in the I4.0, namely ROS 2 and the IEC 61499 standard.

As such, the plan to fuse these mechanisms aims to merge their benefits, while the individual flaws of each are mitigated by the other's strengths. This coalescence would, mostly, provide developers a means to take advantage of the resulting modularity and flexibility, when conceiving complex robotic applications with distributed control systems. At the same time, it should improve upon other contemporarily available options, regarding the formerly discussed issues.

In essence, the ideal solution should be able to introduce a reform to the concepts of scalability, interoperability and adaptability, present in currently adopted technologies.

Consequently, the main goal of this investigation is to study the viability of how the ROS and the IEC 61499 standard can be integrated, taking into consideration their respective architectures and communication models. In this dissertation, we expand on this idea by equipping an IEC 61499 development environment with the capabilities available in the ROS 2 platform. The proposed solution, KAINOSORE, results from this assimilation, as we will be detailing it in later chapters.

As for the implementation and validation of this software nucleus, we discuss the use case of incorporating it within the DINASORE framework, which is itself compliant with the aforementioned standard.

To achieve this consolidation, an extensive analysis of the ROS, IEC 61499 and DINASORE technologies was conducted, as well as a breakdown of how they're applied during the development of distributed control systems and robotic applications, in modern industrial settings. Related works have also been studied and taken into account, over the course of this endeavor, with emphasis on their advantages and shortcomings.

1.4.1 Research Questions

Finally, we set out to answer a few research questions, in the scope of the proposed work hitherto defined. These questions are:

- Is KAINOSORE capable of performing ROS-specific functions, in simulated and experimental settings, using the FB-based architecture of the IEC 61499 standard?
- How does KAINOSORE uphold the scalability, interoperability and adaptability principles, compared to ROS 2 and IEC 61499?

1.5 Dissertation Structure

In addition to the introduction, this dissertation contains 7 additional chapters.

The following section, Chapter 2 describes, from a theoretical standpoint, the fundamental aspects and tools addressed throughout this research. This is supported by a dive into the historically relevant background, which will help us better understand the full scope of this work and frame the conceptualization of the proposed solution.

In Chapter 3, the state of the art is analyzed through a literature review. Related studies conducted prior are objectively examined with insights into how these scientific papers have contributed to this dissertation.

Sequentially, Chapter 4 thoroughly introduces the core of the proposed solution, starting with the hypothesis that led to the conceptualization of KAINOSORE. Followed by a technical description that addresses the metaphysical architecture and utility. Moreover, we supplement this with an explanation of the design principles and objectives abided by.

Additionally, Chapter 5 provides a description of KAINOSORE's implementation and development, in regards to the use case. This overview consolidates the solution with DINASORE.

Chapter 6 includes a descriptive account of the experimental setup and environment, used for validation. Similar to the previous chapter, this is done in the context of DINASORE as the use case. We conclude this section displaying the obtained results.

In Chapter 7 we discuss the results yielded during the testing phase and, based on them, answer the research questions identified in this introduction. Furthermore, we identify current limitations of KAINOSORE.

Finally, Chapter 8 concludes with a summary of what has been discussed so far and the findings from previous chapters. Additionally, we offer insightful comments surrounding future work.

Chapter 2

Background and Fundamental Aspects

In this chapter, we explore the historically relevant background for the topics addressed throughout this investigation, further expanding on the contextualization given in the introductory Chapter 1. Furthermore, we examine the core fundamentals behind this work, i.e., CPS, ROS and IEC 61499, with an overview of their basis and special emphasisis on their most significant novelties and challenges.

2.1 Historical Background

In accordance with the exposition of the introductory chapter, we can divide the industrial metamorphosis into 4 distinct phases, signaled by the respective industrial revolutions.

The first stage was instituted after the First Industrial Revolution and is commonly associated with the mechanization of labour supported by the rise of hydropower and the steam engine, in the 1780s. At this time, however, the exchange of information and goods was still noticeably underdeveloped, meaning that the exact date and implications of it are difficult to pinpoint. Aditionally, this was an ever-changing period in history with many events to be factored in. As such, the First Industrial Revolution is a topic that has been vastly explored throughout history, usually emphasized by the dissimilarities between the form it took and its impact across diferent countries and areas of investigation. Still, despite the vast improvements evident in hindsight, most countries faced some reservations from the populace at the time [41], [15], [45], [25], [46]. Regardless, this flags the beginning of a trend, in which, to a certain degree, conceptual abstraction is present in posterior revolutions, as well as the standardization of machinery in the factory system.

Similarly, the Second Industrial Revolution succeds it, by extending the norms introduced previously into a growing manufactory setting. Once again, its timeline has been the subject of intellectual debate, but the 1870s are generally accepted as the dawn of this era, marked by the scientific and technical advancements in power supply and telecommunications. Earning it the title of Technological Revolution, the emergence of these innovations set the groundwork for the evolution of the factory model into one of mass production and standardization with assembly lines and electricity-powered machines [19], [46]. Once again, besides the objectively positive effects

we're aware of today, this movement met some resistance from the public, but also created some concerns from an industrial standpoint. Some of these concerns were due to the drastic changes that had positively affected the manufacturing efficiency, resulting in overproduction, which has its own implications [22]. Thus, setting up a pattern, still visible in today's industry, of drawbacks appearing as a direct result of rapid and radical improvements in the growing workflow. Moreover, during this time, a strategy of splitting a system into separate behavioural segments was officially introduced in the factory layout, eventually evolving into our current concept of decentralized and event-driven architectures.

So far, these revolutions have been centered around improvements to the physical aspect of the industry. In contrast, the Third Industrial Revolution emerges from a postwar society, with a major focus on information technology (IT) for the processing and communication of information. Ergo, this period became known as "Information Age", among other titles. Characterized by the digitalization and automation of previously analog and manual processes, the Third Industrial Revolution was supported by some key inventions, such as the transistor, the Programmable Logic Controller (PLC) and the Internet. This had a significant impact on the *modus operandi* of the factory system and the robotic elements that composed it, now overhauled with substantial improvements. Alas, as expected from previous revolutions, it also brought some turmoil with the sudden industrial revamp [19], [10]. Nevertheless, it gave the production world access to global distribution and automated robotics.

2.1.1 Industry 4.0

Lastly, we will be discussing the modern era, as the successor to the stages imposed by the first, second and third industrial revolutions. At the time of writing, this is the ongoing phase of industrial shift, but, unlike past events, this is marked by both the 4IR and the Fifth Industrial Revolution (5IR) [4]. It starts in the beginning of the 21st century with the introduction of the I4.0, which came to be known as the Intelligence Age and was, shortly after, followed by the I5.0, resulting in the coexistence of these two notions. Inheriting an aspect of conceptual abstraction from its predecessors, this era is characterized by the merge of the physical, biological and digital dimensions [42]. This is accomplished through the combination of its components, minimizing risks and maximizing performance, for example a CPS with AI capabilities. The level of complexity and flexibility, of the systems it employs, is a hallmark of the stages of its growth, but may also be a hindrance [28], [40], as we've seen in prior revolutions and reflected in Chapter 1. Additionally, the digital and automated protocols used are an evolution stemming from their technological ancestors, with concepts such as the standardization of machinery evolving into advanced robotics with integrated AI and mass machine-to-machine (M2M) communication in an Internet of Things (IoT) structure [6], [30]. Extensive studies and reviews have been conducted to define design principles of I4.0 [43], [20], [21]. Accordingly, in this dissertation, we assume the categorization under 7 axioms, as Figure 2.1 depicts. All in all, this stage is a continuity of the digitalization roadmap, with many key fundamentals, but in the context of this investigation, none is more relevant than the CPS.

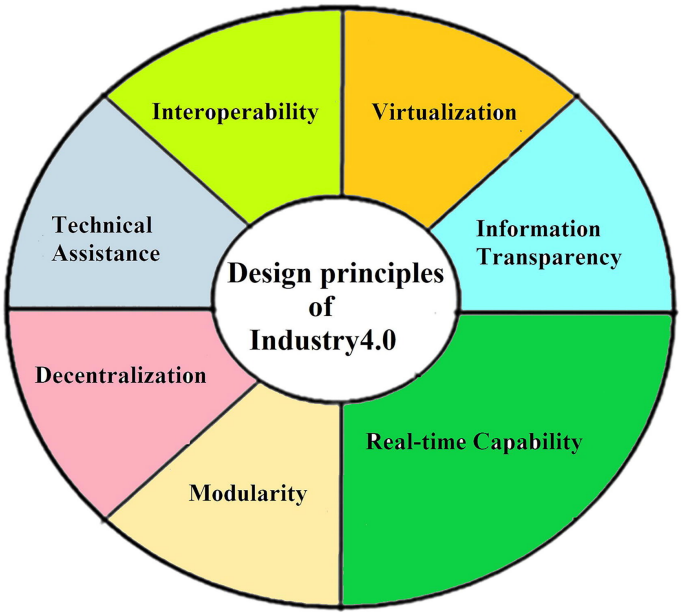


Figure 2.1: Design Principles of Industry 4.0. [43]

2.2 Cyber-Physical Systems

Often embedded into the industrial model of the 4IR, the CPS forms the link between the physical and digital planes by relying on a network of virtual entities that manage and control physical processes. This consolidation emphasizes the autonomy and virtual aspects of modern industrial scenarios, which has contributed to the speculation of its potential [28], [5], [26]. Even though the CPS is an emerging approach in I4.0 [53], based on what's been examined so far, we can trace its origins back to different stages of the industrialization course. Furthermore, this system can be represented as the culmination of the design principles of interoperability, virtualization, information transparency, real-time capability, modularity, decentralization and technical assistance, prevalent in modern industries.

The collaborative nature of a CPS is supported by the collection of sensors and actuators that represent its mechanical actions as input and output data to be interpreted by the control unit, as Figure 2.2 illustrates. As they're usually designed with a decentralized architecture, the digital entity is comprised of multiple interconnected PLCs, whereas the physical layer integrates robotic devices. Additionally, these controllers must be able to communicate with each other and their physical counterparts, using this information exchange as the foundation for event-driven operations and real-time decision-making. Moreover, by creating a feedback loop through data transfer and analytics, the process is able to carry self-optimization and security features.

This philosophy of singular, yet intertwined, elements with a decentralized design acts as a testament of the system's modularity, which has many advantages on its own, but also generates interoperability-wise concerns. Nevertheless, it's important to note that this characteristic is an integral part of the CPS. The modular aspect of this design allows the mechanisms to be controlled and supervised, in tandem, by programmable microcontrollers, operating in real-time and

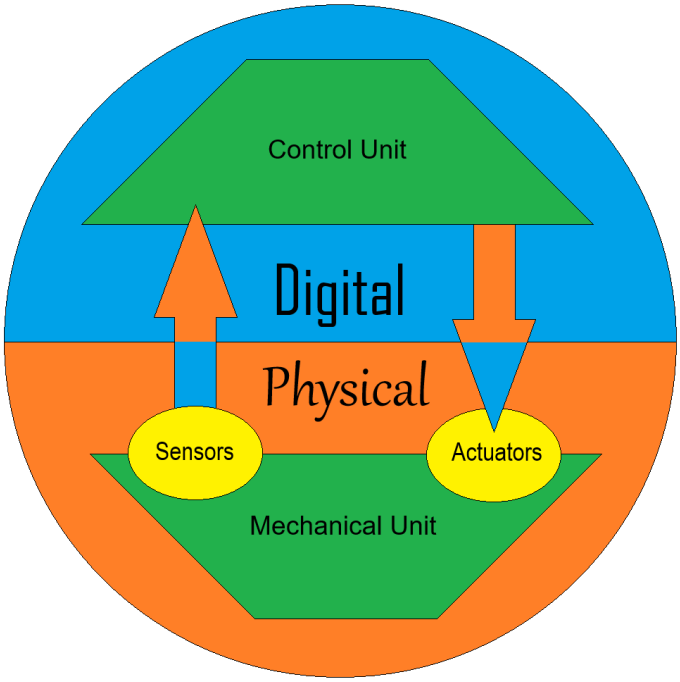


Figure 2.2: General structure of a Cyber-Physical System with Sensors and Actuators to connect the Digital and Physical planes.

providing supplementary fail-safe capabilities, in which a system detects and acts on a specific faulty element, without disrupting the rest of the workflow. In addition to this, the structure can be future-proof and scalable, with the implementation of modifications and upgrades at a component level.

Furthermore, the connections established in a CPS, can be implemented as wired for faster and more stable communications or wireless for increased flexibility and mobility. The latter is able to bypass certain limitations of the former, imposed by physical layout and resource availability constraints, by linking with external servers or cloud environments. In more advanced arrangements, this wireless configuration can even be connected to extrinsic structures, creating a complex CPS composed of smaller ones. In such cases, the control unit of each independent system is able to communicate with the others, similarly functioning as a whole, while retaining its modular composition.

2.2.1 Cyber-Physical Production Systems

Considering the concept of the CPS, in light of the investigation conducted in this work, we can specify it as the CPPS. This is a paradigm of the former, but exclusively directed towards the production and manufacturing processes [14]. Regardless, what has been discussed is still applicable and it complies with the previously categorized design conventions of I4.0, as follows:

- Interoperability - Elements of the control platform communicate with each other and (through sensors and actuators) interact with the mechanical unit.

- Virtualization - An integration of digital entities within a mechanical scenario contributes to the virtualization of the manufacturing workflow.
- Information Transparency - Supported by intrinsic connections, transmitted data is available as often and as precisely, as necessary.
- Real-time Capability - Software implemented in the controllers enables real-time decision-making through data analysis.
- Modularity - A division of labour, in both virtual and physical units, is critical for these systems, providing resilience and scalability.
- Decentralization - Control units are designed and implemented with distributed architectures, manifested as interconnected devices.
- Technical Assistance - By separating the controller section from the mechanical structure, humans and physical processes can synergize without the risks of direct intervention.

As these aspects converge, the production process incorporates a large set of measures for configuration and application diversity, emphasizing its flexibility and dynamism. However, this trait inheritance also translates into issues of the CPS being passed down to the CPPS sector. A decentralized and modular architecture, despite displaying numerous advantages, requires, by definition, connectable devices. Moreover, these components tend to be peculiar in some way or another, as they often come from different manufacturers. As such, the integrity of an encompassing dynamic system is harder to guarantee, pressured by its complexity and lack of universally adopted standards [28], [30], [5].

These issues are part of a greater problem plaguing modern robotics and have long been subject of academic review [8]. Consequently, various solutions have been proposed, but fewer have been deployed as a functionally stable and widely adopted tool. This is, mostly, due to the complexity that these frameworks incorporate and the continuous developments they must undergo, as part of a never-ending CPS evolution. So, in order to respond to these exigencies, open-source approaches were often promoted as the next logical step [8].

2.3 Robot Operating System

In these cirumstances, ROS[1][2] emerged [39] and stood out above the rest, rapidly growing in reputation and development, soon becoming an indispensible tool in automation [3].

Despite its name, ROS was designed, as a middleware, to handle communications of a heterogeneous computer cluster, outside the hosts' operating systems. As such, it relies on a large set of tools, libraries and conventions to streamline the efforts that go into designing, developing and controlling complex robotic systems (in simulated and real environments). Supporting this idea,

[1] https://docs.ros.org/
[2] https://index.ros.org/

hardware abstraction, low-level device control, interprocess communication (IPC) and package management are highlighted benefits of this framework[39], [35].

As a free and open-source software, ROS thrives on community contributions, ensuring the robustness and versatility it's become known for. Consequently, anyone, regardless of their technological background, can have access to the capabilities it provides, which, even in commercial projects, prompts its status as a norm.

