UNIVERSITY OF VERONA

DEPARTMENT OF COMPUTER SCIENCE

GRADUATE SCHOOL OF NATURAL SCIENCES AND ENGINEERING

DOCTORAL PROGRAM IN COMPUTER SCIENCE

CYCLE XXXIV

A Model-based Approach for Designing Cyber-Physical Production Systems

S.S.D. ING-INF/05

Coordinator: _____
Prof. Massimo Merro

Tutor: _____
Prof. Franco Fummi

Doctoral Student: _____
Dott. Stefano Spellini

Abstract

The most recent development trend related to manufacturing is called “Industry 4.0”. It proposes to transition from “blind” mechatronics systems to Cyber-Physical Production Systems (CPPSs). Such systems are capable of communicating with each other, acquiring and transmitting real-time production data. Their management and control require a structured software architecture, which is typically referred to as the “Automation Pyramid”. The design of both the software architecture and the components (i.e., the CPPSs) is a complex task, where the complexity is induced by the heterogeneity of the required functionalities.

In such a context, the target of this thesis is to propose a model-based framework for the analysis and the design of production lines, compliant with the Industry 4.0 paradigm. In particular, this framework exploits the Systems Modeling Language (SysML) as a unified representation for the different viewpoints of a manufacturing system. At the components level, the structural and behavioral diagrams provided by SysML are used to produce a set of logical propositions about the system and components under design. Such an approach is specifically tailored towards constructing Assume-Guarantee contracts. By exploiting reactive synthesis techniques, contracts are used to prototype portions of components’ behaviors and to verify whether implementations are consistent with the requirements. At the software level, the framework proposes a particular architecture based on the concept of “service”. Such an architecture facilitates the reconfiguration of components and integrates an advanced scheduling technique, taking advantage of the production recipe SysML model.

The proposed framework has been built coupled with the construction of the ICE Laboratory, a research facility consisting of a full-fledged production line. Such an approach has been adopted to construct models of the laboratory, to virtual prototype parts of the system and to manage the physical system through the proposed software architecture.

Abstract (Italian)

"Industria 4.0" propone di trasformare i tipici sistemi meccatronici "ciechi" in sistemi di produzione ciber-fisici. Tali sistemi sono in grado di comunicare tra loro, acquisendo e trasmettendo in tempo reale i dati di produzione, volti ad analizzare e ottimizzare i processi produttivi. Il controllo di tali sistemi richiede un'architettura software tipicamente concettualizzata nella "piramide dell'automazione". Il design dell'architettura software e dei componenti fisici è un compito complesso, ove la complessità è indotta dall'eterogeneità delle funzionalità richieste dal mercato.

In tale contesto, l'obiettivo di questa tesi è quello di proporre un framework basato su modelli per l'analisi e la progettazione di linee di produzione, conformi al paradigma Industria 4.0. In particolare, questo framework sfrutta il Systems Modeling Language (SysML) come rappresentazione unificata dei diversi punti di vista di un sistema produttivo. Riguardo i componenti fisici, i diagrammi strutturali e comportamentali di SysML vengono utilizzati per produrre una serie di proposizioni logiche sul sistema e sui componenti in fase di progettazione. Tale approccio è appositamente studiato per la costruzione di contratti Assunzioni-Garanzie. Inoltre, sfruttando tecniche di sintesi reattiva, i contratti vengono utilizzati per la prototipazione di componenti di sistema e per verificare se le implementazioni sono coerenti con i requisiti. A livello di software, il framework propone una particolare architettura basata sul concetto di "servizio". Tale architettura mira a facilitare la riconfigurazione dei componenti e integrare un tecnica di schedulazione avanzata, sfruttando il modello SysML della ricetta di produzione.

Il framework proposto è stato sviluppato assieme alla costruzione del laboratorio ICE, una struttura di ricerca costituita da una linea di produzione reale. Tale approccio è stato adottato per costruire il modello del laboratorio, prototiparne delle parti e per gestire il sistema fisico attraverso l'architettura software proposta.

"Distinguish at all times between the model and the real world. You will never strike oil by drilling through the map!"

Mathematical Models: Uses and Limitations
IEEE Transactions on Reliability (1971)

by Solomon W. Golomb

Acknowledgements

First and foremost, I would like to deeply thank my advisor, Prof. Franco Fummi. We started our collaboration when I was a young bachelor's student, trying to find my place in this world. He always believed in me and provided all the guidance and support I needed, to not only accomplish all the results of this thesis, but to develop as a young researcher. A special mention also goes to my co-advisor, Dr. Michele Lora. His busy work life around the world did not stop him in backing me up and pointing me in the right direction (even if this required taking Skype/Zoom calls at crazy hours!). Thanks also to Prof. Tiziano Villa and Prof. Riccardo Muradore of my internal committee, for the fundamental feedbacks they gave me in these years. Receiving opinions from very different points of view is crucial to explore new and not traveled roads.

I would also like to thank my colleagues and friends, to those that are still in UNIVR and those that are gone for more exotic (or better paid!) job experiences. Thank you for the countless conversations that inspired and helped me achieve the work discussed in this dissertation. Thanks also for spending with me the countless light-hearted moments, which I will always remember with great nostalgia.

I would like to profundly thank my family, my mother Maria Grazia and my father Nello. Thank you for believing in me and for supporting me, even in my deadline-caused insanities. Thank you for teaching me that I can be up to every challenge that may arise in my life. Last but not least, thanks to my partner Arianna. We grew up toghether and we shared the most important achievements of our lives. Thank you for your support and your love. This is only the beginning.

Thanks to all of you,

Stefano

Contents

List of Figures IV

List of Tables VII

List of Listings IX

1 Introduction 1
- 1.1 Introduction 1
- 1.2 Methodology Flow 2

2 Background 5
- 2.1 Modeling 5
  - 2.1.1 AutomationML 5
  - 2.1.2 DIN 8580 Standard 6
  - 2.1.3 ISA-95 Standard 6
  - 2.1.4 System Modeling Language (SysML) 7
- 2.2 Contract-based Desgin and Synthesis 9
  - 2.2.1 Assume-Guarantee (A/G) Contracts 9
  - 2.2.2 Linear Temporal Logic (LTL) 9
  - 2.2.3 General Reactivity (GR(1)) 10
- 2.3 Guiding Case-study Infrastructure 11

3 Cyber-Physical Production Systems Modeling 13
- 3.1 Related Works 15
- 3.2 Methodology Overview 17
- 3.3 Architectural Models Reuse 19
  - 3.3.1 Use Case Basic Components 21
  - 3.3.2 Use Case System Structure 23
- 3.4 Architecture Refinement and CPPSs Design Flow 24

3.4.1 Components Communication Modeling 24
3.4.2 Components Behavior Modeling 26
3.5 Production Recipes Models 27
3.5.1 Task Level 28
3.5.2 Service Level 29
3.5.3 Machine Function Level 31
3.6 Concluding Remarks 32

4 Compositional Design using Assume-Guarantee Contracts 33
4.1 Preliminaries and Related Work 36
4.1.1 The Robot Operating System (ROS) 37
4.1.2 Case study: goods transportation system 38
4.2 Overview 40
4.3 Design problem specification 43
4.3.1 System Contract (C_S) 43
4.3.2 Robot Contract (C_R) 45
4.3.3 Environment Contract (C_E) 45
4.3.4 Mission Contract (C_M) 46
4.4 Synthesis and validation 47
4.4.1 Code generation and simulation 47
4.4.2 Task allocation 47
4.4.3 Simulation for system validation 50
4.5 ROS Code Generation 50
4.6 Experimental results 52
4.6.1 Methodology evaluation 54
4.6.2 Software deployment: Gazebo simulation 55
4.7 Concluding Remarks 58

5 Virtual Prototyping using Assume-Guarantee Contracts 59
5.1 Related Works 61
5.2 Methodology Overview 62
5.2.1 Case study: additive manufacturing and assembly 64
5.3 Components formalization 65
5.4 Code Generation and Application 69
5.4.1 Executable models generation 69
5.4.2 Plant Simulation C-Interface import 70
5.4.3 Experimental Results 71
5.5 Conclusion 74

6 Application to a Service-Oriented Manufacturing Architecture . . 75
6.1 Background . . . 77
6.1.1 Levels of Automation in Manufacturing . . . 77
6.1.2 Communication Protocols . . . 78
6.1.3 Production Scheduling . . . 79
6.2 Services and Data Collection Architecture . . . 80
6.2.1 OPC UA Servers . . . 80
6.2.2 Data Collection Infrastructure . . . 81
6.3 Service-Automation Manager . . . 82
6.3.1 Drivers . . . 82
6.3.2 Core . . . 83
6.3.3 Applications . . . 84
6.4 Architecture Evaluation . . . 85
6.4.1 Qualitative Analysis . . . 85
6.4.2 Overhead Analysis . . . 86
6.5 Services and Models-based Advanced Scheduling . . . 87
6.5.1 Scheduling Problem Statement . . . 88
6.5.2 Scheduling Algorithm . . . 89
6.5.3 Experimental Setup . . . 91
6.5.4 Implementation . . . 92
6.5.5 Results and discussion . . . 93
6.6 Conclusion . . . 95

7 Modeling in Industry 5.0: What Is Missing . . . 97
7.1 From Industry 4.0 to Industry 5.0 . . . 99
7.2 Modeling Industry 5.0: what is missing and possible directions . 102
7.2.1 Uncertainty Requirement . . . 103
7.2.2 Cognition Requirement . . . 104
7.2.3 Safety Requirement . . . 105
7.2.4 SysML support to Industry 5.0 models . . . 105
7.3 Conclusion . . . 106

8 Summary of the Experimental Results . . . 107
8.1 Compositional Design using Assume-Guarantee Contracts . . . 107
8.2 Virtual Prototyping using Assume-Guarantee Contracts . . . 109
8.3 Application to a Service-Oriented Manufacturing Architecture . 111

9 Conclusion and suggestions for future research . . . 115
9.1 Summary of the proposed approach . . . 115
9.2 Directions for future research . . . 117

Summary of the proposed innovative contributions 119
Cyber-physical production systems modeling 119
Compositional design using assume-guarantee contracts 120
Virtual prototypization using assume-guarantee contract 120
Model-based and service-oriented advanced scheduling 120

List of Acronyms 121

References 123

Appendices 133

A Robotic design code 133

B Contract-based Specifications 135

List of Figures

1.1 Overview of the conceptual flow proposed by this thesis. 3

2.1 Schematization of the automation pyramid referenced in Industry 4.0. 7
2.2 SysML diagrams hierarchy, clarifying the diagram types that are adopted or adapted from Unified Modeling Language (UML). 8
2.3 Structure of the advanced manufacturing production line used as a case study in this work. The machines are connected through an articulated software-controlled transportation mechanism. 12

3.1 Depiction of the proposed modeling methodology. The top-down phase (1) implements a modeling approach for production processes and requirements, to map onto the bottom-up platform of plant models (2). Such a platform is constructed enabling models reuse from Automation Markup Language (AML). 13
3.2 AML libraries of SystemUnit classes, roles and interfaces of the *BayGate* system. 21
3.3 Visual representation of the Block Definition Diagram (BDD) derived from AML object classes. 22
3.4 AML InstanceHierarchy representing the *BayGate* sub-system. . . 23
3.5 The SysML Internal Block Diagram (IBD) resulting from the AML InstanceHierarchy. 24
3.6 The refined IBD with additional information about object flows and directions. 25
3.7 *activity diagram* in SysML characterizing components' behaviors. 26

3.8 An example of the proposed three-layer representation. The first layer defines the set of tasks and machines on which they can be allocated. The second layer depicts the concept of "service", which are specified in a control flow graph. Finally, the third layer outlines machines' functions and, therefore behaviors close to Programmable Logic Controllers (PLCs). 28
3.9 An example of the task level of our modeling strategy. The recipe is defined by a set of tasks (in gray), connected to required pieces of equipment (in violet) and materials (in cyan). 29
3.10 The implementation of the "Assemble" task, made of a constrained flow of multiple services. 30
3.11 An implementation example of the "Turn" service. It is composed by two machine functions, executed depending on the actual and target angles of the material. 31

4.1 Overview of problem tackled by the presented approach, and proposed problem decomposition. 33
4.2 Running example: an autonomous goods transportation system, composed of two robots, coordinating to reach six different targets. 39
4.3 General schema of the proposed methodology. Requirements and specifications are formalized and decomposed as a set of *Assume-Guarantee (A/G) contracts* (Step 1). After solving the sub-problems (Steps 2 to 6), their composition is validated through simulation (Step 7). In case the validation results in a positive outcome, Robot Operating System (ROS) code implementing the control strategy of each robot is automatically produced (Step 8). 40
4.4 Weighted graph extracted from the case study. It represents the costs of all the paths from robots to targets. 48
4.5 The case study analyzed in this chapter represented in a Gazebo simulation. Through publisher/subscriber architecture, each robot subscribes for messages from the environment controller, which publishes the set of sensor values and each agent position. A robot command is then produced by the implemented controller, wrapped in another ROS node. 56

5.1 Summary of the contribution: the production line is formalized by a set of contracts under the guidance of the DIN 8580 Standard. The contracts are used for the automatic synthesis of a virtual prototype of the production line. 59

5.2 Overview of the presented approach: starting from a taxonomy of industrial machines, elementary actions associated with this machine are defined. Then, an A/G contracts library defining each action is assembled and synthesized, generating a Plant Simulation-compatible model that can be imported and simulated. 63
5.3 3D representation of the parts that compose the final product of the case study. The (1) and (2) pieces are gathered from the warehouse, while (3) has to be 3D printed. 64
5.4 Interactions between the coordinator contract and the turn action contract, implemented by a set of input/output variables. 68
5.5 The ICE case study being simulated into Plant Simulation. 72

6.1 The classic *Automation Pyramid* (on the left, colored in blue), and the proposed modification of the software architecture (colored in red). The architecture is modified by adding the *Automation Manager* presented in this work. 75
6.2 Automation of the *Supervisory level* by introducing the proposed service-oriented architecture. Arrows show the commands and data flow through the automation pyramid and our novel architecture. 81
6.3 The internal structure of the *Automation Manager*. 83
6.4 The figure shows the experimental setup used to assess the methodology. The three-level model of the production processes is built by modeling the production recipes and the manufacturing line equipment. Then, this representation is used to implement a service-based scheduler. Lastly, the scheduler is executed in a real production environment. 91
6.5 Service-level model of the task *"T2"* represented as a SysML activity diagram. It expresses all the information contained in the second level of our proposed model. Each node is specified by a typed object. The arrows describe the edges of the graph in the middle layer of Figure 3.8. 93

7.1 While in the vision proposed by Industry 4.0 the human is mainly supervising and "observing" the system, Industry 5.0 aims at putting the human back at the center of manufacturing. To pursuit such a task, models of manufacturing systems must be adjoined with models of human behaviors and, thus, novel modeling methodologies must be investigated. 97

7.2 The automation pyramid: the starting point of Industry 4.0. Role of OPC-UA as a unifying protocol focused on data. 100
7.3 Relations between Industry 5.0 requirements asserted in Sec. 7.1, systems types and possible associations with SysML diagrams. 102

List of Tables

3.1 Mapping of AML elements to SysML objects. 19

4.1 Comparison between the time needed to obtain the final control strategy using the holistic system contracts, and decomposing the design problem. The experiments have been carried on by varying the three main dimensions of the problem. 53
4.2 Qualitative comparison of the generated code. 55
4.3 Results obtained by the Gazebo simulation. 57

5.1 Sketch of the coordinator contract. 66
5.2 Sketch of the turn operation contract. 67
5.3 Time required to perform the consistency checking and synthesis step, and the code generation for the non-decomposed (*i.e.*, holistic) system specification, and for the different actions of the robotic assembling station. The numbered columns refer to the machine's elementary actions: (1) Compose, (2) Decompose, (3) Pick, (4) Place, (5) Move, (6) Turn. The last column reports the time required to obtain the machine coordinator. 72
5.4 Time required to perform the consistency checking and synthesis step for all the machines in the production line. The last column reports the total time required for the entire production line. 73

6.1 Comparison of functionalities available when using the traditional software stack against the proposed Service Oriented Architecture (SOA)-enabled architecture. 86

6.2 Comparison between the communication delay derived from a direct connection with OPC Unified Architecture (OPC UA) and with the proposed architecture. 87
6.3 Comparison between the execution time when using the state-of-the-art and the proposed architecture to govern three different complete production recipes. 88
6.4 Comparison between the scheduling of 450 production recipes, using a classical Resource Task Network (RTN)-based representation against the proposed hierarchical modeling approach. 94

8.1 Comparison between the time needed to obtain the final control strategy using the holistic system contracts, and decomposing the design problem. The experiments have been carried on by varying the three main dimensions of the problem. 108
8.2 Qualitative comparison of the generated code. 108
8.3 Results obtained by the Gazebo simulation. 109
8.4 Time required to perform the consistency checking and synthesis step, and the code generation for the non-decomposed (*i.e.*, holistic) system specification, and for the different actions of the robotic assembling station. The numbered columns refer to the machine's elementary actions: (1) Compose, (2) Decompose, (3) Pick, (4) Place, (5) Move, (6) Turn. The last column reports the time required to obtain the machine coordinator. 110
8.5 Time required to perform the consistency checking and synthesis step for all the machines in the production line. The last column reports the total time required for the entire production line. 111
8.6 Comparison between the communication delay derived from a direct connection with OPC UA and with the proposed architecture. 111
8.7 Comparison between the execution time when using the state-of-the-art and the proposed architecture to govern three different complete production recipes. 112
8.8 Comparison between the scheduling of 450 production recipes, using a classical RTN-based representation against the proposed hierarchical modeling approach. 112

B.1 Sketch of the *move* action contract. 135
B.2 Sketch of the *pick* action contract. 136

B.3 Sketch of the *place* action contract. 136

List of Listings

4.1 Code structure used to implement and instantiate the control software into a ROS node. At each update, input message parameters are gathered and passed to the control strategy instance. This produces new output values, that are published to the destination topic. 51

5.1 Sketch of the `simulate` C-Interface function that performs a property abstraction operation over MU physical properties, then it calls the simulate of the synthesized controller and finally back-propagates time, power and MU properties to the simulator. 69

5.2 Sketch of the *EntranceControl* of the *ManipulatorsStation* object in Plant Simulation (written using *SimTalk* language). . . . 71

A.1 Code structure of the synthesized control software. The control logic is implemented by the *executeMachine* function, that models the behavior of an FSM. 133

A.2 Code structure of a generic ROS node. 134

1

Introduction

1.1 Introduction

Market trends of the 21st Century are characterized by high product demands along with high degrees of customization. Traditional production paradigms are not consistent with modern market requirements. Furthermore, manufacturing technologies must evolve to cope with the increasing unpredictability of modern society conditions while guaranteeing cost-efficiency. In this regard, the COVID-19 global pandemics of 2020 is the perfect example: today's manufacturing systems are not ready to efficiently respond to the disruption of supply chains due to sudden shifts of market critical requirements (*e.g.*, Personal Protective Equipment (PPE)) [1]. "Industry 4.0" [2] is meant to assist this transformation, proposing a set of *production systems development guidelines* to a wide range of engineering disciplines, from systems design to product development. Among the promises of the Industry 4.0 trend [3], the concept of *reconfigurability* in manufacturing systems stands out as a key factor to quickly react and adapt the production to frequent and sudden market changes [4].

Such trends are constantly enriching traditional production systems with computational and communication infrastructures, transforming manufacturing lines into every day more complex systems. *Industry 4.0* is particularly pushing the adoption of Cyber-Physical Systems (CPSs), Cloud Computing and Internet of Things (IoT) into systems [3]. Therefore, "blind" manufacturing systems are evolving into connected and intelligent *Cyber-Physical Production Systems (CPPSs)*, also named Industrial CPSs. Indeed, such a transformation introduces unprecedented challenges in their design and optimization [5]: it requires the ability to design systems while considering the production processes, as well as the complex computational infrastructure monitoring and controlling the production processes.

The ongoing transformation is particularly problematic for Small and Medium Enterprises (SMEs). While a large manufacturing corporation may consider redesigning their production plants from scratch to incorporate novel technologies, SMEs are often forced to gradually introduce intelligence in their already existing lines. Furthermore, companies must be able to evaluate in advance the impact of re-designing their production lines. As such, a methodology capable of representing different systems' viewpoints and, thus, enabling a holistic approach to the design, is still missing.

In particular, different standards and languages have been developed in the past to organize the knowledge within the manufacturing context. As an example, the Automation Markup Language (AML) [6] has been proposed for the architectural and plant topology view, while the International Society of Automation (ISA)-95 [7] standard has been used for the business level and the Manufacturing Operations Management (MOM). Therefore, multiple languages are extremely specialized to represent a single aspect, but an all-encompassing approach does not exist. When (re-)designing, configuring, or optimizing a production system, the complexity induced by the presence of such heterogeneous information is hardly bearable if tackled manually. Thus, design automation and model-based techniques become crucial to constructing complex manufacturing systems.

For such reasons, this thesis proposes a design framework for CPPSs based on models. This framework exploits the expressivity of the System Modeling Language (SysML) to guarantee a complete and familiar modeling environment. To enable reuse, an approach is also proposed to import models expressed using other languages (*i.e.*, AML). Furthermore, the formalization of the system's specifications into Assume-Guarantee (A/G) contracts allows designers to verify different properties as well as produce system's implementations.

1.2 Methodology Flow

The overall conceptual framework proposed by this thesis, named Modeling, Formalization & Design for Industry (MOOD4I), is depicted in Figure 1.1. The methodology flow starts from a set of standards and languages to describe various aspects of the production line. In particular, the framework takes as input the production recipe expressed through a graph representation named Resource Task Network (RTN). Furthermore, the structural view of the plant is defined by handling AML specifications and mapping such knowledge to SysML diagrams. The framework also allows importing production requirements structured using the ISA-95 standard. Therefore, one of the objectives

Fig. 1.1: Overview of the conceptual flow proposed by this thesis.

of the proposed framework is to allow designers reusing descriptions of parts of the system that may be already available.

The core of MOOD4I is composed of the SysML language and A/G reasoning through contracts. On the one hand, SysML provides an intuitive and complete modeling language for a broad range of systems. It enables the specification of multiple system's viewpoints, from the architecture to behaviors. Therefore, the proposed design flow can map the system's features described by the input representations to SysML diagrams. On the other hand, MOOD4I exploits A/G contracts to decompose the system design problem, to verify properties (*e.g.*, *feasibility*) and to synthesize implementations using reactive synthesis algorithms. Such a set of techniques provide a connection between formal reasoning and model-based design, especially applied to production systems. Furthermore, models can also be used to enhance the knowledge provided to simulation and scheduling techniques. Particularly in service-oriented production architectures, the concept of service abstracts the bare-metal functionality provided by the piece of equipment, hiding implementation details. While this factor may be useful for process designers, the hidden details could be used by scheduling or optimization techniques to improve the quality of the output. As such, MOOD4I also proposes a modeling strategy for the production process that can scale on the requested level of detail. Based on Figure 1.1, the thesis is structured as follows:

- Chapter 2 presents the necessary background to understand the details of the proposed design framework.
- Chapter 3 describes the entire modeling strategy. It defines how to reuse AML descriptions and the mapping to SysML diagrams. It also introduces the hierarchical model of production recipes, designed to expand the knowledge regarding the production process compared to RTN.

In addition, this thesis proposes to handle the encapsulated knowledge within the models to formalize the design problem through a set of contract-based specifications. Such a set of A/G contracts are employed in two different design flows. In particular:

- Chapter 4, offering compositional contract-based design applied to a robotic system. It also presents how the constructed implementations are integrated into a robotic operating system.
- Chapter 5, explaining a methodology to build a virtual prototype of the production line, starting from a set of A/G contracts. This Chapter also presents how to simulate such a prototype in a plant simulator, to validate its behaviors.

Another approach to exploit the set of developed models is presented in Chapter 6. Here, we describe a service-oriented platform and a software architecture to control manufacturing operations. Such a complex architecture is particularly effective to ease the implementation of reconfigurable manufacturing systems. In fact, Chapter 6 also presents an advanced scheduling algorithm: it exploits both the software architecture and the knowledge of the process enclosed in the hierarchical recipe model presented in Chapter 3. Chapter 7 analyzes what's missing in current languages and modeling methodologies, to be able to capture the requirements of an industry trend that repositions the human at the center of the production: Industry 5.0. Chapter 8 collects all the results of the methodologies proposed by this thesis. Finally, Chapter 9 draws some conclusions and describes possible future works.

2

Background

This chapter provides a set of preliminary concepts that are necessary to understand the methodologies proposed by this thesis. In particular, Section 2.1 focuses on languages and standards aiding the proposed modeling strategy for CPPSs. Section 2.2 defines the concept of A/G contracts and reactive synthesis from contract-based specializations. Finally, Section 2.3 defines the production line used as case-study, guiding the development and providing a real application platform to assess the qualities of our methodologies.

2.1 Modeling

Among the plethora of modeling languages for representing CPPSs, this thesis relies on AML and SysML. Developed specifically for the automation aspect, we exploit the AML standard to specify production systems under the structural viewpoint (*i.e.*, topology). To guide the definition of elementary actions of machines, the Deutsches Institut für Normung (DIN) standard provides different standards related to multiple classes of machinery. In particular, we concentrate on the DIN 8580, providing a taxonomy of manufacturing processes. From a more business-oriented viewpoint, the ISA-95 standard is used to fix a production process common terminology. On top of all those stands SysML, a quite general-purpose system engineering modeling language. SysML is capable of representing both the structural and production processes' viewpoints.

2.1.1 AutomationML

AML is an XML-based data format to exchange information describing manufacturing systems [8]. Different standards are intertwined within AML to

describe multiple aspects of production plants: from topology to the logic controlling machines' microcontrollers.

The Computer Aided Engineering Exchange (CAEX) (IEC 62424) standard provides to AML the features required to represent a topological view of the system, *i.e.*, relations between objects, such as types of machinery and materials. It is object-oriented [9] as it provides system objects' semantics using roles which are defined by a *role class* library. Role classes express the abstract functionality representation of objects, without specifying their implementations. As an example, a "resource" is a role for an object, and can be further detailed to represent a piece of equipment or material. Concrete resource instances are typically specified by *system unit classes* usually containing vendor-specific AML objects. Relations between objects are specified within the *interface classes* library. Furthermore, AML descriptions can be hierarchically organized. In particular, AML's core is the *instance hierarchy*, storing the hierarchy of components and sub-components composing the system.

2.1.2 DIN 8580 Standard

The DIN 8580 standard [10] defines a wide set of processes, products, activities, and facilities connected to the industrial domain. A *manufacturing process* is the production or the transformation of a workpiece. A process can be divided into multiple sub-processes, each of them changing or forming a different property or shape of the processed product. The DIN states that every manufacturing process can be classified into five main groups, according to the type of material transformation they provide, in particular: *Primary shaping*, *Forming*, *Cutting* , *Joining*, *Coating*. These main groups are divided into sub-groups, further characterizing processes by delineating elementary actions associated to the concept of manufacture, *i.e.*, the Joining group is divided into *operations* such as assembling, fastening, soldering, etc.

2.1.3 ISA-95 Standard

The ISA-95 standard has been developed to define the interface between the enterprise structure and the control systems [11]. The standard defines three main categories of interest, each defining a set of information models interfacing the different parts of a manufacturing company: *information models between business and manufacturing operation systems*, *information models for activities defined in manufacturing operation systems*, and *information models within manufacturing operation systems*.

As depicted in Figure 2.1, the automation pyramid is assumed to be composed of 5 different levels, each managed by different systems and different

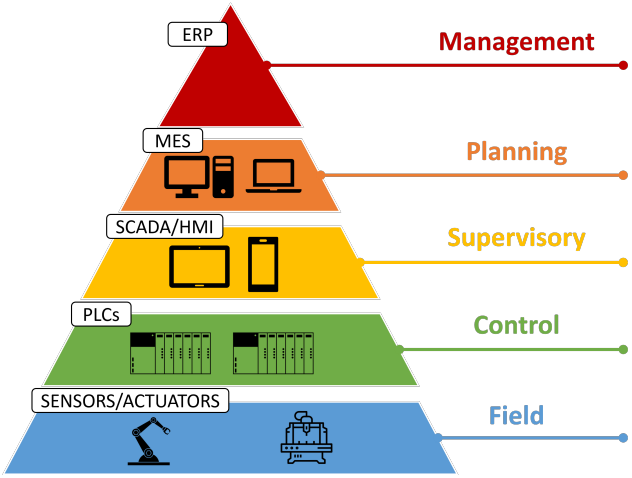

Fig. 2.1: Schematization of the automation pyramid referenced in Industry 4.0.

timeframes. We are interested specifically on levels 3 and 4, operating respectively on Manufacturing Operations and Control Information and Business planning and logistics. Parts 1, 2 and 5 of the standard are dedicated to the proposition of a consistent terminology between automation Levels 4 and 3, to bridge the information gap between two different manufacturing views. More precisely, a set of data models formally details the types of information that pass through both systems, such as the *Product definition* model, which describes processes and requirements to make a product, or the *Resource definition* model, that characterizes available resources such as pieces of equipment, materials and also personnel.

2.1.4 SysML

For software and systems model engineering, one of the most used languages is SysML. As depicted in Figure 2.2, it provides a set of diagrams over those provided by UML, to represent systems and systems-of-systems in addition to plain software. As such, it is natively capable of representing manufacturing systems and expressive enough to enable performing analysis over models. Other than native SysML, specializations have been proposed [12] to aid the development of automation software (*i.e.*, Programmable Logic Controllers (PLCs)) for smart manufacturing systems. Furthermore, models can be used to automatically generate control software to directly integrate into machines. SysML has also been used to ease the development and the integration of a Manufacturing Execution System (MES) in a production line [13].

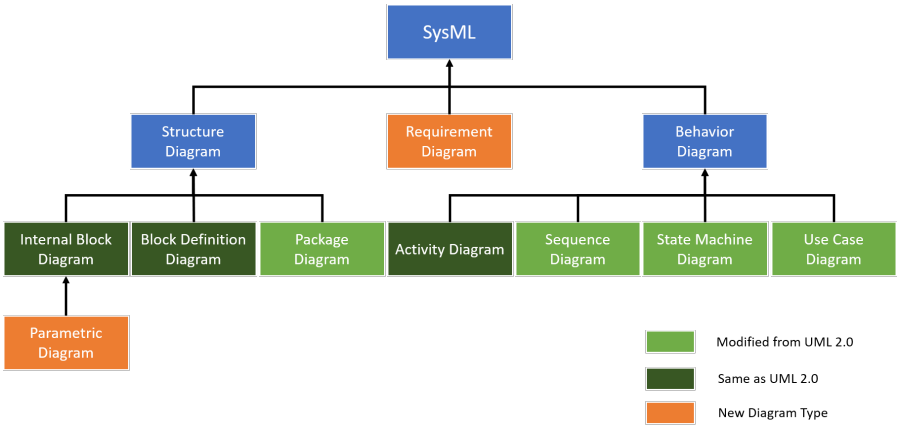

Fig. 2.2: SysML diagrams hierarchy, clarifying the diagram types that are adopted or adapted from UML.

SysML models may be used for many purposes within the design flow of industrial production systems. We hereby exemplify some of them:

- *Verification and Validation*, to evaluate the correctness of requirements and behaviors by models for mission-critical applications, such as industrial systems [14]. So far, methods for the verification and validation of SysML models rely either on formal methods or simulation [15];
- *System Analysis and Optimization*, in which accurate system models may be used to perform in-depth analysis and optimizations. Design-space exploration is intrinsic concept of any Platform-Based Design (PBD) [16] flow, that may be performed on top of sufficiently expressive SysML models [17]. Optimization problems and formal models can be built on top of SysML models [18, 19] and then resolved exploiting existing solvers;
- *Code Generation and Implementation* not limited to software. In fact, SysML is also efficient at capturing features of hardware components and their interactions with the software components. Thus, it may become a fundamental tool to support hardware-software integration. For instance, SysML models may come in handy while integrating a Manufacturing Execution System into a production line [20].

SysML also allows the generation of control software starting from diagrams composing a system model. In particular, SysML diagrams can be used to generate the templates for PLC software consistent with the IEC 61131-3 standard [21] to be later deployed on the system.

2.2 Contract-based Desgin and Synthesis

In this section, we provide a preliminary definition of the theory behind A/G reasoning through contracts. Contract-based specifications can be expressed using different formal languages, such as Linear Temporal Logic (LTL). Furthermore, we describe a contracts synthesis methodology, based on the General Reactivity of rank 1 (GR(1)).

2.2.1 Assume-Guarantee (A/G) Contracts

A contract C for a component M is a triple (V, A, G), where V is the set of the component variables, and A and G are *assertions*, each representing a set of behaviors over V [22]. A represents the *assumptions* that M makes on its environment, and G represents the *guarantees* provided by M under the environment assumptions.

A component M satisfies a contract C whenever M and C are defined over the same set of variables, and all the behaviors of M satisfy the guarantees of C in the context of the assumptions, meaning that M is an implementation of C. Moreover, a component E can also be associated with a contract C as an environment for the contract. We say that E is a legal environment of C, whenever E and C have the same variables and the behaviors implemented by E are a subset of A. The A/G contract theory [22] defines a set of operations, those used in this work are:

- *Composition*: Contracts associated to different components can be combined according to different rules. *Parallel composition* builds complex contracts from simpler ones.
- *Compatibility and Consistency*: C is *compatible* if there exists a legal environment E for it. A contract is *consistent* when the set of implementations satisfying it is not empty.
- *Refinement*: A contract C refines a contract C', written $C \preceq C'$, if and only if $A \supseteq A'$ and $G \subseteq G'$. Conceptually, a contract C refines a contract C' if it relaxes its assumptions while strengthening its guarantees.

2.2.2 Linear Temporal Logic (LTL)

LTL formulas are perfectly suited to model the evolution of a system over time. A component behaviour is expressed considering present and future paths, *i.e.* a condition that will eventually hold in the future. Specifically related to A/G contracts, both assumptions A and guarantees G of a contract C can be specified as LTL formulas [23]: a component M satisfies a contract C if it fulfills the logical implication $A \rightarrow G$, while it is a legal environment for C if it satisfies the formula A [24].

LTL Syntax

Let AP be a set of atomic propostions where $\pi \in AP$ is a Boolean variable. LTL formulas are constructed from atomic propostions $\pi \in AP$ according to the following grammar:

$$\varphi ::= \pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \varphi \tag{2.1}$$

where $\neg$ ("not") and $\vee$ ("or") are Boolean operators, and $\bigcirc$ ("next") and $\mathcal{U}$ ("until") are temporal operators. Given "next" ($\bigcirc$) and "until" ($\mathcal{U}$) operators, additional temporal operators can be derived such as "eventually": $\Diamond\varphi = True\mathcal{U}\varphi$ and "always": $\Box\varphi = \neg\Diamond\neg\varphi$.