The non-proprietary model is one of the philosophies specified per its design [39]. ROS complements this with multi-lingual support, allowing users to collaborate with each other, in spite of personal preferences and knowledge-based limitations in certain laguages. Extending even to processes scripted with different languages communicating through this language-neutral middleware without any setback. Moreover, in an attempt to follow the modern trend of reusability, it depends on libraries to incorporate higher degrees of complexity, thus unburdening the main build as thin and lightweight. Likewise, a modular and tools-based approach helps the management of intricacies in ROS, similar to the modularity's role in CPS, as it's been discussed previously. Finally, the peer-to-peer (P2P) topology reinforces its reliability in multi-host heterogeneous clusters.

2.3.1 Architecture

At its core, ROS architecture follows a simple pipeline of communicating elements, most notably nodes, messages, topics and services [39], [35]. These notions and their interdependence can be summarized as:

- Nodes - These are the components that communicate with each other and execute computational tasks. They are also regarded as a virtual representation of a system's modularity.
- Messages - As the means via which nodes exchange information, messages are deliberately designed data structures.
- Topics - When any node subscribes to any topic, as a subscriber it receives messages being published to that same topic by other nodes (publishers). This is the essence of the publisher-subscriber connection.
- Services - Following a similar system to topic-based interactions, services stand out as a synchronous alternative. A client inputs a message as a request and the server promptly outputs the response message.

2.3.2 Robot Operating System 2

Even though ROS, in its own right, was a successful advancement in robotics with consistent revision, there were still some inherent issues it was unable to outgrow. As such, ROS 2[3] [32] was announced, with the same general goal, basic philosophies and architecture, examined above,

[3] https://github.com/ros2

as its backbone. Nevertheless, the improvements to be made were so vast, the system had to be completely revamped and ROS 2 was programmed from scratch [47], [23].

This development is enhanced with key new features to better accommodate modern industrial growth, such as cross-platform and multi-robot system support, supplemented by higher efficiency, reliability and security, as well as real-time properties [47], [23]. Ergo, this second generation of software placed substantial importance in design principles that abide by the CPS norms previously examined. This is corroborated in [32], describing its distributed architecture as a decentralized and secure response to advanced systems' complexity. Another axiom defined is the level of abstraction it employs to withstand interoperability demands. In addition to this, asynchrony is emphasized in communications to comply with event-driven systems and modularity is adopted throughout multiple stages of the framework.

Regarding the communication pipeline of ROS, as seen above, ROS 2 supplements it by expanding the concept of actions. This was a later addition to ROS, but it's only in the second generation middleware that it became a more prominent element. Like topics, these are, by definition, asynchronous communication interfaces. But, whereas topics focus solely on data exchange, actions are goal-oriented and offer more control through a complex data structure with feedback and the ability to be canceled [32]. This new layout is examplified in Figure 2.3. Furthermore, this overhaul saw significant improve in cybersecurity with the adoption of the Data Distribution Service (DDS) [34] middleware, along with its own Secure ROS 2 (SROS2) module [29].

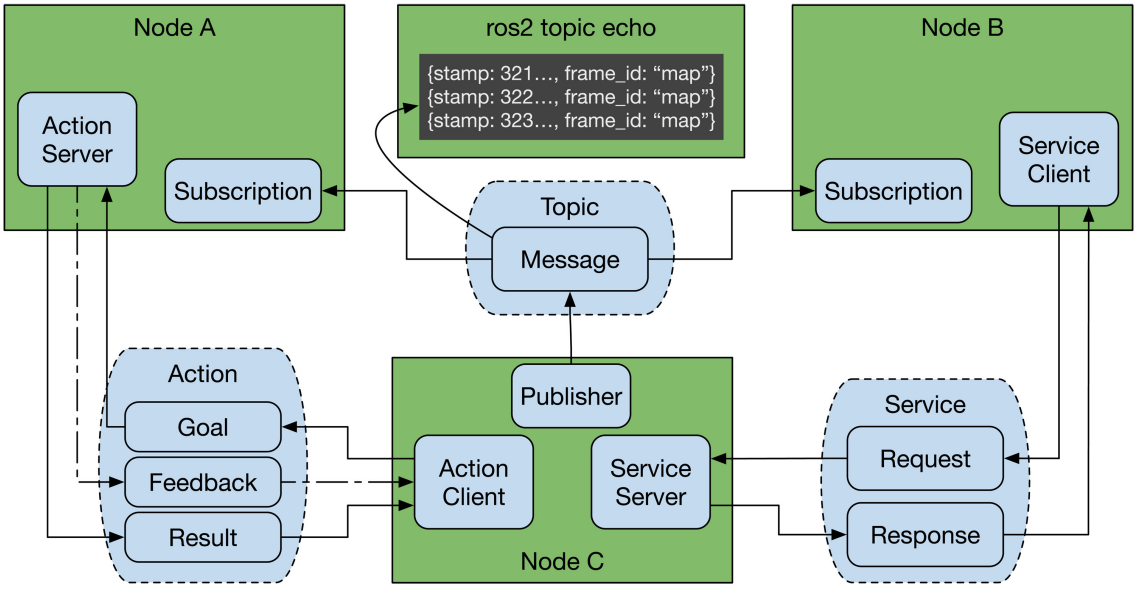


Figure 2.3: ROS 2 node interfaces: Topics, services, and actions. [32]

2.3.3 Pros and Cons

In summary, ROS 2 facilitates the design, management and implementation of large-scale CPPS. It benefits from hardware abstraction and a distributed structure, among other principles, to reliably provide reusability of developed software. Along with improved efficiency, it increases a system's

scalability, making it ideal for low-cost systems that don't require high-end equipment. As such, it has been widely adopted, evolving as a vital framework in robotics. Moreover, as a free and open-source tool, the collaborative nature fostered by its community has also played a significant role, increasing the scope of capabilities it provides. Correspondingly, the interoperability of a CPPS can be enhanced further with the integration of ROS 2.

Considering what has been discussed thus far, ROS 2 presents a significant upgrade to its predecessor, providing a far more robust and complete system. However, despite its efforts to adapt to the ever-changing setting of I4.0, some challenges yet persist, deep-rooted. Most notably, the level of complexity is one of these issues.

Even though it has simplified the layout, control and deployment processes, this tool is still far from simple, itself. The growth of ROS 2, as a framework, allowed it to, more fittingly, accommodate the CPPS's expectations and demands at the cost of its own complexity. This has resulted in a steep learning curve for newer users and an arduous effort for current ones to transition from ROS to ROS 2 or between releases. Ergo, this software's complexity thwarts the proficiency of its users, which also negatively impacts how it handles scalability.

For the remainder of this dissertation, the terms ROS and ROS 2 will be used interchangeably, and, unless explicitly indicated, will refer to ROS 2 as described in this chapter.

2.4 IEC 61499 Standard

Prior to the development of ROS, the International Electrotechinal Comission (IEC) introduced the IEC 61499 Standard [12] [11]. It incorporates conventions and solutions meant to aptly react to challenges of distributed and decentralized systems. Furthermore, similar to ROS, this standard is detached from hardware manufacturers and vendors, which helps sustain a broader public adoption.

As a set of guidelines, it can be integrated into CPPSs and, through the level of software abstraction it enables, has a nonintrusive nature. Accordingly, both in industrial and research environments, this international standard has become well-established.

At its core, the adoption of IEC 61499, in a CPPS, seeks to contribute to the portability, configurability and interoperability of these structures, as Figure 2.4 depicts. Portability ensures that the Software Tools employed in a system are capable of properly using elements and configurations from the Software Libraries compliant with the IEC 61499 standard. Whereas configurability allows for plenty of leeway during the configuration of the devices equipped, through the Software Tools that use the standard-approved Software Libraries. Finally, interoperability guarantees that separate components are able to seamlessly communicate and cooperate with each other.

2.4.1 Architecture

Abiding by the norms set in IEC 61499, a CPPS features a modular architecture, in which the virtual base unit is the function block (FB). As defined by the standard [11], several types of FBs

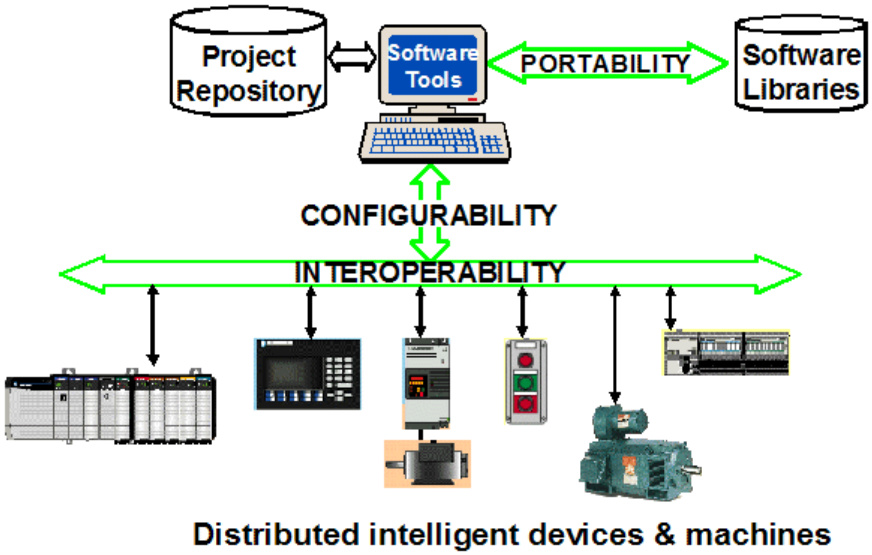


Figure 2.4: Qualities of Open Automation and Control Systems
(Source: [12])

are available, each with different objectives. However, in this investigation, we focus solely on the basic function block (BFB).

These FBs, as illustrated in Figure 2.5, are event-driven reusable components that handle machine states and behaviour. Software developers, adhering to this convention, incorporate their computer code into these modules when designing CPPSs. And, should one of these systems require reconfiguration, these FBs are the means via which the developers do so. Supported by the IEC 61499 standard, a pipeline is created by linking multiple FBs through event and data connections. Moreover, the functionality of the BFB can be split into two different sections: i) the Execution Control Chart (ECC); and ii) the Algorithm.

(i) The ECC is the portion to which event inputs are sent, often connected to the event output of other FBs. It handles state transitions and, sequentially, triggers the execution of the software code implemented in the FB. After it concludes, an event output is usually sent, signaling the completion of the execution.

(ii) The Algorithm encapsulates programmed behaviour through computer processes and algorithms. Prompted by the ECC, it carries out a given operation, often complemented with data inputs. However, it's important to highlight that the processes it integrates only have access to internal data. As such, they're unable to communicate with separate components asynchronous to event edges. The functions, implemented so, may produce data outputs in response.

Regarding the event and data connections, a pair of rules tend to stand out and are particularly important to mention. Firstly, the standard specifies that any given FB must only receive, at most, a single event input in any given time instant. Additionally, the other requirement is that FBs must

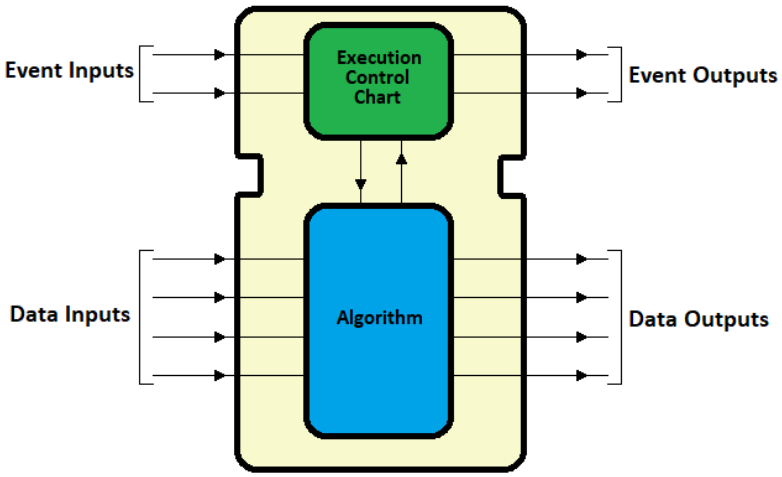


Figure 2.5: Overview of a generic BFB

be deployed with the data connections directly linked with the events that use them. As such, the operations that model a FB application are executed orderly and consistently.

In the end, adopting the standard in question, helps simplify the whole process behind the design, management and implementation of distributed and decentralized control systems. The focus it places on portability and interoperability, is the perfect fit for a nonintrusive synergetic approach. Further emphasis on configurability adds to the influence and control users have. Its FB-based architecture is the hallmark of resuability and straightforwardness that keep it quite user-friendly. However, the level of simplicity provided compromises the flexibility of the system using it.

To ensure that these conventions are followed when plotting, overseeing and deploying a CPPS, it's essential to use an Integrated Development Environment (IDE) compliant with the IEC 61499 standard. Furthermore, these must be equipped with a Runtime Environment (RTE) capable of executing the FB pipeline and may support Software Development Kits (SDKs) to assist with interoperability.

2.4.2 4DIAC-IDE

Among a variety of available software that supports IEC 61499, in the present dissertation, we distinguish the 4DIAC-IDE [44], as an open-source tool under the Eclipse framework.

This IDE enables the creation of FB-oriented structures via a Graphical User Interface (GUI) with a drag & drop display. As such, the design and deployment of FB pipelines in the 4DIAC-IDE is simple, yet thorough, with visual representations for each FB and their event and data connections. Consequently, this makes it ideal for improved monitoring and control of the workflow.

Moreover, it uses XML-based definitions to represent the FBs, allowing them to be ported, to and from, different environments.

Likewise, the 4DIAC project also integrates the IEC 61499 compliant RTE, FORTE. This is the tool in charge of executing the designed system under the IEC 61499 standard rules. Adequately, it follows a sequential event-driven model and has been implemented in an object-oriented language (C++), placing major focus on portability and operability in smaller embedded controllers.

In addition to this, the DINASORE[4] framework will be used for implementation and validation purposes, as it uses 4DIAC and FORTE to comply with the standard.

2.4.3 DINASORE

DINASORE[36] is a particularly appropriate middleware for this investigation. It relies on the 4DIAC and FORTE tools to provide the IDE and RTE, respectively, and complements them with a simplified method for the development of FBs. To accomplish this, DINASORE is implemented in *Python*, which has become one of the most common, if not the most commonly used programming languages worldwide. As such, it allows the behaviour of the FBs it executes to be coded in *Python*, hence, it provides a more accessible alternative to the public, while also enabling the integration of Machine Learning (ML), and other advanced algorithms, in FB-based structures. Additionally, DINASORE implements Open Platform Communications - Unified Architecture (OPC-UA) that enables an easier integration with third-party software via automatically generated data models.

Furthermore, DINASORE corroborates the philosophy of the IEC 61499 standard that has been discussed thus far. It boosts portability as a *Python* environment, designed for both embedded and non-embedded systems, in accordance with the FORTE implementation, which is itself portable. Configurability is another major focus and is upheld per the straightforward and easily understandable *Python* FB paradigm, as well as the ability to reconfigure and moniter them during runtime. As for the interoperability, it uses the 4DIAC project and OPC-UA server integration. Allowing the deployment of a FB pipeline across multiple edge devices, running DINASORE instances. These FB clusters, are composed of BFBs, which is the main type of FB supported in this setup.

Taking this into account, an user is able to easily design, manage and implement a FB pipeline as desired. This development process starts with the creation of two files for each FB. One of them contains the FB's structural metadata via Extensible Markup Language (XML) tags, whereas the other includes the full *Python* implementation dictating the FB's functionalities. Accordingly, the XML file is used inside a 4DIAC workspace to generate the graphical representation of the associated FB, which can then be used to connect to other FBs through the event and data, inputs and outpus. These FBs must all be correctly mapped to the device that will be running the DINASORE instance, and the DINASORE environment must be able to access the corresponding

[4] https://github.com/DIGI2-FEUP/dinasore/wiki

XML and *Python* files previously created. The mapping is available in the 4DIAC's GUI and provides easy management and information transparency, as well as the option to deploy the structure, which will be executed by the running DINASORE.