LTL Semantics

Semantics of an LTL formula φ are defined on an infinite sequence $\sigma = \sigma_1\sigma_2...$ of truth assignments to the atomic propostions $\pi \in AP$, where σ_i denotes the set of atomic propostions that are True at position i. Whether σ satisfies LTL formula φ at position i (denoted $\sigma, i \models \varphi$) is recursively defined as:

- $\sigma, i \models \varphi$ iff $\pi \in \sigma_i$,
- $\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$,
- $\sigma, i \models \varphi_1 \vee \varphi_2$ iff $\sigma, i \models \varphi_1$ or $\sigma, i \models \varphi_2$,
- $\sigma, i \models \bigcirc\varphi$ iff $\sigma, i+1 \models \varphi$
- $\sigma, i \models \varphi_1 \mathcal{U} \varphi_2$ iff there exists $k \geq i$ such that $\sigma, k \models \varphi_2$, and for all $i \leq j < k$, $\sigma, j \models \varphi_1$.

The formula $\bigcirc\varphi$ expresses that φ is True in the next "step" (the next position in the sequence) and the formula $\varphi_1 \mathcal{U} \varphi_2$ expresses the property that φ_1 is True until φ_2 becomes True.
The sequence σ satisfies formula φ if $\sigma, 0 \models \varphi$. The sequence σ satisfies formula $\Box\varphi$ if φ is True in every position of the sequence, and satisfies the formula $\Diamond\varphi$ if φ is True at some position on the sequence.

2.2.3 General Reactivity (GR(1))

LTL formulas can be used as specifications of reactive systems where atomic propositions are divided between the *environment* (*i.e.*, the system input) and the *system* (*i.e.*, the system output). The realizability of LTL is 2-EXPTIME-complete, which makes it practically infeasible [25]. However, for the GR(1) fragment of the LTL, there is an algorithm able to decide the realizability in N^3 [26]. GR(1) synthesis specifications contain assertions over initial states, safety constraints relating to the current and next state, and goals requiring

that an assertion holds infinitely often during a computation. More specifically, a GR(1) synthesis problem is defined as a game between a system player and an environment player, with the following game structure:

- $\mathcal{X}$ input variables controlled by the environment
- $\mathcal{Y}$ output variables controlled by the system
- θ^e assertion over $\mathcal{X}$ characterizing initial states of the environment
- θ^s assertion over $\mathcal{X} \cup \mathcal{Y}$ characterizing initial states of the system
- $\rho^e(\mathcal{X} \cup \mathcal{Y}, \mathcal{X})$ transition relation of the environment
- $\rho^s(\mathcal{X} \cup \mathcal{Y}, \mathcal{X} \cup \mathcal{Y})$ transition relation of the system
- $\varphi = GFJ^e \rightarrow GFJ^s$ winning condition as implication between justice goals J^e of the environment and J^s of the system.

The acceptance condition is finally defined as:

$$(\theta^e \wedge G\rho^e \wedge GFJ^e) \rightarrow (\theta^s \wedge G\rho^s \wedge GFJ^s) \tag{2.2}$$

where $G\rho^e$ and $G\rho^s$ are safety conditions over the environment and the system while GFJ^e and GFJ^s are liveness properties over the environment and the system.

In the literature, many different tools implement reactive synthesis from the GR(1) fragment [27, 28]. These tools accept a GR(1) LTL specifications and return a Mealy Machine implementing a control strategy that allows the system player to win over the environment player. In this work, we are going to use the GR1C tool of the Temporal Logic Planning (TuLiP) toolbox [27] to perform reactive synthesis from GR(1) specifications.

2.3 Guiding Case-study Infrastructure

The methodologies proposed in this thesis have been applied to a manufacturing system available at our research facility[1]: the Industrial Computer Engineering (ICE) laboratory. The laboratory consists of a full-fledged production line, structured as depicted in Figure 2.3. The production plant is composed (from right to left) of a milling machine, a 3D printer, a collaborative robotic assembly cell composed of two robotic arms composing an assembly station, and a Quality Checking (QC) station. Raw materials, the components, as well as the final products, are stored in an automated warehouse.

The transportation system is made by a closed-loop main conveyor belt. Multiple conveyor bays are linked to the main belt to move the materials from the transportation system to the machines and back. The passage of material

[1] https://www.icelab.di.univr.it/

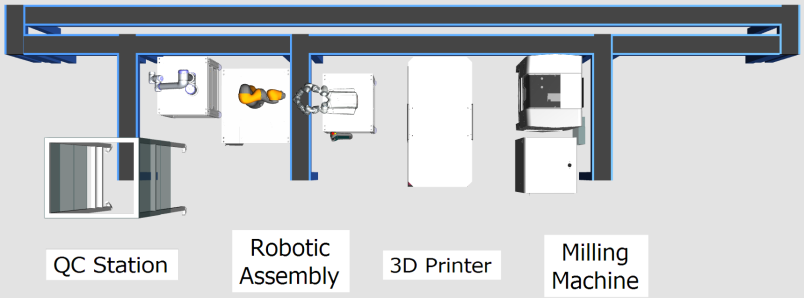

Fig. 2.3: Structure of the advanced manufacturing production line used as a case study in this work. The machines are connected through an articulated software-controlled transportation mechanism.

between the main belt and each bay is managed by a switching mechanism that is guided by sensors detecting and identifying the minipallets moving around the production system.

3

Cyber-Physical Production Systems Modeling

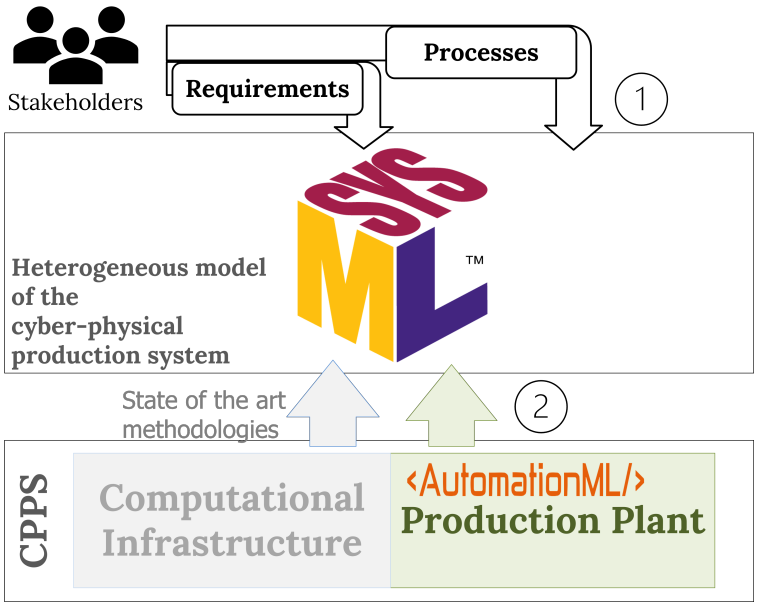

Fig. 3.1: Depiction of the proposed modeling methodology. The top-down phase (1) implements a modeling approach for production processes and requirements, to map onto the bottom-up platform of plant models (2). Such a platform is constructed enabling models reuse from AML.

The first point of the framework proposed by this thesis is a modeling strategy for CPPS. To acquire and structure knowledge about the system, this chapter describes modeling and design flows based on the PBD paradigm [16]. The methodology, outlined in Figure 3.1, supports both the *top-down modeling of requirements and functionalities*, as well as the *bottom-up reuse of components* already existing in the system and available to designers. Core to the methodology is a language able to capture concepts belonging to both "cy-

ber" and "physical" concepts of today's production systems. We chose to rely on SysML [29] for such a task, as it provides a variety of heterogeneous diagrams able to capture the structure, behavior and requirements of systems.

However, SysML falls short when used to carry information about already existing components. For this reason, in the past, methodologies have been developed to convert and import already existing domain-specific models into SysML models. This task has been already carried out extensively for software [30] and hardware [31] components, and network infrastructure [32] enabling modeling of CPS. To enable the design of CPPSs, it is necessary allowing to import models of already existing machines. For this reason, this work proposes a methodology to extract information from AML [6] models of production lines to produce SysML structural diagrams to be used for the design of CPPSs.

Furthermore, CPPSs must be capable of carrying out convoluted production processes, that are implemented by multiple machine operations. Therefore, the complexity of modern production recipes is constantly increasing, especially when dealing with *on-demand production* and *agile manufacturing*. Such principles require restructuring how the concept of "process" (*e.g.*, production recipe) is being expressed and represented. This is especially true when CPPSs are implemented according to the principles of Service Oriented Manufacturing (SOM) [33]: it proposes organizing the work of the machines as a set of *services*. Each service carries out a specific machine's functionality and can be executed on-demand by the system's software architecture. Therefore, tasks composing a recipe may be further characterized as a set of services. In addition, services may have one or more pre- and post-conditions for their correct execution, as well as dependencies with other services. Thus, tasks may be implemented as a logical flow of services, with branches and cycles. Indeed, such a transformation introduces unprecedented challenges in the design of manufacturing systems [5], as it requires dealing with heterogeneous systems implementing complex behaviors to carry out multiple production processes. To represent complex recipes and, therefore, to include the largest set of information on the production process, this chapter presents a multi-level, hierarchical modeling strategy for production processes. Such a modeling strategy is specifically oriented towards enabling the mapping of the machine's functionalities to recipe tasks and, therefore, bridging the automation level with planning/scheduling viewpoints.

The contributions presented in this chapter are:

- a modeling strategy for CPPSs, based on the PBD theoretical framework, where functionalities and requirements are mapped to a bottom-up abstraction of components;

- an implementation of such a modeling strategy in a state-of-the-practice systems modeling language, able to capture the structure of the system, as well as its set of implemented behaviors and requirements;
- to enable components reuse, a mapping between the AML language and SysML;
- a complete and hierarchical representation of production processes, exploiting the concept of tasks, services and machine functions. The relations between such constructs and, thus, the integrated knowledge, allow defining a complex production recipe that can be mapped to a subset of platform components. As such, the overall methodology can be exploited to analyze and explore the design space;
- the application of the aforementioned modeling strategy to a concrete *case-study*, consisting in a full-fledged production line implementing a SOM architecture, different production cells and a flexible transportation system.

This chapter is oganized as follows: Section 3.1 provides a literature analysis on modeling languages and methodologies for CPPSs. Section 3.2 describes the general approach, while Section 3.3 and 3.4 offer more details on the modeling strategy for the architecture and the processes. Then, Section 3.6 draws our conclusions.

3.1 Related Works

In the context of manufacturing and CPPSs, Model-based System Engineering (MBSE) is a quite popular approach to support the design and the development lifecycle. Authors in [12] propose a specialization of SysML to support the development of automation software (*e.g.*, PLC software) for manufacturing systems. Then, they suggest a methodology to automatically generate control software from SysML models of the production plant. [34] proposes to use an AML-based model to create "Plug-and-Produce" facility components, which integrate information about their production capabilities. In this regard, the production resources composing the manufacturing system are modeled in terms of their ability to produce a particular product or to implement a required production "Skill". As such, AML includes the necessary constructs to model such a system viewpoint. The work proposed in [35] presents an extension of UML to support the modeling of mechatronic components and IoT production environments. Therefore, manufacturing system components are adjoined with IoT interfaces at a modeling stage, facilitating their latter integration.

Representing and formalizing the knowledge of entities and services involved in manufacturing is also an open challenge. In this regard, the most widely used tool is ontology, which is defined as a formal and explicit specification of a shared conceptualization [36]. Semantic models and ontologies have been used to specify concepts, resources and entities involved in manufacturing, especially production information systems (*e.g.*, MES) [37]. The models serve as logical definitions that can be exploited to infer concept relations between entities, such as between enterprise, automation and logistics domains [38]. For those reasons, different standards and languages have been proposed to carry out information modeling through ontologies. The most representative ones are the Web Ontology Language (OWL) and the Semantic Web Rule Language (SWRL). Those languages have been used to define semantic annotations of service-oriented architecture, created to control manufacturing systems based on the type of service it is required to provide [39] [40]. The integration of ontologies and, thus, semantic reasoning within SysML has been proposed in the past [41]. Nonetheless, an ontology can be considered as a meta-metamodel, which is often specified by creating custom UML profiles. Consequently, the knowledge defined in the ontology is transferred to the model, and their interoperation is guaranteed.

A systematic review [42] of modeling languages for manufacturing highlighted a gap between the involved research communities. In particular, system engineering and knowledge-representation languages are widely used in this field, but rarely combined. The survey also found out that AML and UML (which is the root of SysML) are quite popular in the field. However, they are rarely used together and the reuse of existing models is hardly tackled.

To the best of our knowledge, the only approach combining AML models and SysML has been proposed in [43]. The authors investigate the commonalities and differences between the structural modeling provided by the two languages. They also propose a SysML profile and an AML metamodel, to boost languages' interoperability. While their objective of reusing models is common to this work, our approach exploits AML models reuse in a PBD framework. As such, it considers also other modeling and system aspects, such as behavioral and parametric facets, implemented in corresponding SysML diagrams. Furthermore, the reuse of AML models is implemented using a direct one-to-one mapping between AML and SysML standard elements. As such, it avoids using a SysML custom profile and additional stereotypes.

Regarding production processes modeling, two popular mathematical optimization models are RTNs [44] and State Task Networks (STNs) [45]. Both focus on formalizing the production recipes as a directed graph. The main difference between STN and RTN is that the former concentrates on expressing the sequence of material states associated with tasks, while the latter also ex-

plicitly includes the allocation of tasks and resources to physical machines. Therefore, the information on such models is limited to the recipe viewpoint, without any knowledge of how tasks are implemented by machines and, thus, without any information about control aspects. To structure recipes within manufacturing information systems (*i.e.*, MESs), different standards have been developed in the past. For example, ISA-95 and 88 provide a consistent terminology to define basic components of a production recipe, such as tasks, materials and pieces of equipment. Furthermore, multiple XML-based languages have been developed to concretely express the concepts present in the standard. As an example, Business To Manufacturing Markup Language (B2MML) [46] and AML [9] can be used to define not only the production process viewpoint, but also the topology and the architecture.

To the best of our knowledge, a model capable of representing manufacturing production in a hierarchical structure has not been proposed. In fact, the analyzed models and standards are focused on a single conceptual production level, such as the recipe level or the control automation level. Meanwhile, a unified model, able to capture at different abstraction levels, the different aspects of a production process, is still missing.

3.2 Methodology Overview

When dealing with today's production systems, the bare manufacturing plant is not the only aspect to consider. For this reason, concerning modeling, the most obvious limitation of AML is the lack of expressiveness to model advanced functionalities making plants smart. Neither the computational infrastructure nor the information flowing through the system's machinery can be represented by AML models. Indeed, PLCopen allows modeling control logic. However, its constructs are not suitable to model anything more complex than control software running on PLCs.

AML expressiveness is limited also when specifying manufacturing functionalities. While it supports structural modeling through CAEX and kinematic modeling through COLLAborative Design Activity (COLLADA) [47], it does not provide constructs useful to specify actions and primitives provided by the single machines composing the systems. The lack of primitives to express machinery's behavior in terms of actions makes it difficult to specify production processes recipes. While it is possible specifying which products, resources and processes are related, AML does not allow specifying how a process is structured to transform resources into products.

For all these reasons, design flows for CPPS require more expressive languages than AML. However, manufacturing systems engineers are already

confident with AML. Furthermore, many AML descriptions are already available for existing manufacturing systems, and the standard is popular also among researchers [42]. Thus, while searching for novel design and modeling approaches, we advocate the importance of letting system designers and engineers continue using their languages of choice. Such a feature has already been investigated when dealing with CPSs [48]. However, the reconciliation to a single language of the production viewpoint of CPPSs has never been proposed. For this reason, this work aims at enabling the integration of existing AML models within SysML-based design flows. AML descriptions of existing manufacturing systems are analyzed and used to generate SysML models. Then, generated models can be used by designers to carry on the top-down phase of designing a CPPS.

Regarding functionalities and processes, the recipe model is structured over three levels, each of them representing a different abstraction of the production:

- the *task level* consists of a task-resources graph. It allows describing the bones of the production process, which are the tasks, their dependencies and the machines on which such tasks can be allocated.
- The *service level* refines the concept of "task", describing the sequence of steps required to be carried out to complete the task. It consists of a directed graph where the edges express the execution flow and each node identify a step of the task.
- The *machine function level* describes the interactions that need to take place between the control software and the machine implementing a service as a directed graph. Therefore, it allows exploiting the basic functionalities provided by the machine to create more complex services.

The presented models integrate the necessary set of information to map the process on the platform of components (*e.g.*, hardware and software). In fact, the proposed PBD flow enables the top-down refinement of services and machine functions to bottom-up abstractions of components' architectures, each implementing specific control behaviors. Therefore, the design flow combines efficiently the production viewpoint with control aspects, by outlining the necessary set of behaviors (*e.g.*, PLCs functions) to carry out a service and, thus, a recipe.

In this way, the designers can model novel functionalities and refine them onto the model of the already existing architecture. The proposed methodology enables better modeling features for designer to tackle advanced manufacturing systems design, while preserving already existing models of systems.

3.3 Architectural Models Reuse

AML models, built upon the CAEX standard, depict the manufacturing system's structure: the components composing the production line and their relations. As such, they can be exploited for the bottom-up phase of the proposed PBD approach, to construct the structural part of SysML models. The methodology hereby described creates the production plant model alongside the computational infrastructure model, within the same SysML description. This is a fundamental feature, enabling models' reuse and boosting the design process for complex systems. To accomplish such a task, we propose a mapping between the two languages. Table 3.1 reports the mapping: the AML elements are categorized in libraries, instances, classes, objects and relations. The table reports, for each AML element, the corresponding SysML elements.

Table 3.1: Mapping of AML elements to SysML objects.

	AML Element	**SysML Element**
Libraries	SystemUnitClassLib	Block Definition Diargam
	RoleClassLib	
	InterfaceClassLib	
Instances	InstanceHierarchy	Internal Block Diagram
Classes	SystemUnitClass	Block
	RoleClass	
	InternalElement	
	InterfaceClass	InterfaceBlock
Objects	Attribute	Property
	Internal Link	Connector
	ExternalInterface	Port
Relations	BaseClass → Class	Generalization
	RoleRequirements → Class	Realization
	SupportedRoleClass → Class	Realization

The main component in an AML model is the *class*: it is typically organized in libraries, depending on the concept or component it represents. In this regard, AML defines three different libraries: the *SystemUnitClassLib*, the *RoleClassLib* and the *InterfaceClassLib*. The *SystemUnitClassLib* represents system components and their relationships. The *RoleClassLib* encapsulates semantic classes that are associated to objects, while the *InterfaceClassLib* allows defining the type of communication between elements.

Since AML specifically represents the structure of the system, it is natural to map such viewpoint to structural SysML diagrams. In particular, an

AML library is directly mapped to a SysML BDD, characterizing the hierarchy of system components (*i.e.*, classes to blocks) and their relations. AML classes frequently enclose attributes to further specify components' properties or parameters. As such, an AML attribute is mapped to a SysML *block property*, typed accordingly to the attribute type. Assigning the correct type to an attribute is not trivial: AML provides a small set of native types (*e.g.*, *integer*, *real*), but in case a physical dimension (*e.g.*, velocity or temperature), a workaround is needed to specify their units of measure.

AML implements different relationships between classes: an AML *system unit class* might be either a base or a specialized class inheriting attributes from a superclass. SysML provides a similar set of constructs between blocks composing a BDD. As an example, the inheritance relation is mapped and represented in SysML through the *Generalization* connector. In addition, at least one role class must be associated with a class or to an *InternalElement*, which is a class instance. As such, the *RoleRequirement* construct represents the requirement that a specific role must be associated with the element. Furthermore, an element associated with multiple roles includes multiple *SupportedRoleClass* AML relationships. In a more general object-oriented fashion, roles can be treated as abstract classes. Therefore, the association of a role to a class or an internal element is mapped to the *realization relation* between SysML blocks, representing abstract and concrete classes.

The AML *ExternalInterface* defines an interface to an external object or class. An interface is typed by an *InterfaceClass*, which encapsulates user-defined attributes to characterize the communication. SysML provides similar constructs within a BDD: the *Port* and the *InterfaceBlock*. In particular, the *InterfaceBlock* is used to type a *Port* object. They are both directly acquired from AML ExternalInterfaces composing the class or from the specific InternalElement.

The *InstanceHierarchy* is a central section of AML: it structurally defines class instances and object connections between ExternalInterfaces. It is natural to correlate this concept to the IBD of SysML, where instances of classes are related to each other through *Ports*. As such, AML ExternalInterfaces typed by InterfaceClasses and connected using *InternalLinks*, are mapped to SysML Ports, typed by InterfaceBlocks and connected via *Connectors*. An important limitation of AML InternalLinks is the lack of expressivity: they do not allow specifying the kind of information passing through the ports being connected. As such, they only represent the connection between elements, without providing any further information.

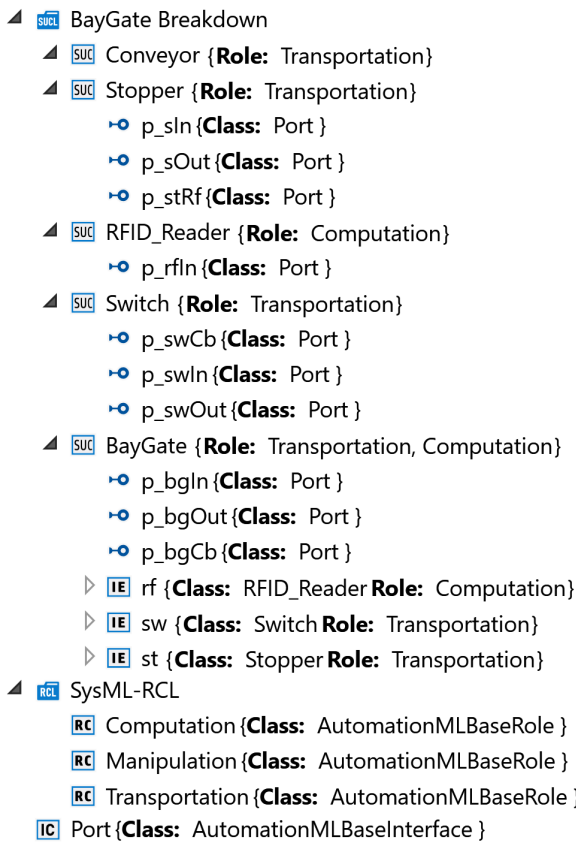

Fig. 3.2: AML libraries of SystemUnit classes, roles and interfaces of the *BayGate* system.

3.3.1 Use Case Basic Components

The set of physical components of the examined portion of our conveyor system are: a conveyor gate, which stops the minipallet, reads its id and deviates it in the conveyor bay whenever necessary. Otherwise, the pallet is unlocked and can continue its journey through the main conveyor. The gate is composed of a minipallet stopper, locking the minipallet in a specific position of the main conveyor. Then, an RFID reader reads a tag fixed on the minipallet frame and, once the supervisor produces a response, the stopper deactivates. Then, a special conveyor, moving along two axes, configures itself to let the unit in the conveyor bay or to let the minipallet flow throughout the gate. Such a system AML model is depicted in Figure 3.2: the SystemUnitClassLib *BayGate Breakdown* consists of a SystemUnit class for each component. It includes a set of base classes of simple components: the *Stopper* and the *RFID_Reader*. The li-

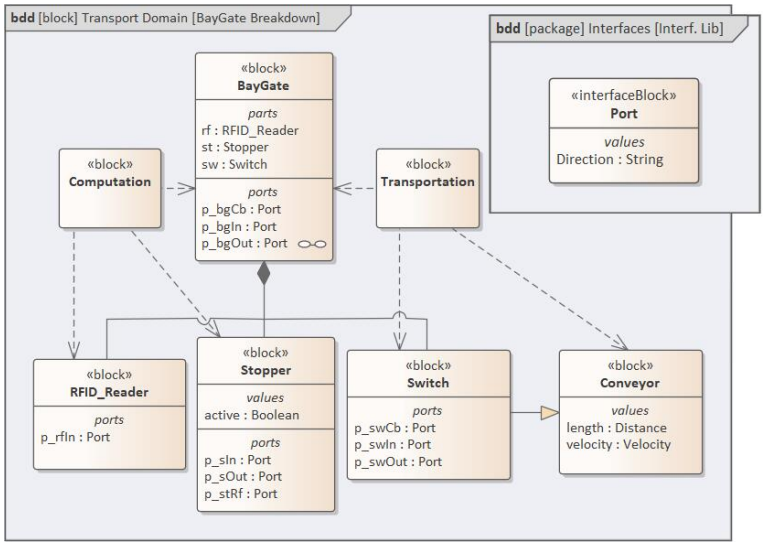

Fig. 3.3: Visual representation of the BDD derived from AML object classes.

brary also defines a subclass: the *Switch* class, which is a specialization of the *Conveyor* base class, inheriting the *length* and the *velocity* attributes. Exploiting such a relationship between classes allows the extension of the subclass with additional class members, such as external interfaces. In particular, the *Switch* class extends the *Conveyor* class by defining three ExternalInterface objects. The ExternalInterface type is defined by the *Port* InterfaceClass specified in an InterfaceClassLib. Finally, the *BayGate*, which is the primary class in the hierarchy, includes port-typed external interfaces and instances of components as class members. Each class has one or multiple associated roles, defined in a RoleClassLib. In this scenario, three role classes are sufficient to represent the semantic category of each object composing the system: *Computation*, *Manipulation* and *Transportation*.

Adopting the mapping rules reported in Table 3.1, each library defined in AML is translated to a SysML BDD. For the sake of clarity, in Figure 3.3 only the “BayGate Breakdown” and “Interface Lib.” diagrams are depicted, omitting the “SysML-RCL” BDD. The entire set of classes defined in AML is mapped to blocks related to each other according to inheritance and composition rules. The block “Conveyor” is, in fact, in a Generalization relationship with the “Switch” class, which is, in turn, associated with the “BayGate” block in a membership relationship. Therefore, the “Switch” class becomes a property “part” of the “BayGate”. Class attributes are transformed to block properties (which become “Values” properties once typed) and ExternalInterfaces become “Ports”, typed by an interfaceBlock. Role classes are related to other

classes using the “Realization” connector. As an example, the “BayGate” class is associated with both the “Computation” and the “Transportation” roles.

3.3.2 Use Case System Structure

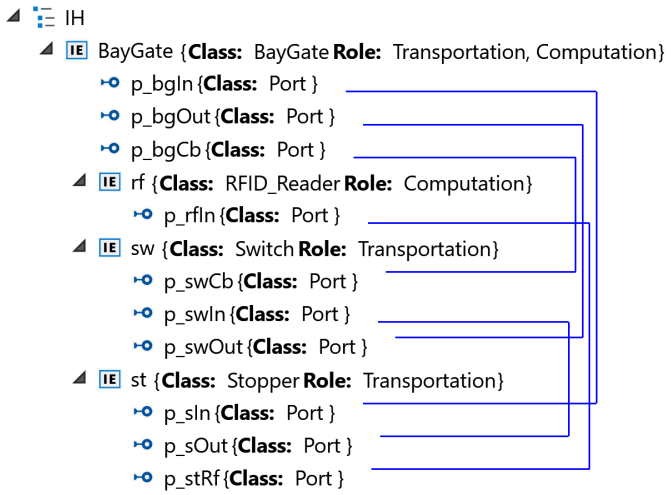

Fig. 3.4: AML InstanceHierarchy representing the *BayGate* sub-system.

The AML *InstanceHierarchy* is a hierarchical structure of objects called *internalElements*, which are instances of classes specified in the AML libraries. Each object’s semantics is defined by associated roles. Furthermore, the object’s interfaces to other objects are concretely used to represent object-to-object connections, other than simply declared. Regarding the conveyor gate example, the structure of the “BayGate”, depicted in Figure 3.4, is consistent with the class defined in the SystemUnitClassLib. However, it is also further characterized by defining the connections between interfaces of its sub-components. In this regard, *InternalLinks* are used to connect ports of the gate components set. The instantiation of a class in the hierarchy also includes assigning a value to its set of attributes.

Parts, *ports* and *values* are specialization of the type *Property* in SysML. For this matter, the purpose of the IBD is to define the internal structure of a Block by means of properties and relationships between properties. The IBD depicted in Figure 3.5 is built upon the BayGate internalElement. Since the internalElement is an instance of a class, it carries the same structure of the related block defined in the BDD. Consequently, class members such as

properties, parts and ports are instantiated and are associated with the corresponding block. In the IBD, InternalLinks between externalInterfaces become connectors between ports.

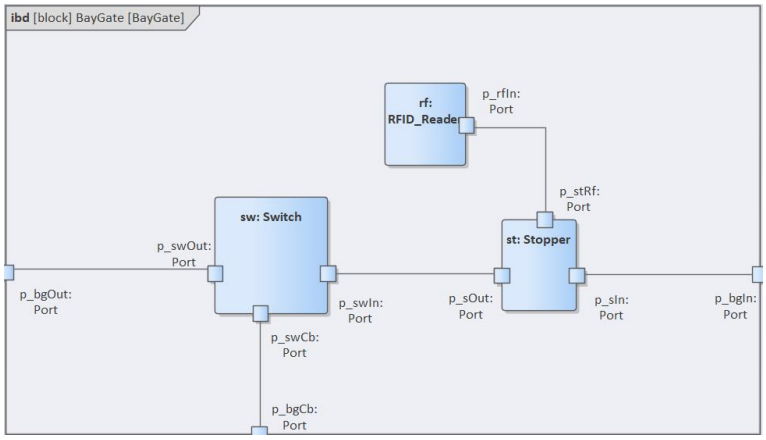

Fig. 3.5: The SysML IBD resulting from the AML InstanceHierarchy.

3.4 Architecture Refinement and CPPSs Design Flow

The SysML models generated from AML descriptions are barebones: they delineate the basic structure of objects with no functionality attached. Furthermore, the communication between objects is not exhaustively defined. As discussed, the reason for such a shortcoming is related to the lack of expressivity of the AML language. On the other hand, SysML does provide multiple constructs to enrich the diagrams obtained from the bottom-up phase. As such, the barebones models act as a *platform* to be refined by the designer, by expressing a larger set of information regarding the system to specify intended functionalities and requirements. Furthermore, SysML provides other diagrams useful to model many more details of different systems' aspects and viewpoints. For example, behavior diagrams such as the *activity diagram* focus deeply on the system functionality, specifying sequences of events. In this Section, we show the applicability of the models generated in the previous section to support a complete CPPSs design flow.

3.4.1 Components Communication Modeling

The diagrams in Figure 3.5 model exclusively the architecture of the gate component, without specifying object flows between one sub-component and

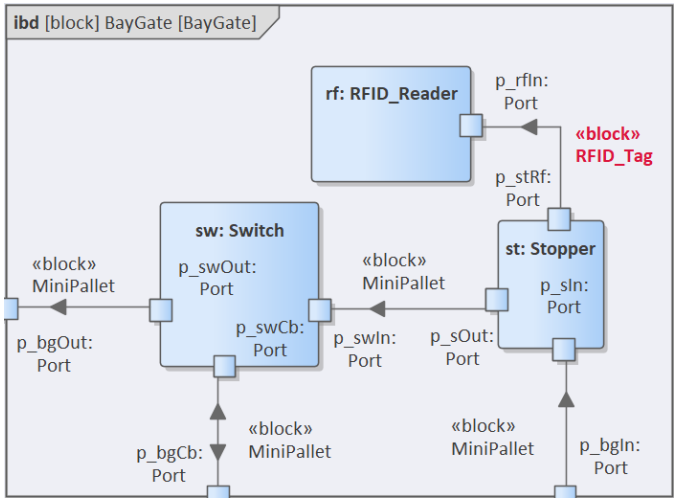

Fig. 3.6: The refined IBD with additional information about object flows and directions.

another. The Figure 3.6 depicts a refined version of the original IBD, defining the type of communication other than outlining the direction of the flow of objects. In particular, *item flows* have been added above simple connectors. This step enables specifying the type of information or object flowing between ports. Thus, it allows defining a piece of functionality performed by the system architecture. In fact, by determining the kind of objects passing through a connection, the connection itself is refined from an abstract concept to a concrete object. As an example, a communication network is very different from a physical object that connects two points in the physical space. However, AML does not have the required expressivity to detail such a set of information that is crucial in today's production systems. Applying this modeling approach, it is possible to discriminate between physical objects and information. In the example, the *MiniPallet* block represents the object moving through the physical components of the conveyor system. Meanwhile, the *RFID_Tag* is the unique identifier of a MiniPallet object. The main difference between these two objects is that they belong to different domains: the minipallet is a physical object, while the RFID tag carries a piece of information that is purely digital. As such, the channel on which these entities move is substantially different. The ability to define components has been enabled by moving the modeling of the system from AML to SysML.

The *Stopper*, modeled in Figure 3.6, acts as the contact point between the plant and the computational platform, carrying out two types of communication. The inward object flow in its input port is typed by a MiniPallet

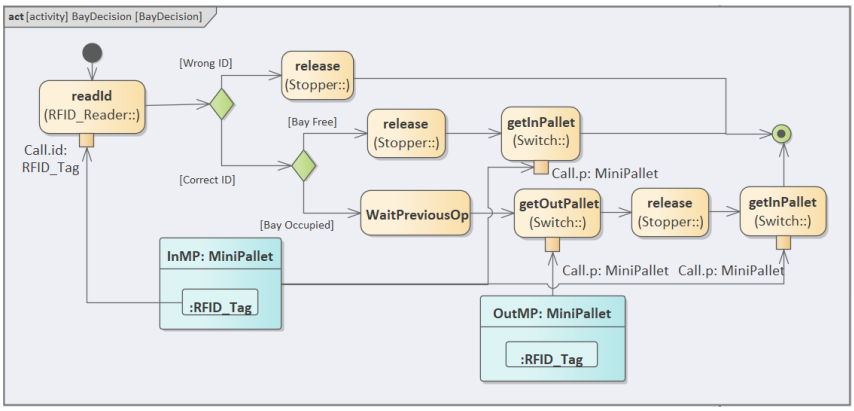

Fig. 3.7: *activity diagram* in SysML characterizing components' behaviors.

block, which represents a physical object. Then, it has two output ports: the *p_stRf* and the *p_sOut* ports. While both output ports share the same type, they accept different flows of objects: the first accepts a digital RFID_Tag object (highlighted in red in Figure), while the second accepts the flow of a physical MiniPallet object.