Table 2.1: Experimental results with average grade [37]

Indicator	**Current**	**DINASORE**
Requirements Achievement	3.0 (Neutral)	4.0 (Agree)
Learnability	2.0 (Disagree)	4.0 (Agree)
Understandability	2.0 (Disagree)	3.0 (Neutral)
User Attraction	1.0 (Strongly disagree)	4.0 (Agree)
Efficiency	2.0 (Disagree)	3.0 (Neutral)
Flexibility	2.0 (Disagree)	4.0 (Agree)
Maturity	3.0 (Neutral)	3.0 (Neutral)

In addition to this, it's important to mention that this tool has been used in research environments and in the case study [37] it was applied in a real industrial environment. In this project, Pinto et al. propose using DINASORE as an alternative to the, at the time, employed monitoring system, which was effortful and time-consuming. As such, this study is complemented by a comparative analysis, with a set of questions in the form of interviews, which resulted in the table 2.1. This is relevant as it provides valuable insight into the public's opinion when compared to a well-established industrial norm.

Chapter 3

Literature Review

The topic of using ROS/ROS 2 applications, in a model-based architecture to manipulate robots, has very limited research to it. Furthermore, whithin this scientific niche, using IEC 61499 compliant FBs as the modular element to build these structures is considerably more underexplored.

In this chapter, we provide a comprehensive review of related projects developed around this subject. Additionally, we complement each analysis with a an impartial overview of its contributions and shortcomings, focusing on its relevance to this dissertation.

3.1 Mapping Approach

Walter and Grave, in the article [48], explore this topic with a ROS-to-IEC 61499 mapping approach. This idea stands out as the authors highlight how it relies on existing IEC 61499 utility to alleviate learning and integration efforts.

It starts by identifying how the lack of an inherent model-based design in ROS is an encumbrance to developers and jeopardizes maintainability. Adding that alternatives (like SysML[50]) do exist, but rely too heavily on language and conventions specific to them. Besides that, these tools are also mentioned as they fail to provide clarity, which would help counteract the complexity associated with ROS-absed application development. Moreover, the authors underline issues that stem from the limited study that has been done conducted in model-based solutions. These are, for example, the scarcity of published papers and their narrowed applications and testing.

Sequentially, they explore the IEC 61499 standard and draw parallels between its architecture and the ROS communication paradigm. At this point, Walter and Grave establish this as the hypothesis behind their approach, emphasizing the mapping procedure as an unilateral adaptation of ROS to IEC 61499. Afterwards, the authors follow with a comprehensive description of how basic ROS elements can, adequately, be manually mapped to their IEC 61499 equivalent. This is corroborated with a clear focus on the similarities of the communication frameworks. However, this is only applicable to the design of the structure. As such, this analysis is succeeded by a clarification of the implementation methodology. In this section, the authors stress how much this

solution relies on automatic code generation and provide 4 distinct options users can choose from. These differ regarding the distribution of effort, between man-made and autonomous techniques.

We agree with the authors on many aspects and recognize the benefits of this method, nevertheless, there are some concerns we feel compelled to raise. Firstly, while the irreciprocal mapping may help overcome design obstacles, it also undermines the native qualities of stand-alone ROS, by limiting its capabilities to the standard's boundaries. Additionally, this technique facilitates the layout of CPPSs via an unambiguous visual representation of the system, but the manual modelling still requires a relatively high degree of ROS expertise. While avoiding additional layers and relying on the innate IEC 61499 utility as simplification mechanisms is commendable, this also restricts ROS and ignores the integration of middleware, per software abstraction, that doesn't demand a new know-how. Furthermore, this approach depends on automatic code generation, distancing the user from the process, which could create a barrier for debugging. However, it contributes greatly with an established correspondence between ROS and IEC 61499 traits.

3.2 ReApp Workbench

Another alternative is introduced by Wenger et al., in [49], with the ReApp project. This is a quintuple tool that seeks to provide a platform for "creating, applying and managing resuable robotics software componnents". These components are designated as 'Apps' and a notable type of App is the 'Skill', which represents ROS nodes' compositions and connections. However, published in 2016, it only targets ROS, as in the predecessor to ROS 2, for the latter hadn't been officially released at the time.

Nevertheless, the authors describe issues still relevant by today's standards, namely, the general absence of interoperability and the labour-intensive processes to configure and deploy robotic applications. This is highlighted as being especially detrimental to smaller environments. Furthermore, several already existing approaches are reviewed, mostly ROS-based ones, of which none are compliant with the IEC 61499 standard.

Wenger et al. expand on the workflow and dynamics of the 5 components that shape this tool. Notably, a cloud-based repository, of previously programmed applications, is used to integrate ROS functionalities with reduced complexity. This creates a graphical layout and a corresponding file for lauchning ROS nodes, promoting reusability. Automated code and model generation complements the previous process to extend the scope of capabilites provided, which is uploadable to the cloud-based ReApp Store [7] repository. At this point, the automatically created elements can be transferred to the ReApp's IDE for integration with the IEC 61499.

The authors emphasize the main objective of this consolidation to be the enhancement of ROS with the realtime behaviour supported in IEC 61499. As such, ReApp uses the realtime environment FORTE, available as part of the 4DIAC project explored in Chapter 2. Users proceed to add the IEC 61499 compliant logic, that will handle machine behaviour, to the C++ code skeleton generated before. This results in an editable FB pipeline, in the form of the skill-based architecture native to ReApp, where each FB maps to ROS components, as Figure 3.1 illustrates. Moreover,

this framework offers an interface for the deployment and configuration of the structure, which also allows parallel ROS and FORTE functionality.

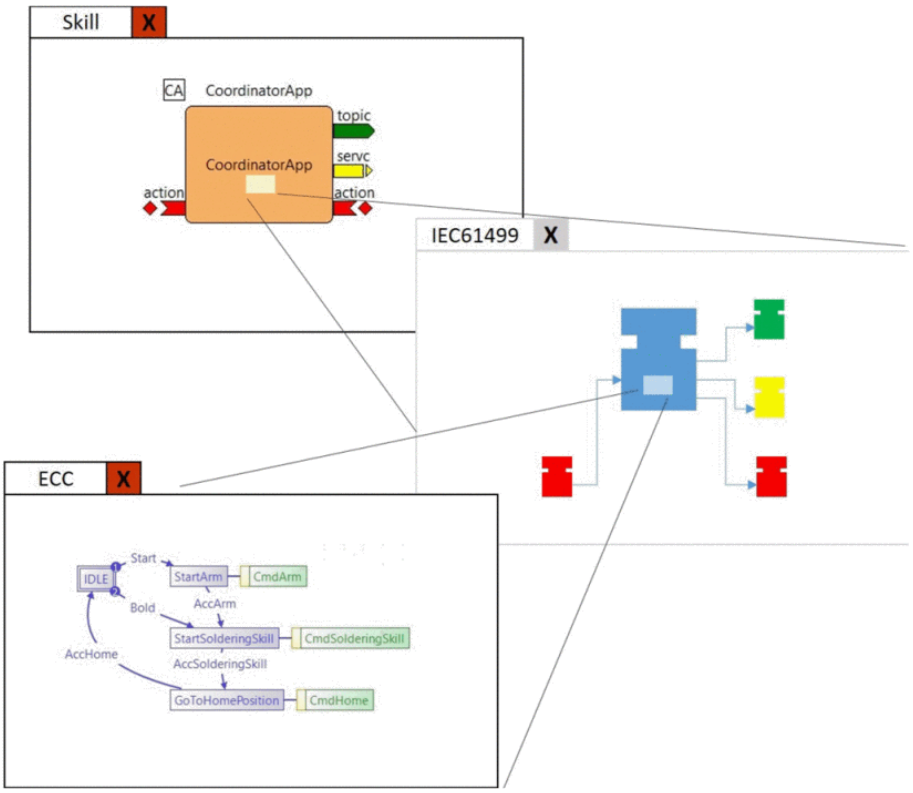


Figure 3.1: Editors for the programming of nodes (blue) according to the IEC 61499 standard [49]

The ReApp workbench introduces a platform for mapping dynamically created IEC 61499 FBs to ROS components. It manages to conceal the complexity of ROS, via software abstraction, making it easier for unfamiliarized users to benefit from ROS utilities. However, these FBs are implemented internally to represent the modules that will be used in the graphical diagram of ReApp's IDE. This means that IEC 61499 simply acts as an intermediary with a focus on providing realtime capabilities to ROS structures. Ergo, users must still undergo a familiarization process with this tool, regardless of their ROS and IEC 61499 skill set. Furthermore, a methodology specific to a single workbench weakens the portability factor of this approach. While the integrated constituents of this tool are able to cooperate aptly and uphold the intended integration, the whole process relies heavily on autonomous code generation. This relieves part of the design effort for developers, but distances them from the creation process, which suggests an obstacle for manual reconfiguration and debugging.

3.3 Additional Works

Both projects examined have focused on providing a generic mechanism of parallel integration, through different methods of mapping ROS to IEC 61499. Nevertheless, a few studies deviate

from this norm and address the consolidation differently.

3.3.1 Sequential Integration

The paper [24] recommends a methodology for workflow and task management through the IEC 61499 Standard, assisted by ROS for implementation purposes. Once again, this pertains to the first generation of ROS. This technique is particular, since it integrates ROS and IEC 61499 sequentially, with robotic cells. Accordingly, Iannacci et al. highlight the importance of having an intuitive interface available, capable of facilitating robot integration, as a response to the increased complexity of modern systems. As such, the proposed solution advances with the introduction of a new FB type, preserving reusable and modular qualities, as a means of contributing to the abstraction and scalability properties of the structure. These FBs are closely based on preexistent FB types inheriting their IEC 61499 compliance and are used to handle the workflow of ROS-operated robots. In this setting, the adaptability and dynamism of this method are highlighted, for the ROS control can be modified without affecting the rest of the structure, as much as the management FBs can be reconfigured without interfering with the ROS nodes. The authors validate this solution with a real use case implementation, emphasizing the modularity and simplification contributions. However, this approach is focused on the management process and doesn't apply to the design and deployment sectors we wish to impact. Nevertheless, it provides relevant insight into the capabilities of IEC 61499 management, in CPPS, linked to ROS for robot manipulation, promoting adaptability.

3.3.2 Specific Mapping

In [17] and [16], the same solution is used and tested with a low-cost single-board computer device controlling a robotic arm for a pick and place operation and a welding system, respectively. This integration relies on specific mapping to connect ROS to IEC 61499. The authors implement a control structure comprised of FBs, that fall under two distinct categories, to form the pipeline. The external design and later deployment of this architecture is done with the 4DIAC, whereas the algorithms are programmed in C++ files. As the first subset of FBs, the so-called Robot Operation FBs are responsible for interacting with the mechanical unit via exchange of ROS messages. This subset can be divided further into readers and writers FBs. The former reads data from each sensor, published in ROS messages, and decodes it into values (in this case, of type REAL) that can be transmitted in IEC 61499 connections to other FBs. On the other hand, writers have a similar structure, but function in reverse, receiving values from IEC 61499 FBs and coding them back to ROS messages, that are then published to the actuators, triggering the robotic arm movement. Between this read/write ROS mechanism, the second subset, Kinematic Controller FBs, is integrated. This handles the actual logic that defines the control of the robot, based on its kinematic model. Even though the mesh of ROS and IEC 61499 capabilities has been successful, the lack of abstraction in this approach compromises interoperability and doesn't tackle the design complexity issue. Nonetheless, this still contributes to facilitate the management process and, compared to

the other analysed works, is the most well tested solution, even being expanded in a later project [9].

3.4 Conclusions

In summary, ROS-based robot manipulation using the IEC 61499 standard is a subject marked by a general shortage of conducted research and developed projects. Moreover, there has been a significant lack of quantitative tests performed and discussed. Amongst the reviewed literature, no single paper seems to offer improvements to the scalability, interoperability and adaptability, while focusing on the design, management and implementation processes simultaneously, usually opting for a quid pro quo alternative. Additionally, most of the work developed around this topic is still aimed at the first version of ROS, which, at the time of writing, has been discontinued. Although these contributions are nevertheless relevant in some way, since a large part of ROS concepts have progressed to ROS 2, this shows that the complexity levels of the latter have become more demanding to unravel.

Out of the projects analyzed, the papers that utilize some form of direct mapping, in their proposal, end up optimizing the implementation of CPPSs, unlike the [24]. However, the generic mapping of [48] and [49], limits the capabilities of standalone ROS. While relying too heavily on automatic code generation may assist the design process, by reducing the expected complexity, it also incurs some undesirable costs. On the one hand, it overly diminishes the human involvement, which weakens the authority and control we have over the system. On the other hand, it isolates the design and deployment processes, as well, creating barriers when it comes to scalability and reconfigurability. In contrast, solutions that impose manual design, tend to lack hardware abstraction and center around very specific use cases.

In general, these projects appear to often rely on a trade-off between the properties of ROS and IEC 61499. Either imposing the limitaions of the standard on ROS for a parallel integration, or using IEC 61499 as a means to manage ROS-based operations without a significant reduction to its complexity. Furthermore, we believe that a typical flaw is the heavy focus on the current state of I4.0 disregarding the teachings and patterns from its historical progress. This is corroborated by the concentration of these developments in seeking tool replacement instead of tool cooperation. Another common shortcoming is, noticeably, not taking full advantage of software and hardware abstractions to allow for greater flexibility.

Chapter 4

Proposed Solution

This chapter concentrates in a theoretical view, as we set up the hypothesis that led us to the proposed solution. Initially, we underpin this basis with a brief review of previously evidenced conclusions. Sequentially, we elaborate on this idea and present the concept of the KAINOSORE approach. This includes an explanation of its structure and utility, assisted by some generic examples. Moreover, we highlight how it differentiates from regular ROS 2 or IEC 61499 applications, emphasizing its cooperative essence. Additionally, we provide an intricate description of the design principles and objectives defined for the development of this approach, as well as how KAINOSORE seeks to realize them.

4.1 Hypothesis

Taking into account all that has been analysed thus far, we recognize that the expansion and magnitude of the industrial process are the biggest obstacles towards a single perfectly functional and universally adopted tool. However, when examining from a solution seeker stance, it's important to avoid the pitfalls of oversimplification and overcomplication. As such, we recognize that the idea of a flawless and complete technique on its own, would be purely utopian. Nevertheless, we also acknowledge the advancements and contributions of current frameworks.

KAINOSORE, the solution described in this investigation, stems from these beliefs and pursues an ideal balance that contemporary technologies have failed to achieve. To do so, we have delved into ROS 2 and IEC 61499, looking to merge them under a single unifying structure capable of mitigating their weaknesses by emphasizing their strengths.

4.2 KAINOSORE Approach

Despite seldomly being integrated under a single system, ROS 2 and IEC 61499 share some key similarities that support a seamless consolidation between both. Besides, these tools aren't mutually exclusive and provide enough software abstraction as to avoid disrupting each other's operations. Consequently, numerous approaches appear to be theoretically feasible.

In this endeavour, we present one of these methods, as KAINOSORE. This can be described as an SDK-like adaptation of ROS 2 elements in a FB-oriented architecture compliant with the IEC 61499 standard. As a result, our focus has been primarily directed towards the development of a set of libraries, that users can rely on, as they would with other SDKs, to execute ROS-specific functionalities, inside the FB's implementation code.