3.4.2 Components Behavior Modeling

Behaviors can be determined in SysML by associating an *operation* to a block of a BDD. Each operation is conceptually similar to a function and is represented as a class method. Therefore, an operation is defined by a set of input parameters and a return value and the model of its behavior. A parameter can be typed by a native type or by another block. Furthermore, an operation has to be defined in a behavioral SysML diagram, which defines its implementation in terms of *actions*. As an example, the activity diagram depicted in Figure 3.7 represents the sequence of actions following the arrival of a mini-pallet at the gate and the decision process that establishes whether it has to be pulled inside the attached bay or it has to be released. The activity diagram separates the types of flows, outlined by arrows between entities, depending on the kind of information they carry: if the flow represents the transfer of control between actions, it is called "control flow", while if the flow serves as the movement of objects, it is called "object flow". Each action in the diagram is a call to an operation defined in the corresponding block of the BDD. As an example, the *readId* action, which is the first action of the activity, takes as an input parameter an *RFID_Tag* object. As such, the flow between the *inMP* object (*i.e.*, an instance of the MiniPallet block definition) and the *readId* action is an object type.

The minipallet destination is decided on the ID of the RFID tag and is implemented through a couple of decision nodes, connected with control flows: if the ID is the wrong one, the pallet is released on the main belt, otherwise, the control passes to another decision node. This entity is in charge of determining whether the bay is free. In such a case, the stopper is released and the pallet is pulled into the bay by the *getInPallet* function of the *Switch* component. In the other scenario, the ingoing minipallet waits for the completion of the previous production operation and the clearance of the bay from the *OutMP* minipallet. Then the ingoing minipallet is pulled in.

While this diagram portrays a simple scenario for demonstration purposes, the complexity of such a kind of modeling constructs enables the specification of intricate behaviors. The expressiveness guaranteed by SysML allows to overcome the limitations of the AML structure and, thus, empowers the modeling of modern intelligent production systems.

3.5 Production Recipes Models

In the following, we present the proposed modeling strategy for production processes. The model is organized through three layers of knowledge, each depicting a particular abstraction perspective: the layers span from a high-level representation of production recipes as a set of tasks to the sequence of functions that realize the actual machines. This section will provide an in-depth presentation of the three levels, paired with their exemplification of the instance depicted in Figure 3.8. The figure reports the model of a production process modeled using the proposed three-layer representation. The example shows a production recipe composed of four tasks, which are “T1”, “T2”, “T3” and “T4”. In the uppermost layer, the production recipe is split into four tasks, which are associated with a subset of capable machines. Then, each task is further refined in the middle layer, where a task is composed of a sequence of services. The concept of service is realized through different implementations. For instance, a service may be related to the behavior of a machine or an interaction with a sensor or an information system (*i.e.*, the MES). These services are further refined in a lower layer, defining precise machine functionality and implementing the behavior represented by the service. For example, the bottom layer of Figure 3.8 depicts the specification of the *Turn* service modeled as a sequence of four machine functions. The “Turn” service is related to a robotic manipulator arm. As such, it outlines the ability of the arm to physically assemble two objects positioned within its operating space. While the Figure reports only the specification of the “Turn” operation, the same type of lower-level representation is available for all the services. The ensemble

of the three levels provides a complete description of the production process, spanning from a business-oriented viewpoint to a control-focused perspective.

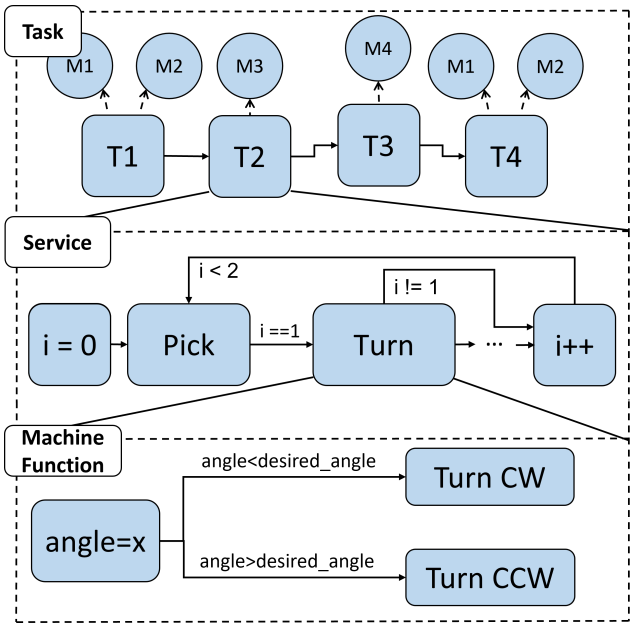

Fig. 3.8: An example of the proposed three-layer representation. The first layer defines the set of tasks and machines on which they can be allocated. The second layer depicts the concept of "service", which are specified in a control flow graph. Finally, the third layer outlines machines' functions and, therefore behaviors close to PLCs.

3.5.1 Task Level

At the highest abstraction level of the representation, a production recipe is modeled as a set of tasks. Concretely, it is expressed by a task-resource graph, which is similar to an RTN. An example of a recipe defined at such a level is depicted in Figure 3.9. Concretely, it represents a simple recipe composed of an "Assemble" task, which is set to compose three lego pieces, and a "Store" task, which transports and stores the assembled lego piece in the warehouse. Such a recipe is constructed through an activity diagram composed of custom "objects", encapsulating a set of information about each conceptual entity, and *object flow* relations. The types of entities involved are: the *task*, the *equipment* and the *material*. Each task (in gray in the Figure) identifies a different macro-step of the recipe, and is characterized by the required amount of raw

materials, and an upper bound for the task duration. The materials (in violet in the Figure) are associated with each task by defining its type, the amount available, and the current position. Furthermore, additional properties such as color and weight can be easily defined. One or more machines (in cyan in the Figure) are associated with each task. A machine is associated with a task if it provides the functionalities required to carry out such a task, and is characterized by a set of attributes: the execution time, the availability, and an upper and lower bound of its power consumption.

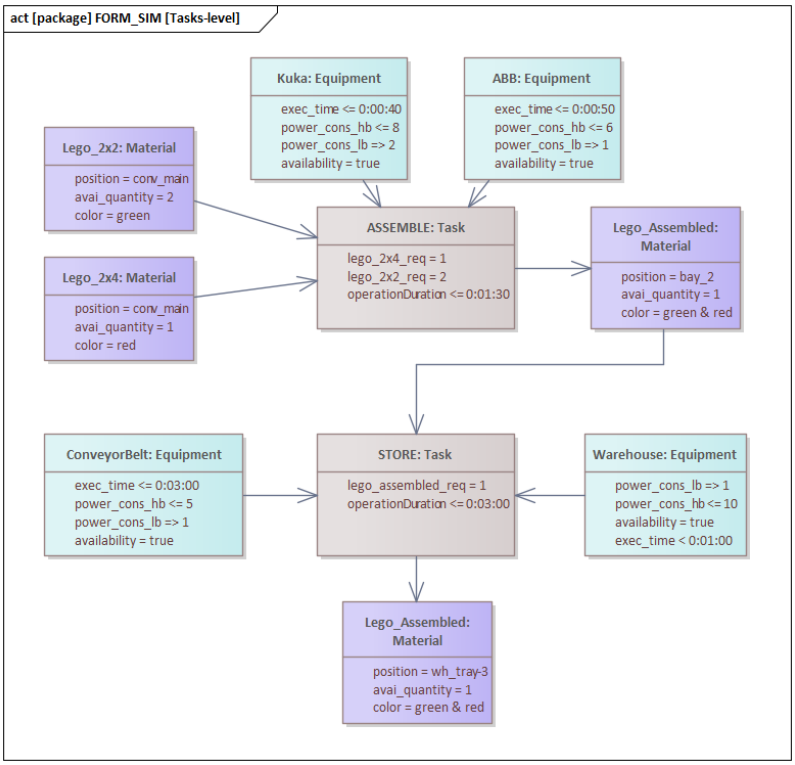

Fig. 3.9: An example of the task level of our modeling strategy. The recipe is defined by a set of tasks (in gray), connected to required pieces of equipment (in violet) and materials (in cyan).

3.5.2 Service Level

The second level refines the tasks-resources representation defined in the upper level. Current manufacturing systems are more and more based on the

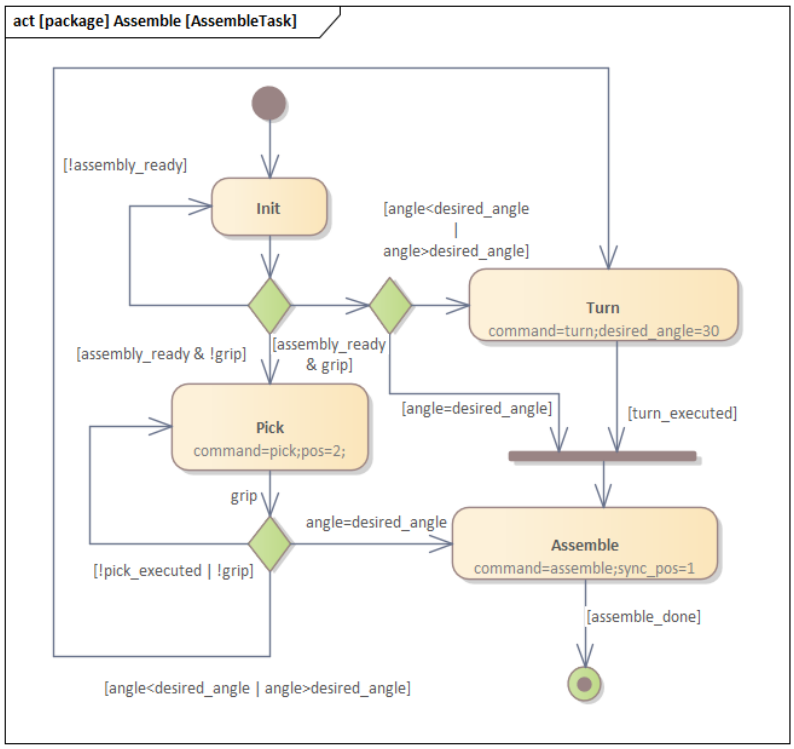

Fig. 3.10: The implementation of the "Assemble" task, made of a constrained flow of multiple services.

concept of Service-oriented Manufacturing, which models the interaction with the machines and sensors through services. Therefore, to describe all the steps in which a task is composed, it is necessary to portray the relationship between tasks and services. For this reason, the second layer describes the sequence of services called to complete the task. Its representation consists of a control flow graph. Each node in the graph represents either a service or a control flow statement. *Control flow statements* model either the definition of variables, arithmetical operations, conditional and iterative clauses, enabling the representation of complex logical flows of services. Edges in the graph specify the order in which the behaviors determined by the nodes are executed.

A service can be either a *machine service* or an *infrastructure service*:

- a *machine service* models a structured behavior of a machine as a sequence of simpler operations called machine functions.

- An *infrastructure service* models the interaction with sensors, actuators and computational resources available in the production system.

Furthermore, each service is characterized by a set of input parameters, constants and output parameters used by the control flow statements.

Figure 3.10 represents the implementation of the "Assemble" task into an activity diagram: it is constructed as a sequence of machine services. The sequence is constrained and guided through decision nodes and control guards, to represent different execution flows. Services may have input streams based on sensors data and, thus, on machines' status. Furthermore, they may produce outputs in terms of commands.

3.5.3 Machine Function Level

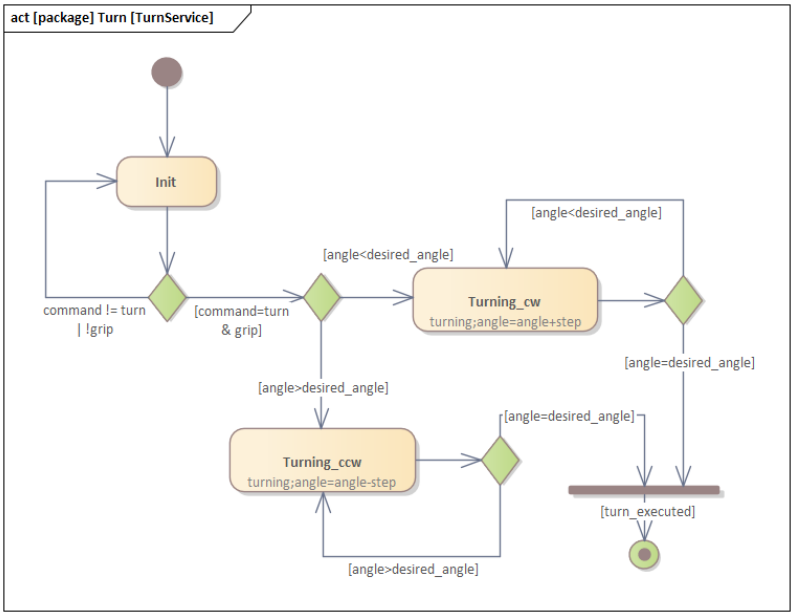

Fig. 3.11: An implementation example of the "Turn" service. It is composed by two machine functions, executed depending on the actual and target angles of the material.

A sequence of sub-services refines the services defined in the second layer. The purpose of the machine function level is to model the actual control behaviors through machine functions. In this regard, machine functions are implemented as behaviors at the PLC level. Each function is characterized by a set of input and output parameters that can be either variables or constants.

Regarding the example reported in Figure 3.11, the service "Turn" is refined in a sequence of two possible machine functions: turn the arm clockwise or counter-clockwise, depending on the actual angle of the piece and the target angle. This level allows planning the execution of the machine services, taking into consideration when a precise atomic function will take place. It also allows getting a more precise estimation of the time required to carry out a sub-task. Consequently, it enables a better forecast on when certain materials are necessary and on the used tools.

It is also necessary to assume that flattening the hierarchy is not a viable option. The model does not allow to conceptually consider a machine function or a service as a task, to include every possible information within a single level. The reason for such a limitation is that, by doing so, the data on the machine's status during the execution of a function or a service would be lost, resulting in an incomplete model.

3.6 Concluding Remarks

The presented modeling strategy proposes the use of PBD as the methodological framework, and SysML as specification language. We presented a methodology to automatically produce SysML models from AML descriptions: we presented the mapping, and we applied it to a real-world industrial transportation system to generate its SysML model from its AML description. Then, we extended the model to specify its functionalities.

To model complex production processes, this chapter presented a multi-level modeling approach to manufacturing processes. According to the service-oriented paradigm, the model organizes hierarchically tasks, services and machine functions, to carry out the production recipe. Such an approach enables precise definition of processes with a more comprehensive knowledge on how tasks are implemented on machines.

4

Compositional Design using Assume-Guarantee Contracts

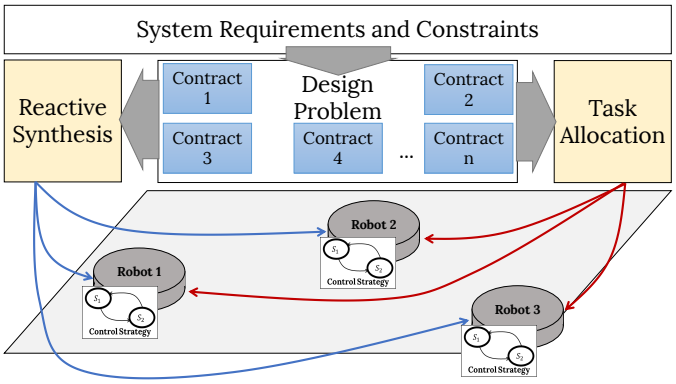

Fig. 4.1: Overview of problem tackled by the presented approach, and proposed problem decomposition.

The MOOD4I framework enables compositional reasoning about system's design. It allows decomposing the design problem and, thus, splitting responsibilities of achieving particular functionalities over the set of components composing the system. Such an approach is guaranteed by employing A/G contracts and contracts theory [22]. In fact, A/G contracts allow to decompose system design among the different components involved (*i.e.*, horizontal decomposition), and among different levels of abstraction (*i.e.*, vertical decomposition). An A/G contract formally represents a component as a pair of sets of behaviors defined over its variables: the *assumptions* and the *guarantees* [22]. A system can be represented as a composition of components (*i.e.*, *horizontal contracts*), while each component may be modeled at different levels of abstraction and according to diverse points of view (*i.e.*, *vertical*

contracts) [24]. Such an approach is particularly important in autonomous systems, and in particular in robotics systems, since they recently started to appear in many human activities: autonomous robotics is now widely used to clean domestic environments, as well as to support industrial processes within production plants. The increasing adoption of these technologies, especially in safety-critical applications, implies the rising importance of reliable, and structured flows along all the steps of the design process, from the requirement specification to the system validation.

Traditionally, the software in charge of controlling robots, as well as other families of cyber-physical systems, is designed on top of dynamical model simulators, and its validation strongly relies on extensive simulation [49]: a practice not providing the rigorousness required by safety-critical applications. Recently, formal methods have been introduced in this field [50, 27], trying to develop correct-by-construction design flows able to synthesize the control software from high-level requirements. However, dealing with such a problem holistically leads to serious complexity issues, especially when considering the constantly increasing size of the applications being designed. Problem decomposition, together with abstraction, will play a central role for future system-level design methodologies [51]. Considering the problem depicted in Figure 4.1, where a set of system requirements must be implemented by a set of collaborating robots, while respecting some constraints, abstraction and problem decomposition may be exploited in different ways. When designing the control software for the agents composing such a system one may consider going straight creating the coordination for the entire ensemble of robots. Otherwise, a designer may structure its design flow by defining the tasks composing a mission. Then, allocating the tasks to the single robots, and creating a control strategy for each of them independently. Of course, this requires to assure that the composition of the robots' behaviors, guided by their task allocations, is an *implementation* of the intended mission. Thus, the design flow needs to be supported by a formalism to describe system components and their abstractions.

We present a *compositional approach to generate the control software* for multi-robot systems. The main innovation of our methodology is the structured decomposition, supported by the A/G contract formalism, of the design process. As Figure 4.1 shows, it starts from a set of requirements and constraints characterizing a mission of the system. The requirements and the constraints are formalized as A/G contracts, to partition the design problem into multiple sub-problems. Then, we propose to use a subset of the contracts to *synthesize a control strategy* for every single robot in the system. Meanwhile, a subset of the requirements is used to perform *tasks allocation* among the different robots. The information contained in the contracts are also used

to automatically generate an executable model of the system for simulation purposes. We exploit system-level simulation to validate the results of the design flow. Once the synthesized control strategies are shown to be correct, they can be translated into the software concretely controlling the robots. In particular, this work presents an approach able to generate code for the popular ROS middleware [52]. More specifically, each agent's control software is wrapped into a ROS node. A set of well-defined messages and topics is created to handle the communication between the generated nodes. We validate the results of the presented methodology by simulating the final system using *Gazebo*, a robotic oriented 3D CAD tool and simulation environment, able to emulate the behavior of robots controlled by ROS software.

The approach is presented through a running example based on a multi-robot goods transportation system. We evaluate the effectiveness of the approach by applying it to scenarios of increasing complexity. The experimental results show the scalability that may be achieved by systematically decomposing the design problem.

The contributions of this chapter can be summarized as follows:

- Definition of a problem decomposition strategy that allows to address compositionally the design of control software for multi-robot systems. In particular, we exploit the A/G contracts to automatically partition the requirements and the specifications of the system into smaller sub-problems to be addressed individually, and obtain a set of control strategies for the components of the system. Then, we define a simulation-based method to validate the composition of the sub-problems solutions with respect to the main problem requirements.
- A technique to automatically generate software code implementing the control strategies generated, and validated, for the components of the system. In particular, the generated code is based on the popular ROS framework.

This chapter is organised as follows. Section 4.1 analyzes the related work. A case study is defined, to illustrate the application of the proposed approach. Section 4.2 provides an overview of the suggested methodology. The contract-based specification of the robotic system is detailed in Section 4.3. The decomposed system is synthesized and validated as described in Section 4.4. Section 4.5 presents the automatic generation of code for ROS. Section 4.6 shows the applicability of obtained implementations and the scalability of the entire methodology. We finally draw some conclusions in Section 4.7.

4.1 Preliminaries and Related Work

The increasing complexity of robotic systems led to the introduction of multiple robot software frameworks [53] aiming at easing the development of robotic software. Notable examples are ROS [52] and OROCOS [54]. These frameworks typically act as middleware, abstracting away the low level details about the physical system and providing high-level primitives to develop functionalities. Usually, designers develop the software on top of such frameworks. Then, they evaluate the correctness of the implemented behavior by using simulators able to reproduce the physical behavior of the system. Meanwhile, the main focus of the robotic software research community is shifting toward the integration of cognitive functionalities to make robots everyday more autonomous [55]. However, all these works rely on extensive simulation to reason about the correctness of the system being controlled by the developed software. Thus, none of them is able to provide strong guarantees about the correctness of the software being produced.

Formally specifying high-level behaviors of robotic systems to obtain a provably correct implementation is not a new task in the autonomous robotics research area. Our preliminar work presented in [56] proposes the idea of using A/G contracts, in order to reduce the complexity required to synthesize a control strategy for autonomous multi-robot systems. Even though it introduces the guiding ideas supporting this aspect of the presented work, it presents only preliminary results without providing a structured methodology. It decomposes the design problem only horizontally: it decomposes the problem over the different components only; it does not decompose over the different aspects and abstractions of the same problem (*i.e.*, vertical decomposition [57]). Furthermore, it does not target the generation of actual code able to run on real systems. On the contrary, the presented contribution exploits the ability of A/G contracts to decompose the problem also vertically. Moreover, the different parts resulting from the decomposition are used to solve different sub-problems in a structured methodology, able to generate ROS code that can be run on actual robots.

Another end-to-end methodology to generate executable code that implements high-level robot behaviors has been proposed in [58]. The methodology is applied to the DARPA Robotic Challenge (DRC), consisting in a semi-autonomous mission executed by a ground robot operating in a dangerous environment for humans. This approach assumes that each sub-component of the system is already defined in a centralized way, while concentrating on the reactive mission plan applied to a high-level view of available actions and behaviors. The approach has been exploited by the same team [59] to specify system capabilities that are mapped onto the task definition and execution.

The integration of robotic control-software obtained by formal specifications and GR(1) synthesis in ROS is addressed in [60]. The authors developed a framework to automatically synthesize a ROS node implementing GR(1) specifications. The approach relies on the reactive synthesis algorithms by *Slugs* [28]. Finucane *et al.* [61] developed a toolbox (*i.e.*, *LTLMoP*) to develop and test high-level robot controllers from problem specification in natural English. This tools collection is also able to set up a visual simulation of the robot mission execution. All these contributions rely on the automatic synthesis of control strategies from LTL specifications [62]. However, none of the previous works addressed and exploited a structured problem decomposition to reduce the complexity of the design process.

Our methodology has similar objectives to the previously described works. Furthermore, we also rely on the same theoretical pillars, such as the automatic generation of control strategies from GR(1) specifications [63]. However, the presented work is able to keep manageable larger and more detailed definitions of the system being designed thanks to the compositional reasoning enabled by A/G contracts formalism. In addition, the robotics systems we consider are composed of multiple agents interacting with each other, producing an extremely complex problem to specify and synthesize in practice. As we will discuss, traditional synthesis from temporal logic specifications is not feasible, taking into account the necessity for an optimized specification method and synthesis algorithms. In addition to the background concepts on A/G contracts, LTL and reactive synthesis presented in Chapter 2, we hereby detail the target simulation platform. Furthermore we present the robotic case study used as a guiding example.

4.1.1 The Robot Operating System (ROS)

ROS [52] is a collection of packages for developing robotic software, from low-level device control to communication protocols between robot components. Tipically, a robotic system is organized in ROS as a set of distributed processes connected to each other. As such, a ROS-based system is usually represented by a graph depicting its architecture where nodes represent the processes implementing the computation in the system. For the same reason, in the ROS terminology, the term *node* is synonym of process. Each node of the graph corresponds to a component of the system, *e.g.*, a sensor, an actuator, or an algorithm implementing a functionality. Each node is usually connected to other nodes using different styles of communication, such as synchronous Remote Procedure Call (RPC)-style services or asynchronous data streaming over *topics*. More specifically, a topic is an abstraction used to identify a specific message subject or content, while ROS messages through nodes can be

considered as particular data structures, composed by multiple fields with specific field types (*i.e.*, *int8* or *string*). A *message_generation* package is available to define custom system-level messages: by specifying a set of data fields and types, a *C++* header file is generated, which can be used by nodes implementations to pack and unpack topic's messages structures. A ROS node is generally implemented by using a well-defined *client library*. Such a library provides an application programming interface in different languages, such as *C++* or *Pyhton*. In particular, the *roscpp* library provides interfaces and standard functions to create a ROS node in *C++*, capable of interacting with topics and services. A less runtime-performant but more intuitive method of designing nodes is provided by the *rospy* client library, which is similar to *roscpp* in terms of functionalities.

The ROS ecosystem is completed by some tools for analysis and simulation. *Gazebo* [64] is a 3D CAD tool, that allows designing the mechanical parts of robots. Furthermore, it provides a robust 3D simulation environment. It includes a solid physical engine and multiple extension packages, such as *gazebo_ros_pkgs*, that allow the integration of newly defined ROS nodes into Gazebo logic and visual models. More specifically, Gazebo interacts with ROS by using a set of defined messages and services, which acts as an interface between a standalone simulation client and various ROS nodes representing a robotic system.

4.1.2 Case study: goods transportation system

For the sake of clarity, we exemplify the concepts described on a running example. The case study is inspired by autonomous multi-robot goods transportation systems. Figure 4.2 gives a schematic representation of the system being considered. A set of robots moves in a two-dimensional space, such as a building floor. The robots are required to move a set of objects to specific positions in space. Such positions are called *targets*. They are indicated in Figure 4.2 by the star icons. The order in which the targets must be reached is imposed by the system requirements, *i.e.*, the requirements define a partial order over the set of targets. Each robot may move in four directions (*i.e.*, up, down, left and right). Robots must avoid crashing into eventual obstacles and into each other. Each robot is equipped with proximity sensors, providing to the robot information about the physical objects present in the robot's immediate surrounding environment. In practice, at each instant the robot is aware of the space available in the four directions it can move. Figure 4.2 depicts the case in which two robots (*i.e.*, `R1`, and `R2`) must collaborate to complete all six tasks. The right side of Figure 4.2 reports the main variables used in

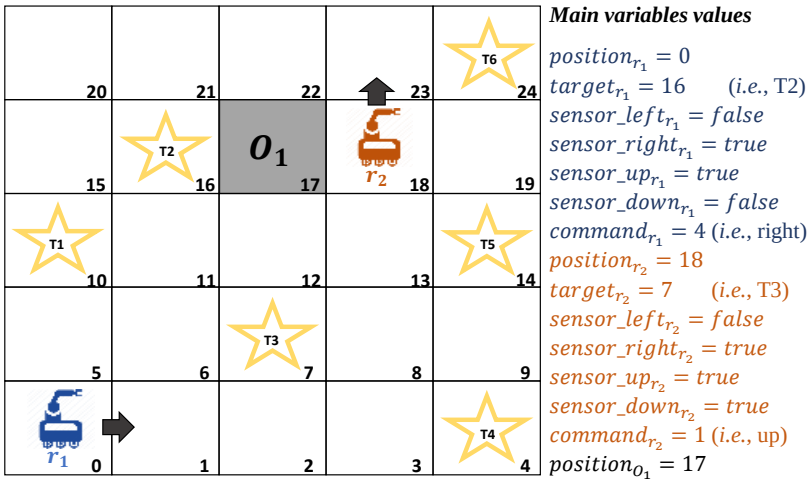

Fig. 4.2: Running example: an autonomous goods transportation system, composed of two robots, coordinating to reach six different targets.

the formalization described in Section 4.3, and their values according to the state depicted in the left side of the figure.

It is important noticing that the methodology can be extended to many classes of robotic systems. In fact, it is trivial to extend the formalization described in Section 4.3 to consider three-dimensional spaces. Such an extension requires to simply add a further variable modeling the third dimension for every variable referring to the position of a system component. The extension allows to extend the field of applicability to autonomous unmanned vehicles [65]. However, using a three-dimensional case study would not add any information about the methodology, while complicating the formulas used in the formalization step.

Furthermore, the approach can be extended to deal with robots composed of multiple independent mobile parts moving in the three-dimensional space. In such a case, the formalization would have multiple components (*i.e.*, the mobile parts of the robot) rather than multiple robots. Indeed, a set of logical assertions must be added, during the formalization step, modeling the constraints about the relative positions of the components. As long as the different mobile parts act independently to one another, the approach can be used to generate a ROS node for each mobile part to be controlled.

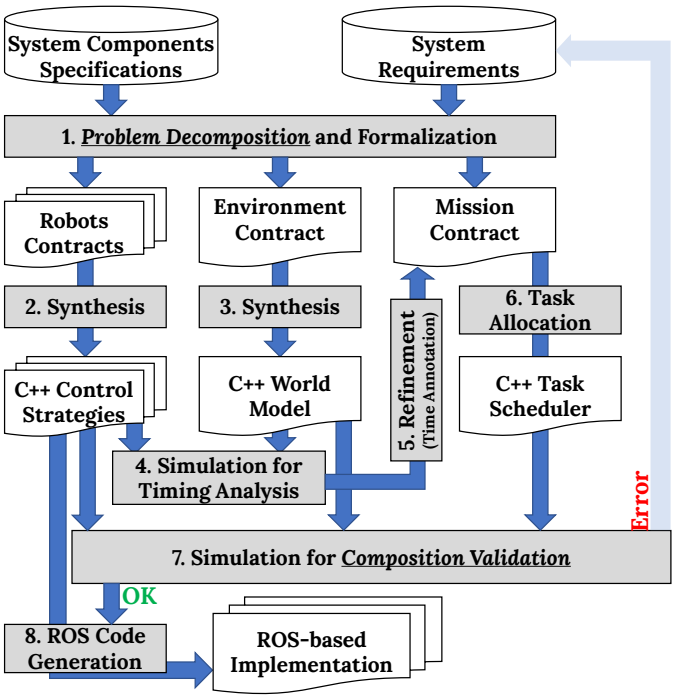

Fig. 4.3: General schema of the proposed methodology. Requirements and specifications are formalized and decomposed as a set of *A/G contracts* (Step 1). After solving the sub-problems (Steps 2 to 6), their composition is validated through simulation (Step 7). In case the validation results in a positive outcome, ROS code implementing the control strategy of each robot is automatically produced (Step 8).

4.2 Overview

Given a multi-robot system and a set of targets to be reached by the system, the final objective of our work is to automatically generate the control software for each robot composing the system. The generated software is targeted to the ROS framework, and it allows the ensemble of robots to accomplish all the required targets. Figure 4.3 summarizes all the phases composing the proposed flow, enumerated within gray boxes.

Initially, the design problem is characterized by a set of requirements and specifications. The *system components specifications* define all the resources available in the system, as well as the constraints on their use. On the other hand, the *system requirements* specify which are the objectives of the system, such as the operations that must be accomplished by the robots and the timing constraints on such operations. Requirements may differ between different systems, and they can be captured in different ways. Many different

approaches have been proposed to specify and formalize requirements and components for robotics and cyber-physical systems in general [66, 67]. Requirements management methodologies allow to automatically cast the formalization of requirements and components within a specific theory or design flow [68]. For instance, a structured management of requirements is helpful when aiming at automatically partitioning a complex design problem by using A/G contracts [69, 70]. While requirements management is not in the scope of this work, we assume that the design process starts from requirements managed by such a kind of techniques. In the case study introduced in Section 4.1.2, the requirements are the targets to be reached and the timing within these must be reached. Its components are indeed the robots, but also the two-dimensional physical space in which the robots move. As the physical space is a component of the system, then the constraints imposed by the physics are part of the system component specifications. Thus, for instance, the presence of physical obstacles, as well as the reaction time, or the features of the robots' sensors are part of the system components specifications.

The *Problem decomposition and formalization step* (*i.e.*, Step 1) formalizes the specifications and requirements composing the design problem by using A/G contracts. Alternatively, the design problem may be represented by a single contract having all the identified requirements as guarantees, and assuming all the identified constraints, as many state-of-the-art approaches do [62]. Theoretically, such a holistic contract may also be processed to perform reactive synthesis, and to generate a control software for the entire system [27]. However, the problem would be computationally intractable [69]. Conversely, the presented approach partitions the design problem into multiple sub-problems, each of them represented by a contract. As such, the design problem formalization is partitioned into the following contracts:

- multiple *Robot Contracts*, one for each robot in the system, describing each robot constraints and behavior. Each contract assumes the characteristics of the environment, while guaranteeing to react to external stimuli properly.
- one *Environment Contract* describing the constraints imposed by the system, and assuming the behavior of the robots, *i.e.*, it assumes the robots properly reacting to environmental stimuli. It must guarantee the assumptions made by the robot contracts.
- one *Mission Contract* describes the tasks the robots must accomplish. It assumes the ability of the robots of reaching each target within a given time limit. It guarantees the existence of a sequence of tasks for each robot, such that the entire ensemble of robots completes all the required tasks.

The defined contracts are used in different manners. Each robot contract is used to synthesize a *control strategy* for each robot by applying reactive synthesis (Step 2). Thus, for each robot contract it is produced a Mealy Machine implementing the contract. The generated Mealy Machine is translated into an equivalent *C++ executable model of the control strategies* for the robot. The same flow is applied to the environment contract (Step 3), thus producing a *C++ model of the world* containing the robots (*i.e.*, the environment).

The implementations obtained by synthesizing the robot and environment contracts are merged and compiled together to create a custom simulator for the system. Such a custom simulator provides higher simulation speed by integrating the simulation engine and all the models into a single executable [71], and it is used to perform multiple *simulations* necessary to evaluate the time required by each robot to reach each target (Step 4). The information retrieved by the simulation is used to *refine the mission contract*: for each robot and each task, the mission contract assumes the timing value retrieved from the simulations as the time the robot needs to reach the target.

The mission contract is given to a *task allocation* algorithm that takes care of allocating the tasks to the single robot (Step 6). The procedure generates a *C++ task scheduler* assigning to each robot the tasks to perform. After this step, we have both a strategy for each robot and a task allocation for the entire system. However, each sub-problem has been solved by using different assumptions. As such, it is necessary to verify that the composition of the solution is still a solution for the initial specification. Proving the correctness of the composition using formal methods leads to intractable complexity. For this reason, the presented approach performs *simulation for composition validation*. The C++ models of the control strategies, environment (*i.e.*, the world model), and the C++ task scheduler can be compiled together to create an executable specification of the entire system, thus composing the solutions of the sub-problems into a fast custom simulator. The composed system is simulated for all the initial conditions that are admissible according to the initial specification assumptions (Step 7). In the case none of the simulations violates the initial requirements, then the C++ control strategy of each robot is automatically translated into a ROS-based implementation (Step 8).

On the contrary, in the case the simulation for composition validation step fails, then the designer may either try to generate an alternative task allocation, or to relax the system requirements and iterate the design flow. In either cases, the trace generated by the failing simulation may be helpful to identify the conflicting requirements [72, 73]. However, a more in-depth investigation of this strategy is beyond the scope of this work.

4.3 Design problem specification

For each instance of the design problem, three contract-based specifications have been created: a contract representing the entire problem holistically, *i.e.*, the *System Contract* (C_S); a contract specifying a single robot instance, *i.e.*, the *Robot Contract* (C_R); a contract that describes the environment in which various robots move, *i.e.* the *Environment Contract* (C_E); and a contract specifying the tasks to be performed to complete the mission, *i.e.*, the *Mission Constract* (C_M). Contracts C_R, C_E and C_M are obtained by decomposing C_S. An implementation of the composition of the multiple instances of C_R (one for each robot in the system), C_E and C_M must implement the initial contract C_S.

Such a partition decomposes the problem *horizontally* (*i.e.*, among the different components of the system) by dividing the problem among the different robots composing the system. It also decomposes the problem *vertically* (*i.e.*, among different levels of abstraction), as the Mission Contract may be seen as *an abstraction of the composition* of robots and environment. In fact, while satisfying the mission contract requires the existence of a task assignment, satisfying the composition of the Robot Contracts and the Environment Contract requires to identify a control strategy for each problem that is able to implement the solution of the Mission Contract. The following of this section provides the details of these specifications.

4.3.1 System Contract (C_S)

The representation of the two-dimensional space is discretized and represented as an occupancy grid [74]. Each robot r is characterized by four boolean variables, expressing the four directional sensors (*i.e.* $sensor\_up$, $sensor\_down$, $sensor\_left$, $sensor\_right$), an integer variable for the position (*i.e.* $position = n$), and an integer variable representing the robot command (*i.e.* $command = c$). A boolean value for each cell of the occupancy grid is used to store whether a position is free or not (*i.e.* $free_p$ where p is the index of the cell). Target positions are given as input to each robot. Finally, each robot stores an integer value to represent the number of steps it performed (*i.e.*, how many times its position changed). The system specification is within the GR(1) fragment.

The *initialization assumptions* predicates about the robots' initial positions and sensor values:

- two robots cannot have the same position, for each grid cell n:

$$\neg(position_{r_i} = n \wedge position_{r_j} = n \wedge i \neq j)$$

- For every position n, a conjunction over all the robots defines the *occupation status* of the cells:

$$free_n = (\bigwedge_{r_i} \neg(position_{r_i} \neq n))$$

- Sensors values are set according to cells adjacent in the occupancy grid and the grid boundaries. For instance, the assertion:

$$position_{r_i} = 18 \rightarrow (sensor\_left_{r_i} = free_{17})$$

  specifies that the value of the left sensor of a robot in position 18, depends on the occupancy state of position 17. C_S specifies one assertion such as this for each position and each sensor.

The *initialization guarantees* set the initial command of every robot to `stay`, and its steps counter to zero. Initialization assumptions have to hold also at each time step, so they are also duplicated as safety guarantees.

The *safety assumption* properties define the robots motion:

$$\square((position_r = n \wedge command_r = c) \rightarrow \bigcirc(position_r = m))$$

where m is the position adjacent to the position n, in the direction specified by the command c. Then, safety assumptions model the global time of the system advancing at every evaluation step:

$$\square(timer = t \rightarrow \bigcirc timer = t + 1)$$

The *safety guarantees* specify that the robot decide the next command based on the sensor values. Thus, considering the running example in Section 4.1.2, and the command *up* the system guarantees are as follows:

$$\square(\bigcirc(command_{r_i} = up) \rightarrow sensor\_up_{r_1})$$

The same guarantee must be inserted for each robot, and each command modifying the status of the system, in the case study: *up*, *down*, *left* and *right*. C_S guarantees also the correct evolution of the step counters:

$$\square(((position_{r_i} \neq \bigcirc position_{r_i}) \wedge steps_{r_i} = n) \rightarrow (\bigcirc steps_{r_i} = n + 1))$$

The only *goal* of the system is that each robot r_i target should be reached within a certain amount of time T, *i.e.*,

$$\square\Diamond(position_{r_i} = target_{r_i} \wedge timer \leq T)$$

4.3.2 Robot Contract (C_R)

The *A/G* contract of every robot instance is an abstraction of C_S (*i.e.*, $C_S \preceq C_R$) obtained by increasing the set of assumptions by assuming an environment that allows the robot to move freely (*i.e.* $A_S \subset A_R$). The resulting A/G contract responds with the appropriate command value to input sensors values provided by the environment. For every position n, assumptions are as follows:

$$\square(position_{r_i} = n \rightarrow$$
$$(sensor\_up_{r_i} \wedge sensor\_down_{r_i} \wedge sensor\_left_{r_i} \wedge sensor\_right_{r_i}))$$

However, it is possible to define different configurations inserting assumptions about obstacles. Considering the example in Figure 4.2, it is possible to assume the existence of the obstacle O_1 by inserting assumptions that forces different values for the sensors, such as:

$$\square(position_{r_i} = 12 \rightarrow$$
$$(\neg sensor\_up_{r_i} \wedge sensor\_down_{r_i} \wedge sensor\_left_{r_i} \wedge sensor\_right_{r_i}))$$

However, C_R does not consider the other robots in the system, thus assuming that other robots do not act as obstacles.

The guarantees expressed in the C_R are the subset of guarantees of C_S that generates the value for the next command to send according to the sensors values.

In particular, if the target position is reached, the only possible command is `stay`:

$$\square((position_r = target_r) \rightarrow \bigcirc(possible\_command = stay))$$

Finally, if the output command is different than `stay`, the robot has moved. Thus, the time required by the agent to reach the desired target is incremented by 1.

4.3.3 Environment Contract (C_E)

The environment contract is an abstraction of the general system contract modeling the physical environment in which the robots act. The occupancy grid is represented by a set of boolean variables, one for each cell. Each variable values true if and only if the corresponding block is free. The environment contract assumes the behavior of the robots. Thus, its behavior is based on the command values assumed the robots would generate. Its output is the

set of positions for the robots, as well as the sensor values for each robot. More specifically, for each robot r, for each position n and each command c, an assertion is in charge of computing the next position of the robot. Assertions describing robot movements are:

$$\square(position_r = n \wedge command_r = c \rightarrow \bigcirc (position_r = m))$$

The environment contract also guarantees some trivial constraints of the physical system, *e.g.*, two robots can never share the same position. Thus, for each pair of robots r_i and r_j:

$$\neg(position_{r_i} = n \wedge position_{r_j} = n \wedge i \neq j)$$

4.3.4 Mission Contract (C_M)

The mission contract formalizes system requirements specifying the timing constraints of the target to be reached by the robots, as well as their ordering. It assumes the main features of the system, *i.e.* the number of cooperating robots, as well as the time needed by each robot to complete each target. Furthermore, it assumes that no obstacle can create a situation such that a robot gets stuck in some positions in the two-dimensional space. Meanwhile, the contract guarantees a certain time-bound within every task of the system will be completed. That is, the mission will be completed within N steps.

The maximum time required by each robot to move to every target is hard to be computed. Retrieving such time requires computing, for each robot, all the possible paths between every pair of targets in the system. For this reason, system simulation is used to enrich the contract with the travel time information for each robot and each target. The execution of synthesized robot controllers into the generated environment model ensures that a robot r_n can reach a target t_m in k steps. The contract is defined as follows:

- *Assumptions*: each robot possible path to each available target is structured as a triple (I, D, T) where I is the initial robot position, D is the destination cell and T is the time required to reach that cell.
- *Guarantees*: mission requirements can be fulfilled since a set of targets can be reached in a predictable time limit.

The consistency of the mission contract provides a possible allocation of tasks to the robots that may potentially realize initial requirements and constraints. This contract is an abstraction of C_S, as it defines a larger set of assumptions: it assumes for each robot, and each pair of targets, the maximum travel time required by the robot to travel between the two targets. As such, while the solution obtained satisfy C_M, it may not satisfy all of its refinements such as

C_S. Thus, the solution must be validated. The following section details how the allocation is generated and validated.

4.4 Synthesis and validation

After decomposing the design problem, the sub-problems represented by the contracts defined above must be solved. Then, their solutions must be combined to obtain a solution to the design problem. Thus, each contract must undergo different steps, as introduced above in Section 4.2. This section details the different steps.

4.4.1 Code generation and simulation

Contracts describing the robots and the environment of the system undergo reactive synthesis. They are formalized within the GR(1) fragment of the LTL. Thus, each of them can be used to synthesize a control strategy by using a reactive synthesis tool. In our case, we rely on GR1C [27]. After being synthesized, each strategy is expressed as a Mealy Machine in JSON format.

An automatic tool has been developed to generate executable C++ code starting from each of the Mealy Machine descriptions. It includes a JSON parser creating an intermediate representation of a Mealy Machine object. Then, the tool generates a C++ class implementing the Mealy Machine by exploiting automatic homogeneous code generation [75]. This allows creating an executable specification for each A/G contract emulating the behavior of the component specified by the contract.

Composing the executable models of robots and environment enables early simulation for the system. Each C++ class representing a robot is instanced by a top-level component, together with the class generated by the environment contract. The inputs of the robots are the outputs of the environment model and vice versa. A scheduling procedure [75] is in charge of reproducing the concurrency, and managing the communication and synchronization among the different components of the system.

4.4.2 Task allocation

The simulation environment defined above can be used to retrieve the set of steps required by robots to reach all the available targets. This is done by simulating the system multiple times. At each simulation, one robot moves between a pair of targets. The process is repeated for each robot, and each pair of targets to be reached consecutively. The information gathered by such

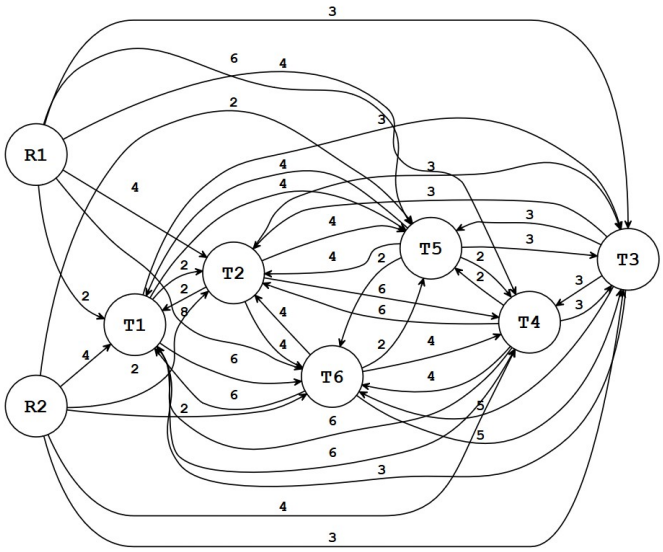

Fig. 4.4: Weighted graph extracted from the case study. It represents the costs of all the paths from robots to targets.

simulation phase is then used to generate a tasks allocation over the available robots.

The task allocation problem can be resolved in multiple ways, and much research has been already carried on in this field [76]. Furthermore, the solution generated by performing reactive synthesis from the system contract C_S, without decomposing the design problem, already embeds in itself the solution of the task allocation sub-problem. The framework we implemented to evaluate the positive impact of problem decomposition relies on a procedure made in-house. However, it is important noticing that any other technique for task allocation can be used to address this step. For the sake of completeness, we report the details about our solution to better clarify how to connect other solutions to our methodology.

The procedure encodes the problem as a *directed weighted graph* $G = (V, E)$. Vertices V represent either robots initial positions and target positions. Edges E and their associated weight represent *cost of paths* connecting nodes. A partial or total ordering relation may be defined over the targets by the mission specification, *i.e.*, a set of constraints about the order in which the tasks must be performed. Furthermore, the mission may impose that two consecutive tasks are carried on by the same robot. This may be useful to represent a robot moving an object from one target to another. Figure 4.4 shows the graph extracted by the case study described in Section 4.1.2.

Algorithm 1 Task allocation

Require: graph of costs, task_list
Ensure: minimize total mission cost and robot cost differences
1: $robots \leftarrow [r_1, r_2, ..., r_n]$
2: $costs \leftarrow [0, 0, ..., 0]$
3: $tmp\_costs \leftarrow [0, 0, ..., 0]$
4: $allocation \leftarrow \emptyset$
5: **procedure** ASSIGN_TASKS
6: **for** task in task_list **do**
7: **for** r in robots **do**
8: $tmp\_costs[r] \leftarrow min\_path(graph, r, task) + costs[r]$
9: $best\_robot \leftarrow min\_cost\_move(tmp\_costs)$
10: $costs[best\_robot] \mathrel{+}= tmp\_costs[best\_robot]$
11: $allocation.insert(best\_robot, task)$
12: $remove\_edges(graph, task)$
13: $tmp\_costs \leftarrow [0, 0, ..., 0]$
14: **return** $allocation$

Algorithm 1 takes the graph representation as input, to produce a task allocation while minimizing the overall mission cost, *i.e.*, the sum of all the robots actions. It also aims at keeping balanced the number of actions performed by each robot. Thus, trying to increase concurrency, and decreasing the overall mission duration. Line 1 to 3 initialize some lists: the list of n `robots`, the `costs` of each robot, and a list that will be used to store temporary values, one for each robot. Then, the list of the `allocations` is initialized empty: it will contain a list of robot/task pairs, each indicating that a task must be performed by the assigned robot. The procedure iterates over the targets in the `task_list` (Line 6). Then for each robot, the `min_path` function, based on the Dijkstra shortest path algorithm, returns the cost for the robot of the shortest path to the given target. Then, it allocates the target to the robot having the minimum value in `costs`, after summing the cost to reach the considered target (Lines 10-11). A pair is added to the allocation list, to indicate which robot will perform the task (Line 12). Then, the `remove_edges` sub-routine updates the graph by removing all the edges entering the node representing the allocated task. Then, the algorithm iterates to the following target in `task_list`. Finally, the procedure will return the list of allocated tasks (Line 16).

4.4.3 Simulation for system validation

Generating a control strategy for each robot of the system by synthesizing the complete system contract C_S provides a control strategy that already contains the solution of all the design sub-problems. Meanwhile, in our approach, every sub-problem is solved by using a set of assumptions that is larger than the set of assumptions of C_S. The task allocation is generated under the assumptions of the Mission Contract (C_M), that is an abstraction of the System Contract (C_S). Thus, C_M makes more assumptions with respect to C_S. For this reason, the solution of C_M may or may not be a solution also for C_S. Consequently, it is necessary to validate the generated task allocation. The final step of the design flow is a System-level validation through simulation. The objective is to verify that task allocation to each robot is feasible and the control software will be able to complete the given mission while respecting the constraints imposed by C_S. The simulation relies on the executable models generated by the approach in Section 4.4.1, while being driven by the generated task allocation. The allocation is encoded as a C++ static scheduler assigning the tasks to the different robots instantiated by the C++ executable model of the environment.

The different C++ models are integrated and compiled together into an executable system-level model of the system. Such a model is simulated for all the possible combination of initial conditions admissible according to the system contract. Finally, if the system-level validation returns a positive outcome, then the control strategies generated are *correct under the assumptions defined by the initial requirements*, and formalized by the system contract C_S. It is important to keep in mind that if the actual system does not respect assumptions specified in the requirements, then the generated implementation may be unable to control the system properly.

Once validated, the C++ implementations of the control strategies for the robots may be used as control software for the multi-robot system. Next Section describes how the validated C++ control strategies are used to produce ROS nodes.

4.5 ROS Code Generation

The integration into ROS of the generated implementations requires to create a ROS node for each component of the defined system (*i.e.*, a robot). Listing A.2 depicts the structure of a generic ROS node. The set control strategies generated and validated for each robot in the system is the starting point to create multiple ROS nodes, each enclosing the control strategy of one component. The structure of such control strategy is depicted in the listing A.1.

In particular, it outlines the code implementation of the Finite State Machine (FSM) that realizes the control strategy for such an agent.

Listing 4.1: Code structure used to implement and instantiate the control software into a ROS node. At each update, input message parameters are gathered and passed to the control strategy instance. This produces new output values, that are published to the destination topic.

```
1   /// @brief Code structure used to implement and instantiate the control
2   /// software into a ROS node.
3   msgs::env_output inputs;
4
5   void callback(const msgs::env_outputConstPtr &msg) {
6       inputs = *msg;
7   }
8
9   int main(int argc, char** argv) {
10      ros::init(argc, argv, "robot_controller");
11      ros::NodeHandle nh("~");
12      ros::Rate r(1);
13
14      Robot robot_ctrl;
15
16      ros::Publisher p = nh.advertise<msgs::robot_output>
17      ("/r01/robot/output", 1);
18      ros::Subscriber s = nh.subscribe<msgs::env_output>
19      ("/r01/environment/output", 1, &callback);
20
21      msgs::robot_output output;
22
23      while (ros::ok()) {
24          robot_ctrl.executeMachine(inputs.up0, inputs.down0,
25          inputs.left0,inputs.right0, inputs.target0,
26          inputs.position0);
27
28          output.steps = robot_ctrl.steps_out;
29          output.command = robot_ctrl.command_out;
30          output.target = inputs.target0;
31
32          p.publish(output);
33
34          ros::spinOnce();
35          r.sleep();
36      }
37
38      ros::shutdown();
39
40      return 0;
41  }
```

The listing 4.1 describes the standard structure of a robot node implementation in *C++*, using the *roscpp* library. Input and output messages have a precise structure of type `msgs::robot_output` and `msgs:env_output`

(Lines 3, 21). The first is composed of a set of boolean values, one for each directional sensor, and two integer variables, one representing the current position and one for the assigned target. The output message is characterized by two integer values, one for the produced command and one for the executed movements count.

Initially the ROS node is initialized (Line 10) and its handle is defined (Line 11), with its update rate (Line 12). At Line 14, the Robot controller is declared and instantiated. Then, a subscriber and a publisher are created to manage the message passing protocol: the publisher (Line 16) sends messages of type `msgs::robot_output` to the */r01/robot/output* topic, while the subscriber component listens to */r01/environment/output* for `msgs:env_output` messages, calling a callback function (Line 5) to handle the received message and to fill the internal `inputs` structure. The *while* loop at Line 23 simulates the robot controller with input parameters gathered from the received message (Lines 24-26) and publishes an output message representing the computed values from the controller (Lines 28-32).

A node subscription, as well as publication, is not constrained to a single topic: the environment node, in our case, is composed of N subscribers and publishers, attached to N topics, where N is the number of robots acting in the system. The mission execution is handled by another similarly defined node: it subscribes to each robot position, checking whether the current target is reached. In that case, it publishes to the environment node the next target.

The execution of each node is started by publishing an initial message to each robot, containing the initial conditions of the entire robotic system. It is terminated when each robot reaches all the targets the mission nodes has assigned to it.

4.6 Experimental results

Every phase of the proposed design flow has been automated. We set up a proof-of-concept tool-chain created specifically to bind A/G contract reasoning and techniques based on simulation. Each sub-component or aspect of the problem is specified through an A/G contract in the SPIN syntax of LTL [77]. Consistency checking of each contract is performed by using *GR1C* [27]. The tool is also able of performing reactive synthesis to produce a Mealy Machine implementing the given specification. GR1C produces a JSON description of the state machine whenever the specification is realizable. The JSON specification is parsed by a tool we built on top of the *HIFSuite* APIs. Relying on the HIFSuite we can exploit its automatic C++ generation tool [78] to generate the executable model of the original specification. The task planning

Table 4.1: Comparison between the time needed to obtain the final control strategy using the holistic system contracts, and decomposing the design problem. The experiments have been carried on by varying the three main dimensions of the problem.

Problem Dimension			Non-decomposed system formalization			Decomposed system formalization			
# Blocks	# Robots	# Targets	Synthesis time (s)	Code Generation (s)	Total Time (s)	Synthesis time (s)	Code Generation (s)	Simulation for Validation (s)	Total Time (s)
9	2	2	20.36	20.47	**40.83**	6.85	37.31	0.01	**44.17**
16	1	2	31.48	25.83	**57.31**	20.54	40.46	0.03	**61.21**
16	2	2	3924.18	1906.16	**5831.37**	33.13	244.48	0.02	**277.63**
16	2	4	3924.22	1910.79	**5835.01**	33.17	247.97	0.04	**281.18**
16	2	6	3924.60	1913.95	**5838.55**	33.17	247.96	0.07	**281.20**
16	3	6	Time Out (6 hours)	Time Out (6 hours)	**Time Out (6 hours)**	71.59	380.86	0.04	**452.49**
16	4	6	Time Out	Time Out	**Time Out**	184.35	810.34	0.05	**994.74**
16	4	8	Time Out	Time Out	**Time Out**	184.35	814.56	0.06	**998.97**
25	2	4	Time Out	Time Out	**Time Out**	55.54	292.94	0.08	**348.56**
25	2	6	Time Out	Time Out	Time Out	55.54	295.08	0.09	**350.71**
25	3	9	Time Out	Time Out	Time Out	142.79	504.25	0.12	**647.16**
25	4	12	Time Out	Time Out	Time Out	238.83	943.75	0.17	**1182.75**
25	5	15	Time Out	Time Out	Time Out	312.64	1351.49	0.21	**1664.34**

algorithm has been implemented in Python. Finally, we developed a backend producing the final ROS implementation of the control software as described in Section 4.5.

A series of Python scripts has been produced to generate various scenarios of the problem. The scenarios we vary the number of deployed robots and the size of the environment. For all the scenarios, we produced the holistic (*i.e.*, non-decomposed) specification of the system to be solved by state-of-the-art techniques based on reactive synthesis [62]. Then, we generated the decomposed specification as discussed in Section 4.3, and we apply the proposed approach.

In Section 4.6.1, we compare the time necessary to obtain a valid control strategy using our approach, against the state-of-the-art reactive synthesis of the non-decomposed system specification. For each specification, we set a timeout of six hours. Then, we compare qualitatively the code generated by the two approaches.

We also evaluate the applicability of the proposed methodology by building virtual models of the scenarios we considered using the Gazebo simulator [79]. Then, we deployed the ROS code generated by our methodology to control the dynamical models of the robots instantiated in Gazebo. Then, we monitor the behavior of the system to evaluate if it meets the initial requirements, and it is thus properly controlled by the generated code. Contrary to the system-level simulation relying on the code synthesized by the contracts (*i.e.*, steps 4 and 7 of Figure 4.3), the simulation performed using Gazebo is not a step of the methodology. However, it allows evaluating the properties of

the generated code once deployed in a real system. Section 4.6.2 provides the details about the Gazebo simulations.

4.6.1 Methodology evaluation

Table 4.1 reports the time required by the various design phases, *i.e.*, the realizability, synthesis, the control software generation and its execution based on the finite state machines produced by reactive synthesis tools. The different entries have been obtained by varying the three main design problem dimensions, *i.e.*, the number of blocks in the two-dimensional space representation, the number of robots, and the number of targets. The *Non-decomposed formalization* columns report the time required to synthesize the control strategy from the contract representing the system "holistically". The last four columns report the time required by applying the presented approach. In both cases, we reported the time necessary to perform reactive synthesis, code generation, and the total time required. In the case of the decomposed system formalization, we report also the simulation time required for validating the generated code. It is worth noticing that the code generated from the non-decomposed system formalization does not require further validation, as its synthesis relies on a state-of-the-art, and proved to be correct-by-construction [63], synthesis methodology.

The state-of-the-art approach [62] using the non-decomposed system formalization suffers the computational complexity of the reactive synthesis algorithm. Thus, the synthesis process requires more than six hours for the majority of our benchmarks. The complexity rises significantly by increasing the number of robots in the system. This can be explained by the substantial number of safety invariants needed to model the set of guarantees of the holistic representation. By increasing the granularity of the grid representation, the problem's complexity rises more gradually, since most of safety invariants regarding the environment shape and size compose the set of assumptions, that are more easily managed. On the other hand, the approach proposed in this work allows synthesizing also the instances that are otherwise intractable. In particular, it shows good scalability also when increasing the number of robots in the system, as highlighted by the experiments using the 16 and 25 blocks occupancy grids.

Notice that whenever the set of targets of the mission is changed, then it will be possible to perform only the task allocation while maintaining the previously synthesized strategies for the single robots. In this case, the time required by the task allocation procedure is negligible in comparison to the entire design flow, as the algorithm required at most 1.3 seconds in our experiments.

Table 4.2: Qualitative comparison of the generated code.

Blocks	Robots	Target	Synthesized States		Number of ROS Messages	
			Non-decomp.	Decomp.	Non-decomp.	Decomp.
9	2	2	10704291	2664 * 2	15	19
16	1	2	3435704	8432	35	39
16	2	2	72938223	8432 * 2	20	24
16	2	4	72938223	8432 * 2	40	48
16	2	6	72938223	8432 * 2	55	67

Table 4.2 provides a qualitative comparison between the code produced by our approach, and the code generated by using the non-decomposed system specification. We compare the number of states composing the synthesized Mealy Machines. Then, for each scenario we simulate the generated code using Gazebo, we monitor their behavior and we quantify it by counting the ROS messages used to control the system. It is important noticing that in ROS messages are the main software primitive.

Using the non-decomposed specification leads the generated Mealy Machines to grow exponentially. Meanwhile, using the presented approach, *i.e.*, synthesizing multiple Mealy Machines from the multiple contracts composing the decomposed system specification, allows generating smaller Mealy Machines. Thus, the proposed methodology provides a more compact implementation for the same given set of requirements. Meanwhile, the generated software requires slighter more messages to control the system. This is due to the fact that using our methodology, each robot needs to broadcast the information about the target aimed by the robot. Thus, for each target, two additional messages must be broadcast to all the robots in the system. Additional details about the Gazebo simulation are provided below.

Overall, this set of experiments shows that the proposed decomposition strategy allows managing systems otherwise intractable.

4.6.2 Software deployment: Gazebo simulation

The concrete applicability of the methodology has been evaluated by deploying the code being generated to control the mechanical models of real robots running in Gazebo [79]. In this work we use different instances of the open source robot Turtlebot3 [80]. Gazebo provides great adherence to systems physical reality, as well as integration with ROS. These features, make Gazebo simulation a widely used practice to evaluate the behavior of a robot system in absence of the final hardware systems.

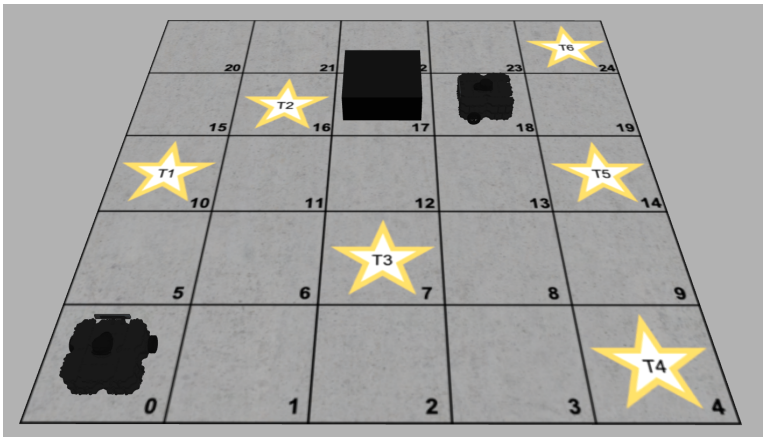

Fig. 4.5: The case study analyzed in this chapter represented in a Gazebo simulation. Through publisher/subscriber architecture, each robot subscribes for messages from the environment controller, which publishes the set of sensor values and each agent position. A robot command is then produced by the implemented controller, wrapped in another ROS node.

Additional topics must be defined to set-up a 3-dimensional simulation of robots physical behavior. The `turtlebot3_simulation` package exposes a specific node that is controllable publishing on the already defined topic `cmd_vel`. It also provides information about the robot position in the simulated Gazebo environment through its odometry functionality. The messages of the `cmd_vel` topic are of *Twist* type, that expresses both linear and angular velocity through vectors. Moreover, the odometry messages are constructed by a pose point with x, y and z components, together with a quaternion that models the actual orientation of the robot.

For each robot, we create a new node to instantiate the control strategy generated by the presented approach for the robot, and to interface the generated control strategy with the robot hardware. First, cell positions are mapped into a range of x and y coordinates. Then, since the contract-based specification does not take into account the rotation of the robot, the turtlebot3 has to be rotated to face the direction of the target cell before moving: from odometry messages the destination position and rotation can be calculated using traditional algebraic formulas. More specifically, the angle facing the target is determined in radians by calling the `atan2(x,y)` function. Moreover, the actual Euler rotation angle of the robot is collected by its odometry module, applying the `euler_from_quaternion` function to the rotation quaternion sent on the odometry topic.

Table 4.3: Results obtained by the Gazebo simulation.

Blocks	Robots	Targets	Steps	ROS Messages	Simulation Time (s)
9	2	2	3	19	20.53
16	1	2	7	39	98.21
16	2	2	4	24	34.78
16	2	4	8	48	85.67
16	2	6	11	67	110.24
16	3	6	8	52	75.27
16	4	6	7	47	60.02
16	4	8	10	66	82.86
25	2	4	12	68	104.13
25	2	6	15	87	119.44
25	3	9	17	103	133.26
25	4	12	20	124	94.86
25	5	15	19	125	70.32

Figure 4.5 shows the simulation environment emulating the case study used throughout the chapter. We monitored the Gazebo simulation of different scenarios used to evaluate the methodology. For each scenario we used one of the possible initial conditions. Table 4.3 reports for each scenario, the number of steps performed by the robots (*i.e.*, the sum of commands generated by the control strategies of all the robots), the number of ROS messages passing in the system, the time needed to simulate the scenario in Gazebo. In all the monitored cases, the system is implementing the requirements without violating any constraint. The simulation time using Gazebo is many order of magnitude higher than the system-level simulation used for validation (Table 4.1). This is due to the fact that Gazebo simulates every detail of the systems' physical behavior. Meanwhile, the system-level simulation emulates the details interesting the control strategies of the different robots, while relying on a coarse abstraction of the system's kinematics. It is also worth noticing that the time required by Gazebo to simulate does not depend on the scenario's parameters. The time is affected by the parallelism due to the presence of multiple robots, as well as the equations to be solved for emulating the dynamical models.

In conclusion, the proposed decomposition allows producing more compact ROS code, able to fulfill the initial design requirements, while slightly increasing the number of messages.

4.7 Concluding Remarks

In this chapter, we proposed an approach to exploit assume-guarantee reasoning to decompose, and make more efficient, the design of robotic control software. The approach decomposes the main design problem into multiple sub-problems, formalized as A/G contracts. Then, the composition of the sub-problems solutions is validated through simulation to verify that the solution implements the initial requirements. The experimental results show the advantage of decomposing the design problem, highlighting a substantial reduction of the required design time.

5

Virtual Prototyping using Assume-Guarantee Contracts

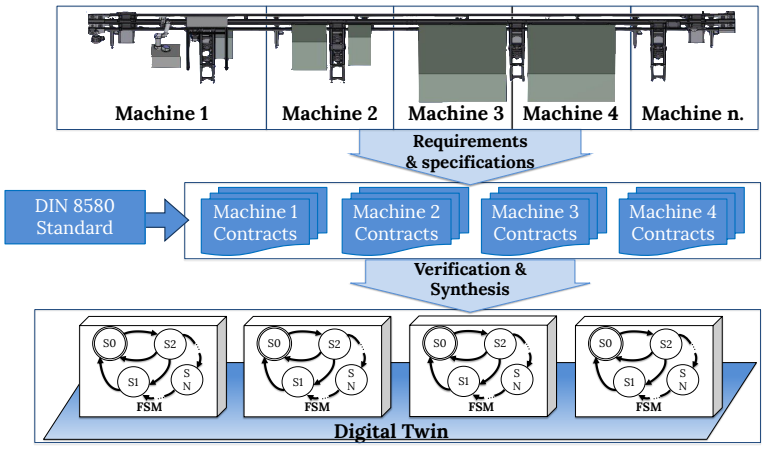

Fig. 5.1: Summary of the contribution: the production line is formalized by a set of contracts under the guidance of the DIN 8580 Standard. The contracts are used for the automatic synthesis of a virtual prototype of the production line.

To evaluate the feasibility of a production process over a newly constructed system, or to estimate its performance, MOOD4I proposes to exploit a similar approach as described in Chapter 4. In particular, this chapter presents an approach, summarized in Figure 5.1, exploiting a contract-based representation to build virtual prototypes of production lines. The behavior of each production machine available in the line is represented by a set of A/G contracts, whose assumptions and guarantees are expressed by using the LTL. The definition of machine's behaviors is guided by the *DIN* standard 8580 on industrial machinery [10], in order to conceptually identify the granularity of

the base actions considered in our approach. Therefore, such a set of actions represent the *machine functions* available for a specific machine, as described in Section 3.5. This work targets the construction of a set of implementations starting from contracts, formalized systematically exploiting the models presented in Chapter 3. In fact, each contract defined to model the system is synthesized into an FSM implementing the behavior specified by the contract. This step relies on state-of-the-art synthesis techniques [63, 81], and it produces a C++ implementation of the contract. The machine's executable model is transformed into a block to be imported into a commercial industrial plant simulator [82]. Then, the plant simulator can be used to simulate the entire system. The virtual prototypes generated by the proposed methodology may be used to perform system-level validation through simulation. The advantage of creating system simulations from the formal specification of its sub-components is the possibility to formally validate the behavior of the single components. Thus, leaving to simulation only the burden to validate the composition.

The methodology presentation is paired with its application for the formalization of a robotic arm manipulator of the ICE laboratory, and in particular to the "Turn" operation provided by the robot. Then, we apply the methodology to the entire production line comprising the robotic arm and we report the obtained results.

The contributions of this chapter can be summarized into three main pillars:

- A classification of elementary actions of machines based on industrial standards (*e.g.*, DIN 8580) that is used to guide the formalization in A/G contracts.
- The exploitation of reactive synthesis techniques to produce implementations from A/G contracts. Such a set of implementations represent the machine's constrained behaviors in terms of FSMs.
- The integration of synthesized machines' behaviors into a plant simulation software. Such a simulation is targeted toward verifying the correctness of the composition of the behaviors, to provide a correct production strategy (*i.e.*, recipe).

This Chapter is organized as follows: Section 5.1 collects the state-of-the-art related to system-level simulation techniques and simulation models construction. Section 5.2 gives and overview of the overall proposed methodology while also defining the production case-study. Section 5.3 describes the formalization of A/G contracts starting from the DIN 8580 standard. Then, Section 5.4 details the synthesis of implementations from contracts and how

to integrate the obtained code into the plant simulator. Finally, Section 5.5 gives some concluding remarks.