Structurally, KAINOSORE follows the same FB-based architecture as other IEC 61499 compliant tools. Using custom-made provided or created FB models along with already existing ones, one can design and implement a whole pipeline. Connecting FBs via events and data links also ensures the sequential event-driven execution paradigm defined by the standard. Likewise, the computational behaviour of each FB is written as an event-driven component, nevertheless, developers can opt to add ROS 2 configurations, such as message publishers and subscribers or service clients. These functionalities are provided with a high degree of software abstraction, internally handling ROS 2 code-related interpretations. In addition to this, KAINOSORE provides templates and alternatives for a wide range of regularly used ROS 2 capabilities.

KAINOSORE can even be used as a common factor to link both approaches. So, instead of assuming a single technique, users can design a system with dual-control (i.e., ROS 2 and IEC 61499 jointly controlling the same system). This gives developers greater authority over resource distribution and offers unparalleled cooperation between users with different skill sets. As a result, a team of engineers can design, manage and implement a dual-control CPPS, even if its members are split between ROS 2 and IEC 61499 proficiency-wise.

Furthermore, this implementation aims to exploit the inherently implied synergies between ROS 2 and IEC 61499. Due to the collaborative nature of KAINOSORE, we believe that the strengths of each software can be used to overcome the limitations of the other, resulting in a mostly advantageous system.

As we've discussed in previous chapters, ROS 2 operates with a distributed architecture, as do IEC 61499 compliant systems, pointing to a possible connection. However, whereas the IEC norms establish data exchange via data links, the ROS 2 communication protocol relies on messages. This variation is extremely relevant, since the data inputs and outputs of the former are very restrictive, only allowing for a specific data type in each link. Besides that, the standard imposes an additional limitation, by confining the transmission of information to an event-driven scheme. In contrast, a message type data structure in ROS 2 can be composed of several data types including nested messages. Additionally, ROS 2 nodes can publish and subsribe data asynchronously, over the publisher-subscriber connection. As such, KAINOSORE takes advantage of this, by allowing a developer to select the preferred method of data exchange, when designing the FB's algorithm, for each specific case. This can be either asynchronously via a publisher-subscriber connection, event-driven with the ROS 2 message structure or simply event-driven single type input and output.

Another parallel between ROS 2 and IEC 61499 is that, even though both seek to facilitate the design, management and implementation of large-scale CPPSs, the extent to which they accomplish this, varies. On the one hand, ROS 2 is undergoing constant development, which grows its range of capabilities but increases its level of complexity. On the other hand, the standard took a

more direct path towards the same goal. It mainly focuses on keeping its complexity levels down, so, in return, the rules defined are much more restricitve. KAINOSORE, however, exploits this divergence by adopting the simpler user-friendly architecture, specified in IEC 61499 and complementing it with a set of straightforward ROS-adapted functions. Essentially, expanding the clear and unambiguous FB-based system with functionalities, previously limited solely to the domain of ROS 2 tools and libraries.

Likewise, as a technique framed to ROS-based robot manipulation using the IEC 61499 standard, KAINOSORE is designed to support the same capabilities available in ROS 2 and IEC 61499 systems.

Following an objective analysis and review of related works and current technologies, consolidated with an overview of their historically relevant background in industrial settings, we have identified key characteristics that our proposed solution should abide by. These include a set of custom design principles and a compliance with the 3 conceptual tenets defined in Chapter 1 (i.e., scalability, interoperability and adaptability).

4.3 Architecture

In terms of architecture, KAINOSORE stands out with a five-layer integration structrue, illustrated in Figure 4.1. This design seamlessly connects control-level interactions with the communication model via the five different layers, split as follows:

1. **Physical and device layer**: includes industrial assets such as robotic manipulators, sensors, actuators, and external equipment. This layer represents the interface with the physical environment.

2. **ROS 2 communication layer**: provides distributed communication and service interaction through ROS nodes, topics, services, and actions, supported by DDS middleware. This layer is responsible for data exchange and interaction with robotics software components.

3. **KAINOSORE integration layer**: acts as the core integration mechanism, embedding ROS communication primitives into IEC 61499 through ROS-enabled Function Blocks. This layer enables the interaction between asynchronous communication and structured control logic.

4. **IEC 61499 orchestration layer**: defines the application control logic using FBs, event/data connections, and control pipelines designed in 4DIAC and executed in a compliant runtime environment. This layer is responsible for explicit orchestration and execution sequencing.

5. **Runtime visibility and interoperability layer**: provides system monitoring and external interaction through OPC UA and TCP/IP interfaces, supporting configuration, inspection, and integration with industrial systems without interfering with control execution.

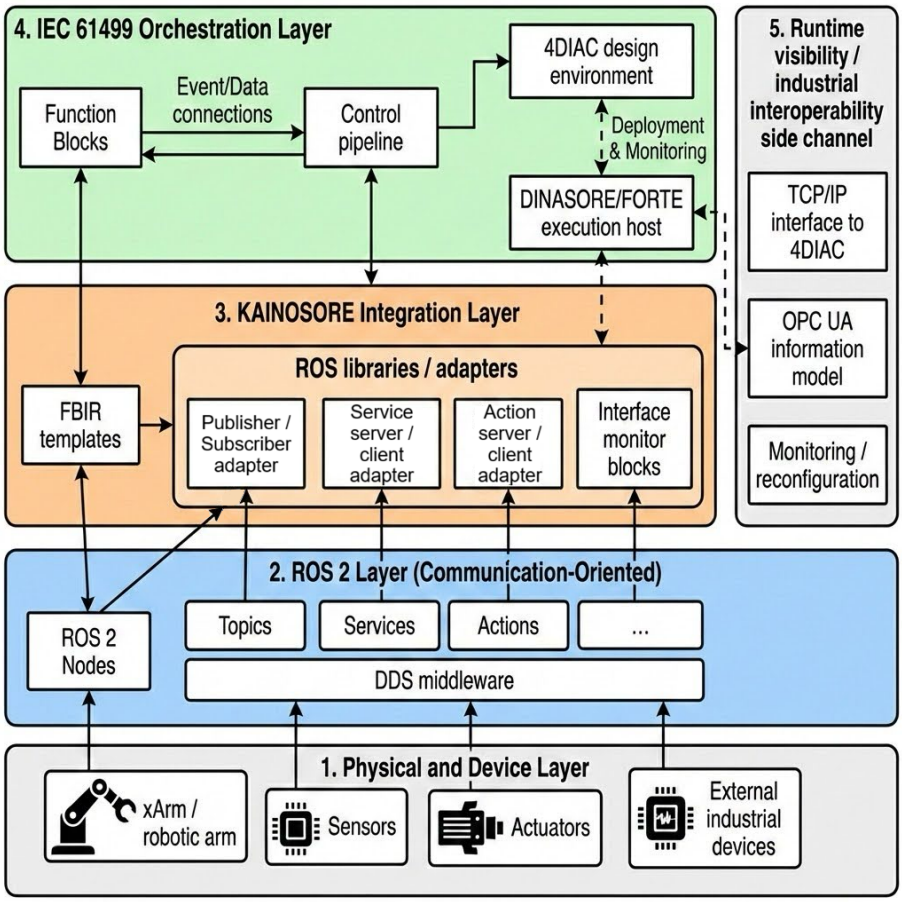


Figure 4.1: KAINOSORE layered architecture integrating ROS 2 communication with IEC 61499-based orchestration through ROS-enabled Function Blocks

Consequently, KAINOSORE operates as the nucleus that integrates the distributed communication model of ROS 2 with the orchestration and execution structures of IEC 61499, essentially, enabling ROS-based interactions to be incorporated into structured control pipelines without requiring external translation mechanisms or middleware coupling.

As such, this consolidation enables cooperative behaviour between the ROS and IEC 61499 systems, within the same CPPS, resulting in a flexible, yet structured, unifying design.

4.4 Design Principles

During the design cycle of KAINOSORE, certain traits have been endorsed to mesh and expand the concepts behind its creation. These can be epitomized as: i) Tool-Cooperation, ii) Abstraction, iii) Modularity, iv) Reusability, v) Easy-to-Learn Easy-to-Use and vi) Free and Open-Source.

(i) Tool-Cooperation

The historical analysis of the I4.0, as the consequence of previous industrial progress, reveals a pattern of exponential complexity sustained by the cooperation of previous norms

with innovative techniques. This collaborative nature has been accentuated in modern industry more than ever. Furthermore, ROS 2 and IEC 61499 are well established advancements within CPPS and each offer a multitude of capabilities apt to this industrial era.

As a result, we direct a major focus towards providing tool-cooperation instead of tool replacement. Consolidating both technologies under an unifying software nucleus, allows systems to be built around their advantages and, at the same time, avoid their drawbacks. Hence, KAINOSORE innovates per the same idea that has always been pivotal in the industrial evolution.

(ii) Abstraction

Abstraction is a concept that has been growing throughout modern history and seems to have become a perpetual idea for future advancements. Its presence in I4.0 technologies, specially future-proofed ones, is so prominent, it may go unnoticed or overlooked. However, it plays a vital role when addressing other fundamentals, such as flexibility, interoperability and adaptability.

In KAINOSORE, abstraction pertains to both hardware and software aspects. Regarding the mechanical units of heterogeneous systems, this approach provides a standardized interface that allows users to interact with different equipments without requiring individual code specifications. Moreover, KAINOSORE handles ROS 2 to IEC 61499 standard integration under the hood. This enhances software abstraction through interchangeable communication templates that resemble other libraries' operations.

(iii) Modularity

As an integral part of CPPSs, modularity is expected of any tool targeting this field, and KAINOSORE is no exception. This platform consolidates the modular structures of node-based communications with event-driven data exchange, by operating both configurations in parallel. Additionally, the FB-based representation of IEC 61499 complements the ROS 2 structure built around nodes, via a visually modular display, with unambiguity and increased information transparency. This results in an user-friendly interface adequate to introduce begginers to robotics and flexible enough to accommodate to a specialist's expectations.

(iv) Reusability

This idea is the zenith of abstraction and modularity converging. As CPPSs evolve and grow, their complexity can increase exponentially. In this environment, reusability is of the utmost importance, capable of acting as the counterbalance to it. Furthermore, it substantiates a system's scalability and adaptability in an ever-changing setting, while promoting a simpler design and lowering the skill demands for developers. Subsequently, this concept also reinforces community-driven developments, which further extend the range of accessible tools and capabilities, as well as the robustness and resilience of a system. In KAINOSORE, this concept is particularly relevant.

Ergo, we amplify the reusability aspect through an integration of the software and hardware abstraction with the modular architecture provided. ROS 2 and IEC 61499 benefit from reusable components within their distributed structures. As such, KAINOSORE is able to exploit reusable nonspecific code available for ROS 2 implementation, which then combines with the software abstraction discussed previously. This allows users to produce a reusable FB that can be organized in the model-based display of IEC 61499. In addition to this, the FB interface enables high-level costumization through enhanced information transparency and ROS 2 complements it with high node configurability and parametrization. In KAINOSORE, these capabilities are embraced to support a greater level of reusability.

Besides that, KAINOSORE also supplies a collection of reusable FB templates. These autonomously perform some of the most common operations specific to ROS 2, in an IEC 61499 compliant environment. In the end, this facilitates the design, management and implementation of a CPPS.

(v) Easy-to-Learn Easy-to-Use

The issue of high levels of complexity in CPPSs is a recurring topic in I4.0. Especially, when it comes to the layout, control and deployment processes, as it compromises scalability, interoperability and adaptability. As we've already discussed, ROS 2 and the IEC 61499 standard tackle this issue differently. Analysing both methods we concluded that, while commendable in many aspects, none of this techniques accomplishes the desired balance between capability and simplification.

KAINOSORE aims to offer the same tools and conventions, as ROS 2, in a simplified FB-based environment. To maintain a clear and easily understandable system, we rely on the abstraction characterized above. This autonomously attaches ROS 2 operations to a visual model of the event-driven modular organization. As a result, the monitoring of data exchange and workflow is also bolstered per visual medium.

(vi) Free and Open-Source

The principle of Free and Open-Source software has been explored in previous sections and we view it as a critical part of industrial evolution. Since the beginning, it has been a major philosophy of ROS and ROS 2 and, as a set of guidelines, the IEC 61499 standard is also publicly available to follow. This maxim is what enables continuous community-driven development and feedback. Moreover, it sets the precedent for achieving maximum interoperability and broader adoption. Hence, KAINOSORE was designed to be a non-commercial free and open-source system.

These characteristics are crucial to this approach and congregate on the centripetal philosophies defined as the main objectives we set out to achieve with our design.

4.5 Design Objectives

The primary goals we determined, for the development of this project, can be summarized as scalability, interoperability and adaptability. These are sustained by the design principles of KAINOSORE and, collectivelly, embody the paragon of a software's capability to handle design, management and implementation of growing CPPSs.

In the scope of CPPSs, scalability refers to how effectively a framework accommodates a system's growth. As more equipment is integrated into the structure, increasing its complexity and amount of resources used, KAINOSORE should be able to reliably operate without compromising performance or consistency. This is applicable to the design, management and implementation processes. Accordingly, KAINOSORE relies on its design principles to offer a myriad of optimization libraries and tools that facilitate procedures. Additional support is provided by a clear FB-based architecture that reinforces information transparency and simplification. Moreover, by preserving a parallel tool-cooperation, KAINOSORE contributes with a flexible equilibrium for resource management.

On the other hand, interoperability ensures that interconnected componenents, within the same structure, are capable of working together in a smooth and competent manner. This is elevated with seamless real-time data sharing links and hardware abstraction. From what we've explored, ROS 2 contributes with synchronous and asynchronous connections. Whereas IEC 61499 depends on event signals to transmit information. Hence, KAINOSORE comprehensively offers data exchange through the ROS 2 communication middleware assisted with an event-driven link-up option. This hybrid approach differentiates from simply using ROS 2 by allowing greater flexibility in resource management. Furthermore, our solution boosts interoperability to a whole new domain of virtual information sharing. This is manifested as currently employed ROS 2 systems can be integrated with IEC 61499 environments without requiring modifications. As such, tool-cooperation is essential as an amplifier of this behaviour.

Finally, adaptability is vital in CPPSs, allowing them to dynamically adjust and respond with real-time decision-making. Concurrently with the other design goals, this works to uphold the future-proof aspect of modern systems. The IEC 61499 standard supplies KAINOSORE with a modular architecture that exploits an event-based interface similar to state machines. This method of behaviour control and deployment has always been a trademark for decision-making capabilities. KAINOSORE relies on ROS 2 communications to enhance this with real-time proficiency and dynamism. Additionally, as the proposed solution is implemented in *Python*, CPPSs can be designed with FBs that incorporate advanced algorithms, such as ML and neural networks.

Chapter 5

Implementation

In this chapter we delve into the practical implementation of the proposed solution. Accordingly, we explore the software requirements and structure of KAINOSORE as a consequence of the hypothesis and conceptualization detailed in Chapter 4. This is done in the context of the DINASORE application use case. Furthermore, we highlight the functional differences between DINASORE and KAINOSORE and finalize with an use case scenario based on the official ROS 2 tutorial[1]

5.1 Requirements

In accordance with what's been discussed, prior to this chapter, the implementation of KAINOSORE was done using the DINASORE[2] [36] framework. Essentially, it functions as an extension that enables DINASORE to integrate ROS 2 mechanisms and components into the workflow of IEC 61499 compliant FB pipelines. As such, it must satisfy the different requirements for each of these tools.

First and foremost, it's important to note that, even though this approach is heavily derived from the structure found in DINASORE, it has only been tested in computer systems running Ubuntu 22.04.5 LTS (Jammy Jellyfish)[3], both natively and in a virtual environment. This is, in part, due to the ROS 2 *Humble Hawksbill*[4] system requirements, as this was the distribution used during the implementation and validation stages. Finally, for compatibility purposes, *Python* 3.10.12[5] was selected.

Even though the ROS ecosystem was natively designed in C++, it has also been supported with a *Python* Application Programming Interface (API), in response to this language's increase in popularity. Appropriately, this facilitates the integration of ROS 2 with KAINOSORE.