5.1 Related Works

Several tools have been developed in order to model and simulate industrial production systems that simplify the validation and the optimization of a manufacturing plant [83]. System-level simulation may rely on models at different abstraction levels and using diverse models of computations [84]. Such abstractions may go from the physical level, where every mechanical detail of each production machine is modeled by a set of differential equations, to the functional level, where the system is modeled as a set of machine actions and interactions between multiple machines. Indeed, the more detailed is a model, the more complex will be its simulation. As such, the choice of model's abstraction level must be based on the model's purpose. Discrete-event models [85] represent systems as the set of events occurring throughout its life. As such, they provide a high-level of abstraction while representing the essential details of the system behavior. For this reason, they are widely used to simulate manufacturing systems, and many commercial simulation tools relying on the event-based paradigm are available [86]. Different system stakeholders aim at evaluating various features of the manufacturing system using simulation. For instance, a designer may be interested in the validation of the production process, while a system engineer may aim at evaluating whether an already deployed production line could be optimized to increase the manufacturing line throughput. For this reason, each simulator can simulate different aspects of a manufacturing system.

In most cases, simulators are equipped with placeholder components depicting generic manufacturing processes. However, they usually provide extension mechanisms to precisely define the machine's behaviors by constructing and importing new custom models. Most of the available simulators provide interfaces to a well-known programming language (*e.g.*, C/C++), thus allowing the definition and the import of custom models into the simulator. Tecnomatix Plant Simulation [82] is an industrial-grade tool [87], widely popular among industrial actors being also recognized as one of the most versatile tools available for the simulation of industrial processes [88]. It provides many interfaces to external tools and languages.

In order to simulate a manufacturing plant, it is necessary to produce its model first. While this is reasonably doable when designing a line from scratch, creating models of already existing machines may be an error-prone process that may lead to inaccurate models [75]. Discrete-event models may

be also derived automatically by analyzing the system behavior [89]. However, the quality of the produced models will be constrained by the quality of the executed scenarios. As such, it may be ideal to start from formal specifications of systems and components when aiming at producing the executable models used for simulation. A formalization for manufacturing control systems has been developed for verification purposes in [90]. The authors developed a framework to automatically translate specifications to LTL formulas and using model checking techniques to verify the plant consistency. In [91], a method to formally specify industrial component behaviors has been proposed: it focuses on control logic components, the sensor/actuator behaviors and it presents how to build a formal specification to verify their correctness. However, none of these approaches relies on simulation but only on formal methods that usually lead to complexity issues. System execution is used to perform runtime verification of industrial systems [92]. However, this is usually applied to the control systems, rather than on the actions physically performed by the production line. Furthermore, it usually does not consider the state of the product and its evolution.

A combination of formal methods with simulation is described in [93]. The authors developed a formalism to specify properties over the system while being expressive enough to be translated into an FSM useful to simulate the system.

In this work, we extend the state-of-the-art by proposing a systematic (*i.e.*, guided by the DIN 8580 standard) formalization of the production machines features. Then, we present a methodology producing the digital-twin of a manufacturing system from its formalization.

5.2 Methodology Overview

The proposed methodology builds a formal representation of a production line through executable models of manufacturing machines, and it composes them into its virtual prototype. The models are meant to be simulated for evaluating the feasibility and correctness of the production line before building, deploying, and assembling the real plant. Figure 5.2 summarizes the steps of the proposed approach to validate a production line.

Initially, the components are classified according to the DIN 8580 standard, cataloguing actions and industrial processes associated with production machines. For each machine, the DIN standard identifies and details a set of actions that the machine is built to execute. Each action identified in the standard is formalized as an A/G contact. An action describes a specific behavior of the component defining a set of guarantees (*e.g.*, the ability to perform

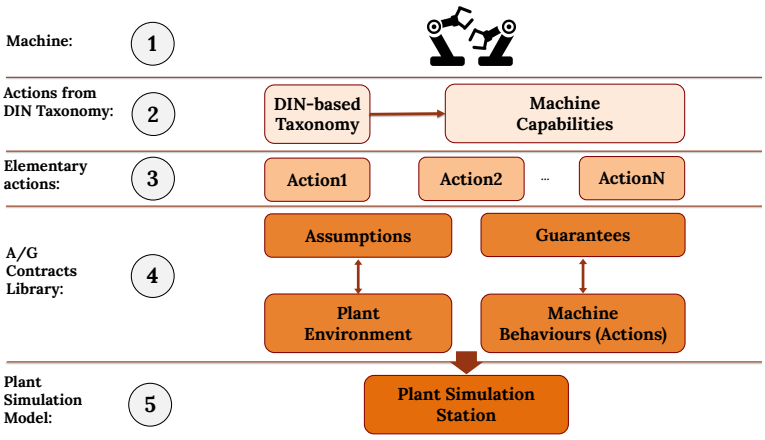

Fig. 5.2: Overview of the presented approach: starting from a taxonomy of industrial machines, elementary actions associated with this machine are defined. Then, an A/G contracts library defining each action is assembled and synthesized, generating a Plant Simulation-compatible model that can be imported and simulated.

a certain modification to the worked material shape), and assumptions that specify the conditions necessary to perform the action (*e.g.*, the presence of the worked material within the action's range of the component). Each manufacturing machine is represented by a contract that is the composition of the contracts modeling the machine's actions. This allows representing a manufacturing machine as the composition of its actions. As such, it would be possible to verify the consistency of the machine formally. However, verifying the consistency of the composition is affected by computational complexity issues [69]. For this reason, a *coordination contract* is specified to model the machine coordinating the actions identified. The coordination contract assumes the actions' behaviours, while it guarantees the possibility to perform all the actions. Such formalization of a manufacturing machine leads to a two-layer hierarchy of contracts: the first layer being composed by the actions contract, while the second layer is the coordination contract. Such hierarchical decomposition allows verifying each action separately and then to verify the coordination between actions. Thus, it allows decomposing the verification problem both horizontally (*i.e.*, over the different actions), and vertically (*i.e.*, over the two different layers of the contracts hierarchy).

All the contracts are specified using the GR(1) fragment of LTL. This allows keeping the check of consistency computationally tractable. The methodology goes on by checking the consistency of each contract. Then, for each contract, it synthesizes a control strategy, *i.e.*, an implementation of the guarantees

given the assumptions. This allows producing an FSM implementing the specification expressed by the contract.

The final step of the methodology creates the digital-twin of each machine and then the one for the entire manufacturing line to be executed within Tecnomatix Plant Simulation. It generates a Plant Simulation "*Stations*", a simulator-specific object simulating the behavior of a machine's controller. Plant Simulation adopts the concept of *Mobile Unit (MU)* to represent an abstraction of every physical object that moves throughout the production plant. Thus, each Station describes the processing of an MU, outlining the variation of the object's properties. In other words, each Station simulates the behavior of a machine manipulating the working material. The Stations are instantiated within a plant model in the simulator according to the manufacturing line's initial specifications, thus assembling the plant's complete digital-twin from the component manufacturing machines models. Finally, simulating the plant model allows validating the production line.

The following of the chapter describes each step of the methodology applied to the case study presented hereby.

5.2.1 Case study: additive manufacturing and assembly

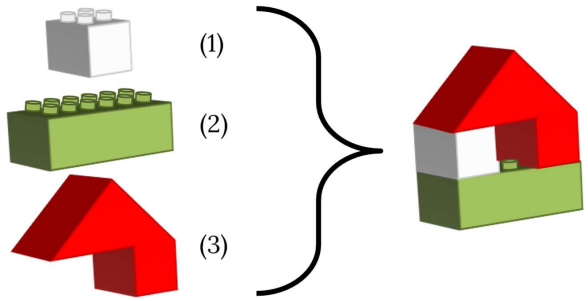

Fig. 5.3: 3D representation of the parts that compose the final product of the case study. The (1) and (2) pieces are gathered from the warehouse, while (3) has to be 3D printed.

The manufacturing system in the ICE laboratory is used to produce the object depicted in Figure 5.3. The first operation is the 3D printing of a small plastic LEGO®-like brick (*i.e.*, piece number (3) in Figure 5.3). Once printed, the piece is checked for defects by using the QC station. Meanwhile, two plastic bricks (*i.e.*, pieces (1) and (2) of Figure 5.3) are gathered from the warehouse to be assembled by the robotic assembly station. Finally, the printed

piece is assembled to the ensemble of the other two pieces. The final product is verified in the QC cell for assembly defects. Finally, the product is transported to the warehouse.

The analysis of the actions in the DIN 8580 taxonomy highlights that some of these can be implemented by the composition of more elementary actions. In the example above, the identified actions can be decomposed as follows:

- *Dismantle*: Decompose;
- *Emptying*: Pick, Turn, Place;
- *Shift one into another*: Pick, Move, Place;
- *Screw*: Turn;
- *Clamp*: Pick, Move, Turn, Place;
- *Embedding*: Pick, Move, Place;

Therefore, we identify the elementary actions of the collaborative manipulators:

- *Compose*: constitute or make up a piece;
- *Decompose*: separate into two elements the piece;
- *Pick*: collect the piece from a specific location;
- *Place*: drop the piece to a specific location;
- *Move*: move a piece;
- *Turn*: rotate the piece to a specific angle.

In the following, the description of the formalization will be paired with its exemplification of the “Turn” operation, performed by the manipulator robot.

5.3 Components formalization

In the following, we assume that the same concepts can be applied to different machines, while we will concentrate on the formalization of the functionality associated to the robotic assembly cell (*i.e.*, the collaborative manipulators). The two collaborative robots can manipulate objects and precisely assemble multiple pieces of material into a single product. The actions identified in the previous section can be further divided into two subgroups: those requiring a single manipulator arm, and those actions performing an assembly or disassembly procedure, thus requiring both manipulator arms to be active. The proposed formalization specifies a sequential behavior of the system and its components, enforcing its constrained evolution in discrete-time steps. At the chosen abstraction, action’s specification differ due to the MU characteristics and whether to model the behavior of the manipulators. A cooperative action (*e.g.*, Compose) is specified to happen at a synchronization point in space.

Table 5.1: Sketch of the coordinator contract.

	Coordinator C_{coord}
Contract	$C_{coord} = (V_{coord}, A_{coord}, G_{coord})$
Variables	$V_{coord} = \{grip, command, turn\_executed, place\_executed,$ $pick\_executed, available, angle, desired\_angle\}$
Assumptions	$A_{coord} = \{\square\lozenge(available), \square\lozenge(turn\_executed)\}$
Guarantees	$G_{coord} = \{(\square(turn\_executed \rightarrow \bigcirc(available))),$ $(\square(command = pick \wedge pick\_executed \rightarrow grip = true)),$ $(\square(command = place \wedge place\_executed \rightarrow grip = false)),$ $(\square(angle \neq desired \wedge grip = true \rightarrow command = Turn))\}$

An elementary action is the specification of a process onto a work-material (*i.e.*, a transformation defined in the DIN taxonomy). Its contract-based specification aims to model the modifications applied to the MU. The level of abstraction used to represent the work-material depends on the action. For instance, an appropriate abstraction of the MU to model a subtractive (or additive) manufacturing process should rely on a 3D grid. The grid abstracts through discretization the shape of the working-material. Meanwhile, in the contract representing the "Turn" action, the MU can be represented by its rotation only. Thus, the MU is specified as a 2D projection of its 3D shape.

For each machine, the proposed methodology creates a contract for each *elementary action* and a *coordinator* contract. Each action contract assumes the action's environment and, as such, describes the type of processing and desired shape, place, or orientation of the MU. The contract guarantees the component's behavior, by discretizing the movement of the machine and, therefore, the transformation of the MU. On the other hand, the coordinator contract assumes the whole set of elementary actions the machine is capable of. It also assumes additional properties to represent the shape and position of the product after the machine processing. The position specification is used to guarantee that the processed work-piece is placed in a specific cell of the 2D grid spatial representation, as well as its orientation and its shape (in case of a composition or separation operation).

The scenario considered in the case study outlines the rotation action of the MU performed by the manipulator arm. Two contract-based specifications are created to represent such an action: the coordinator contract and the "Turn" action contract. The manipulator coordinator contract, depicted in

Table 5.2: Sketch of the turn operation contract.

	Operation C_{turn}
Contract	$C_{turn} = (V_{turn}, A_{turn}, G_{turn})$
Variables	$V_{turn} = \{command, turn\_executed, grip, angle, desired, step, turning\}$
Assumptions	$A_{turn} = \{\Box(command = Turn \rightarrow grip = true), \Box\Diamond(command = Turn)\}$
Guarantees	$G_{turn} = \{(\Box(command = Turn \leftrightarrow \bigcirc(turning))), (\Box(turning \wedge (angle < desired) \rightarrow \bigcirc(angle = angle + step))), (\Box(angle = desired \rightarrow \bigcirc(angle = desired)))$

Table 5.1, assumes that the machine is always eventually available to perform the required action. Furthermore, a variable shared between components may be controlled by the environment in at least one of the contracts, creating a circular dependency. Thus, the environment may trivially control the variable to satisfy the assumptions even when the guarantees are falsified. This troublesome condition may be overcome by adding a liveness assumption that forcing the environment to avoid trivial assignments of variables. For example, the action completion (*e.g.*, the assumption on `turn_executed` in the coordinator) is the liveness property solving such an issue for the coordinator contract. On the other hand, the contract guarantees that whenever an action has completed its execution (in this case, the "Turn" action), the machine becomes available in the next discrete-time step. Another safety guarantee is in place to model that whenever the "Pick" action is completed, the `grip` variable becomes `true`. Contrariwise, another safety guarantee states that whenever "Place" action is completed, the `grip` variable becomes `false`. The last safety guarantee property states that the "Turn" command is produced whenever the actual angle of rotation of the MU is not equal to the desired one and, at the same time, the gripper has the material in its claws.

Table 5.2 represents the contract modeling the "Turn" action. The contract assumes that the manipulator has already completed a "Pick" action: a safety assumption requires that whenever the coordinator issues a "Turn" action, the gripper has the work material in its clamps. The action contract also assumes that the command is set to the constant "Turn" (*i.e.*, the turn action is requested by the coordinator) infinitely often: this liveness property is necessary to manage the same circular dependency issue as the coordina-

tor contract. The first safety guarantee of the action contract ensures that the `turning` variable is `true` if and only if the command received is "Turn". This variable represents the turning mode of the component and it is useful for defining the status of the system. In fact, the second guarantee assures that the MU is actually rotated of a certain angle (at discrete time-steps) while the turning status is active and while the actual angle is less than the desired angle. The last two safety guarantees model the finished action status, signaling that when the actual angle is equal to the desired one, the `turn_executed` variable become `true`.

Among the set of variables defined in each contract, a subset is related to the system while another subset is related to the environment. As such, system variables are controlled by the system and, therefore, can be viewed as the system's output or internal variables. On the other hand, environment variables act as input variables for the system. Figure 5.4 represents the interactions between the coordinator contract and the turn contract, by outlining the input and output variables between each contract. In particular, the coordinator produces the command enabling the execution of an action. In addition, it provides the `grip` variable required for the initialization of the specific turn action. The "Turn" contract provides to the coordinator the `turn_executed` variable, that signals that the execution of such action is terminated.

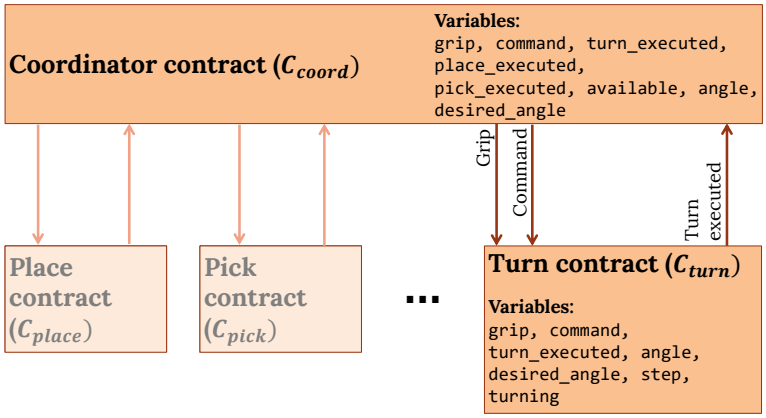

Fig. 5.4: Interactions between the coordinator contract and the turn action contract, implemented by a set of input/output variables.

For reference, other action contracts related to the manipulator arm (*e.g.*, move, pick and place) are collected in the Appendix B. Once completed the formalization of each action and machine, the contracts can be synthesized individually into its functional implementation.

5.4 Code Generation and Application

In this section, we show the application of the above formalization for the virtual prototyping of the production system. In particular, we show the steps necessary to generate the code implementing the contracts modeling the system, their interfacing with a commercial simulator of industrial production systems. Finally, we report the main results about the synthesis and code generation of the virtual prototype of the case study.

5.4.1 Executable models generation

The Mealy Machine synthesized by GR1C is represented by using a JSON format. We developed an automatic tool to generate the equivalent executable C++ code starting from each of the Mealy Machines synthesized from contracts. It translates the JSON model of each machine into an intermediate representation. Then, the tool generates a C++ class implementing the machine by exploiting automatic homogeneous code generation techniques [75]. This allows creating, for each A/G contract, an executable specification emulating the behavior of the component specified by the contract.

Listing 5.1: Sketch of the `simulate` C-Interface function that performs a property abstraction operation over MU physical properties, then it calls the simulate of the synthesized controller and finally back-propagates time, power and MU properties to the simulator.

```
1   /// @brief Code structure of the C-Interface function, that performs
2   /// a property abstraction operation over MU physical properties.
3   extern "C" __declspec(dllexport)
4   void simulate(UF_Value* ret, UF_Value* arg) {
5   UF_Value &mu=arg[0]; UF_Value &station=arg[1];
6   int operation=arg[2].value; //Pick, Place, ...
7   double angle = MU_READ_ANGLE(mu);
8   // ... other properties
9   bool end;
10  do {
11      manipulator->simulate(angle,...);
12      end = manipulator->end;
13  } while(end);
14      ST_WRITE_TIME(station, calcTime(operation));
15      ST_WRITE_POWER(station, calcPower(operation));
16      MU_WRITE_ANGLE(mu,toAngle(manipulator->angle));
17      // ... other changed properies
18  }
```

The integration relies on the interface provided by the simulator. In this work, we focus on the C-Interface provided by Tecnomatix Plant Simulation,

which allows loading custom shared libraries into the simulator, and SimTalk, the Plant Simulation internal scripting language. However, many other simulators provide conceptually similar interfaces [75].

The simulator provides a C interface that cannot call C++ methods and neither instantiate classes. Because of this limitation, `instantiate` and `deinstantiate` methods need to be implemented separately. Their purpose is to initialize and destroy the C++ class before and after the simulation execution. The `UF_Value` structure is defined in the `cwinfunc.h` header file and it is used to perform data exchange between the component implementation and the simulator environment. Each `UF_Value` contains two values: the data to exchange and its datatype. Each simulator datatype is mapped to the least bit-consuming C data-type providing enough bits: for example, `time` and `acceleration` are mapped to `double`, `string` to `char*` and so on. The C-Interface requires that each function expects an `UF_Value` to use as an argument and another one to bring its return value. To interact with the C++ class a `simulate` method is then added. It performs the following operations:

1. a *property abstraction* translates the MU physical features into a boolean representation (Listing 5.1, line 7);
2. the C++ controller is simulated with the provided inputs till the *end* output is raised indicating the operation completion (Listing 5.1, lines 10-13);
3. time and power consumption are calculated according to the simulation, and exported to the Plant Simulation *Station* object (Listing 5.1, lines 14-15);
4. the simulation results are converted into physical properties which are back-propagated to the MU (Listing 5.1, line 16).

Such a set of operations allows the product state to be kept consistent between machines, by storing it in the MU object.

Finally, the C++ strategy enriched with C-Interface headers and the previously defined functions is compiled into a shared library. The compiled file can be loaded by Plant Simulation.

5.4.2 Plant Simulation C-Interface import

Plant Simulation strongly relies on object-oriented concepts. Each simulator object is an instance of a class with its properties and methods. The import phase generates special *Station* objects that uses the previously generated shared library functions to simulate their operation.

Listing 5.2: Sketch of the *EntranceControl* of the *ManipulatorsStation* object in Plant Simulation (written using *SimTalk* language).

```
1   -- @brief Sketch of the EntranceControl in SimTalk.
2   -- load dll
3   var fd := loadLibrary(".\printer.dll")
4   if fd > 0 -- check loading phase
5   -- simulate the controller
6   callLibrary(fd, "simulate", @, ?)
7   -- unload dll
8   freeLibrary(fd)
9   else
10  -- stop the station
11  ?.Failed := trueer changed properies
```

The `ManipulatorsStation` object is created by deriving the *Station* class and editing its `EntranceControl` method. The method written using the *SimTalk* language and it is automatically called by the simulator whenever an MU enters the station. Its purpose is to call the `simulate` function of the synthesized C++ controller. The code is sketched in Listing 5.2 and it relies on the *SimTalk* methods `loadLibrary`, `callLibrary` and `freeLibrary` which are the ones developed to handle external libraries. The *loadLibrary* method (Listing 5.2, line 3) opens the provided file path and returns its file descriptor, or a number lower than zero when an error occurs. Such an eventuality is checked in Listing 5.2 at line 4 and sets the station to a `Failed` state (in this state the object cannot process any MU), leading to the end of simulation. The `callLibrary` method calls a function into the provided library file descriptor by name. It is used to execute the external code (Listing 5.2, line 6). Two special arguments are provided:

- @ represents the current MU;
- ? indicates the *ManipulatorsStation*.

The `freeLibrary` method unloads a previously loaded file (Listing 5.2, line 8). The `ManipulatorsStation` object is finally saved and made available to the simulator libraries.

5.4.3 Experimental Results

We build a tool-chain implementing the proposed methodology: the consistency checking and synthesis of the contracts is performed by using GR1C [27]. Meanwhile, we develop a tool receiving as input the JSON specifications produced by GR1C, and implementing the code generation described above. We applied the tool-chain to build a virtual prototype of the ICE laboratory case study. Then, we used the prototype to validate the system through simulation,

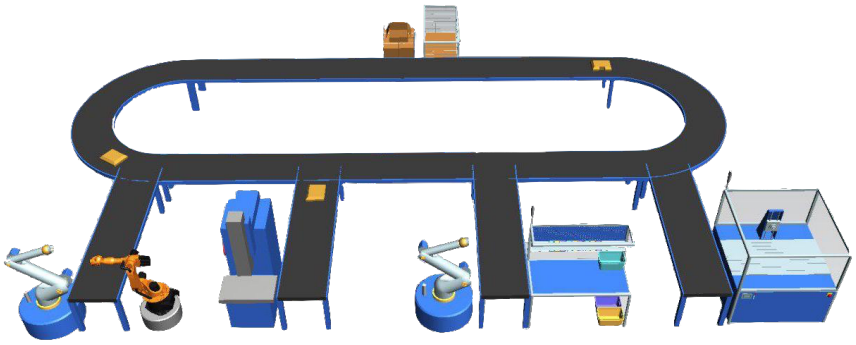

Fig. 5.5: The ICE case study being simulated into Plant Simulation.

and to evaluate the power consumption of the different machines involved in the production system.

Table 5.3: Time required to perform the consistency checking and synthesis step, and the code generation for the non-decomposed (*i.e.*, holistic) system specification, and for the different actions of the robotic assembling station. The numbered columns refer to the machine's elementary actions: (1) Compose, (2) Decompose, (3) Pick, (4) Place, (5) Move, (6) Turn. The last column reports the time required to obtain the machine coordinator.

	Holistic	**Decomposed System**						
	System	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**	**Coord.**
Consistency Checking & Synthesis (s)	Time Out	0.20	0.20	0.19	0.17	0.13	0.11	0.24
Code generation (s)	-	0.12	0.12	0.08	0.08	0.07	0.06	0.15
Total Time (s)	-	**0.33**	**0.32**	**0.27**	**0.25**	**0.20**	**0.17**	**0.39**

Table 5.3 reports the time necessary to generate the virtual prototype of the collaborative robotic assembly machine. It reports distinctly the time required to perform the synthesis from contracts, that also incorporates the time required for the consistency checking, and the time required for the code generation. The *holistic system* column refers to the machine specified by a single A/G contract: it specifies all the operations of a single machine and their coordination. Therefore, it is specified using a different modeling approach. The same reactive synthesis tools and algorithms are used for both the decomposed and the non-decomposed scenarios. The *decomposed system* columns report the synthesis and code generation time required when us-

ing the decomposed system specification, as proposed in this work. The non-decomposed (*i.e.*, holistic) system specification leads to complexity issues, as its consistency check and synthesis reaches the time-out we set to six hours: this contract is characterized by a much higher number of LTL properties and, consequently, is harder to carry out the consistency check and the synthesis. On the other hand, decomposing the system specification into multiple sub-problems allows keeping the required time extremely limited.

Table 5.4: Time required to perform the consistency checking and synthesis step for all the machines in the production line. The last column reports the total time required for the entire production line.

	3D Printer	**Conveyor Belts**	**Quality Check**	**Robotic Assembly**	**Milling Machine**	**Total**
Consistency Checking & Synthesis (s)	0.25	0.67	2.58	1.24	0.48	**5.22**
Code generation (s)	0.07	0.20	0.76	0.68	0.20	**1.91**
Total Time (s)	0.32	0.87	3.34	1.92	0.68	**6.39**

Table 5.4 reports the results for all the machines in the production line. It is reported both the time required for the synthesis from the A/G contracts and the time required for the code generation. Each column refers to a machine. For instance, Column (4) refers to the robotic assembly station. As such, its values are the sum of the values in Table 5.1. For all the machines, the processing time is minimal. The most time-consuming specification is the Quality Checking cell. This is due to the fact that the cell relies on cameras. Thus, its specification has to model multiple two-dimensional spaces to represent the signals analyzed by the cameras. The last column reports the total synthesis and code-generation time. It shows the efficiency of our methodology that allows generating virtual prototypes for production lines from their specifications.

The validation of a production line consists of verifying that the composition of each manufacturing step fulfills the production requirements, following the components specifications. The process validation is obtained by simulating the manufacturing plant agents with appropriate inputs and verifying that each component produces expected outputs. To perform such a simulation, we load into Plant Simulation the components generated by our methodology, and we build the simulation scenario, as depicted in Figure 5.5.

5.5 Conclusion

We presented a methodology to build the virtual prototype of a production line through the formalization of its specifications and automatic code generation. We have shown the effectiveness of the proposed methodology by applying it to a real production line. We were able to generate its virtual prototype, and use the generated virtual prototype to analyze the correctness of the line.

The experiments showed that the virtual prototypes for production machines are generated in a few seconds, after having formalized the production line specification as temporal contracts. This could be particularly hard for untrained designers, but it sensibly reduces its complexity starting from a library of pre-designed contracts associated to each action of a taxonomy, like the DIN 8580.

6

Application to a Service-Oriented Manufacturing Architecture

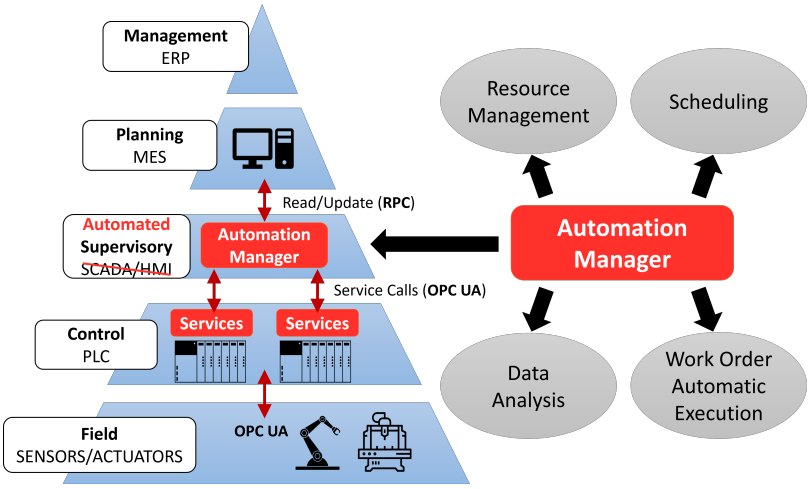

Fig. 6.1: The classic *Automation Pyramid* (on the left, colored in blue), and the proposed modification of the software architecture (colored in red). The architecture is modified by adding the *Automation Manager* presented in this work.

The reconfiguration of a production line is a multi-layered problem, bridging business aspects to automation control [94]. Traditional monolithic information systems, currently used by most companies (*e.g.*, MESs), do not provide the necessary flexibility and agility to support efficient system reconfiguration. The concept of SOA has been recently proposed in the industrial automation context [33]. SOA systems distribute the responsibility of carrying out functionalities across the different manufacturing components. As such, the granularity and flexibility of production processes based on services are

some of the fundamental pillars for carrying out an efficient reconfiguration. Nonetheless, the adoption of SOA-based technologies in a production environment are far from being an easy task, especially considering the reluctance of stakeholders to embrace new (and potentially production-breaking) technologies [95]. Especially at the business level, getting rid of a monolithic MES in favour of a distributed architecture may not be a viable option for companies.

In this context, this chapter proposes a variation to the classic automation pyramid. It modifies the traditional software architecture by automating the supervisory layer, which becomes *Automated Supervisory*. The modified layer, as depicted in Figure 6.1, encloses the *Automation Manager*, which is composed of multiple software modules carrying out a specific functionality. In particular, the Automation Manager provides:

- *Seamless integration* with existing software architecture,
- *Automated reconfiguration* of the production system,
- *Easy integration* of new technologies,
- *Autonomous execution* of production orders,
- *Resource management*
- *Advanced scheduling*,
- Support for *data analysis*.

The flexibility of such an approach is guaranteed by its compatibility with standard industrial communication protocols (*i.e.*, OPC UA [96]), to interact with services at PLC level. Furthermore, the adoption of the ISA-95 standard ensures terminology and data compatibility with most commercial MESs, thus easing the adaptation of existing manufacturing systems to implement the proposed software architecture. To concretely implement a reconfiguration strategy, this chapter also proposes an advanced scheduling algorithm. It exploits the information structured in the three different abstraction levels of the process model described in Section 3.5. The algorithm takes advantage of the restructured information to make more precise decisions on whether a process can be interrupted, interleaved and preempted, thus potentially improving the performance of the production system. Furthermore, the scheduler is reactive: it can react to unforeseen events (*e.g.*, new high-priority orders or machine failures) and reschedule the production.

To demonstrate the applicability and the efficiency of the proposed architecture, we integrated the Automation Manager into a real production line equipped with machines providing SOA features. The model-based scheduler is developed as a module of the same Automation Manager and, thus, is part of the same software architecture. We show that, despite a little overhead, the advantages in configuration simplicity and software flexibility provided by the proposed architecture are relevant. Furthermore, we evaluate the results

of the proposed model-based scheduling algorithm. We show how exploiting hierarchically structured information allows increasing the average machine utilization and throughput, while minimizing the makespan, especially for high-priority orders.

Therefore, the summary of the contributions of this Chapter are:

- A software architecture that facilitates production reconfiguration. It provides control over the production line by exploiting the concept of "services". Through the OPC UA protocol, the proposed software is capable of driving machines' execution in a more granular fashion than traditional MESs.
- A service-based scheduling algorithm that exploits the underlying control provided by the service-oriented architecture. Furthermore, it takes advantage of the knowledge integrated into the hierarchical recipe model proposed in Section 3.5, improving preemption and interleaving.

This Chapter is structured as follows: Section 6.1 reports some key background concepts on SOM, industrial communication protocols and production scheduling. Section 6.2 describes a service-based data collection architecture that is driven by the Automation Manager, which is illustrated in Section 6.3. The evaluation of the advantages and overheads of the proposed architecture is reported in Section 6.4. Then, Section 6.5 characterizes the service and model-based scheduling algorithm. It also discusses the improvements in scheduling and reconfiguration efficiency against a traditional approach. Section 6.6 concludes the Chapter.

6.1 Background

In this section, we present some background concepts to better understand the proposed architecture. In particular, we start with a description of the classical structure of the manufacturing software in subsection 6.1.1 to highlights on which level we place our solution. Then, in subsection 6.1.2 we present the communication protocol adopted within the proposed architecture. Lastly, we give a brief introduction to production scheduling in subsection 6.1.3, which is implemented in our architecture.

6.1.1 Levels of Automation in Manufacturing

The architectural model, followed by smart manufacturing and also referred to as the *Automation Pyramid* is shown on the left of Figure 6.1. It consists of five layers with different structures, requirements, and temporal constraints:

- The *Field level* consists of sensors, actuators, and applications with real-time behavior that physically act on the production floor.
- The *Control level* is where the automation starts: it consists of PLCs taking information from all the sensors to make decisions on how to control the actuators to complete a programmed task.
- The *Supervisory level* provides high-level supervision with Supervisory Control and Data Acquisition (SCADA) and Human-Machine Interaction (HMI) controlling multiple machines and collecting data from them.
- The *Planning level* contains the MES monitoring the entire manufacturing processes transforming raw materials into finished products. It also provides a complete real-time vision of the different shop floor metrics potentially useful to optimize the production processes.
- The *Management level* connects the production infrastructure with the Enterprise Resource Planning (ERP) system focused on the administration of the company resources, such as purchases, sales, and financial reports.

To simplify the integration of new technologies within manufacturing it is necessary to define a standard terminology and a unique data representation used throughout all the automation levels. ISA-95, also known as the IEC 62264 standard [7] aims at guiding the development and integration of manufacturing software within a business. It defines functionalities, responsibilities, standard terminology, and data exchange between the enterprise and process control levels. This allows simplifying the integration and development of software within the manufacturing industry.

The communication between different layers of the automation pyramid use interface exposed at each level. In the lower levels characterized by real-time constraints we found Profinet, EtherCAT, Sercos. The other level use protocol based on Ethernet, such as OPC UA.

6.1.2 Communication Protocols

Among the plethora of today's manufacturing standards, we propose an architecture that is consistent with ISA-95, since all internal data structure is modeled according to the standard's specifications. Furthermore, all communication with the machines is based on the OPC UA protocol.

Nowadays, OPC UA plays a very dominant role in industrial applications, as it became a de facto standard for Machine to Machine (M2M) communication in industrial automation. OPC UA is a platform-independent service oriented protocol developed by the OPC Foundation and standardized in IEC 62541 [96]. The communication is based on a client/server structure, where the server contains and exposes the information model, which is a graph

structure representing a wide variety of information, such as type hierarchies and inheritance. The basic components of this structure are nodes containing the data and the references that create relationships between nodes. An OPC UA client communicates with the server through a standardized set of services, such as management of the information model, read/write of nodes, and methods calls. Due to its versatility, OPC UA allows to model data transport compliant with the ISA-95 standard [97].

Distributed applications need scalable, fault-tolerant and with low latency communication channels to interact with each other. Apache Kafka is a distributed publish-subscribe messaging system developed to process real-time data with low latency. It is designed to be scalable, durable, and fault-tolerant [98]. Messages are stored as records including any kind of information, and they are stored until a specified retention period has passed. Kafka records are organized into topics, where producer applications publish new messages and consumers read the data by subscribing to a specific topic. Topics are divided into several partitions, this allows the creation of a group of consumers that increase the processing throughput by splitting the data between them.