[1] https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools.html
[2] https://github.com/DIGI2-FEUP/dinasore/wiki
[3] https://www.releases.ubuntu.com/releases/22.04.5/
[4] https://docs.ros.org/en/humble/Releases/Release-Humble-Hawksbill.html
[5] https://www.python.org/downloads/release/python-31012/

Additionally, KAINOSORE relies on the 4DIAC project, which includes the IDE with an ideal GUI for FB pipeline configurations via a drag & drop feature. It also provides FORTE, the RTE necessary for IEC 61499 compliance.

5.2 Structure

Having fulfilled the previous requirements, the way KAINOSORE behaves (when ROS 2 isn't being used) is identical to how DINASORE would, in the same setting. Particularly, when it comes to the development process of CPPSs. This is because KAINOSORE is built upon the DINASORE software, branching off with the introduction of FBs that rely on ROS-specific utilities. In light of the proposed solution, we will be expanding on the development of modular structures, as it incorporates concepts introduced in previous chapters. This includes the design, management and deployment of FB pipelines.

Expectadly, these are modular systems, comprised of interconnected reusable FBs, compliant with the IEC 61499 standard. Thus, these modules are defined by a set of inputs and outputs, that are further divided into events and variables. Successively, each FB is assembled from two concomitant files, namely an XML-based one and a *Python* script. The former, defines the structure of the FB using XML tags, per the norms found in *LibraryElement.dtd*[6] set by the HOLOBLOC[7] project. On the other hand, the *Python*-coded script dictates the functionalities of the FB, following a *Python* class template, with a *schedule* method.

5.2.1 Using 4DIAC

Once these files have been created, developers can use the 4DIAC-IDE to render a virtual representation type for the corresponding FB, based on the XML metadata described. Moreover, the same definition file can be used to generate multiple instances of the same FB, as these are reusable elements. To build a pipeline, the event and data outputs of a FB must be connected to the event and data inputs, respectively, of another. This creates an event-driven sequential architecture, in which event links trigger the orderly execution of the next element, while the data links transmit information between them. To ensure a consistent and predictable performance, the events and variables must be properly identified and their data types and dependencies explicitly defined. Additionally, the XML definition structure includes a tag for classifying FBs, under several categories, which impacts their behaviour. In the current scope, this is used to distinguish FBs with an asynchronous execution from those with a cyclical one. However, this tag is only handled by the KAINOSORE framework and doesn't affect 4DIAC directly. Furthermore, this IDE is used to map the configuration built, to the device that will be running it, even allowing the decentralization of a system with networked controllers. Afterwards, users can deploy these structures to the running KAINOSORE environment, which will execute them, according to the modeled architecture.

[6] https://www.holobloc.com/xml/
[7] https://www.holobloc.com/

5.2.2 Communication Interfaces

KAINOSORE relies on two separate communication interfaces to allow for reliable extrinsic connections. Firstly, the Transmission Control Protocol (TCP)/Internet Protocol (IP) is used to establish a connection with 4DIAC. This is how the IDE is able to interact with KAINOSORE. As for the other interface, an automatically generated OPC-UA data model contains the configuration of the system, in a locally stored XML file. Both methods can be utilized during runtime to reconfigure the pipeline and/or monitor each FB. As a result, this preserves high configurability, which is an optimal quality for typical IEC 61499 systems, while boosting information transparency and flexibility. Nevertheless, these interfaces have several distinguishing features.

To enable the TCP/IP connection, KAINOSORE executes a TCP server through which the 4DIAC-IDE is capable of instructing it based on the layout defined. This includes the creation of each deployed FB, as well as their connections, with specified origins and destinations. These can be between two FBs linked through events and data, or manually defined values, directly entried into a FB's input using the IDE. Additionally, watches can be added to help with runtime monitoring of the workflow. The final instruction starts the execution of the FBs, however, the IDE can still be used to stop the running process.

Conversely, the OPC-UA interface is mostly directed towards the monitoring of the pipeline. It relies on the OPC-UA data model, previously mentioned, to store the deployed configuration, which is dictated by the instructions set by 4DIAC. As such, this file is automatically generated by KAINOSORE, upon the system's initialization, and updates during runtime, according to incoming 4DIAC commands. Moreover, this technique allows the configuration and state of operation to be saved, and restored at a later moment, avoiding a complete reboot of the process every time it's terminated.

5.2.3 Execution Model

So far, this description has been primarily focused on the general architecture and visual representation of the pipeline and the FB elements shaping it. Hence, the 4DIAC-IDE and the OPC-UA protocol have been central points of this discussion. However, the actual behaviour and performance of these structures relies much more on the features of the KAINOSORE environment, and the execution model implemented.

Following the deployment procedure, with the aforementioned communications properly established, KAINOSORE begins the execution process. This is characterized by the multithread technique, as each FB is assigned to a different thread. Despite this, thread-objects aren't actually executed in parallel, as KAINOSORE relies on a scheduling approach to manage time slices attributed to each FB's execution. Furthermore, the order, in which a series of FBs is executed, is dynamically determined, based on the event inputs being transmitted. When one of these event inputs is triggered, the corresponding FB reads the current data values and, through the *schedule* method, performs the actions coded according to the specific event received. Subsequently, once these actions are successfully computed, they return the output values, in the same sequence that

has been defined in the metadata file. As such, these outputs are promptly transmitted to the appropriate inputs of the next FB, in the chain of operations. Thus, this requires KAINOSORE to access both the XML and *Python* files, which should be locally stored, in the proper directory.

5.2.4 Programmable Function Block Behaviour

In line with the *modus operandi* described, the behaviour of each FB is programmed, via Object-Oriented Programming (OOP), into a *Python* script, and encoded in a straightforward class template. Nevertheless, this implementation must be consistent with the structure defined in XML, most notably, the class and FB type share the same name, and the inputs and outputs coincide with the respective interfaces. In addition to this, the object model, by default, is composed of two separate methods.

Firstly, the initialization function (*init*), which is run when the corresponding FB thread object is orignially created, in each execution process (typically used to set up local variables). And, most importantly, the *schedule*, responsible for prompting a specific functionality, based on the received inputs, and returning a list of outputs. Accordingly, the latter method is only triggered when an event signal has been transmitted to one of its event inputs.

By definition, each FB should include the 'INIT' event input, which is automatically triggered by KAINOSORE, upon deployment. As such, the *schedule* function can be used to implement any initialization process and set up local variables. Furthermore, cyclical FBs also need an additional 'READ' input and 'READ_O' output, as these are used, by the KAINOSORE environment, to handle executions in loop, via internally sent signals upon the completion of each cycle.

5.3 ROS-based Integration

Thus far, the described structure is directly inherited from the DINASORE framework. However, KAINOSORE diverges by allowing users to integrate ROS-specific utilities within FB pipelines. This mesh, of ROS 2 and IEC 61499, is done by coding the *Python* file, of any given FB, with the inclusion of certain libraries available in the KAINOSORE environment. Moreover, these libraries are bare-bones adaptations of the *Python* API of ROS 2 *Humble Hawksbill*, used to emulate its behaviour in an IEC 61499 compliant system. Throughout the remainder of this dissertation, to distinguish a FB that uses ROS 2 from one that doesn't, we will be referring to the former as a Function Block Integrating ROS (FBIR).

Besides this set of libraries, KAINOSORE is equipped with a collection of reusable (and editable) FBIR templates that can be incorporated directly into a CPPS, as other FBs would. This is because their definitions have been preemptively set in the appropriate XML and *Python* files, adhering to the previously characterized rules for IEC 61499 compliant structures. Furthermore, FBIRs are fully compatible with other FBs, so KAINOSORE can still execute an heterogeneous system, composed of both elements.

Sequentially, the templates have been created with software abstraction in mind, to serve as a guide, and present a straightforward integration of the libraries. As such, this helps preserve

the Easy-to-Learn Easy-to-Use principle of KAINOSORE. Accordingly, developers can design a pipeline by customizing the templates, building a structure around them, or creating FBIRs from scratch. Additionally, FBIRs can be broadly divided into two main sets: i) Interactive and ii) Monitor.

(i) Interactive:

Simply put, these FBIRs are responsible for engaging with the ROS 2 ecosystem, in an impactful way. This includes direct control over a node, as well as operations that affect its behaviour. The nodes interacted with, can be started via FBIRs, or by ROS 2 separately, extending to nodes started prior to the execution of the KAINOSORE engine.

(ii) Monitor:

As the name suggests, these FBIRs are mainly used for monitoring purposes and don't impact a node's behaviour. Even though they still engage with the ROS 2 ecosystem, it would be more accurate to describe this interaction as: the nodes impacting the Monitor FBIR. Identically to the Interactive FBIRs, the nodes in question include any active node, regardless of how and when it was started, as long as they are able to communicate with the system.

An example of an Interactive FBIR can be found in the *ROS2_PUB* template, as Figure 5.1 illustrates. This FBIR model receives the name of a topic, the user wishes to upload a message to, and the message to be published, with the specified data. These inputs are of type string and KAINOSORE autonomously handles the interpretation of the data and converts it to the proper format before uploading it. If a subscriber is found in this topic, the message is published, if not, KAINOSORE continues listening on the topic until a node subscribes to it. Moreover, since nodes are reusable elements, several instances of the same type can be concurrently active. As such, namespaces can be used to distinguish them.

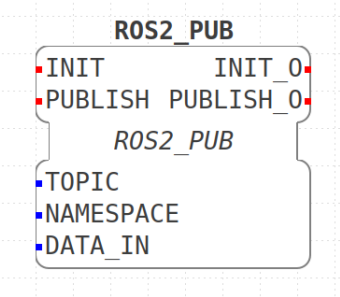


Figure 5.1: Structure of the ROS2_PUB template

As for the Monitor FBIRs, a common configuration is using these to list the interfaces[8] of active nodes. These can be subscribers, publishers, service or action clients and service or action servers. Additionally, there are also templates that listen for topics, which includes both publishers

[8]https://docs.ros.org/en/humble/Concepts/Basic/About-Interfaces.html

and subscribers, as well as for services and actions, which include both servers and clients. Figure 5.2 depicts some examples of FBIRs with this utility. By default, these models execute cyclically and output the name of the first five interfaces of a specific type, with realtime updates. Nevertheless, these structures can easily be modified, in the XML and *Python* files, to only update when triggered by a specific event edge and output any number of interfaces at a time. Furthermore, if there are more data outputs than interfaces, the remaining variables simply return an empty string.

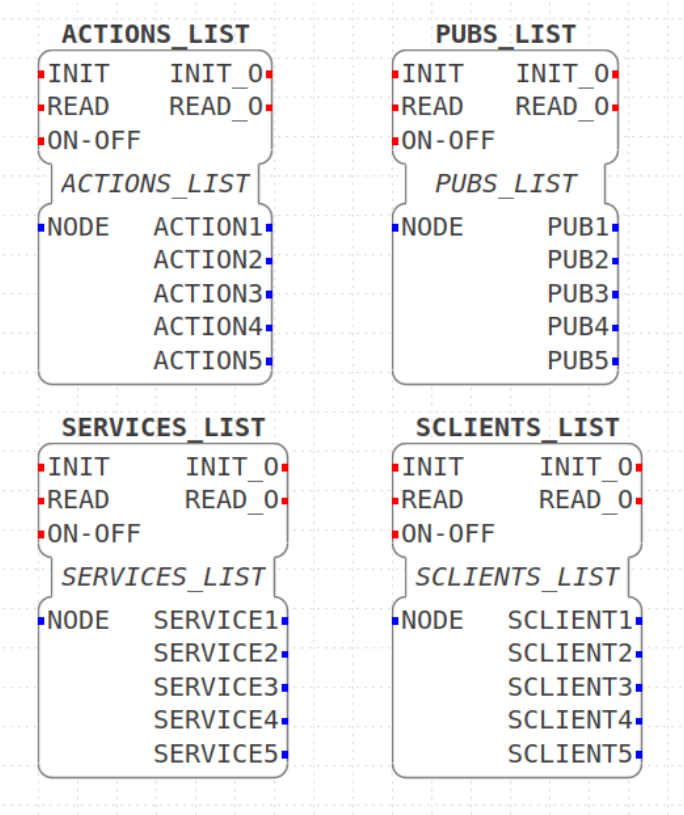


Figure 5.2: Examples of Monitor FBIRs templates:
ACTIONS_LIST (top left), *PUBS_LIST* (top right), *SERVICES_LIST* (bottom left),
SCLIENTS_LIST (bottom right)

Despite this, a single FBIR can have both Interactive and Monitor qualities, as these definitions are purposefully ambiguous and intended to be taken as conceptual abstractions. It's also possible for a FBIR to not be literally restricted within this categorization, which is the case of the *INTERFACE_FORMAT_FB* template, illustrated in figure 5.3. This model receives, as data input, the name of a topic, service or action and outputs a string with the format, and data types, of the message that's used to interact with it. Even though the output value is a string, due to IEC 61499 norms, the template first returns a list and then converts it to a string inside the *schedule* method, so both types are usable. However, the functionality of this type of FBIR is mostly used to assist with certain utilities for the development of more complex ones, and not as a standalone element per se. For example, a FBIR dynamically checking the publishers of an active node (*PUBS_LIST*), reading their messages (*ROS2_SUB*), processing and converting them to different message types

(*INTERFACE_FORMAT_FB*), and, finally, publishing these to other active nodes (*ROS2_PUB*).

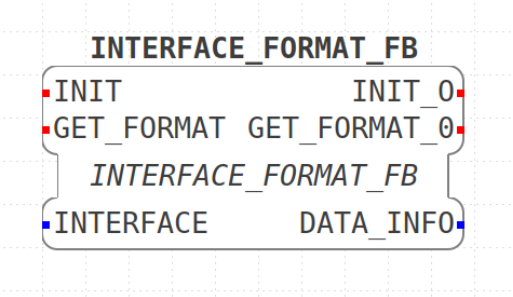


Figure 5.3: Structure of the *INTERFACE_FORMAT_FB* template

FBIRs operate by creating basic nodes, with a specific objective, directly connected to the *daemon*[9] process, which is the background service that handles node discovery. Thus, FBIRs rely on ROS 2 communication interfaces to interact with nodes, topics, services and actions. As a result, any modification, regarding these ROS 2 components, is recognized by KAINOSORE in realtime, whether it originates from a FBIR or not, which is an important factor that promotes interoperability and adaptability.

In addition to this, the development of these templates has been focused around the core elements of ROS and the wide range of utilities they provide for design, management and implementation of CPPSs.

5.4 *Turtlesim* Use Case

A common package used in the official ROS 2 tutorials is the *turtlesim*[10]. This is convenient, since *turtlesim* is a simple simulator that covers the basics of ROS 2, and serves as the introduction to more advanced concepts and features. As such, we will be exploring a simple illustrative process of a pipeline, solely composed of FBIR templates, that interacts with the *turtlesim* package. Figure 5.4 depicts the state of the workflow after a single iteration of the example operation. The majority of these modules have already been introduced, however, this system also incorporates new ones. Most importantly, the *NODE_FB_TEMPLATE*, which initiates a node based on its package, executable, nodename and parameters. It is also used to update a given node (during runtime), with parameters defined in a *YAML* file and, finally, shut it down, if the specified node is running.