While Kafka is more suitable for low-latency and high throughput applications, RabbitMQ performs best when request-response messages and security are the primary concerns [99]. It is a message broker and queuing server that supports different messaging protocols, among which the most important are Advanced Message Queuing Protocol (AMQP) and Streaming Text Oriented Messaging Protocol (STOMP) [100]. Producers publish messages into a queue, which are stored until the consumer retrieves them. It also supports delivery acknowledgment and the possibility to assign permission such as rights to read and write to different users.

6.1.3 Production Scheduling

Efficient production scheduling is crucial to achieve efficient manufacturing processes. An optimal schedule allows increasing the productivity, maximizing throughput, and minimizing delay and interruption of production. In manufacturing, a schedule is an optimal allocation of *recipes* in a specific timeframe. In particular, a recipe is represented by a set of jobs (sequence of operations) that can be carried out by a subset of machines with respect to different constraints. Thus, production scheduling requires extensive knowledge of the tasks offered by each machine as services. Practical scheduling problems are proved to be NP-hard [101] and, therefore, difficult to solve due to the number and variety of jobs and potentially conflicting goals. Furthermore, when manufacturing systems encounter unexpected conditions, such as machine

breakdown, rush orders, process time delay, the schedule may no longer be the optimal one or may become not feasible due to these unexpected events. In the literature, there are many proposed solutions to solve the scheduling problems, most of them are based on Artificial Intelligence techniques such as Ant Colony Optimization [102], Particle Swarm Optimization [103], Artificial Bee Colony [104], Genetic Algorithms [105]. There are also other solutions based on state-space search algorithms, Partial Order Planning [106], PERT Method [107], Mixed Integer Linear Programming (MILP) [108], and CSP [109].

6.2 Services and Data Collection Architecture

A market characterized by rapid changes in consumer demands requires software architectures able to provide agile reconfiguration. However, today's solutions are still based on a monolithic MES structure, where the reconfiguration of the production process requires a long configuration time and the integration of new software and technologies is challenging.

The reference system is constructed as modular SOA-enabled architecture located in a Kubernetes cluster connecting the automation level to classical MES solutions. An overview of the architecture's structure is provided in Figure 6.2. It consists of different applications, communicating with each other through Kafka and RabbitMQ. Therefore, the integration of new applications is transparent and does not require stopping the entire production. The communication is realized through OPC UA clients directly connected with the machines and exposing their functionalities. The main actor is the *Automation Manager* managing the communication with the MES and extending its functionalities to add reconfiguration of the production line, autonomous execution of production orders and advanced scheduling. This section describes the main components of the architecture, while Section 6.3 deep dives into the Automation Manager.

6.2.1 OPC UA Servers

Each machine in the system is equipped with an *OPC UA server* module. The piece of equipment and its server module are strongly intertwined. The data model (in particular the machine status and sensors data) exposed by the server depends on the functions developed to externally control the machine. To create the *OPC UA server* module, the machine's functions are wrapped and exposed as *OPC UA methods*. Then, the *OPC UA data model* is enriched with state variables that a client can read to know the status of the running operations.

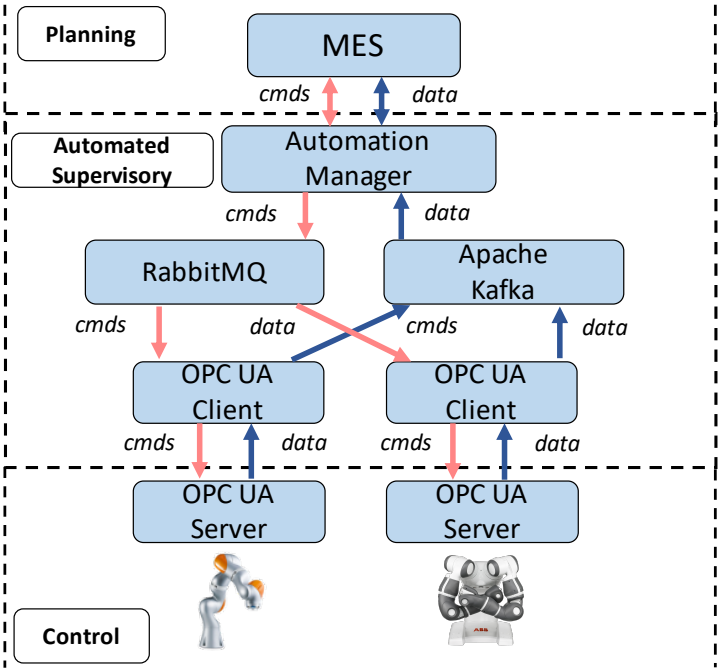

Fig. 6.2: Automation of the *Supervisory level* by introducing the proposed service-oriented architecture. Arrows show the commands and data flow through the automation pyramid and our novel architecture.

6.2.2 Data Collection Infrastructure

An Industrial Internet of Things (IIoT) Data Collection Infrastructure monitors the connected equipment and stores the gathered data. It acts as a service platform allowing, among other functionalities, to interact with the *OPC UA servers* through the RPC paradigm. All the architecture nodes are deployed into the Kubernetes cluster. The Data Collection Infrastructure features a multi-node *Apache Kafka* instance handling data streams. Meanwhile, a multi-node *RabbitMQ* instance is in charge of handling remote procedure calls through queues.

The Data Collection Infrastructure communicates with the equipment through multiple *OPC UA Client* nodes: an active instance (and configuration) is active for each *OPC UA server*. This node creates a persistent connection with the machine and creates an *OPC UA subscription*, specifying the *OPC UA variables* to monitor. Each time a variable changes, the client is notified with the new value, which is written to the configured Kafka topic. The *OPC UA client* nodes are also connected to the RabbitMQ instance and listen for RPC requests from the configured queues. Each RPC request contains the method

to call, its arguments and an universally unique identifier (*i.e.* the `uuid`) used to route the answer to the correct sender. Allowed requests for this client are: the *read* of variable, the *write* on a variable, or the *invoke* executes an *OPC UA method*.

Figure 6.2 shows the data and command flows into the Data Collection Infrastructure and the Automation Manager. Each machine has its own Kafka topic and RabbitMQ remote procedure call queue, managed by the corresponding OPC UA client instance. The Automation Manager (or a generic controller) can exploit this platform by executing machine operations through RabbitMQ and getting equipment statuses and sensors data through Apache Kafka.

6.3 Service-Automation Manager

The *Automation Manager* is the core component connecting the different pieces of software deployed in the Kubernetes cluster and the MES. It is organized in three different layers and many sub-components, depicted in Figure 6.3, necessary to handle the communication with the MES and the machines transparently. The top layer contains the software *Driver* interfacing the Automation Manager with the upper layers of the automation pyramid; the bottom layer is the software *Driver* connecting the manager with the lower layers of the pyramid; the middle layer contains the manager *Core*, a set of *Applications*, and a *Logger*. The Automation Manager is compliant with ISA-95 standard. Thus, it is compatible with any existing software infrastructure based on the same standard. It takes in input information about the plant structure from both the MES, which contains the production line structure and recipes and from a set of configuration files describing the machine capabilities and recipe implementations. These characteristics are meant to ease as much as possible its integration within already existing manufacturing plants.

6.3.1 Drivers

The *Driver* levels contain the components enabling the communication with other pieces of software. The *Equipment Connector* exposes for the other levels basic functionalities of OPC UA, such as variable read/write, methods call, and subscriptions for data changes. It communicates with the OPC UA clients connected to the machines within the cluster. The *MES Connector* manages the communication with the MES. It is implemented as an RPC client calling functions defined by an RPC server connected directly with the MES. As an advantage of using RPC interfaces, the integration with any other MES only

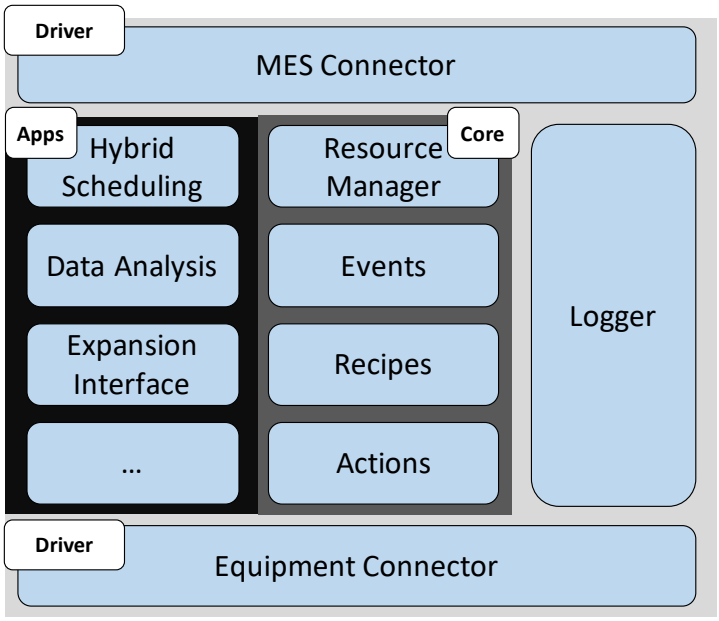

Fig. 6.3: The internal structure of the *Automation Manager*.

requires to simply implement the corresponding RPC server. This driver allows navigating within the MES configuration and notifying the actions executed by the architecture, such as execution of operations and reconfiguration. The last driver is the *Logger*. It publishes log messages on two different Kafka topics: one consists of messages useful for debugging purposes of the system; the other includes messages logging actions executed by the architecture, such as the execution of recipes or machine services. This allows notifying the entire architecture of the status of each level.

6.3.2 Core

The second level contains the *Core* components, defining and implementing industrial processes. The MES represents production recipes as a sequence of dependant tasks, where each one is associated specifically to a *class* of working cells (*e.g.*, a work center). On the one hand, this allows to model production processes at a higher level of abstraction, hiding unnecessary implementation details. On the other hand, this is not enough to execute tasks without human intervention. Therefore, the recipe representation in the Core of the Automation Manager is extended with a low-level model that describes the implementation of tasks on the working cells. This representation consists of a sequence of actions with input and output parameters and an execution

flow, formalized a directed cyclic graph: the actions are nodes of the graph, connected by directed edges to represent dependencies. An action can be a service exposed by the Equipment Connector or a logical construct (*e.g.*, creation of variables, the sum of variables, if, cycle, *etc.*) proposed by the Core. This extension allows executing tasks with a simple visit of the graph nodes. Thus, the software also integrates a parsing module able to interpret the graph and carry out the actions defined by it while following the dependency flow. Then, the actual execution of tasks is managed by the *Resource Manager*. It retrieves the manufacturing structure from the MES and, for each working cell, it connects to the correct machine's client. This ensures that when an operation is executed on a working cell, it has access solely to those clients. It also guarantees that a maximum of one operation is executed on a working cell at the same time.

6.3.3 Applications

The scheduling of production processes on different machines in a dynamic environment is still an open problem. In the literature, this problem is known as Dynamic Flexible Job Shop Scheduling (DFJSS). Although there are many solutions to the static counterparts of this problem, known as Job Shop Scheduling (JSS) and Flexible Job Shop Scheduling (FJSS), when we introduce the dynamic component that characterizes real systems the solutions for these problems are not applicable. The reason is that every time an unexpected event occurs (*e.g.*, the arrival of new orders, machine breakdowns and delays), the schedule is no longer optimal or even not feasible anymore. Therefore, it must be recalculated, which is a time-consuming process. To solve this problem, the most used solution consists of totally reactive scheduling where the decisions are taken dynamically when the machines are in an idle state [110]. Nonetheless, exploiting exclusively dynamic techniques does not provide any certification of optimality. Therefore, a promising direction is to introduce static-reactive scheduling, characterized by a first phase that produces a static schedule of the jobs, dynamically updated on the arrival of events [111]. In the first implementation of the proposed architecture, we opted for this hybrid approach to implement scheduling. It consists of a static phase exploiting constraint programming to produce an optimal solution while minimizing energy consumption and delays. Then, a dynamic component continuously recalculates the scheduling to react whenever an unexpected event occurs, such as new job arrivals and machine breakdowns.

Furthermore, to achieve more precise scheduling, we developed a data analysis application receiving timing data about executed production processes. The gathered data is used to update the completion time estimation

of production processes. To support the integration of applications based on different technologies, an expansion interface exposes the functions of the SOA-enabled architecture's core.

6.4 Architecture Evaluation

The evaluation of the methodology has been carried out in the Industrial Computer Engineering (ICE) Laboratory (Figure 2.3). The production line is governed by a state-of-the-art system implementing the automation pyramid, centered around a commercial, monolithic MES offering some advanced features. Each machine composing the plant exposes services through OPC UA servers, while the commercial MES is orchestrating the execution of the different production processes. However, production orders are manually executed by operators, and features such as reconfiguration or advanced scheduling are not available. Therefore, to assess the applicability of the proposed architecture, we implemented and deployed it to the ICE laboratory production line. We first defined and implemented the Kubernetes cloud architecture. Subsequently, we configured and deployed the Automation Manager software modules. To test the novel functionalities, such as scheduling, resource management, recipe representation and communication with the MES and machines, we have implemented and tested three different production processes on the production line.

First, we evaluate the proposed software architecture qualitatively, by analyzing the newly available features. Then, we evaluate it quantitatively by measuring the overhead introduced due to the new functionalities.

6.4.1 Qualitative Analysis

We provide an evaluation of the proposed architecture's effectiveness in Table 6.1. It summarizes the functionalities implemented by the traditional architecture compared to those provided by the presented solution. Among the new features introduced, the Automation Manager reduces the effort necessary to configure and reconfigure the entire software architecture, from the low-level task implementation to the high-level recipe representation and machine structure. It does so by providing a set of configuration files that contain the structure of the production line and the implementation of production processes, in terms of sequences of tasks. This allows creating a flexible implementation of productive processes, based on actions (*e.g.*, services) exposed by the machines and recipes specified within our novel architecture as a sequence of these actions. It also introduces the possibility to adapt the realization of production processes at run-time, to minimize the total execution

time or to pursue a specific production objective. Furthermore, the hybrid scheduler continuously adapts the production plan as a response to unforeseen events and manages the execution of production processes on the entire plant. Therefore, it reconfigures the system by automatically changing the sequence of manufacturing operations during a certain time frame. In addition, our novel architecture offers a containerized environment where new business functionalities can be developed as single applications on top of the Automation Manager.

Table 6.1: Comparison of functionalities available when using the traditional software stack against the proposed SOA-enabled architecture.

Functionalities	**Traditional Pyramid**	**Automated SOA**
Data collection	✓	✓
Product monitoring	✓	✓
Process monitoring	✓	✓
Inventory tracking	✓	✓
Advanced production planning	✓	✓
Resource management	✓	✓
Automatic reconfiguration		✓
Autonomous process execution		✓
Integration of new software modules		✓
Run-time adaptive scheduling		✓
Reduced configuration time		✓
Reduced deployment time		✓

6.4.2 Overhead Analysis

Additional features come at a price in terms of computational overhead, as it is reasonable to expect. Table 6.2 reports the overhead necessary to call OPC UA functions comparing a direct connection with the machines and through the proposed architecture. We compared the delay of different services, such as read/write of variables, method calls, and subscription to variables. For each operation, the last line reports the additional overhead (in percentage) required when using the proposed architecture.

The additional overhead introduces a significant communication delay, as it ranges between 40% and 70%. However, this communication delay is in the

Table 6.2: Comparison between the communication delay derived from a direct connection with OPC UA and with the proposed architecture.

Transport Type	Read (s)	Write (s)	Methods (s)	Subscription Update (s)
OPC-UA	0.008	0.009	0.010	0.150
SOA	0.013	0.013	0.014	0.255
Overhead	62.50%	44.45%	40.00%	70.00%

context of complex physical processes, such as those involved in a manufacturing line. For this reason, we also evaluated the behavior of the proposed architecture by using it to coordinate different manufacturing processes. Table 6.3 reports the total execution time for three production recipes of different sizes. The table compares the time required using the state-of-the-art architecture with our proposed solution. The total execution times do not consider the transportation time required to move materials on the conveyor belts. This is because transportation data is highly variable and influenced by many physical factors that do not depend only on the control software architecture. The fourth and fifth columns of the table report the execution times obtained with the two different configurations. The last column reports the overhead introduced by the proposed architecture. Considering the number of service calls of each recipe we can see that the delay introduced is minimal and consequently negligible from the total execution time.

Furthermore, comparing Tables 6.2 and 6.3, it is worth noticing that while the additional overhead is significant when considering single operations, it becomes negligible in the context of a complete manufacturing process. In most cases, the introduced overhead is not a limitation. In fact, even at higher operation frequencies, physical processes typically dominate computational processes in terms of execution times. This is due to the fact that manufacturing processes are strongly dominated by mechanic operations, in which timing is measured in tens rather than tenths of seconds.

6.5 Services and Models-based Advanced Scheduling

In this section, we propose a reactive-dynamic scheduling algorithm exploiting the proposed three-level modeling approach. The scheduling algorithm exploits the information represented in the model, aiming at minimizing the makespan of the production while maximizing the machine utilization. The algorithm exploits the increased granularity of the model to schedule the sub-

Table 6.3: Comparison between the execution time when using the state-of-the-art and the proposed architecture to govern three different complete production recipes.

Recipe	Tasks	Service Calls	OPC UA Time (s)	SOA Time (s)	SOA Overhead
1	4	54	70.34	70.85	0.72%
2	5	44	66.73	67.04	0.46%
3	11	132	158.83	159.63	0.50%

tasks (*i.e.*, services) specified within the tasks. Furthermore, knowing the encapsulated services within a sub-task, it is possible to identify the required resources (*i.e.*, tools, materials *etc.*). This allows to schedule these sub-tasks taking into account the various delays, such as waiting times for the retrieval of missing resources.

6.5.1 Scheduling Problem Statement

Let M be a set of the machines available in the production system. Let $R = \{r_1, r_2, \ldots, r_n\}$ be the set of recipes. The relation $R_p = \{(r_n, p_n) \mid r_n \in R, p_n \in \mathbb{N}\}$ associates a recipe r_n with a priority p_n. Let $T_n = \{t_1, t_2, \ldots, t_n\}$ the set of tasks of a recipe n. For each task $t_i \in T$, $ST_i = \{st_{i1}, st_{i2} \ldots, st_{ij}\}$ is the set of sub-tasks of t_i, as modeled in the hierarchical representation of the production process. The set of possible allocations between sub-tasks ST_i and machines M is defined by the relation $PA = \{(st_{ij}, m_k, d_{ij}) \mid st_{ij} \in ST_i, m_k \in M, d_{ij} \in \mathbb{N}\}$, where d_{ij} is the duration of the sub-task st_{ij} allocated on the machine m_k. The actual allocation between sub-tasks and machines is formalized as a relation $S = \{(st_{ij}, m_k, d_{ij}, \tau_{ij}) \mid st_{ij} \in ST_i, m_k \in M, d_{ij} \in \mathbb{N}, \tau_{ij} \in \mathbb{Z}^+\}$, where τ_{ij} is the starting time of the sub-task on a machine expressed as positive integer number.

The objective of the scheduling algorithm is to find an assignment S, that minimizes the makespan while maximizing the machine utilization U_k. The makespan MS is computed as the dIfference between the starting time of the latest allocated sub-task plus its duration, and the starting time of the first allocated sub-task:

$$MS = \max_{\forall st_{ij} \in S} (\tau_{ij} + d_{ij}) - \min_{\forall st_{ij} \in S} \tau_{ij} \tag{6.1}$$

The machine utilization u_k, is defined as a percentage of the total production time in which the machine k is busy:

$$u_k = \frac{\sum_{\forall \langle st_{ij}, m_k \rangle \in S} d_{ij}}{MS} \cdot 100 \tag{6.2}$$

6.5.2 Scheduling Algorithm

The scheduling algorithm takes decisions based on the information stored in the model and the priorities assigned at runtime to the recipes. In particular, whenever a new order arrives, the recipes of the requested products are added to the set of recipes to be scheduled. Meanwhile, in case of delays or machine unavailability, the priority of the corresponding recipe is lowered. This allows to allocate other tasks that are actually "runnable", optimizing the machines utilization. Once the missing materials or machines become available, the priority of the tasks is restored. Then, the scheduler is invoked each time an unexpected event occurs. An unexpected event could be the arrival of new orders, delays in retrieving materials or machines incorrectly set-up for an allocated task.

Algorithm 2 Service-based Scheduling

```
1: τ ← getCurrTime()
2: for {(st_ij, m_k, d_ij, τ_ij)} in S do
3:     if τ < τ_ij then
4:         S ← S \ {(st_ij, m_k, d_ij, τ_ij)}
5: R ← sort R by p
6: for r_n in R do
7:     for t_i in T_n do
8:         assign(t_i)
```

The algorithm is made of two procedures. The first is described by the Algorithm 2. Its main functionality is to update the schedule S at each invocation. In particular, at line 1, the procedure retrieves the current time. Lines 2-6 remove from the previous schedule all the sub-tasks that are not already allocated and started on a machine at time τ. Line 7 sorts the set of recipes according to their priority. We chose this sorting parameter to handle high-priority orders first. Then, the algorithm evaluates each sorted recipe (line 8-12). For each $r_n \in R$, and each task $t_i \in T_n$, the algorithm invokes the support function `assign(`t_i`)` (line 10).

The $assign(t_i)$ function is described by the Algorithm 3. It provides a machine assignment for every sub-task composing a task. It does so by searching for unused time slots in the schedule of each machine. Therefore it aims at maximizing the busy-times of machines and, thus, at maximizing the utilization of such a piece of equipment. The function only schedules sub-tasks that are not executed or already allocated. In fact, as implemented in lines 2-4, the algorithm checks whether the sub-task st_ij is included in the set of sub-

Algorithm 3 Assign function

```
    Input: t_i
 1: for st_ij in t_i do
 2:     if st_ij in S then
 3:         skip
 4:     st_m ← 0
 5:     st_end ← ∞
 6:     st_τ_ij ← ∞
 7:     for {(st_ij, m_k, d_ij)} in PA do
 8:         τ_ij = findFreeTimeSlot(st_ij, m_k)
 9:         if (τ_ij + d_ij) < st_end then
10:             st_m ← m
11:             st_τ_ij ← τ_ij
12:             st_end ← (τ_ij + d_ij)
13:     S ← S ∪ {(st_ij, st_m, d_ij, st_τ_ij)}
```

task already allocated S. If it is, the sub-task is skipped. Lines 5-7 define local variables used to save the best solution found, which is the first machine that is able to complete the sub-task. For each possible allocation in PA, the algorithm searches (lines 8-15) for the best one, identified as the earliest starting sub-task on a machine m_k. The function $findFreeTimeSlot(st_{ij}, m_k)$ at line 9 retrieves the starting time-slot of the sub-task st_{ij} executed by the machine m_k. In particular, the function searches the first free time-slot capable of containing st_{ij}, within the schedule S of the machine m_k. It also considers the machine function layer of the model, to assess whether a dependency within two sub-tasks composing the same task is present. A dependency means that the sub-tasks may share resources (*i.e.*, materials) and, thus, must be allocated on the same machine. In such a situation, the function returns $+\infty$. Lines 10-14 check whether the time-slot found by the $findFreeTimeSlot(\dots)$ function is better than the one found in the previous cycle. In such a case, the sub-task is allocated to the machine (lines 11-13) at the found time-slot, and then the schedule is updated (line 16).

The algorithm also maximizes the utilization of the machine: a more granular representation of tasks as sub-tasks allows the scheduler to allocate sub-tasks in smaller time-slots than taskConsidering the equation 6.2, the implemented allocation strategy allows shrinking the gap between the numerator and denominator. The complexity of the algorithm is linear on the cardinality of S; the complexity of the *findFreeTimeSlot* is also linear as it relies on interval trees.

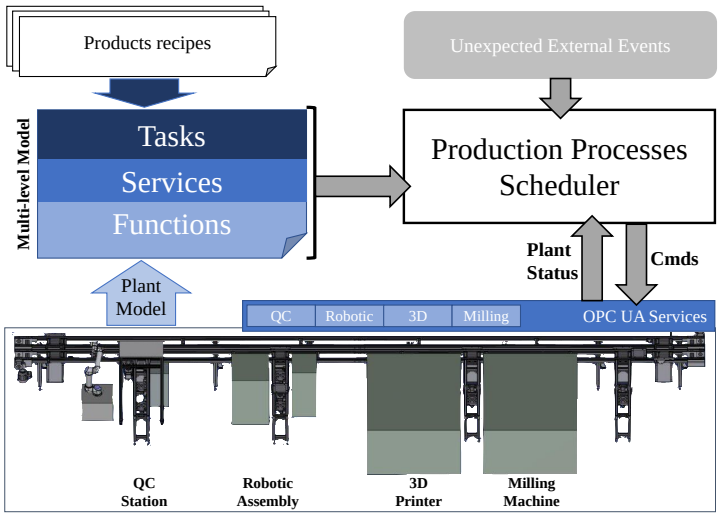

Fig. 6.4: The figure shows the experimental setup used to assess the methodology. The three-level model of the production processes is built by modeling the production recipes and the manufacturing line equipment. Then, this representation is used to implement a service-based scheduler. Lastly, the scheduler is executed in a real production environment.

6.5.3 Experimental Setup

Figure 6.4 summarizes the experimental setup used to evaluate the advantages the proposed approach. We applied the presented modeling and scheduling techniques to the ICE laboratory. The system is able to implement the Service-oriented Manufacturing paradigm: each machine exposes to the user its functionalities as a set of services. The communication is realized through the OPC UA protocol: a well-known standard for industrial machine communication [96]. Furthermore, the system is governed by a commercial MES communicating with the machines.

The production recipes available for the plant have been modeled according to the three-level modeling approach described in Section 3.5. Each level in the models have been expressed using *SysML activity diagrams*. SysML provides an intuitive and well-defined graphical language, explicitly tailored to express the structure and behavior of complex systems. Thus, SysML aims at easing the specification and modeling phase for all the three levels of the production model. Furthermore, SysML supports the XML Metadata Interchange (XMI): an XML-based format easing data exchange, manipulation, and analysis. Thus, allowing to easily implement procedures able to analyze and manipulate the data carried by models.

Then, we implemented the algorithm, and made it able to take as input the proposed three-level model, expressed using the SysML syntax. The algorithm implementation is interfaced with the MES to monitor unexpected external events. Finally, it interfaces with the OPC UA servers deployed on the production plant, in order to monitor the state of the system, and to send commands to the machines to execute the decisions the algorithm takes.

We first describe how production recipes are expressed in SysML, while following the proposed hierarchical modeling strategy. Then, we provide more details about the algorithm implementation and its interfacing with the production plant's infrastructure. Lastly we report the results obtained by exploiting the implemented scheduler. We compare the results to those achieved by using the native scheduler provided by the commercial MES governing the system.

6.5.4 Implementation

The production recipes described by applying the proposed hierarchical modeling approach are concretely expressed using SysML. Each level described in section 3.5 is modeled as an *activity diagram*. We started defining the data types related to *task*, *service*, and *machine*: the essential types allowing to express all the information necessary within each model. Then, we described the production recipes at the first level of the hierarchy. We modeled the nodes of the graph as an object of type *task*, on which we specified the parameters, such as *name*, *materials*, *etc*. Each task is assigned to one or more *machine* objects. On these objects, we specified the parameters related to the machine-task relation, such as tools, electrical consumption, execution time, *etc*. The dependencies between tasks are specified through control-flow arrows.

Each task described in the first level is further refined at the second level of the hierarchy. The correspondent nodes of the graph are represented as an object of type *service*. The service nodes store the information of each service, *i.e.*, *inputs*, *outputs*, *etc*. Thus, leading to activity diagrams as the one reported in Figure 6.5. It represents the service-level model, expressed as a SysML activity diagram, of task *T2* of the case study previously shown in Figure 3.8.

Furthermore, the model is annotated with the information about the calls to the OPC UA protocol required to invoke each service. The order of OPC UA service calls is modeled by using control-flow arrows, while conditional branching in invocation sequences is modeled by *decision nodes* in the activity diagram.

Finally, each machine function represented in the lower level is also modeled by a SysML activity diagram. The activity diagram models the PLC behaviors related to machine service. The nodes of the graph are modeled as

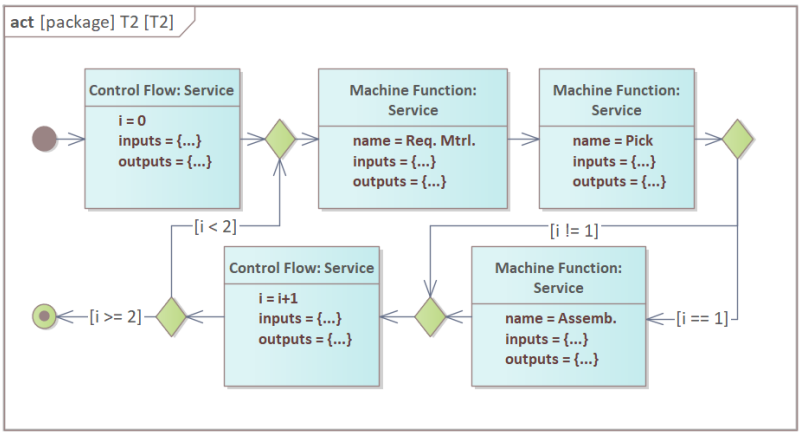

Fig. 6.5: Service-level model of the task *"T2"* represented as a SysML activity diagram. It expresses all the information contained in the second level of our proposed model. Each node is specified by a typed object. The arrows describe the edges of the graph in the middle layer of Figure 3.8.

an object of type *service*, which contains the information related to PLC functionalities, such as *inputs*, *outputs*, *etc*. The nodes of the graph are specified with control-flow arrows that allow modeling the sequence of the low-level services.

We automatically extract the essential information stored in the XMI description of the SysML model, and we store such information into a JSON file. This representation will provide the input to our implementation of the algorithm described in section 6.5. The scheduler is implemented as a software module of the Automation Manager, describe in Section 6.3. The closed-loop communication between the scheduler, the Automation Manager and the MES allows interracting with the machines, retrieving the current state of the plant, and fetching new orders. The communication is carried out through the OPC UA protocol.

6.5.5 Results and discussion

We compare the results obtained by the scheduler implemented by the commercial MES, which relies on a classical RTN-based task representation, against our service-based scheduler, which exploits the proposed hierarchical information model. Table 6.4 reports the result obtained with the two different approaches. As a benchmark, we input 450 production processes instances (*i.e.*, production orders), randomly generated from a pool of 4 different recipes. The first row compares the total makespan obtained by the

Table 6.4: Comparison between the scheduling of 450 production recipes, using a classical RTN-based representation against the proposed hierarchical modeling approach.

Param	RTN Representation	Hierarchical Representation	Diff. (%)
Cycle Time	20:56:19 (h)	20:17:23 (h)	-3,1%
Avg. Change Overtime	5:42:47 (h)	5:05:16 (h)	-10,95%
Avg. Utilization	74,47%	80,85%	+6,38%
Throughput	235,96 (u)	245,75 (u)	+4,14%
Avg. Time To Complete HP	1:02:08 (h)	0:57:08 (h)	-8,07%
Avg. Time To Complete LP	2:19:25 (h)	2:16:45 (h)	-1,92%

two approaches, calculated using Equation 6.1 and Equation 6.2. The proposed scheduler is able to reduce the total execution time needed to complete the 450 orders by almost 40 minutes. Thus, providing an improvement of 3.1% with respect to the state-of-the-practice approach. The next three lines show the comparison of the average amount of time in which the machines are not used (*e.g.*, the change time), the average machines utilization, and the throughput. The proposed approach decreases the average change time, while increasing both the average machine utilization and the system throughput. Thus, the proposed approach successfully hits its target of reducing the makespan, while increasing the average machine utilization.

The improvement is due to the scheduling algorithm enabling a more precise interleaving than the traditional task-resources representation. In fact, dividing one task of a recipe in sub-tasks means partitioning the time necessary to complete such a task. Therefore, the algorithm is able to fill machines downtimes with sub-tasks of different tasks or even of different recipes. As a consequence, the throughput also increases due to the algorithm ability to fill the production line's execution time-span more efficiently. Lastly, the last two rows compare the average time to complete production recipes at high priority (HP) and low priority (LP). The completion time for a recipe is calculated as the difference between the time instant in which the last task ends and the time instant in which its first task starts. While reducing the completion time for both new high and low priority orders, the proposed representation allows handling more efficiently the arrival of new higher priority orders. In general, the scheduler routine in charge of managing the priorities of the tasks

is able to handle more accurately the allocation of sub-tasks whether certain conditions are met (*e.g.*, required materials availability).

Nonetheless, such a methodology can be applied only to manufacturing systems built both conceptually and concretely around the concept of "service". In fact, it assumes that the various machines composing the plant are capable of implementing and exposing functionalities enclosed in such constructs. Furthermore, the presented work assumes that the production domain (*i.e.*, the type of manufacturing industry) handles processes that can be interrupted, with materials temporarily stored in buffers. Therefore, additional constraints and aspects would be needed for this work to be applicable to the production of non-durable goods (*e.g.*, food and beverages).

6.6 Conclusion

This chapter presented a variation of the traditional automation pyramid software stack to control SOA systems, which introduces manufacturing services at PLC level and an Automation Manager to control them. The interaction of the manager with the other pyramid's software components relies on well-known manufacturing communication standards. The results of the application of the proposed architecture to a real production plant showed that the overhead introduced is negligible. Nonetheless, the added functionality over state-of-the-art architecture proved to increase production flexibility.

Furthermore, by exploiting the multi-level modeling approach to manufacturing processes, this chapter proposes a scheduling procedure for production processes with more comprehensive knowledge of how tasks are implemented on machines. Therefore, the proposed scheduler is aware of the status of the machines and buffers and, thus, whether and when the task may be preempted. We assess the proposed approach by modeling in SysML four production recipes and scheduling a high number of instances. The results show that the conjunction of the modeling approach with the proposed scheduling algorithm, offers better performances than a traditional method based on a plain task-resources representation. Overall, the achieved increased efficiency should allow reducing the production costs of the single units being produced.

7

Modeling in Industry 5.0: What Is Missing

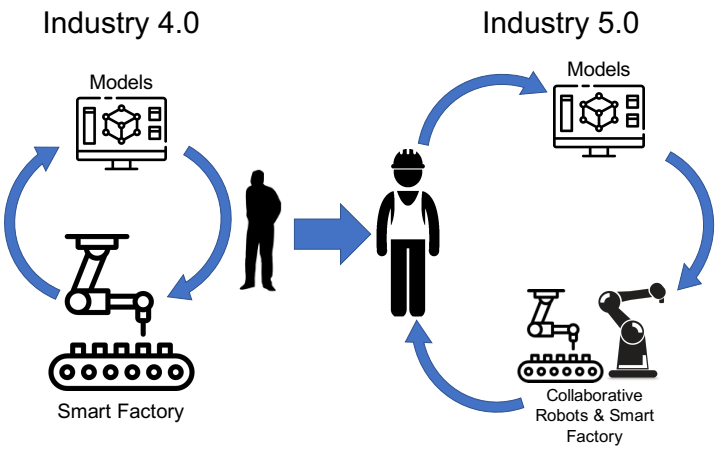

Fig. 7.1: While in the vision proposed by Industry 4.0 the human is mainly supervising and "observing" the system, Industry 5.0 aims at putting the human back at the center of manufacturing. To pursuit such a task, models of manufacturing systems must be adjoined with models of human behaviors and, thus, novel modeling methodologies must be investigated.