Using the 4DIAC-IDE, we deploy this structure to a running KAINOSORE environment. Since the *PUBS_LIST*, *SUBS_LIST* and *SERVICES_LIST* are cyclical, we start by activating their 'ON-OFF' switch. Sequentially, we manually trigger the 'EXECUTE' event that launches the *turtlesim* node with the name 'turtle_kaino'. For simplicity purposes, the package and executable of this node have been preemptively assigned, in the *Python* file that encodes its behaviour, and can be verified in the output data. Furthermore, at this point, the 'PARAMS' input is empty, so the

[9] https://docs.ros.org/en/humble/Concepts/Basic/About-Command-Line-Tools.html#ros-2-daemon-background-discovery-service

[10] https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Introducing-Turtlesim/Introducing-Turtlesim.html

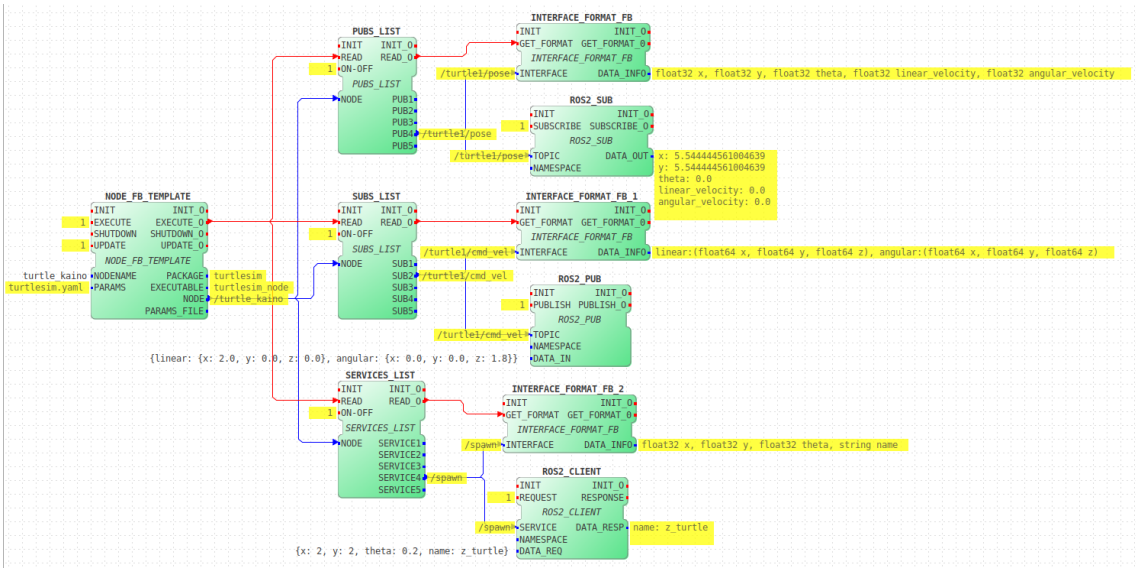


Figure 5.4: *Turtlesim* Pipeline Example

node starts with its default values. The first state of the simulator displays a blue background with a single turtle (identified as 'turtle1') in the center of it (figure 5.5a). Accordingly, this prompts the Monitor FBIRs to read the publishers, subscribers and service clients of the running *turtlesim*. Since this node has no service servers, the services listed only return the service clients. Afterwards, we enter the string 'turtlesim.yaml' and trigger the 'UPDATE' event. This *YAML* file is stored locally in the KAINOSORE engine and simply alters the background color of the simulator to a purplish hue. Following this, the 'SUBSCRIBE', 'PUBLISH' and 'REQUEST' events are manually triggered, in this sequence. The first is used to read the values being published by the 'turtle_kaino' node in the '/turtle1/pose' topic output by 'PUBS_LIST'. Secondly, the values defined by the string in 'DATA_IN' are published to the topic '/turtle1/cmd_vel'. As 'turtle_kaino' has an active subscriber to this topic (seen by the output 'SUBS_LIST'), 'turtle1' moves with the input trajectory. Finally, the 'ROS2_CLIENT' FBIR automatically creates a client and requests the '/spawn' service, from the node's service client, to spawn a new turtle, named 'z_turtle', in the bottom left corner of the same simulator, as figure 5.5b illustrates.

In addition to this, the three instances of 'INTERFACE_FORMAT_FB' are purely informative and are being used to corroborate the values and data types found throughout this system.



(a) Original *turtlesim* (b) Final *turtlesim*

Figure 5.5: Original and final stages of the *turtlesim* simulator

Chapter 6

Validation

In this chapter, we provide an in-depth description of the methodologies used to validate the proposed solution, in a relevant setting. Whereas chapter 5 introduced an use case with the *turtlesim* simulator, this chapter explores a scenario in an experimental environment with a robotic arm. This includes a detailed dive into the tests performed and metrics used. Finally, we present the obtained results.

6.1 Setup

One of the most common operations in CPPSs, and automation in general, is the pick and place process. This task is comprised of two main functions and a mechanical unit, usually a robotic arm, capable of grabbing and/or holding the desired object. As the name suggests, the goal of pick and place movements is to relocate a given item to a target location, which can range from inventory organization to assembly lines. Additionally, precision is a trademark requirement for robotic units and can be integral to the operations of CPPSs. As such, we designed the testing environment as a simple use case of pick and place, where a robotic arm picks an object from a small tray and places it on a separate nearby platform, piling these items up without knocking them down.

The experimental setup used for the validation of KAINOSORE, is also utilized with DINASORE and ROS 2, for comparison purposes. As such, every test in the relevant laboratory environment has been performed on an Ubuntu 22.04.5 LTS (Jammy Jellyfish)[1] (64-bit) system. The machine uses an octa-core *Intel(R) Core(TM) i7-7700HQ @ 2.8GHz* Central Processing Unit (CPU), with 16Gb of Random Access Memory (RAM) and 125Gb of disk space. The distribution of ROS 2 used was the *Humble Hawksbill*[2], which is compatible with the *Python* 3.10.12 version, used throughout the implementation and validation processes.

Additionally, we use the *xArm 5*[3] robotic arm (Figure 6.1), manufactured by *UFactory*, equipped with a vacuum gripper and an Alternating Current (AC) control box. This robot has 5 Degrees of

[1] https://www.releases.ubuntu.com/releases/22.04.5/

[2] https://docs.ros.org/en/humble/Releases/Release-Humble-Hawksbill.html

[3] https://www.ufactory.us/product/ufactory-xarm-5

Freedom (DoF) for a comprehensive range of motion within a 700mm reach and repeatability of ± 0.1mm. It provides *Python*, ROS and C++ SDKs for easier control, which includes *Python* 3.10.12 and ROS 2 *Humble Hawksbill* support in Ubuntu 22.04, making it ideal for this comparison.



Figure 6.1: *UFactory xArm 5*

Both KAINOSORE and ROS 2 use the *cyclone* DDS ROS Middleware (RMW) and require the ROS 2 *humble* environment to be sourced, in the terminal that's running the tests. Furthermore, the setup script of the mechanical unit[4] must also be sourced.

6.1.1 Test Case Scenario

The experimental scenario, illustrated in Figure 6.2, is composed of the *UFactory xArm 5*, a *5*x*5* tray, twelve red pieces and a separate platform. Additionally, the robotic arm is mounted on a table, equipped with a vacuum gripper and establishes a TCP/IP connection, with the aforementioned Ubuntu-based machine, via an ethernet cable.

Simply put, the objective of this pick and stack operation is for the robot to take a piece from the tray and place it on the platform. All the pieces are functionally identical and start on the tray, positioned in order from one through twelve, completely occupying the first two rows and two extra spaces on the third. They are moved, one by one, onto the platform, placing each on top of the last. Furthermore, the state of the robotic arm assumes four main categories: Resting, Picking, Transporting and Placing. Assuming this classification, we can define the workflow as follows:

1. Resting:

   Starting in a predefined *START_POSE* position, an automated process of setup and configuration of the robot, according to the test conditions, is triggered. This consists of establishing the TCP/IP connection, cleaning warnings and errors, setting the load and offset values and

[4] https://github.com/xArm-Developer/xarm_ros?tab=readme-ov-file#45-source-the-setup-script



Figure 6.2: Pick and stack laboratory architecture scheme (not to scale)

unlocking the robot's motion actuators, which includes setting the mode and state to allow movement instructions. Lastly, we ensure the machine is in the correct position by prompting it to move to *START_POSE*.

2. Picking:

   During this stage, the system computes where on the tray, is the piece to be picked, based on the current size of the stack on the platform. For the purpose of this experiment, this is calculated with a local variable, incrementing after each iteration. Sequentially, the robot moves to the piece's x and y coordinates maintaining a preestablished safe distance on the z axis. Lastly, it descends, grabs the piece and ascends back.

3. Transporting:

   While holding a piece, the arm moves to an interim position, between the tray and the platform, to safely adjust the z coordinate to the height of the stack. This way, the robot will always follow the same consistent pathing and we ensure that the stack isn't knocked down during the next operation.

4. Placing:

   This movement is the parallel of the Picking function. The machine moves to the x and y coordinates of the stack, while keeping a safe interval above it. Afterwards, it descends,

carefully sets the piece on top of the last, and ascends back. In the case of the first iteration, the piece is placed directly on the platform.

5. Transporting:

   Similar to before, the robot moves to a temporary position between the tray and the platform, however, no piece is being held. Likewise, this ensures a safe and consistent trajectory.

6. Resting:

   Finally, once an iteration has been completed, the *xArm 5* unit returns to *START_POSE* and updates any necessary local variables, awating new instructions.

6.2 Metrics

During the experimental phase, we have focused primarily on quantitative measurements to compare KAINOSORE, DINASORE and ROS 2, under the same conditions. As such, we use a set of metrics for FB pipeline comparison, proposed in [52], and another performance related one.

6.2.1 Function Block Pipelines

These measures have been presented by Zhabelova and Vyatkin to reflect IEC 61499 compliant software's distinct characteristics, and later adapted in [38] to better conciliate with state of the art technologies, still adhering to this standard. While these metrics are directed towards FB-based architectures, some are nonetheless pertinent to the script used for the ROS 2 test cases. Accordingly, the measurements carried out consist of four subsets.

Firstly, we analyze structural complexity, which is often identified as graph based. As such, we use McCabe's cyclomatic complexity to indicate the number of paths in a control flow graph. In this case, we calculate it based on the number of transitions (N_{edges}) and states (N_{nodes}) in the ECC, and *if/else* statements (N_{ifs}) and loops (N_{loops}) in the algorithm of the FB, given by Equation 6.1.

$$\begin{aligned} v_{ECC}(G) &= N_{edges} - N_{nodes} + 2 \\ v_{Alg}(G) &= N_{ifs} + N_{loops} + 1 \end{aligned} \tag{6.1}$$

Considering the number of distinct operators (n_1) and operands (n_2), as well as the total number of operators (N_1) and operands (N_2), of a given FB, we can estimate its volume and entropy. These are part of Halstead's metrics and the full set is:

- Program Vocabulary:

$$n = n_1 + n_2 \tag{6.2}$$

- Program Length:

$$N = N_1 + N_2 \tag{6.3}$$

- Estimated Length:

$$\hat{N} = n_1 log_2(n_1) + n_2 log_2(n_2) \tag{6.4}$$

- Purity Ratio:

$$PR = \frac{\hat{N}}{N} \tag{6.5}$$

- Program Volume:

$$V = N log_2(n) \tag{6.6}$$

- Difficulty:

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2} \tag{6.7}$$

- Program Effort:

$$E = D \cdot V \tag{6.8}$$

Additionally, we evaluate the High-level design complexity of a given FB i, which consists of structural complexity, (Equation 6.9), data complexity (Equation 6.10) and system complexity (Equation 6.11). These require the number of output events (f_{out}), data inputs (N_{in}) and data outputs (N_{out}).

$$S(i) = f_{out}^2(i) \tag{6.9}$$

$$DC(i) = \frac{[N_{in} + N_{out}]}{f_{out}^2(i) + 1} \tag{6.10}$$

$$C(i) = S(i) + DC(i) \tag{6.11}$$

Finally, we examine Maintainability (*MI*), as a culmination of previous measures, to compute how easily a component can be modified without impacting too heavily other FBs in the pipeline. Considering that the Lines of Code (LOC), in this case, represent the sum of the LOC of a FB's algorithm and its states plus transitions, we can use Equation 6.12, to determine the values of *MI*. It's important to note that a higher *MI* equates to a more maintainable structure.

$$MI = 171 - 5.2ln(V) - 0.23v(G) - 16.2ln(LOC) \tag{6.12}$$

6.2.2 Performance

In addition to this, we also compare the three alternatives (i.e., KAINOSORE, DINASORE and ROS 2) based on performance metrics, which include the time it takes for a given task to complete and the amount of resources used. The latter consists of CPU usage and free RAM available, during the operation.

To measure the duration of the process, we implemented two separate timers in each solution. The first timer determines how long the appropriate setup and configuration steps took until the robot was ready to start the loop. The other one, gauges the duration of each iteration, from the moment the event is triggered until the robot returns to the *START_POSE*. As such, we use these values to assess the interval of time it takes for each solution to complete the whole operation and the stacking of every piece, for a given number of iterations.

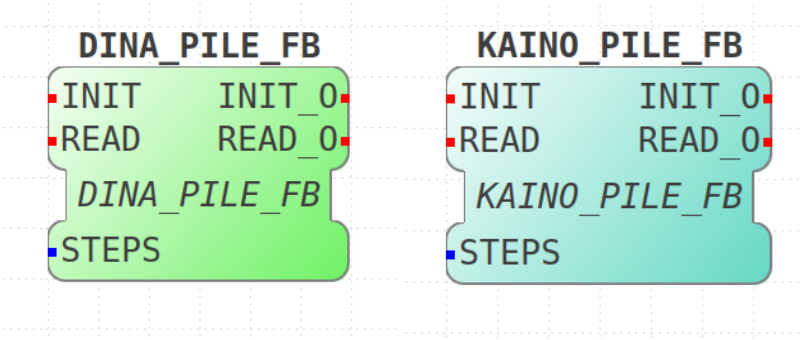
On the other hand, we measure the amount of resources used via the *nmon* monitor tool [18]. These include average CPU usage and maximum at any given instance, and average and minimum free RAM available. Ergo, this measurement takes into account the whole length of the process.

6.3 Tests Description

In accordance with what's been discussed thus far, we designed the experimental scenarios, following a single template along the development of every solution. Additionally, every test has been performed in the same initial conditions, that is, the *xArm 5* machine powered off and prepositioned to *START_POSE*. Furthermore, every solution, once triggered, autonomously operates in a loop for a given number of iterations.

Hence, we examine five different alternatives, performing the same pick and stack operation, previously described. For each of these scenarios, we conduct two sets of testing, each comprised of three identical ones. One of the sets addresses a pick and stack of six pieces, while the other runs the same operation for twelve pieces. There are thirty test scenarios in total, however, we couple the identical ones to calculate the average for each measurement taken. Thus, the five structures tested are: *DINASORE (single FB)*, *DINASORE (pipeline)*, *KAINOSORE (single FBIR)*, *KAINOSORE (pipeline)* and *ROS 2*.

DINASORE (single FB) and *KAINOSORE (single FBIR)* execute a single component, as depicted in Figure 6.3, handling every movement and action in sequence. This reduces the number of connections in a system to rely on a more asynchronous approach, while implementing a straightforward algorithm.



(a) *DINASORE (single FB)* (b) *KAINOSORE (single FBIR)*

Figure 6.3: Deployed structures for single component sequential workflow

However, both *DINASORE (pipeline)* and *KAINOSORE (pipeline)* rely on an architecture with four interconnected components (Figure 6.4). Similar to the previous scenario, the architecture of *DINASORE (pipeline)* is the same as *KAINOSORE (pipeline)*, with modifications only in the algorithm of each FB. Moreover, this is a purposefully exaggerated structure, to accentuate the differences between a pipeline and a single component, executing the same tasks. In this case, the *MANAGER_FSM* handles computing processes, variables and state transitions, while the three remaining elements focus on a single function in a more generalized way. One of the modules handles movement instructions to a position input by *MANAGER_FSM*, while another only controls the setup and configuration steps. In addition to this, the third module simply instructs the gripper to activate or deactivate, according to the received input. This allows certain operations to work in parallel, at the cost of increased complexity, while improving modularity and reusability. The *MANAGER_FSM* is the same for *DINASORE (pipeline)* and *KAINOSORE (pipeline)*, thus, it doesn't require any ROS 2 integration.