Industrialization underwent four main evolutions in human history, impacting on population and economic growth, and establishing important social changes. Each industrial revolution has brought to humankind new technical innovations, made possible by the acquisition of a a better understanding of the natural environment and its resources: the use of steam, fossil fuels and electrical energy have contributed to lifting the burden of moving heavy and complex physical machines from animals and humans. The last iteration

of such revolutions, dubbed "Industry 4.0" [112], proposes to transition from mechatronics systems to CPSs incorporating data and intelligence, that in the context of manufacturing has been dubbed CPPS. Such systems are capable of communicating with each other, acquiring and transmitting real-time production data used to optimize production processes. This main contribution leads to increased production throughput while reducing costs and waste. All these innovations have been made possible through the adoption of key-enabling technologies such as IoT and Cloud computing. Furthermore, the development of virtual models of the manufacturing system, *i.e.*, *Digital Twins*, enables the simulation of the entire production and allows performing what-if analysis.

The pursuit for production optimality and efficiency proposed by modern manufacturing often leads to an inevitable conclusion: human labor is neither as efficient nor as cost-effective as machine labor. This is particularly true in repetitive and dull production tasks that characterize the "mass production" trend. Consequently, the cost in terms of manpower will be quite evident in the following years, when the Industry 4.0 principles will be fully implemented. Indeed, on a global scale, human labor is implemented by consumers. Thus, negatively affecting human labor may lead to decreased demand, making the investments to reach full automation not sustainable.

In this context, pushed by human resilience, the concept of Industry 5.0 [113] is emerging, intending to find a sustainable trade-off between automation and human labor. As depicted in Figure 7.1, it proposes to put the human back at the center of manufacturing: rather than exploiting the manpower and the human muscles, it capitalizes on human brainpower, adding to the production loop the creativity and the problem-solving abilities that cannot be transferred to autonomous machines. In this context, nonetheless, autonomous and intelligent systems are a fundamental addition to achieving the maximum process efficiency: collaborative robots (cobots) will be able to support the human during the production, by observing, learning and offering help when needed.

In an Industry 5.0 context, the design of cognitive and collaborative manufacturing systems is even more complex, since the process must deal with uncertainties of humans' behaviors. In future industries, collaborative robots (Cobots) are not only required to autonomously offer assistance, by recognizing the type of job the human worker is executing, but must also keep a safe working condition by not creating dangerous situations. As such, modeling such systems requires a language and a modeling environment able to capture the complexity of their possible behaviors, to perform in-depth analysis and carry out safe control strategies and AI algorithms.

By establishing the novel modeling requirements being posed by Industry 5.0 and how collaborative systems should be designed, this chapter defines "what is missing" in this field. Therefore, we assume the modeling state-of-the-art presented by Chapter 3, where we identified the modeling trends of traditional production systems, while we investigate the research work being accomplished on Industry 4.0 CPPSs. The contributions proposed in this chapter are:

- The identification of novel requirements of the Industry 5.0 paradigm. Such a set of requirements are specifically meant to deal with the human-centric vision of future production lines.
- A discussion on how modeling languages and methodologies could be adapted to support the design of such a class of systems. In particular, we focus on trying to imagine how the SysML language features would be able to respond to Industry 5.0 requirements.

This chapter is organized as follows. Section 7.1 defines Industry 5.0 and the requirements of such a manufacturing trend. Section 7.2 instead proposes possible research directions on modeling languages and methodologies compatible with Industry 5.0 requirements. In addition, Subsection 7.2.4 proposes a development direction for SysML. Finally, Section 7.3 draws our conclusions.

7.1 From Industry 4.0 to Industry 5.0

The starting point of the Industry 4.0 evolution has been the so-called software automation pyramid (see Figure 7.2). Under this view, there is a continuous bi-directional information flow from the actual plant to the management software (*e.g.*, ERP). Level by level, electrical signals become strings of bits, then abstract data types, then data objects and finally services. A huge variety of protocols has been created to model the exchange of information among the different levels. Few of them, are able to act as a unifying protocol managing all kinds of information exchange between all levels of the automation pyramid. The possibility of abstracting actual protocols has been the basis of Industry 4.0 since an entire production plant can become a service-oriented software architecture. This sensibly reduces the complexity of developing software applications managing data from the sensor/actuator level up to the cloud infrastructure. Data managing is thus becoming the core problem of a production line and Industry 4.0 focused its revolution on using such data. This allows optimizing the production, make predictions and facilitate reconfiguration, thus moving from mass production in large batches to personalized

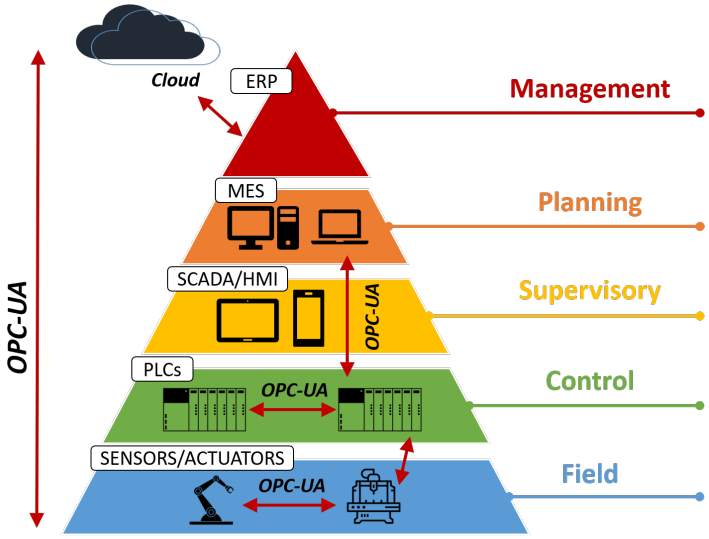

Fig. 7.2: The automation pyramid: the starting point of Industry 4.0. Role of OPC-UA as a unifying protocol focused on data.

production in small and differentiated batches. Research and development in this area allowed to define protocols able to implement these requirements, among them OPC-UA [96] is one of the most successful. However, such protocols are useful to manage the heterogeneous information in the system once the system is up and running, while they are not meant to express the information of a system being designed (or redesigned). Thus, alongside protocols, models became necessary to design and implement systems able to fulfill the main design requirements of Industry 4.0 manufacturing systems. In particular, six main design requirements can be listed to implement Industry 4.0 principles:

- **Interoperability**: the ability of cyber-physical systems (*i.e.*, work assembly stations), humans and Smart Factories to connect and communicate with each other via the Internet of Things and the Internet of Services;
- **Reconfigurability**: the set of operations to produce a good (recipe) must no longer be fixed and statically scheduled, the system must adapt to the variations of its surrounding conditions and production requirements;
- **Virtualization**: a virtual copy of the Smart Factory which is created by linking sensor data (from monitoring physical processes) with virtual plant models and simulation models;

- **Decentralization**: the ability of cyber-physical systems within Smart Factories to make decisions on their own Real-Time Capability: the capability to collect and analyze data and provide the insights immediately;
- **Service Orientation**: offering of services (of cyber-physical systems, humans and Smart Factories) via the Internet of Services;
- **Modularity**: flexible adaptation of Smart Factories for changing requirements of individual modules.

The core (a bit naïf) assumption of Industry 4.0 is that the production is completely automatic, thus the role of humans is limited to control and supervision. However, this is not feasible in the majority of production lines, particularly because flexibility, adaptation and precision of robotic activities are still far away from human abilities. People must still play a productive role in an Industry 5.0 production line.

However, differently from the third industrial era, Industry 5.0 promotes a significant change in the adaptation philosophy: people have not to adapt their behavior to the machines, but **machines must automatically adapt their activities to the human ones**. Thus, there is a new list of design requirements to implement this new Industry 5.0 principle:

- **Uncertainty**: this is related to the unpredictability of the human behaviors that must be represented under some level of uncertainty. Indeed, uncertainty impacts how interoperability and reconfigurability are implemented by Industry 5.0 systems, as subsystems must reconfigure and interoperate by considering that other subsystems may be humans. Facing the non-determinism becomes crucial and this must be taken into consideration by all languages, models and theories;
- **Cognition**: an evolved MES is necessary to control the production line; it cannot be based on simple deterministic algorithms, but it has to implement cognitive methods that must be continuously reinforced by the experience; in other words, AI and Machine Learning (AI/ML) must become the core technologies to program this kind of architectures in order to allow the system to co-exist with natural intelligence of human agents;
- **Safety**: a higher level of safety must be reached to guarantee the cooperation between humans and robots, since all operations scheduled by a robot must be checked for safety before their execution by considering the current and the possibly predictable human behaviors.

Indeed, these new requirements require a technological leap, as well as an advancement of the tools necessary to support the design of the new technologies that must be supported by novel modeling and design languages and methods. The next section analyze how to fulfill these requirements, by

analyzing what is missing and by proposing a future SysML development direction.

7.2 Modeling Industry 5.0: what is missing and possible directions

Industry 5.0 is still in its infancy. To the best of our knowledge, no research effort exists today in proposing modeling tools or methodologies for this trend. In particular, modeling CPSs with humans in the loop is still an open task [114, 115].

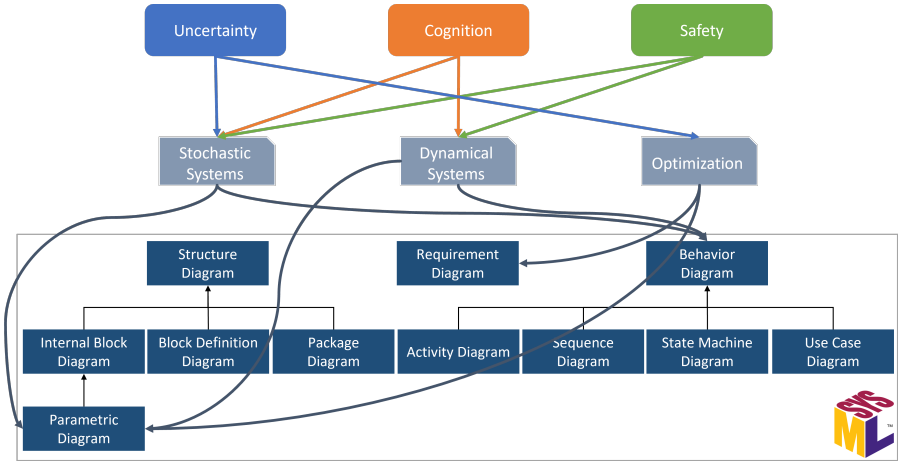

Fig. 7.3: Relations between Industry 5.0 requirements asserted in Sec. 7.1, systems types and possible associations with SysML diagrams.

As stated in Sec. 7.1, one of the main requirements of Industry 5.0 is reconfigurability: it involves guaranteeing a certain resiliency and adaptation degree of the CPPS to real-time changes in the environment. In this regard, to carry out the best reconfiguration strategy, it would be fundamental to have models spanning over business, processes and control viewpoints: integrating the widest possible set of information increases the quality of the control strategy carried out by the reconfiguration procedure. By also considering humans in the manufacturing loop, a degree of complexity is added on top, since it involves modeling nondeterministic behaviors. Of course, models are not enough in this case to represent all the possible behaviors. It is necessary to employ cognitive methods, which require experimenting with AI techniques to carry out control strategies based on observations. Nonetheless, models

can aid the "learning" procedures of AI algorithms, by providing a basic set of information that can be helpful to interpret the current system status. This is particularly true also for safety considerations: the cooperation between humans and robots can be dangerous for human operators. Therefore, possible critical situations must be predicted and avoided. Different techniques can be exploited, most of them based on pattern recognition and AI-based prediction using different types of learning techniques. In all these scenarios, models can aid the process of carrying out the best possible strategy. These techniques introduce in the system different levels of uncertainties, making systems non-deterministic. While multiple mathematical frameworks allow reasoning on non-deterministic systems, no system modeling language at the state of the art provides the designer the tools necessary to specify such systems efficiently.

To support the engineering of Industry 5.0 systems, engineers must be put in the condition to comfortably specify non-deterministic behaviors intrinsic to humans' behaviors, and those introduced when using AI techniques. As such, future system modeling language must allow specifying uncertainties and their semantics must be built on top of probabilistic and stochastic mathematical formalisms. Furthermore, to be effective for system design, they must allow designers to select the appropriate degree of abstraction and is capable of scaling on the level of details.

Reusing models is also a winning strategy in the context of modeling complex manufacturing systems: as presented in Chapter 3, it allows to cut down the modeling effort in a multi-faceted scenario. SysML is a suitable language because it is expressive and it is extensible with profiles and stereotypes. Furthermore, it is XMI-based and, thus, it can be manipulated and translated to other languages (*e.g.*, formal languages, programming languages, etc.).

Figure 7.3 depicts the set of Industry 5.0 characterizing features introduced in Sec. 7.1, *i.e.*, *reconfigurability*, *uncertainty*, *cognition*, and *safety*. The figure relates each feature with the mathematical formalisms required to represent them.

7.2.1 Uncertainty Requirement

Uncertainty requirement strongly impacts on the interoperability and the reconfigurability of the system. The system must be able to interoperate with non-epistemic agents (*i.e.*, human beings), that may potentially act irrationally or through apparently unexplicable paths. Thus, machines may interpret and predict the behavior of these agents only through models contemplating probabilistic, as well as statistical behaviors. At the same time, uncertainties in the environment surrounding the machines may lead to the

necessity of system adaptation, *i.e.*, reconfiguration. This implicates carrying out optimization procedures to maximize the effectiveness of the adaptation. Furthermore, system reconfiguration may still need to consider future possible uncertainties that may be caused by the intervention of human agents, as well as by external causes. As such, stochastic models used to represent the uncertainty of the system may act as triggers for the models specifying the system reconfiguration.

Thus, modeling uncertainty in the context of Industry 5.0 requires the ability of capturing stochastic systems formalisms, which typically involve representations based on probabilistic and statistical methods, as well as optimization specifications. Formal techniques able to capture, within the same framework both optimization and statistical models are gaining maturity [116, 117, 118]. However, a language capable of modeling these aspects within the same framework, while providing an intuitive syntax and semantics to designers has yet to be proposed.

7.2.2 Cognition Requirement

Cognition requirement not only associated with stochasticity but also to system's dynamics. In fact, the cognition process must implement techniques dedicated to recognizing patterns of the dynamical behavior of a certain component or set of components. On the other hand, recognizing the role of the human actor in Human-Robot Collaboration (HRC) is also a key component of Industry 5.0 systems. Such a process involves determining a set of tasks that the human is capable of carrying out in the manufacturing process. It is also necessary to estimate his/her performances, to evaluate whether the robot's cooperation would be beneficial to the overall production. In HRC tasks, the decision-making procedures of robots must deal with the mental state of the human as well as the sense of trust he/she has in the robotic collaborator. Such collaboration facets must exploit inference models and reinforcement learning, to correctly perceive the human's status and intentions, dealing also with uncertainty. Furthermore, it is also necessary to strengthen communication models, to provide convincing decisions explanations to human operators.

Modeling cognitive-aware systems, thus, requires methodologies and languages able to capture multiple facets of human behaviors and emotional processes. Mathematical models have been proposed to guide the decision process and control synthesis of HRC, exploiting non-learning techniques such as optimization based on Markov Chains [119] and learning techniques such as Reinforcement Learning [120]. Nonetheless, an inclusive framework for the

modeling and specification of cognitive-aware systems has yet to be proposed, limiting the applicability of such techniques to domain experts.

7.2.3 Safety Requirement

Safety requirement is a feature associated with both stochastic and dynamical systems. In Human-Robot interactions and collaborations, safety can be related to avoiding both physical and psychological harm. Different categories of methodologies have been developed to provide safety in HRC environments [121]. *Safety through control* proposes methods to prevent unwanted contacts between the human and the robot by detecting at run-time unwanted system's states and reacting to recover from a dangerous situation. *Safety through motion planning* category proposes more proactive approaches suggesting human-aware motion planning techniques exploiting dynamical models. Such models exploit differential equations and, thus, a strong mathematical foundation to define the kinematics of the robotic systems. In methodologies related to *Safety through prediction*, the motion planner is adjoined with predicted human behaviors and movements, to proactively propose movements instead of continuously replanning. *Safety through psychological considerations* is more focused on extra-functional motion properties, *e.g.*, acceleration, velocity, distance from the human, etc, since they are the most influential parameters on human's well-being.

In general, safety requires setting the boundaries of unwanted behaviors. At the state of the art, the specification of such requirement still relies mostly on complex mathematical notations, based on complex logic, automata, and equations [122]. The use of formal mathematical frameworks is imposed by the critical importance of the concepts being expressed. However, this makes the life of engineers harder. Some attempts of proposing domain specification languages able to simplify the specification task, while still guaranteeing strong formal support, have been already presented [69, 123]. However, no general designer-friendly language has been proposed so far, able to intuitively and effectively capture different types of safety requirements, while providing the formalisation required to tackle safety issues.

7.2.4 SysML support to Industry 5.0 models

While many features of Industry 4.0 may be represented by using SysML and other state-of-the-art modeling and description languages, this is not the case of industry 5.0. Neither SysML nor other modeling languages at the state-of-the-art provide the necessary language standardized constructs to specify every type of Industry 5.0 system. Focusing on SysML, among the different

diagrams, it implements the parametric diagram type, that can capture, to a certain degree, the dynamics of a system. However, such a type of diagram is not able to represent in a standardized manner, for example, probability distributions or stochastic equations for numerical approximations. As such, the language has to be extended to robustly support the design such systems. To optimize a system according to the system's requirements is a process involving, as an example, linear programming techniques and an optimization model. Such a model should be designed with the support of SysML diagrams, such as the requirements diagram. However, in requirements diagrams, requirements are specified only in natural language. Therefore, a methodology to extend the expressiveness of SysML able to include formal specifications is still missing.

Thus, a research effort is necessary to extend the existing modeling and specification languages, and to define new ones as well, able to effectively support engineers. This effort should move toward easier ways of specifying models which are supported by stochastic, dynamical and optimization mathematical frameworks underneath, to allow the designer to better tackle reconfigurability, uncertainty, cognition and safety requirements of Industry 5.0 systems.

7.3 Conclusion

The complexity of Industry 5.0 systems poses new challenges in their design. A unifying modeling methodology, able to thoroughly capture all the aspects and viewpoints does not exist. Methodologies based on SysML prove that such a language has the appropriate degree of expressiveness to model complex CPPSs, as shown in Chapter 3. Therefore, we proposed future directions to model Industry 5.0 systems exploiting SysML and tackling new design requirements. Such a set of requirements emerged by (re-)positioning humans inside the production line and, therefore, trying to exploit humans' capabilities while assuring a strict safety degree.

8

Summary of the Experimental Results

This Chapter collects the experimental results of the methodologies proposed by this thesis, applied to the production line presented in Section 2.3. In particular, it reports the results of the compositional design approach proposed in Chapter 4, applied to the robotic case study defined in Subsection 4.1.2. Such a case study is similar to mobile robots available in the ICE Laboratory. Thus, the generated software could be easily implemented in such Automated Guided Vehicles (AGVs). Furthermore, experimental results for the virtual prototyping flow of Chapter 5 are collected and reported. Such prototypes are a direct representation of the structure and functionalities of machines composing the ICE production line. Finally, the Chapter details the results of the software architecture proposed in Chapter 6. The architecture is currently installed in the ICE laboratory. The experimental data of the advanced scheduling algorithm described in Section 6.5 reflect actual production executed on the real plant.

8.1 Compositional Design using Assume-Guarantee Contracts

Table 8.1 reports the time required by the various design phases, *i.e.*, the realizability, synthesis, the control software generation and its execution based on the finite state machines produced by reactive synthesis tools. The different entries have been obtained by varying the three main design problem dimensions, *i.e.*, the number of blocks in the two-dimensional space representation, the number of robots, and the number of targets. The *Non-decomposed formalization* columns report the time required to synthesize the control strategy from the contract representing the system "holistically". The last four columns report the time required by applying the presented approach. In both cases,

Table 8.1: Comparison between the time needed to obtain the final control strategy using the holistic system contracts, and decomposing the design problem. The experiments have been carried on by varying the three main dimensions of the problem.

Problem Dimension			Non-decomposed system formalization			Decomposed system formalization			
# Blocks	# Robots	# Targets	Synthesis time (s)	Code Generation (s)	Total Time (s)	Synthesis time (s)	Code Generation (s)	Simulation for Validation (s)	Total Time (s)
9	2	2	20.36	20.47	**40.83**	6.85	37.31	0.01	**44.17**
16	1	2	31.48	25.83	**57.31**	20.54	40.46	0.03	**61.21**
16	2	2	3924.18	1906.16	**5831.37**	33.13	244.48	0.02	**277.63**
16	2	4	3924.22	1910.79	**5835.01**	33.17	247.97	0.04	**281.18**
16	2	6	3924.60	1913.95	**5838.55**	33.17	247.96	0.07	**281.20**
16	3	6	Time Out (6 hours)	Time Out (6 hours)	**Time Out (6 hours)**	71.59	380.86	0.04	**452.49**
16	4	6	Time Out	Time Out	**Time Out**	184.35	810.34	0.05	**994.74**
16	4	8	Time Out	Time Out	**Time Out**	184.35	814.56	0.06	**998.97**
25	2	4	Time Out	Time Out	**Time Out**	55.54	292.94	0.08	**348.56**
25	2	6	Time Out	Time Out	Time Out	55.54	295.08	0.09	**350.71**
25	3	9	Time Out	Time Out	Time Out	142.79	504.25	0.12	**647.16**
25	4	12	Time Out	Time Out	Time Out	238.83	943.75	0.17	**1182.75**
25	5	15	Time Out	Time Out	Time Out	312.64	1351.49	0.21	**1664.34**

we reported the time necessary to perform reactive synthesis, code generation, and the total time required. In the case of the decomposed system formalization, we report also the simulation time required for validating the generated code. The approach proposed in this work allows synthesizing also the instances that are intractable by the state-of-the-art approach. In particular, it shows good scalability also when increasing the number of robots in the system, as highlighted by the experiments using the 16 and 25 blocks occupancy grids.

Table 8.2: Qualitative comparison of the generated code.

Blocks	Robots	Target	Synthesized States		Number of ROS Messages	
			Non-decomp.	Decomp.	Non-decomp.	Decomp.
9	2	2	10704291	2664 * 2	15	19
16	1	2	3435704	8432	35	39
16	2	2	72938223	8432 * 2	20	24
16	2	4	72938223	8432 * 2	40	48
16	2	6	72938223	8432 * 2	55	67

Table 8.2 provides a qualitative comparison between the code produced by our approach, and the code generated by using the non-decomposed system specification. We compare the number of states composing the synthesized Mealy Machines. Then, for each scenario we simulate the generated code using Gazebo, we monitor their behavior and we quantify it by counting the

ROS messages used to control the system. It is important noticing that in ROS messages are the main software primitive.

Using the non-decomposed specification leads the generated Mealy Machines to grow exponentially. Meanwhile, using the presented approach, *i.e.*, synthesizing multiple Mealy Machines from the multiple contracts composing the decomposed system specification, allows generating smaller Mealy Machines. Thus, the proposed methodology provides a more compact implementation for the same given set of requirements.

Overall, this set of experiments shows that the proposed decomposition strategy allows managing systems otherwise intractable.

Table 8.3: Results obtained by the Gazebo simulation.

Blocks	Robots	Targets	Steps	ROS Messages	Simulation Time (s)
9	2	2	3	19	20.53
16	1	2	7	39	98.21
16	2	2	4	24	34.78
16	2	4	8	48	85.67
16	2	6	11	67	110.24
16	3	6	8	52	75.27
16	4	6	7	47	60.02
16	4	8	10	66	82.86
25	2	4	12	68	104.13
25	2	6	15	87	119.44
25	3	9	17	103	133.26
25	4	12	20	124	94.86
25	5	15	19	125	70.32

Table 8.3 shows that the simulation time using Gazebo is many orders of magnitude higher than the system-level simulation used for validation. This is due to the fact that Gazebo simulates every detail of the systems' physical behavior. Meanwhile, the system-level simulation emulates the details interesting the control strategies of the different robots, while relying on a coarse abstraction of the system's kinematics.

8.2 Virtual Prototyping using Assume-Guarantee Contracts

Table 8.4 reports the time necessary to generate the virtual prototype of the collaborative robotic assembly machine. It reports distinctly the time required

Table 8.4: Time required to perform the consistency checking and synthesis step, and the code generation for the non-decomposed (*i.e.*, holistic) system specification, and for the different actions of the robotic assembling station. The numbered columns refer to the machine's elementary actions: (1) Compose, (2) Decompose, (3) Pick, (4) Place, (5) Move, (6) Turn. The last column reports the time required to obtain the machine coordinator.

	Holistic	**Decomposed System**						
	System	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**	**Coord.**
Consistency Checking & Synthesis (s)	Time Out	0.20	0.20	0.19	0.17	0.13	0.11	0.24
Code generation (s)	-	0.12	0.12	0.08	0.08	0.07	0.06	0.15
Total Time (s)	-	**0.33**	**0.32**	**0.27**	**0.25**	**0.20**	**0.17**	**0.39**

to perform the synthesis from contracts, that also incorporates the time required for the consistency checking, and the time required for the code generation. The *holistic system* column refers to the machine specified by a single A/G contract: it specifies all the operations of a single machine and their coordination. Therefore, it is specified using a different modeling approach. The same reactive synthesis tools and algorithms are used for both the decomposed and the non-decomposed scenarios. The *decomposed system* columns report the synthesis and code generation time required when using the decomposed system specification, as proposed in this work. The non-decomposed (*i.e.*, holistic) system specification leads to complexity issues, as its consistency check and synthesis reaches the time-out we set to six hours: this contract is characterized by a much higher number of LTL properties and, consequently, is harder to consistency check and synthesize. On the other hand, decomposing the system specification into multiple subproblems allows keeping the required time extremely limited.

Table 8.5 reports the results for all the machines in the production line. It is reported both the time required for the synthesis from the A/G contracts and the time required for the code generation. Each column refers to a machine. For instance, Column (4) refers to the robotic assembly station. As such, its values are the sum of the values in Table 5.1. For all the machines, the processing time is minimal. The most time-consuming specification is the Quality Checking cell. This is due to the fact that the cell relies on cameras. Thus, its specification has to model multiple two-dimensional spaces to represent the signals analyzed by the cameras. The last column reports the total synthesis and code-generation time. It shows the efficiency of our methodol-

Table 8.5: Time required to perform the consistency checking and synthesis step for all the machines in the production line. The last column reports the total time required for the entire production line.

	3D Printer	Conveyor Belts	Quality Check	Robotic Assembly	Milling Machine	Total
Consistency Checking & Synthesis (s)	0.25	0.67	2.58	1.24	0.48	**5.22**
Code generation (s)	0.07	0.20	0.76	0.68	0.20	**1.91**
Total Time (s)	0.32	0.87	3.34	1.92	0.68	**6.39**

ogy that allows generating virtual prototypes for production lines from their specifications.

8.3 Application to a Service-Oriented Manufacturing Architecture

Table 8.6: Comparison between the communication delay derived from a direct connection with OPC UA and with the proposed architecture.

Transport Type	Read (s)	Write (s)	Methods (s)	Subscription Update (s)
OPC-UA	0.008	0.009	0.010	0.150
SOA	0.013	0.013	0.014	0.255
Overhead	62.50%	44.45%	40.00%	70.00%

Table 8.6 reports the overhead necessary to call OPC UA functions comparing a direct connection with the machines and through the proposed architecture. We compared the delay of different services, such as read/write of variables, method calls, and subscription to variables. For each operation, the last line reports the additional overhead (in percentage) required when using the proposed architecture. The additional overhead introduces a significant communication delay, as it ranges between 40% and 70%. However, this communication delay is in the context of complex physical processes, such as those involved in a manufacturing line.

Table 8.7 reports the total execution time for three production recipes of different sizes. The table compares the time required using the state-of-the-art architecture with our proposed solution. The total execution times do not

Table 8.7: Comparison between the execution time when using the state-of-the-art and the proposed architecture to govern three different complete production recipes.

Recipe	Tasks	Service Calls	OPC UA Time (s)	SOA Time (s)	SOA Overhead
1	4	54	70.34	70.85	0.72%
2	5	44	66.73	67.04	0.46%
3	11	132	158.83	159.63	0.50%

Table 8.8: Comparison between the scheduling of 450 production recipes, using a classical RTN-based representation against the proposed hierarchical modeling approach.

Param	RTN Representation	Hierarchical Representation	Diff. (%)
Cycle Time	20:56:19 (h)	20:17:23 (h)	-3,1%
Avg. Change Overtime	5:42:47 (h)	5:05:16 (h)	-10,95%
Avg. Utilization	74,47%	80,85%	+6,38%
Throughput	235,96 (u)	245,75 (u)	+4,14%
Avg. Time To Complete HP	1:02:08 (h)	0:57:08 (h)	-8,07%
Avg. Time To Complete LP	2:19:25 (h)	2:16:45 (h)	-1,92%

consider the transportation time required to move materials on the conveyor belts. This is because transportation data is highly variable and influenced by many physical factors that do not depend only on the control software architecture. The fourth and fifth columns of the table report the execution times obtained with the two different configurations. The last column reports the overhead introduced by the proposed architecture. Considering the number of service calls of each recipe we can see that the delay introduced is minimal and consequently negligible from the total execution time.

Table 8.8 compares the results obtained with a traditional scheduler based on RTN and our service-based scheduler, which exploits the proposed hierarchical information model. As a benchmark, we input 450 production processes instances (*i.e.*, production orders), randomly generated from a pool of 4 different recipes. The first row compares the total makespan obtained by the two approaches. The proposed scheduler is able to reduce the total execution time needed to complete the 450 orders by almost 40 minutes. Thus, providing an

improvement of 3.1% with respect to the state-of-the-practice approach. The next three lines show the comparison of the average amount of time in which the machines are not used (*e.g.*, the change time), the average machines utilization, and the throughput. The proposed approach decreases the average change time, while increasing both the average machine utilization and the system throughput. Thus, the proposed approach successfully hits its target of reducing the makespan, while increasing the average machine utilization.

9

Conclusion and suggestions for future research

This chapter concludes this thesis by summarizing the methodologies composing the conceptual design framework MOOD4I. It outlines the contributions of each approach and also suggests future research directions, aiming to further extend the possibilities of the framework.

9.1 Summary of the proposed approach

The *Industry 4.0* is a new industrial automation trend introduced at the Hanover Fair in 2011. It proposes to revolutionize the entire manufacturing world by integrating many new technologies, to massively improve production volumes and quality while creating and optimizing business models to obtain an always-increasing cost efficiency. In this context, Cyber-Physical System (CPS) provides a new degree of controllability to production machines, while supplying extremely precious data about the executed processes. The integration of CPSs in manufacturing systems (*i.e.*, industrial machines), generated a new class of systems, named Cyber-Physical Production Systems (CPPSs). The complexity of designing production lines composed of CPPSs is rising along with new market trends oriented toward product customization and on-demand production. Therefore, this thesis proposes a conceptual design framework, to cope with novel Industry 4.0 paradigms and systems.

The starting point of this thesis is to gather the constitutional set of information about the production plant to be designed. In particular, Modeling, Formalization & Design for Industry (MOOD4I) requires to define the topology of the production line, the production requirements, and the processes that will be applied to the system. Such a set of information can be expressed in different languages (*e.g.*, Automation Markup Language (AML) and System Modeling Language (SysML)) and according to multiple standards (*e.g.*,

International Society of Automation (ISA)-95). Nonetheless, a fundamental feature of a modeling framework is to reuse models already available to the designers, constructed using other languages. Therefore, MOOD4I proposes a mapping from AML and SysML, to reuse AML descriptions to construct equivalent SysML. Furthermore, the part of this thesis related to modeling CPPSs proposes a strategy to hierarchically define a production recipe. Such a modeling strategy is aimed at encapsulating the most complete knowledge of the process, from the business to the automation viewpoints.

The second part of the thesis describes a formalization approach to the design problem. In particular, the formalization exploits the Assume-Guarantee (A/G) reasoning through contracts, to decompose the problem and, therefore, to individually deal with parts of the system. The formalization makes use of both the knowledge of the system gathered from the SysML models and the knowledge about the machines provided by a manufacturing processes taxonomy (*e.g.*, the DIN 8580). The set of formalized contracts decomposing the design problem is used by the framework to achieve two targets: the construction of a virtual prototype that can be simulated to assess the feasibility of a recipe or to generate implementations to be deployed onto a real system. Both targets rely on reactive synthesis algorithms, to obtain implementations from contract-based specifications.

The third part of the thesis deals with the reconfiguration of CPPSs, which is a fundamental task to be carried out by flexible manufacturing systems implementing agile production. The reconfiguration problem originates from a business point of view and actuates on the automation layer. Therefore, to bridge such viewpoints, the proposed architecture is centered on the concept of "production services". Each service implements a precise functionality at the control level and it is also visible from a business viewpoint. It contributes to the overall production process in a specific manner. Therefore, a key role in the architecture is played by the Automation Manager component, which is the software "glue" between the MES and the PLCs composing the system. To fully exploit the potential of such an architecture and the fact that MOOD4I proposes an in-depth production recipe modeling strategy, the final part of the thesis also defines an advanced scheduling algorithm. Such an algorithm is executed at runtime and reacts to events that may happen during the production (*e.g.*, new orders, or machine stops). It handles such events by proposing an interruption and interleaving strategy of processes, exploiting the knowledge of the process provided in the SysML model. The last part of the thesis can, therefore, be considered as a platform on which the design framework (and, thus, the model-based techniques proposed) can be applied. Assuming the functionality of components is implemented through services, the MOOD4I

framework is capable of modeling, prototyping, and design implementations while guaranteeing their correctness.

The last part of the thesis takes a look beyond Industry 4.0, where machines and humans will collaborate to carry out even more complex production tasks. In fact, Industry 5.0 suggests putting the human back at the center of manufacturing, to exploit its intuitions and ingenuity for a more efficient and resilient manufacturing. Therefore, this part of the thesis carries out an speculative analysis on potential limitations of actual design languages and methodologies, to understand Industry 5.0 requirements and outline research directions to cope with such demands.