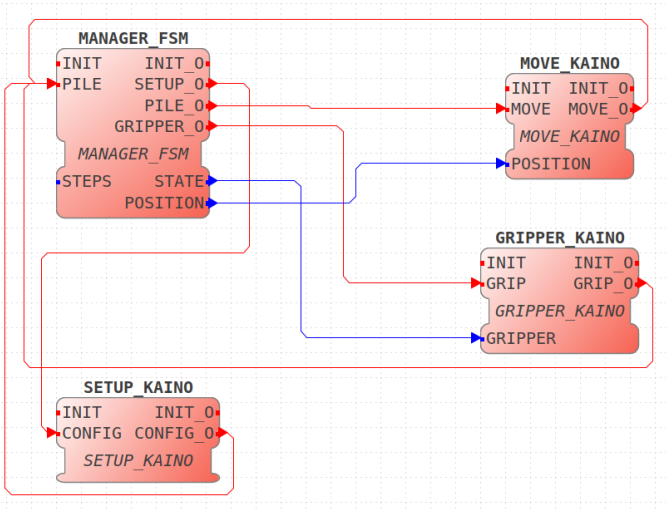


Figure 6.4: Deployed structure for pipeline parallel workflow
(Example: *KAINOSORE (pipeline)*)

The *ROS 2* system is encoded in a separate *Python* script and uses the same template as the others, to mimic the behaviour of a FB, for a far more reliable comparison. This approach, as well as the *KAINOSORE (single FBIR)* and *KAINOSORE (pipeline)* ones, use the *cyclone* DDS RMW to communicate with the robotic arm to send service calls. Whereas the *DINASORE (single FB)* and *DINASORE (pipeline)* use the *Python* SDK provided by the manufacturer.

Except for *ROS 2*, every system is run on a single controller architecture deployed in 4DIAC and using the FORTE RTE, as Figure 6.5 illustrates.

6.4 Results

Taking into consideration prior discussions, we use five different structures for developing a comparative analysis. These are: (I) *DINASORE (single FB)*, (II) *DINASORE (pipeline)*, (III) *KAIN-*

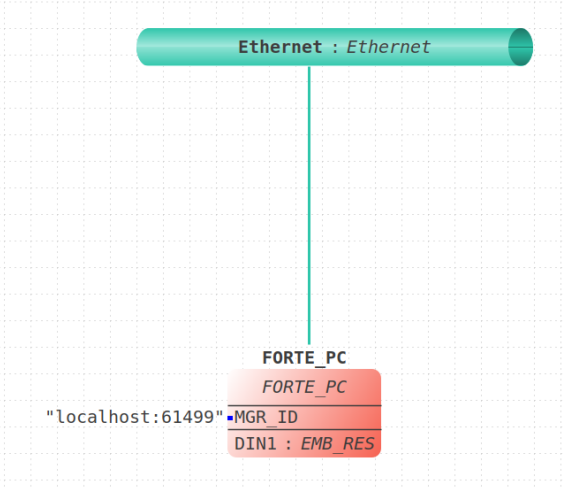


Figure 6.5: System configuration for the *DINASORE (single FB)*, *DINASORE(pipeline)*, *KAINOSORE (single FBIR)* and *KAINOSORE (pipeline)* scenarios

OSORE (single FBIR), (IV) *KAINOSORE (pipeline)* and (V) *ROS 2*. Additionally, considering the metrics used, we can split the findings into two separate sets. Firstly, we present the FB pipeline metrics overview, followed by the results of the performance tests.

6.4.1 Function Block Pipeline Results

Table 6.1 displays the results obtained by calculating the aforementioned FB-based measurements, namely, *McCabe's*, *Halstead's*, High-level design complexity and Maintainability. As we've mentioned previously, some of the metrics used are incompatible with the structure deployed for the *ROS 2*.

Table 6.1: Function Block Pipeline Metrics (N/A: Not Applicable)

Metric	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
Cyclomatic Complexity	24	38	5	28	10
Program Vocabulary	27	73	26	70	27
Program Length	628	1003	636	950	841
Estimated Length	109.717	418.581	103.848	396.220	109.717
Purity Ratio	0.175	0.417	0.163	0.417	0.130
Program Volume	2986	6208	2989	5822	3999
Difficulty	69.66	52.2	73.93	51.72	91.75
Program Effort	208006	323965	221007	301169	374893
Structural Complexity	4	28	4	28	N/A
Data Complexity	0.2	0.58	0.2	0.58	N/A
System Complexity	4.2	28.58	4.2	28.58	N/A
Maintainability	41.75	27.72	45.42	30.49	38.34

6.4.2 Performance Results

The results yielded by performanced-related tests can be divided further into two main groups. One for resource consumption and another for time taken.

Regarding resource consumption, we present the results of each system for the pick and stack operation with six pieces (Table 6.2), twelve pieces (Table 6.3) and, finally, we compare both in Table 6.4 given by Equation 6.13. Furthermore, these results are the average of the three tests run per scenario

$$X_{cmp}(i) = 100 \cdot \frac{T_{wlv}(i) - S_{ix}(i)}{S_{ix}(i)} \tag{6.13}$$

Note that $X_{cmp}(i)$, $T_{wlv}(i)$ and $S_{ix}(i)$ represent the values of the comparison, six-piece scenario and twelve-piece scenario, respectively, for a given i metric.

Table 6.2: Resource Consumption in **Six** Piece Pick and Stack Scenario

Metric	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
Average CPU (%)	1.443	1.592	1.381	1.658	0.633
Maximum CPU (%)	14.733	9.367	22.8	19.6	6.2
Average Free RAM (MB)	10969.436	10808.024	10829.217	10767.517	11900.028
Minimum Free RAM (MB)	10939	10779	10807.9	10738.567	11890.8

Table 6.3: Resource Consumption in **Twelve** Piece Pick and Stack Scenario

Metric	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
Average CPU (%)	0.760	1.278	0.908	1.131	0.538
Maximum CPU (%)	14.567	9	16.8	16.7	5.767
Average Free RAM (MB)	10892.188	10801.741	10815.657	10755.691	11891.008
Minimum Free RAM (MB)	10843.8	10765.7	10797.667	10732.033	11885.5

Table 6.4: Resource Consumption Comparison

Metric	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
Average CPU (%)	-47.316	-19.735	-34.247	-31.792	-14.971
Maximum CPU (%)	-1.131	-3.915	-26.316	-14.796	-6.989
Average Free RAM (%)	-0.704	-0.058	-0.110	-0.125	-0.076
Minimum Free RAM (%)	-0.870	-0.123	-0.061	-0.095	-0.045

In the case of the time measurements, we assume a similar exposition, comparing the time taken for the *Setup and Config* phase, the total duration for all the pieces to be stacked and the total time taken in the operation (which is the sum of the previous metrics). The Table 6.5 represents the pick and stack for the six-piece, similarly, Table 6.6 for the twelve-piece pick and stack.

Table 6.5: Duration of **Six** Piece Pick and Stack Operation

Task	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
Setup and Config (s)	4.216	3.511	4.351	3.681	4.005
Pieces Total (s)	55.529	55.506	56.604	56.664	56.354
Operation Total (s)	59.745	59.018	60.955	60.344	60.359

Table 6.6: Duration of **Twelve** Piece Pick and Stack Operation

Task	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
Setup and Config (s)	4.215	3.511	4.316	3.664	4.018
Pieces Total (s)	111.402	111.339	113.731	113.587	113.049
Operation Total (s)	115.617	114.850	118.047	117.252	117.067

Chapter 7

Discussion

In this chapter, we interpret the previously obtained results and report our findings, based on what's beed discussed throughout this dissertation. Accordingly, we answer the research questions posed in chapter 1 and provide an objective analysis of the current limitations of this work.

7.1 Findings

Keeping with the same designations from chapter 6, we assume the five tested structures as: (I) *DINASORE (single FB)*, (II) *DINASORE (pipeline)*, (III) *KAINOSORE (single FBIR)*, (IV) *KAINOSORE (pipeline)* and (V) *ROS 2*.

7.1.1 Function Block Pipeline Discussion

Looking at the results of the FB pipeline metrics (Table 6.1), certain values stand out, above the others.

Firstly, in regards to McCabe's *cyclomatic complexity*, both pipeline systems are, expectadly, the worst performers, since the number of transitions and states escalate very easily in more complex structures. This is also accentuated by the fact that the pipeline layout used is, admittedly, an exaggerated example for comparison purposes. On the other side of the spectrum, however, the FBIR scenario is vastly superior to the FB of DINASORE. This is due to the need for an additional condition, in the *Python* SDK implementation, that checks if the instruction has been successfully received by the robot, whereas the service interface of ROS-based communication already integrates this by receiving a response message after the request. Even though ROS 2 doesn't have a FB interface, to follow the same template as the other alternatives, it relies on *if/else* conditions to simulate the ECC's behaviour. As a result, this shows how KAINOSORE benefits from the consolidation of ROS communication interfaces with a FB-based architecture to surpass the other solutions, avoiding their drawbacks.

Sequentially, we advance to Halstead's metrics. In this set of measurements, little difference can be seen between DINASORE and KAINOSORE. On the one hand, in a single element scenario, DINASORE barely surpasses KAINOSORE. On the other hand, the latter slightly outperforms the former in modular architectures. Despite this, the biggest distinction is in system (V), as ROS vastly underperforms, considering that it only uses a single script, while scenarios (II) and (IV) rely on four scripts each. This is because, ROS requires a more elaborate script and has many intrincancies, which help support the range of capabilities.

In the third set of metrics, we explore the high-level design metrics. Not much information can be taken from this, since we followed the same workflow and design template, as strictly as we were able, for every solution, to ensure the most impartial and consistently objective results. As such, KAINOSORE and DINASORE adhere to the same architecture and there is no variation between them. Furthermore, these measures aren't applicable to the ROS 2 script.

To finalize the FB pipeline metrics, we use the maintainability index, which relies on the prior measurements of volume and McCabe's formula, as well as the LOC. This may well be the most important metric out of all the FB pipeline ones, in the scope of this work, since high maintainability is integral to uphold scalability and adaptability. Accordingly, the system with the highest, is the single FBIR approach (III), followed by the single DINASORE FB (I) and, closely enough, the ROS 2 script (V). Even though the LOC of (I) and (III) is very similar, McCabe's indicator helps enhance this distinction. Additionally, (V) has the most LOC out of the three, which is represented by a lower maintainability value. Regarding pipeline-based systems, KAINOSORE manages to outperform DINASORE but the difference isn't much, especially considering that both values are substantially lower than the other three.

7.1.2 Resource Consumption Discussion

In regards to the Tables 6.2, 6.3 and 6.4, we can compare the resource consumption of each tool used (i.e., DINASORE, KAINOSORE and ROS).

Solution (V) is, evidently, the most efficient consumption-wise in the six-piece scenario, as its CPU usage is, on average, less than half of its counterparts and, the maximum value it hits, is also relatively small. Meanwhile, the highest average CPU usage is with the (IV) structure, which integrates the KAINOSORE-based pipeline. However, when comparing these values with the twelve-piece operation we obtain the Table 6.4, which shows a notable change in CPU usage. This means that, when this operation takes, roughly, twice as long to complete (See Tables 6.5 and 6.6) solution (I) drops average CPU usage to nearly a half of what it was, and solutions (III) and (IV) drop close to a third. In contrast, (V) has the least significant drop out of the scenarios explored and, if we look at the data collected by *nmon*, there's a clear spike when DINASORE and KAINOSORE structures are first initialized (notably higher in the latter). This is because both approaches initiate a threading process, while ROS just initiates node connections, which use less CPU. Moreover, KAINOSORE also initiates node connections, explaining the higher spike. In addition to this, the difference in average CPU usage between DINASORE and KAINOSORE is generally negligible and, according to these results, either of these systems could use less than the

other in these conditions, this metric isn't impactful enough to be a factor when deciding which one to use. However, ROS still outperforms both in this aspect thanks to its consistency in avoiding significant spikes.

Considering RAM consumption, ROS surpasses both DINASORE and KAINOSORE, once again, mainly because efficient resource management is a big part of the ROS project. On the other hand, integration of DINASORE, and in turn KAINOSORE, with microcontrollers is still growing. Furthermore, the DINASORE-based solutions, still consume less memory than KAINOSORE, in the tested environment. The difference of available RAM on average betwee DINASORE's solution (I) and ROS' (V) is nearly 10%, however, between the former and KAINOSORE's (III), it's barely 1%. Moreover, none of the systems suffer any relevant alteration when extending from six to twelve pieces. Therefore, this metric would only be significant in microcontroller environments, which isn't the case.

Even though KAINOSORE had the most overall resource consumption, the difference is negligible considering the size of the process and the resources of the machine. Additionally, the KAINOSORE scenarios involved concurrent use of ROS and IEC 61499, however, the proposed approach offers enough flexibility for the user to handle resource management by splitting the labour more evenly between ROS and IEC 61499.

7.1.3 Timers

Lastly, we compare the five alternatives, given the amount of time it took for each of them to complete the operation. We have used two separate timers, as indicated previously, one for the *Setup and Config* process and the other for each iteration. Since each piece starts in a different position, this affects how long each iteration takes. However, the positions are consistent across the different solutions, so this doesn't pose a problem. Moreover, in every scenario, the robotic arm is initialized with radians for angular measures.

Analyzing the results in Tables 6.5 and 6.6, we immediately notice that the biggest changes happen during the *Setup and Config* in scenarios (II) and (IV). While the solution (V) is able to perform the *Setup and Config* related actions quicker than (I) and (IV), if only by a slight margin, DINASORE and KAINOSORE can circumvent this by using the modular architecture of IEC 61499 to execute tasks in parallel. Besides this, DINASORE performs better in gereral with the biggest distinction being in the longer twelve-piece process, while variations between ROS and KAINOSORE are mostly negligible.

7.2 Answering Research Questions

Taking into consideration everything that has been discussed thus far, we can answer the research questions initially posed in 1.4.1, as follows:

- *Is KAINOSORE capable of performing ROS-specific functions, in simulated and experimental settings, using the FB-based architecture of the IEC 61499 standard?*

To answer this question, we have designed and implemented a functional FBIR-based system 5.4 in Chapter 5 that interacts with the *turtlesim* simulator, which is only possible via ROS 2 communication. In regards to the experimental setting, in Chapter 6 we did an intricate analysis of a FBIR-based system with two different architectures (single FBIR 6.3b and pipeline 6.4) that we use to control a robotic arm in a pick and stack operation, through ROS integration.

- *How does KAINOSORE uphold the scalability, interoperability and adaptability principles, compared to ROS 2 and IEC 61499?*

  To answer this question we invoke prior discussions regarding the functionalities and principles of KAINOSORE, and corroborate with conducted experiments. Interoperability can be verified with the *Tool-Cooperation* approach that emphasizes the idea of ROS and IEC 61499 working in tandem, which allows KAINOSORE to merge the interoperability of these systems under a single framework, increasing its reach, just like when we used FBIRs to control a robot via the ROS 2 communication interfaces. Furthermore, the FB pipeline metrics have highlighted how KAINOSORE fares, compared to standalone DINASORE and ROS, in regards to its lower complexity and higher maintainability, which are the two core pillars of scalability and adaptability. Additionally, we have also discussed resource consumption, comparing a shorter and a longer versions of the same process, learning that ROS was the most resource-efficient approach, however, KAINOSORE and DINASORE were the most improved when the system grew in complexity.

7.3 Main Advantages of KAINOSORE

KAINOSORE stands out above ROS and DINASORE in many aspects.

As it currently stands, we believe that it provides the best alternative to dynamically create functional ROS elements and customize them to specific situation without the labourious and time-consuming configuration and build processes of normal ROS systems. One of the reasons for this is that it uses software abstraction instead of autonomous code generation. As such, the user still has full control over CPPSs processes.

DINASORE usually restricts the control of robots to *Python*, whereas KAINOSORE allows for the ROS operation to be developed in C++ for faster control, while maintaining a *Python* implementation and interface. On a similar note, KAINOSORE is capable of providing numerous functionalities that aren't accessible to *Python*. The *xArm 5* robotic arm used in the experimental phase serves as a good example of this. As the number of packages available for ROS, to control that robot, far exceeds the functionalities available in the *Python* SDK.

Additionally, we believe that KAINOSORE is capable of providing far more flexibility and balance for resource management. Even though KAINOSORE is using more resources, the amount used is still smaller than the sum of the standalone ROS and DINASORE.

7.4 Main Limitations of KAINOSORE

Despite this, KAINOSORE still has certain limitations.

Firstly, we recognize that this implementation could benefit from optimization algorithms, as a means to reduce resource consumption. Additionally, KAINOSORE creates and shuts down ROS elements with more frequency than ROS 2 does, while using a threading methodology, which causes it to use up more resources.

In addition to this, there is currently no implementation for KAINOSORE to request actions and, by using threading, the *launch* tool of ROS is incompatible with KAINOSORE.

Chapter 8

Conclusion and Future Work

The history behind decentralized robotic systems is vast and its roots extend throughout several generations of industrial revolutions. As such, these structures have undergone numerous radical changes to reach this point and become the backbone of I4.0. However, this is still a fairly abstract concept, which provides plenty of leeway for the industry to grow.

This also means that there is no clear path for it to follow, in order to evolve, so, many advancements are constantly being developed in parallel. The most successful ones end up branching even further, and farther apart. As a result, industrial processes have become a mesh of technologies, as evidenced by modern CPPSs architectures. Hence, complexity became trademark for the design, management and implementation of these structures, which, in turn, compromises their scalability, interoperability and adaptability aspects.

Nevertheless, these three characteristics are particularly important for the progress of the industrial world. As such, various mechanisms have emerged to uphold them, of which, ROS and the IEC 61499 Standard are highlighted for their qualities. Regardless, they are also criticized for their disadvantages. So, modern research and projects tend to focus on finding alternatives to these tools, as a means to circumvent their drawbacks.

However, as we dive deeper into the study of decentralized robotic systems, the clearer it seems to us that these tools can effectively be mutally beneficial, as they share many common principles and goals. But, despite their popularity and similarities, a symbiotic connection between them has regularly been overlooked. Hence, throughout this dissertation we have explored how and if these frameworks can truly be merged in such a way that their benefits would overcome their individual flaws.

8.1 Main Contributions

In this dissertation, we have examined the evolution of industrial systems and modern processes to better understand the issues of scalability, interoperability and adaptability in the desing, management and implementation of CPPSs. Afterwards, we paired this study with a literature review to comprehend modern tendencies and shortcomings.

Taking into account our thorough invstigation, we conjured an hypothesis, and used it to establish a set of design principles and objectives, which we believe should be core tenets for any tool trying to tackle these issues. The objectives are the aforementioned scalability, interoperability and adaptability and the principles can be summarized as:

- Tool-Cooperation
- Abstraction
- Modularity
- Reusability
- Easy-to-Learn Easy-to-Use
- Free and Open-Source

From these characteristics, we developed KAINOSORE as a means to conciliate ROS and IEC 61499 under a single software nucleus. Implementing it and testing it in a simulated environmnet. Following this we also designed a system for laboratory testing across various measurements. Furthermore, we use these results to corroborate our hypothesis, principles and objectives.

We believe KAINOSORE, or at least its philosophy, to be the first steps towards a fully symbiotic integration of ROS and IEC 61499, preserving an ideal ratio of capabilities and complexity.

8.2 Future Work

Admittedly, the proposed solution, KAINOSORE, still has a lot of room to grow.

We direct future work towards two main directions. One, is the optimization of KAINOSORE, which can benefit from implementation of advanced *Python* algorithms and better resource management. While the other, is the development of prototypes that abide by the same hypothesis, principles and objectives, expanding DINASORE to accommodate this integration, but through different methods other than KAINOSORE.

References

[1] John" "Conway. *"IEC 61499: The Industrial Automation Standard for Portability that Unleashes Industry 4.0"*. Tech. rep. "Schneider Electric", 2020.

[2] Amr Adel. "Future of industry 5.0 in society: human-centric solutions, challenges and prospective research areas". In: *Journal of Cloud Computing* 11 (Sept. 2022). DOI: 10.1186/s13677-022-00314-5.

[3] Michel Albonico et al. "Software engineering research on the Robot Operating System: A systematic mapping study". In: *Journal of Systems and Software* 197 (2023). ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2022.111574.

[4] Sabah Ali, Hayder Al-Sultan, and Mithaq Rubaie. "Fifth Industrial Revolution: (New Perspectives)". In: *International Journal of Business, Management and Economics* 3 (July 2022), pp. 196–212. DOI: 10.47747/ijbme.v3i3.694.

[5] R. Baheti and G. Gill. *Cyber-physical Systems*. Vol. 12. issue 1. 2011, pp. 161–166.

[6] Mohd Bahrin et al. "Industry 4.0: A review on industrial automation and robotic". In: *Jurnal Teknologi* 78 (June 2016). DOI: 10.11113/jt.v78.9285.

[7] Ana Sasa Bastinos et al. "ReApp Store-A Semantic AppStore for Applications in the Robotics Domain." In: *ISWC (Industry Track)*. 2014.

[8] Herman Bruyninckx. "Robotics Software: The Future Should Be Open [Position]". In: *IEEE Robotics & Automation Magazine* 15.1 (2008), pp. 9–11. DOI: 10.1109/M-RA.2008.915411.

[9] Gustavo Caiza et al. "Flexible robotic teleoperation architecture for intelligent oil fields". In: *Heliyon* 6.4 (2020). DOI: 10.1016/j.heliyon.2020.e03833.

[10] M. Castells. *The Rise of The Network Society: The Information Age: Economy, Society and Culture*. Information Age Series vol. 1. Wiley, 2000. ISBN: 9780631221401.

[11] James Christensen et al. "The IEC 61499 Function Block Standard: Overview of the Second Edition". In: Jan. 2012.

[12] James H. Christensen. *A Standard for Software Reuse in Embedded, Distributed Control Systems*. https://www.holobloc.com/papers/iec61499/overview.htm [Accessed: February 2026]. 2022.

[13] European Commission, Directorate-General for Research, and Innovation. *Industry 5.0 – Human-centric, sustainable and resilient*. Publications Office, 2020. DOI: doi/10.2777/073781.

[14] Baudouin Dafflon, Nejib Moalla, and Yacine Ouzrout. "The challenges, approaches, and used techniques of CPS for manufacturing in Industry 4.0: a literature review". In: *The International Journal of Advanced Manufacturing Technology* 113 (7 2021), pp. 2395–2412. ISSN: 1433-3015. DOI: https://doi.org/10.1007/s00170-020-06572-4.

[15] Phyllis Deane. *The first industrial revolution*. Cambridge University Press, 1979.

[16] Carlos A. Garcia et al. "An Approach of Cyber-Physical Production Systems Architecture for Robot Control". In: *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*. 2018, pp. 2847–2852. DOI: 10.1109/IECON.2018.8591286.

[17] Carlos A. Garcia et al. "Robotic Arm Manipulation Under IEC 61499 and ROS-based Compatible Control Scheme". In: *Information and Communication Technologies of Ecuador (TIC.EC)*. Ed. by Miguel Botto-Tobar et al. Cham: Springer International Publishing, 2019, pp. 358–371. ISBN: 978-3-030-02828-2.

[18] Nigel Griffiths. *nmon_analyser: A free tool for producing AIX performance reports*. https://developer.ibm.com/articles/au-nmon_analyser/ [Accessed: February 2026]. 2020.

[19] Peter P. Groumpos. "A Critical Historical and Scientific Overview of all Industrial Revolutions". In: *IFAC-PapersOnLine* 54.13 (2021). 20th IFAC Conference on Technology, Culture, and International Stability TECIS 2021, pp. 464–471. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2021.10.492.

[20] Mario Hermann, Tobias Pentek, and Boris Otto. "Design Principles for Industrie 4.0 Scenarios". In: *2016 49th Hawaii International Conference on System Sciences (HICSS)*. 2016, pp. 3928–3937. DOI: 10.1109/HICSS.2016.488.

[21] Mario Hermann, Tobias Pentek, and Boris Otto. "Design Principles for Industrie 4.0 Scenarios: A Literature Review". In: (Jan. 2015). DOI: 10.13140/RG.2.2.29269.22248.

[22] E.J. Hobsbawm and C. Wrigley. *Industry and Empire: From 1750 to the Present Day*. Industry and empire. New Press, 1999. ISBN: 9781565845619.

[23] Deanna Hood and William Woodall. *ROS 2 update*. 2016.

[24] Niccolò Iannacci et al. "Robotic cell work-flow management through an IEC 61499-ROS architecture". In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016, pp. 1–7. DOI: 10.1109/ETFA.2016.7733528.

[25] Hobsbawnm E. J. *The Age of Revolution, 1789-1898*. 1996. ISBN: 0679772537.

[26] N. Jazdi. "Cyber physical systems in the context of Industry 4.0". In: *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*. 2014, pp. 1–4. DOI: 10.1109/AQTR.2014.6857843.

[27] John Kerr and Kevin Nickels. "Robot operating systems: Bridging the gap between human and robot". In: *Proceedings of the 2012 44th Southeastern Symposium on System Theory (SSST)*. 2012, pp. 99–104. DOI: 10.1109/SSST.2012.6195127.

[28] Edward A. Lee. "Cyber Physical Systems: Design Challenges". In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008, pp. 363–369. DOI: 10.1109/ISORC.2008.25.

[29] Yanan Liu et al. "Formal Analysis and Verification of DDS in ROS2". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2018, pp. 1–5. DOI: 10.1109/MEMCOD.2018.8556970.

[30] Yang Lu. "Industry 4.0: A survey on technologies, applications and open research issues". In: *Journal of Industrial Information Integration* 6 (2017), pp. 1–10. ISSN: 2452-414X. DOI: https://doi.org/10.1016/j.jii.2017.04.005.

[31] Yuqian Lu et al. “Outlook on human-centric manufacturing towards Industry 5.0”. In: *Journal of Manufacturing Systems* 62 (2022), pp. 612–627. ISSN: 0278-6125. DOI: https://doi.org/10.1016/j.jmsy.2022.02.001. URL: https://www.sciencedirect.com/science/article/pii/S0278612522000164.

[32] Steven Macenski et al. “Robot Operating System 2: Design, architecture, and uses in the wild”. In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[33] Saeid Nahavandi. “Industry 5.0—A Human-Centric Solution”. In: *Sustainability* 11.16 (2019). ISSN: 2071-1050. DOI: 10.3390/su11164371. URL: https://www.mdpi.com/2071-1050/11/16/4371.

[34] Gerardo Pardo-Castellote, Bert Farabaugh, and Rick Warren. “An introduction to dds and data-centric communications”. In: *Real-Time Innovations* (2005).

[35] U. C. Patkar et al. “Robot Operating System: A Comprehensive Analysis and Evaluation”. In: *International Journal of Intelligent Systems and Applications in Engineering* 12.7s (2023), pp. 516–520.

[36] Eliseu Pereira, Joao Reis, and Gil Gonçalves. “DINASORE: A Dynamic Intelligent Reconfiguration Tool for Cyber-Physical Production Systems”. In: *Eclipse Conference on Security, Artificial Intelligence, and Modeling for the Next Generation Internet of Things (Eclipse SAM IoT)*. 2020, pp. 63–71. URL: http://ceur-ws.org/Vol-2739/#paper_9.

[37] Gonçalo Pinto et al. “Digital Factory: An Industrial Case Study for Function Block-Oriented Development”. In: *2023 IEEE 9th World Forum on Internet of Things (WF-IoT)*. 2023, pp. 01–06. DOI: 10.1109/WF-IoT58464.2023.10539486.

[38] Rui Pinto. “Self-Immunity in Industrial Cyber-Physical Systems”. PhD thesis. July 2022.

[39] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. en. In: (), p. 6.

[40] Bharti Rana and Sanjay S. Rathore. “Industry 4.0 – Applications, challenges and opportunities in industries and academia: A review”. In: *Materials Today: Proceedings* 79 (2023). International Conference on Startup ventures: Technology Developments and Future Strategies, pp. 389–394. ISSN: 2214-7853. DOI: https://doi.org/10.1016/j.matpr.2022.12.162.

[41] W.W. Rostow. *The World Economy: History & Prospect*. University of Texas Press, 1978. ISBN: 9780292790087.

[42] Klaus Schwab. *The Fourth Industrial Revolution: What it means, how to respond*. Edward Elgar Publishing Ltd., 2024, pp. 29–34. DOI: 10.4337/9781802208818.00008. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85206190805&doi=10.4337%2f9781802208818.00008&partnerID=40&md5=ba8f98c3bcc10ce76142fb127dc08f8a.

[43] Arun Kumar Sharma et al. “A study of trends and industrial prospects of Industry 4.0”. In: vol. 47. 2021, pp. 2364–2369. DOI: 10.1016/j.matpr.2021.04.321.

[44] Thomas Strasser et al. “Framework for Distributed Industrial Automation and Control (4DIAC)”. In: *2008 6th IEEE International Conference on Industrial Informatics*. 2008, pp. 283–288. DOI: 10.1109/INDIN.2008.4618110.

[45] M. Teich and R. Porter, eds. *The Industrial Revolution in National Context: Europe and the USA*. Cambridge University Press, 1996. ISBN: 9780521409407.

[46] Murat Caner Testik and Ozgun Sarikulak. "Change points of real GDP per capita time series corresponding to the periods of industrial revolutions". In: *Technological Forecasting and Social Change* 170 (2021), p. 120911. ISSN: 0040-1625. DOI: https://doi.org/10.1016/j.techfore.2021.120911.

[47] Dirk Thomas et al. *ROS 2.0: Developer preview*. 2014.

[48] Jörg Walter and Malte Grave. "Model-Based Design of Distributed ROS2 Systems Using IEC 61499". In: *2024 28th International Conference on Methods and Models in Automation and Robotics (MMAR)*. 2024, pp. 515–520. DOI: 10.1109/MMAR62187.2024.10680751.

[49] Monika Wenger et al. "A model based engineering tool for ROS component compositioning, configuration and generation of deployment information". In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016, pp. 1–8. DOI: 10.1109/ETFA.2016.7733559.

[50] Tomasz Winiarski. "MeROS: SysML-Based Metamodel for ROS-Based Systems". In: *IEEE Access* 11 (2023), pp. 82802–82815. DOI: 10.1109/ACCESS.2023.3301727.

[51] E.J. Woodhouse. "Curbing overconsumption: challenge for ethically responsible engineering". In: *IEEE Technology and Society Magazine* 20.3 (2001), pp. 23–30. DOI: 10.1109/44.952762.

[52] Gulnara Zhabelova and Valeriy Vyatkin. "Towards software metrics for evaluating quality of IEC 61499 automation software". In: *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. 2015, pp. 1–8. DOI: 10.1109/ETFA.2015.7301502.

[53] Keliang Zhou, Taigang Liu, and Lifeng Zhou. "Industry 4.0: Towards future industrial opportunities and challenges". In: *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*. 2015, pp. 2147–2152. DOI: 10.1109/FSKD.2015.7382284.