9.2 Directions for future research

The directions for future works may be a consequence of the limitations of the current framework or may arise from the desire to improve and explore new paths. The list of the following ideas comprises both categories, in which some are already under study:

- The formalization process of A/G contracts specified in both Section 4.3 and Section 5.3 is not automated. To achieve such a goal, the set of SysML diagrams must be mapped to a formalization framework that is built around A/G contracts. In this regard, the Contract-based Heterogeneous Analysis and System Exploration (CHASE) framework [69] would perfectly suit such a task: it provides a library of constructs that, on the one hand, can be related to SysML diagrams elements and, on the other hand, have a direct mapping to logic formulas. Therefore, by exploiting such a framework, MOOD4I could automatically produce the set of A/G contracts that constitutes the specifications of the design problem.
- As Chapter 7 defined, Industry 5.0 systems have an additional set of requirements to be compliant with. Other than refining modeling languages (*e.g.*, SysML) for Industry 5.0, the architecture and the scheduling algorithm proposed in Chapter 6 must be improved. As an example, the uncertainty of human behaviors requires scheduling algorithms that must be able to handle and react to such a class of unexpected events. A key role in this context may also be played by the models: categorizing the events caused by human behaviors and understanding them is the first task for the system to correctly react and reconfigure.
- The virtual prototyping and compositional design methodologies presented in Chapter 4 and Chapter 5 rely on simulation to verify the correctness of the system. Whether the simulation may fail, the designer has no clue which part of the specification is not correct. Contrariwise, formal

verification techniques (*e.g.*, model checking) can provide a counterexample as proof of property violation. In this context, Counterexample-guided Abstraction Refinement (CEGAR) methodologies provide a way to refine the specifications based on such proof. CEGAR techniques, nonetheless, have a common problem: the interpretation of the counterexample is hard and, therefore, it is difficult to correctly understand the incorrect part of the specification to refine. In this regard, a possible future work could integrate CEGAR techniques in MOOD4I and exploit SysML, to construct diagrams implementing the counterexample.

Summary of the proposed innovative contributions

This chapter reports the innovative contributions to the *State of the Art* by the work proposed in this thesis. The articles are grouped into four main groups:

- the first group is the *cyber-physical production systems modeling*, which includes methodologies and concepts from Chapter 3 and Chapter 7;
- the second group includes articles related to *compositional design using assume-guarantee contracts*, which refers to the methodology shown in Chapter 4;
- the third group comprises articles related to *virtual prototypization using assume-guarantee contracts*, which has been described in Chapter 5;
- and the final group consists of methodologies related to *model-based and service-oriented advanced scheduling*, illustrated in Chapter 6.

The contributions inside each group are chronologically ordered.

Cyber-physical production systems modeling

1. **S. Spellini**, S. Gaiardelli, M. Lora and F. Fummi, "*Enabling Component Reuse in Model-based System Engineering of Cyber-Physical Production Systems*" in 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2021, pp. 1-8, doi: 10.1109/ETFA45728.2021.9613572.
2. S. Gaiardelli, **S. Spellini**, M. Lora and F. Fummi, "*Modeling in Industry 5.0: What Is There and What Is Missing: Special Session 1: Languages for Industry 5.0*" in Forum on specification & Design Languages (FDL), 2021, pp. 01-08, doi: 10.1109/FDL53530.2021.9568371.

Compositional design using assume-guarantee contracts

1. **S. Spellini**, M. Lora, S. Chattopadhyay and F. Fummi, "Work-in-Progress: Introducing Assume-Guarantee Contracts for Verifying Robotic Applications" in International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2018, pp. 1-2, doi: 10.1109/CODESISSS.2018.8525885.
2. **S. Spellini**, M. Lora, F. Fummi and S. Chattopadhyay, "*Compositional Design of Multi-Robot Systems Control Software on ROS*" in ACM Trans. Embed. Comput. Syst. 18, 5s, Article 71 (October 2019), 24 pages

Virtual prototypization using assume-guarantee contracts

1. R. Chirico, **S. Spellini**, M. Panato, M. Lora and F. Fummi, "*A Contract-based Methodology for Production Lines Validation*" in IEEE 17th International Conference on Industrial Informatics (INDIN), 2019, pp. 695-698, doi: 10.1109/INDIN41052.2019.8972100.
2. **S. Spellini**, R. Chirico, M. Lora and F. Fummi, "*Languages and Formalisms to Enable EDA Techniques in the Context of Industry 4.0*" in Forum for Specification and Design Languages (FDL), 2019, pp. 1-4, doi: 10.1109/FDL.2019.8876899.
3. **S. Spellini**, R. Chirico, M. Panato, M. Lora and F. Fummi, "*Production Recipe Validation through Formalization and Digital Twin Generation*" in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2020, pp. 1698-1703, doi: 10.23919/DATE48585.2020.9116343
4. **S. Spellini**, R. Chirico, M. Panato, M. Lora and F. Fummi, "*Virtual Prototyping a Production Line Using Assume-Guarantee Contracts*" in IEEE Transactions on Industrial Informatics, vol. 17, no. 9, pp. 6294-6302, Sept. 2021, doi: 10.1109/TII.2020.3038679.

Model-based and service-oriented advanced scheduling

1. S. Gaiardelli, **S. Spellini**, M. Panato, M. Lora and F. Fummi, "*A Software Architecture to Control Service-Oriented Manufacturing Systems*" in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2022, pp. 40-43, doi: 10.23919/DATE54114.2022.9774522
2. S. Gaiardelli, **S. Spellini**, M. Lora and F. Fummi, "*A Hierarchical Modeling Approach to Improve Scheduling of Manufacturing Processes*", in IEEE International Symposium on Industrial Electronics, 2022, pp. TBA, doi: TBA

List of Acronyms

A/G	Assume-Guarantee
AGV	Automated Guided Vehicle
AML	Automation Markup Language
AMQP	Advanced Message Queuing Protocol
B2MML	Business To Manufacturing Markup Language
BDD	Block Definition Diagram
CAEX	Computer Aided Engineering Exchange
CEGAR	Counterexample-guided Abstraction Refinement
CHASE	Contract-based Heterogeneous Analysis and System Exploration
COLLADA	COLLAborative Design Activity
CPPS	Cyber-Physical Production System
CPS	Cyber-Physical System
DFJSS	Dynamic Flexible Job Shop Scheduling
DIN	Deutsches Institut für Normung
DRC	DARPA Robotic Challenge
ERP	Enterprise Resource Planning
FJSS	Flexible Job Shop Scheduling
FSM	Finite State Machine
GR(1)	General Reactivity of rank 1
HMI	Human-Machine Interaction
HRC	Human-Robot Collaboration
IBD	Internal Block Diagram
ICE	Industrial Computer Engineering
IIoT	Industrial Internet of Things
IoT	Internet of Things
ISA	International Society of Automation
JSS	Job Shop Scheduling

LTL	Linear Temporal Logic
M2M	Machine to Machine
MBSE	Model-based System Engineering
MES	Manufacturing Execution System
MILP	Mixed Integer Linear Programming
MOM	Manufacturing Operations Management
MOOD4I	Modeling, Formalization & Design for Industry
MU	Mobile Unit
OPC UA	OPC Unified Architecture
OWL	Web Ontology Language
PBD	Platform-Based Design
PLC	Programmable Logic Controller
PPE	Personal Protective Equipment
QC	Quality Checking
ROS	Robot Operating System
RPC	Remote Procedure Call
RTN	Resource Task Network
SCADA	Supervisory Control and Data Acquisition
SME	Small and Medium Enterprise
SOA	Service Oriented Architecture
SOM	Service Oriented Manufacturing
STN	State Task Network
STOMP	Streaming Text Oriented Messaging Protocol
SWRL	Semantic Web Rule Language
SysML	System Modeling Language
UML	Unified Modeling Language
XMI	XML Metadata Interchange

References

1. M. S. Kumar, R. D. Raut, V. S. Narwane, and B. E. Narkhede, "Applications of industry 4.0 to overcome the covid-19 operational challenges," *Diabetes & Metabolic Syndrome: Clinical Research & Reviews*, vol. 14, no. 5, pp. 1283–1289, 2020.
2. H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," *Business & information systems engineering*, vol. 6, no. 4, pp. 239–242, 2014.
3. R. Drath and A. Horch, "Industrie 4.0: Hit or hype? [industry forum]," *IEEE Industrial Electronics Magazine*, vol. 8, no. 2, pp. 56–58, 2014.
4. Y. Koren, U. Heisel, F. Jovane, T. Moriwaki, G. Pritschow, G. Ulsoy, and H. Van Brussel, "Reconfigurable manufacturing systems," *CIRP Annals*, vol. 48, no. 2, pp. 527–540, 1999.
5. L. Ribeiro, "Cyber-physical production systems' design challenges," in *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, 2017, pp. 1189–1194.
6. AutomationML Consortium, "AutomationML," 2006. [Online]. Available: https://www.automationml.org/o.red.c/home.html
7. International Society of Automation, "ISA-95 Standard," 2000. [Online]. Available: https://www.isa.org/
8. R. Drath, "Let's talk AutomationML what is the effort of AutomationML programming?" in *Proc. of IEEE ETFA*, Sep. 2012, pp. 1–8.
9. N. Schmidt and A. Lüder, "AutomationML in a Nutshell," *AutomationML eV*, 2015.
10. German Institute for Standards, "Din standard 8580," 2003. [Online]. Available: https://www.din.de/en
11. D. Brandl, "What is ISA-95? Industrial Best Practices of Manufacturing Information Technologies with ISA-95 Models," 5 2008. [Online]. Available: http://www.apsom.org/docs/T061_isa95-04.pdf
12. B. Vogel-Heuser, D. Schütz, T. Frank, and C. Legat, "Model-driven engineering of manufacturing automation software projects - a sysml-based approach," *Mechatronics*, vol. 24, no. 7, pp. 883–897, 2014, 1. Model-Based Mechatronic System Design 2. Model Based Engineering.

13. L. Piétrac, A. Lelevé, and S. Henry, "On the use of sysml for manufacturing execution system design," in *ETFA2011*, 2011, pp. 1–8.
14. R. Baduel, M. Chami, J.-M. Bruel, and I. Ober, "SysML Models Verification and Validation in an Industrial Context: Challenges and Experimentation," in *European Conference on Modelling Foundations and Applications*. Springer, 2018, pp. 132–146.
15. M. Debbabi, F. Hassaine, Y. Jarraya, A. Soeanu, and L. Alawneh, *Verification and validation in systems engineering: assessing UML/SysML design models*. Springer Science & Business Media, 2010.
16. K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523–1543, 2000.
17. M. Mura, L. G. Murillo, and M. Prevostini, "Model-based design space exploration for RTES with SysML and MARTE," in *Proc. of IEEE/ECSI FDL*, 2008, pp. 203–208.
18. P. Leserf, P. de Saqui-Sannes, and J. Hugues, "Multi domain optimization with SysML modeling," in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 2015, pp. 1–8.
19. S. Kanthabhabhajeya and B. Lennartson, "Modeling and optimization of synchronous behavior for packaging machines," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 1684–1691, 2014.
20. L. Piétrac, A. Lelevé, and S. Henry, "On the use of sysml for manufacturing execution system design," in *ETFA2011*, 2011, pp. 1–8.
21. M. Jamro and B. Trybus, "An approach to SysML modeling of IEC 61131-3 control software," in *2013 18th International Conference on Methods Models in Automation Robotics (MMAR)*, 2013, pp. 217–222.
22. A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. A. Henzinger, and K. G. Larsen, "Contracts for system design," *Foundations and Trends® in Electronic Design Automation*, vol. 12, no. 2-3, pp. 124–400, 2018.
23. P. Nuzzo, A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli, "Are interface theories equivalent to contract theories?" in *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Oct 2014, pp. 104–113.
24. P. Nuzzo, A. L. Sangiovanni-Vincentelli, D. Bresolin, L. Geretti, and T. Villa, "A platform-based design methodology with contracts and related tools for the design of cyber-physical systems," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2104–2132, 2015.
25. R. Rosner, "Modular Synthesis of Reactive Systems," *PhD thesis. The Weizmann Institute of Science*, 1991.
26. A. P. Nir Piterman and Y. Saar, "Synthesis of Reactive(1) Designs," in *Proceedings of VMCAI 2006*, 2006.
27. I. Filippidis, S. Dathathri, S. C. Livingston, N. Ozay, and R. M. Murray, "Control Design for Hybrid Systems with TuLiP: The Temporal Logic Planning Toolbox,"

in *Proceedings of IEEE Conference on Control Applications (CCA) 2016*, 2016, pp. 1030–1041.
28. R. Ehlers and V. Raman, "Slugs: Extensible GR(1) Synthesis," in *Proceedings of International Conference on Computer Aided Verification (CAV)*. Springer, 2016, pp. 333–339.
29. Object Management Group (OMG), *OMG Systems Modeling Language (OMG SysML), Version 1.6*, Object Management Group Std., 2019. [Online]. Available: http://www.omg.org/spec/SysML/1.6/
30. E. Korshunova, M. Petkovic, M. van den Brand, and M. Mousavi, "CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code," in *2006 13th Working Conference on Reverse Engineering*, 2006, pp. 297–298.
31. N. Bombieri, E. Ebeid, F. Fummi, and M. Lora, "On the Reuse of Heterogeneous IPs into SysML Models for Integration Validation," *J. Electron. Test.*, vol. 29, no. 5, pp. 647–667, oct 2013.
32. E. Ebeid, F. Fummi, and D. Quaglia, "Model-Driven Design of Network Aspects of Distributed Embedded Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 4, pp. 603–614, 2015.
33. T. Lojka, M. Bundzel, and I. Zolotová, "Service-oriented architecture and cloud manufacturing," *Acta polytechnica hungarica*, vol. 13, no. 6, pp. 25–44, 2016.
34. P. Danny, P. Ferreira, N. Lohse, and M. Guedes, "An automationml model for plug-and-produce assembly systems," in *Proc. of IEEE INDIN*, 2017, pp. 849–854.
35. K. Thramboulidis and F. Christoulakis, "UML4IoT-A UML-based approach to exploit IoT in cyber-physical manufacturing systems," *Computers in Industry*, vol. 82, pp. 259–272, 2016.
36. T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220, 1993.
37. S. Jaskó, A. Skrop, T. Holczinger, T. Chován, and J. Abonyi, "Development of manufacturing execution systems in accordance with industry 4.0 requirements: A review of standard- and ontology-based methodologies and tools," *Computers in Industry*, vol. 123, p. 103300, 2020.
38. O. Harcuba and P. Vrba, "Ontologies for flexible production systems," in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2015, pp. 1–8.
39. E. Negri, L. Fumagalli, M. Garetti, and L. Tanca, "Requirements and languages for the semantic representation of manufacturing systems," *Computers in Industry*, vol. 81, pp. 55–66, 2016.
40. C. Hildebrandt, M. Glawe, A. W. Müller, and A. Fay, "Reasoning on engineering knowledge: Applications and desired features," in *European Semantic Web Conference*. Springer, 2017, pp. 65–78.
41. S. Feldmann, K. Kernschmidt, and B. Vogel-Heuser, "Combining a sysml-based modeling approach and semantic technologies for analyzing change influences in manufacturing plant models," *Procedia CIRP*, vol. 17, pp. 451–456, 2014, variety Management in Manufacturing.

42. A. Wortmann, O. Barais, B. Combemale, and M. Wimmer, "Modeling languages in Industry 4.0: an extended systematic mapping study," *Software and Systems Modeling*, vol. 19, no. 1, pp. 67–94, 2020.
43. L. Berardinelli, S. Biffl, A. Lüder, E. Mätzler, T. Mayerhofer, M. Wimmer, and S. Wolny, "Cross-disciplinary engineering with AutomationML and SysML," *at - Automatisierungstechnik*, vol. 64, pp. 253 – 269, 2016.
44. C. C. Pantelides, "Unified frameworks for optimal process planning and scheduling," in *Proceedings on the second conference on foundations of computer aided operations*, 1994, pp. 253–274.
45. E. Kondili, C. Pantelides, and R. Sargent, "A general algorithm for short-term scheduling of batch operations—i. milp formulation," *Computers & Chemical Engineering*, vol. 17, no. 2, pp. 211–227, 1993.
46. I. Harjunkoski and R. Bauer, "Sharing Data for Production Scheduling Using the ISA-95 Standard," *Frontiers in Energy Research*, vol. 2, 10 2014.
47. The Khronos Group Inc., Sony Computer Entertainment Inc., "COLLADA 1.5 – Digital Asset Schema Release Specification," 2008. [Online]. Available: https://www.khronos.org/collada/
48. M. Lora, S. Vinco, and F. Fummi, "Translation, Abstraction and Integration for Effective Smart System Design," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1525–1538, 2019.
49. D. Bresolin, L. Di Guglielmo, L. Geretti, R. Muradore, P. Fiorini, and T. Villa, "Open problems in verification and refinement of autonomous robotic systems," in *Proceedings of Euromicro Digital System Design (DSD) 2012*. IEEE, 2012, pp. 469–476.
50. H. Kress-Gazit, "Robot challenges: Toward development of verification and synthesis techniques," *IEEE Robotics & Automation Magazine*, vol. 18, no. 3, pp. 22–23, 2011.
51. K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 19, no. 12, pp. 1523–1543, 2000.
52. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.
53. P. Iñigo-Blasco, F. Diaz-del Rio, M. C. Romero-Ternero, D. Cagigas-Muñiz, and S. Vicente-Diaz, "Robotics software frameworks for multi-agent robotic systems development," *Robotics and Autonomous Systems*, vol. 60, no. 6, pp. 803–821, jun 2012.
54. H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3. IEEE, 2001, pp. 2523–2528.
55. H. Abbas, I. Saha, Y. Shoukry, R. Ehlers, G. Fainekos, R. Gupta, R. Majumdar, and D. Ulus, "Special Session: Embedded Software for Robotics: Challenges and

Future Directions," in *Proceedings of the International Conference on Embedded Software, (EMSOFT) 2018*, 2018.

56. S. Spellini, M. Lora, S. Chattopadhyay, and F. Fummi, "Work-in-progress: Introducing assume-guarantee contracts for verifying robotic applications," in *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2018, pp. 1–2.
57. P. Nuzzo and A. L. Sangiovanni-Vincentelli, "Hierarchical system design with vertical contracts," in *Principles of Modeling*. Springer, 2018, pp. 360–382.
58. S. Maniatopoulos, P. Schillinger, V. Pong, D. C. Conner, and H. Kress-Gazit, "Reactive high-level behavior synthesis for an atlas humanoid robot," in *IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 4192–4199.
59. J. W. McDonald Hayhurst and D. C. Conner, "Towards capability-based synthesis of executable robot behaviors," in *SoutheastCon 2018*, April 2018, pp. 1–8.
60. K. W. Wong and H. Kress-Gazit, "From high-level task specification to robot operating system (ros) implementation," in *2017 First IEEE International Conference on Robotic Computing (IRC)*, April 2017, pp. 188–195.
61. C. Finucane and H. Kress-Gazit, "Ltlmop: Experimenting with language, temporal logic and robot control," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2010, pp. 1988–1993.
62. H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE transactions on robotics*, vol. 25, no. 6, pp. 1370–1381, 2009.
63. R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Saar, "Synthesis of reactive (1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911–938, 2012.
64. Open Source Robotics Foundation (OSRF). (2018) Gazebo simulator. [Online]. Available: https://www.openrobotics.org/
65. A. Desai, I. Saha, J. Yang, S. Qadeer, and S. A. Seshia, "Drona: A Framework for Safe Distributed Mobile Robotics," in *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2017, pp. 239–248.
66. R. Passerone, I. B. Hafaiedh, S. Graf, A. Benveniste, D. Cancila, A. Cuccuru, S. Gerard, F. Terrier, W. Damm, A. Ferrari *et al.*, "Metamodels in europe: Languages, tools, and applications," *IEEE Design & Test of Computers*, vol. 26, no. 3, pp. 38–53, 2009.
67. Y. A. Feldman and H. Broodney, "A cognitive journey for requirements engineering," in *INCOSE International Symposium*, vol. 26, no. 1. Wiley Online Library, 2016, pp. 430–444.
68. P. Nuzzo, A. L. Sangiovanni-Vincentelli, and R. M. Murray, "Methodology and tools for next generation cyber-physical systems: The icyphy approach," in *INCOSE International Symposium*, vol. 25, no. 1. Wiley Online Library, 2015, pp. 235–249.
69. P. Nuzzo, M. Lora, Y. A. Feldman, and A. Sangiovanni-Vincentelli, "CHASE: Contract-based requirement engineering for cyber-physical system design," in

Proceedings of IEEE/ACM Design Automation and Test in Europe (DATE) 2018, 2018, pp. 839–844.

70. N. Bajaj, P. Nuzzo, M. Masin, and A. Sangiovanni-Vincentelli, "Optimized selection of reliable and cost-effective cyber-physical system architectures," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 561–566.
71. M. Lora, S. Vinco, and F. Fummi, "Translation, abstraction and integration for effective smart system design," *IEEE Transactions on Computers*, 2019.
72. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM (JACM)*, vol. 50, no. 5, pp. 752–794, 2003.
73. D. Kroening, A. Groce, and E. Clarke, "Counterexample guided abstraction refinement via program execution," in *International Conference on Formal Engineering Methods*. Springer, 2004, pp. 224–238.
74. A. Elfes, "Using occupancy grids for mobile robot perception and navigation," *Computer*, no. 6, pp. 46–57, 1989.
75. F. Fummi, M. Lora, F. Stefanni, D. Trachanis, J. Vanhese, and S. Vinco, "Moving from co-simulation to simulation for effective smart systems design," in *Proceedings of IEEE/ACM Design Automation and Test in Europe (DATE) 2014*, 2014, p. 286.
76. B. P. Gerkey and M. J. Matarić, "A formal analysis and taxonomy of task allocation in multi-robot systems," *The International Journal of Robotics Research*, vol. 23, no. 9, pp. 939–954, 2004.
77. G. G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, p. 279, 1997.
78. N. Bombieri, G. Di Guglielmo, M. Ferrari, F. Fummi, G. Pravadelli, F. Stefanni, and A. Venturelli, "HIFSuite: Tools for HDL code conversion and manipulation," *EURASIP Journal on Embedded Systems*, vol. 2010, no. 1, p. 436328, 2010.
79. N. Koenig and A. Howard, "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robotics and Systems (IROS)*, vol. 3. IEEE, 2004, pp. 2149–2154.
80. Robotis, "Turtlebot3," 2017. [Online]. Available: http://www.robotis.us/turtlebot-3/
81. S. Spellini, M. Lora, F. Fummi, and S. Chattopadhyay, "Compositional design of multi-robot systems control software on ros," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, oct 2019.
82. Siemens, "Tecnomatix Plant Simulation," 2017.
83. D. Mourtzis, M. Doukas, and D. Bernidaki, "Simulation in manufacturing: Review and challenges," *Procedia CIRP*, vol. 25, pp. 213–229, 2014.
84. M. Lora, S. Vinco, and F. Fummi, "Translation, abstraction and integration for effective smart system design," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1525–1538, Oct 2019.
85. J. Banks, J. Carson, B. L. Nelson, and D. Nicol, *Discrete-event system simulation*. Pearson, 2005.

86. L. M. S. Dias, A. A. C. Vieira, G. A. B. Pereira, and J. A. Oliveira, "Discrete simulation software ranking — a top list of the worldwide most popular and used tools," in *2016 Winter Simulation Conference (WSC)*, 2016, pp. 1060–1071.
87. L. Büth, N. Broderius, C. Herrmann, and S. Thiede, "Introducing agent-based simulation of manufacturing systems to industrial discrete-event simulation tools," in *Proc. of IEEE INDIN*, July 2017, pp. 1141–1146.
88. "Simulation software survey," 2017. [Online]. Available: https://www.informs.org/ORMS-Today/OR-MS-Today-Software-Surveys/Simulation-Software-Survey
89. M. V. Moreira and J.-J. Lesage, "Fault diagnosis based on identified discrete-event models," *Control Engineering Practice*, vol. 91, 2019.
90. S. Preuße and H. Hanisch, "Verifying functional and non-functional properties of manufacturing control systems," in *Proc. of the 3rd IFAC DCDS*, 2011, pp. 41–46.
91. O. Ljungkrantz, K. Akesson, M. Fabian, and C. Yuan, "Formal specification and verification of industrial control logic components," *IEEE Transactions on Automation Science and Engineering*, vol. 7, no. 3, pp. 538–548, July 2010.
92. R. Savolainen, S. Sierla, T. Karhela, T. Miettinen, and V. Vyatkin, "A framework for runtime verification of industrial process control systems," in *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*. IEEE, 2017, pp. 687–694.
93. A. Yacoub, M. Hamri, and C. Frydman, "A method for improving the verification and validation of systems by the combined use of simulation and formal methods," in *2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*, Oct 2014, pp. 155–162.
94. H. A. ElMaraghy, "Flexible and reconfigurable manufacturing systems paradigms," *International journal of flexible manufacturing systems*, vol. 17, no. 4, pp. 261–276, 2005.
95. K. Tiwari and M. S. Khan, "Sustainability accounting and reporting in the industry 4.0," *Journal of Cleaner Production*, vol. 258, 2020.
96. "OPC Unified Architecture specification - Part 1: Overview and concepts release 1.04 OPC Foundation," 2017.
97. M. V. García, E. Irisarri, F. Pérez, M. Marcos, and E. Estevez, "From isa 88/95 meta-models to an opc ua-based development tool for cpps under iec 61499," in *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, 2018, pp. 1–9.
98. J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, 2011.
99. P. Dobbelaere and K. S. Esmaili, "Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper," in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*. Association for Computing Machinery, 2017, pp. 227–238.
100. "Rabbitmq." [Online]. Available: https://www.rabbitmq.com/
101. J. Błażewicz, W. Domschke, and E. Pesch, "The job shop scheduling problem: Conventional and new solution techniques," *European Journal of Operational Research*, vol. 93, no. 1, pp. 1–33, 1996.

102. M. Dorigo and C. Blum, "Ant colony optimization theory: A survey," *Theoretical Computer Science*, vol. 344, no. 2, pp. 243–278, 2005.
103. R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization: An overview," *Swarm Intelligence*, vol. 1, 10 2007.
104. D. Karaboga and B. Basturk, "A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (abc) algorithm," *Journal of Global Optimization*, vol. 39, pp. 459–471, 11 2007.
105. M. Affenzeller, S. Winkler, S. Wagner, and A. Beham, *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications*. Chapman and Hall/CRC, 03 2009.
106. J. Heizer, *Production and Operations Management*. Allyn and Macon, Needham Heights, Massachusetts, 1991.
107. s. Nahmias, *Production and operations analysis*. McGraw-Hill,, 2015.
108. H. Askari Nasab, Y. Pourrahimian, E. Ben-Awuah, and S. Kalantari, "Mixed integer linear programming formulations for open pit production scheduling," *Journal of Mining Science*, vol. 47, pp. 338–359, 05 2011.
109. Y. Krotov, "Csp production planning system," *AISTech - Iron and Steel Technology Conference Proceedings*, vol. 2, pp. 767–772, 01 2006.
110. I. A. Chaudhry and A. A. Khan, "A research survey: review of flexible job shop scheduling techniques," *International Transactions in Operational Research*, vol. 23, no. 3, pp. 551–591, 2016.
111. O. Cardin, D. Trentesaux, A. Thomas, P. Castagna, T. Berger, and H. B. El-Haouzi, "Coupling predictive scheduling and reactive control in manufacturing hybrid control architectures: state of the art and future challenges," *Journal of Intelligent Manufacturing*, vol. 28, no. 7, pp. 1503–1517, 2017.
112. M. Hermann, T. Pentek, and B. Otto, "Design principles for industrie 4.0 scenarios," in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, 2016, pp. 3928–3937.
113. S. Nahavandi, "Industry 5.0—a human-centric solution," *Sustainability*, vol. 11, no. 16, p. 4371, 2019.
114. C. Raymond and D. Prun, "Extending mbse methodology and sysml formalism to integrate human considerations," in *Proc. of HCI-Aero*, 2016, pp. 1–4.
115. Z. Liu and J. Wang, "Human-cyber-physical systems: concepts, challenges, and research opportunities," *Frontiers Inf. Technol. Electron. Eng.*, vol. 21, pp. 1535–1553, 2020.
116. Y. Shoukry, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada, "Smc: Satisfiability modulo convex programming," *Proceedings of the IEEE*, vol. 106, no. 9, pp. 1655–1679, 2018.
117. P. Nuzzo, J. Li, A. L. Sangiovanni-Vincentelli, Y. Xi, and D. Li, "Stochastic assume-guarantee contracts for cyber-physical system design," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 1, pp. 1–26, 2019.
118. C. Oh, E. Kang, S. Shiraishi, and P. Nuzzo, "Optimizing assume-guarantee contracts for cyber-physical system design," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 246–251.

119. M. Chen, S. Nikolaidis, H. Soh, D. Hsu, and S. Srinivasa, "Trust-aware decision making for human-robot collaboration: Model learning and planning," *ACM Transactions on Human-Robot Interaction (THRI)*, vol. 9, no. 2, pp. 1–23, 2020.
120. L. Roveda, J. Maskani, P. Franceschi, A. Abdi, F. Braghin, L. Molinari Tosatti, and N. Pedrocchi, "Model-based reinforcement learning variable impedance control for human-robot collaboration," *Journal of Intelligent & Robotic Systems*, vol. 100, no. 2, pp. 417–433, 2020.
121. P. A. Lasota, T. Fong, and J. A. Shah, *A survey of methods for safe human-robot interaction*. Now Publishers Delft, The Netherlands, 2017, vol. 104.
122. A. Platzer, *Logical analysis of hybrid systems: proving theorems for complex dynamics*. Springer Science & Business Media, 2010.
123. A. Pinto and A. L. Sangiovanni Vincentelli, "Csl4p: A contract specification language for platforms," *Systems Engineering*, vol. 20, no. 3, pp. 220–234, 2017.

A

Robotic design code

The integration into ROS of a code implementation requires to create a ROS node for each component of the defined system (*i.e.*, a robot). Listing A.2 depicts the structure of a generic ROS node. The set of control strategies generated and validated for each robot in the system, as described in Section 4.3, is the starting point to create multiple ROS nodes, each enclosing the control strategy of one component. The structure of such a control strategy is depicted in the listing A.1. In particular, it outlines the code implementation of the FSM that realizes the control strategy for such an agent.

Listing A.1: Code structure of the synthesized control software. The control logic is implemented by the *executeMachine* function, that models the behavior of an FSM.

```
1     String state;
2     int32_t steps_out;
3     int32_t target_out;
4
5     void Robot::executeMachine( bool up_in, bool down_in, bool left_in,
6         bool right_in, int32_t target_in, int32_t pos_in )
7     {
8         if (state == std::string("0"))
9         {
10            if (up_in == false && down_in == false && left_in == false
11                && right_in == false && target_in == 0L && pos_in == 0L)
12            {
13                state = "20";
14                steps_out = 1L;
15                command_out = 0L;
16            }
17            else if (up_in == true && down_in == false && left_in == false
18                && right_in == false && target_in == 0L && pos_in == 0L)
19            {
20                state = "28";
21                steps_out = 1L;
22                command_out = 0L;
23            }
24            ...
25        else if(state == std::string("1"))
26        {
27            ...
28        }
29        ...
30    }
```

Listing A.2: Code structure of a generic ROS node.

```
/// @brief Code structure representing a generic ROS node.
Input inputs;

void callback(const messages::input &message) {
    inputs = *message;
}

int main(int argc, char** argv) {
    init(argc, argv, "controller");
    NodeHandle nh("~");
    Rate r(1);

    Controller implementation_ctrl;

    Publisher p = nh.advertise<Output>("topic", 1);
    Subscriber s = nh.subscribe<Input>
    ("topic",1, &callback);

    Output outputs;

    while (ros::ok()) {
        implementation_ctrl.executeMachine(input);
    outputs = implementation_ctrl.output
        p.publish(outputs);
        spinOnce();
        r.sleep();
    }

    shutdown();
    return 0;
}
```

B

Contract-based Specifications

The set of elementary actions identified for the manipulator arm is formalized in a set of A/G contracts. Table B.1 reports the specification of the *move* action, which represents the movement between two positions in space of a piece that has been picked by the manipulator. Table B.2 reports the A/G contract related to the *pick* action, which controls the picking of an object with the manipulator clamp. Similarly, Table B.3 illustrates the contract that specifies the placing of an object in the range of action of the arm.

Table B.1: Sketch of the *move* action contract.

	Operation C_{move}
Contract	$C_{move} = (V_{move}, A_{move}, G_{move})$
Variables	$V_{move} = \{command, move\_executed, xpos, ypos, xtarget,$ $ytarget, grip, moving\}$
Assumptions	$A_{move} = \{\Box(command = move \rightarrow grip = true),$ $\Box\Diamond(command = move)\}$
Guarantees	$G_{move} = \{(\Box(command = move \leftrightarrow \bigcirc(moving))),$ $(\Box(moving \wedge (xpos < xtarget) \rightarrow \bigcirc(xpos = xpos + step))),$ $(\Box(moving \wedge (ypos < ytarget) \rightarrow \bigcirc(ypos = ypos + step))),$ $(\Box(moving \wedge (xpos > xtarget) \rightarrow \bigcirc(xpos = xpos - step))),$ $(\Box(moving \wedge (ypos > ytarget) \rightarrow \bigcirc(ypos = ypos - step))),$ $(\Box((xpos = xtarget \wedge ypos = ytarget) \rightarrow \bigcirc(move\_executed)))\}$

Table B.2: Sketch of the *pick* action contract.

	Operation C_{pick}
Contract	$C_{pick} = (V_{pick}, A_{pick}, G_{pick})$
Variables	$V_{pick} = \{command, pick\_executed, grip, picking, pos\}$
Assumptions	$A_{pick} = \{\neg(grip), \square\lozenge(command = pick)\}$
Guarantees	$G_{pick} = \{(\square(command = pick \wedge pos \rightarrow \bigcirc(picking))),$ $(\square(picking \rightarrow \bigcirc(pick\_executed \wedge grip))),$ $(\square\lozenge(pick\_executed \wedge grip))\}$

Table B.3: Sketch of the *place* action contract.

	Operation C_{place}
Contract	$C_{place} = (V_{place}, A_{place}, G_{place})$
Variables	$V_{place} = \{command, place\_executed, grip, placing, pos\}$
Assumptions	$A_{place} = \{\neg(grip), \square\lozenge(command = place)\}$
Guarantees	$G_{place} = \{(\square(command = place \wedge pos \rightarrow \bigcirc(placing))),$ $(\square(placing \rightarrow \bigcirc(place\_executed \wedge \neg grip))),$ $(\square\lozenge(place\_executed \wedge \neg grip))\}$