IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 14 March 2024, accepted 30 March 2024, date of publication 8 April 2024, date of current version 24 June 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3385634

RESEARCH ARTICLE

Enabling Service-Oriented Manufacturing Through Architectures, Models, and Protocols

SEBASTIANO GAIARDELLI[1], (Student Member, IEEE), STEFANO SPELLINI[2], MARCO PANATO[2], CARLO TADIELLO[2], MICHELE LORA[1], (Member, IEEE), DONG SEON CHENG[1], AND FRANCO FUMMI[1], (Member, IEEE)

[1]Department of Engineering for Innovation Medicine (DIMI), University of Verona, 37134 Verona, Italy
[2]FACTORYAL S.r.l., 37057 San Giovanni Lupatoto, Italy

Corresponding author: Sebastiano Gaiardelli (sebastiano.gaiardelli@univr.it)

This work was supported in part by European Union's Horizon 2020 Research and Innovation Program through the Marie Skłodowska-Curie under Grant 894237, and in part by European Union Next-GenerationEU (Piano Nazionale di Ripresa e Resilienza (PNRR)—Missione 4, Componente 2, and Investimento 1.5—D.D. 1058 23/06/2022) through the PNRR Research Activities of the Consortium Interconnected North-East Innovation Ecosystem (iNEST) under Grant ECS_00000043.

ABSTRACT Modern production lines are often composed of machinery from different vendors that must be connected with each other to increase the overall interoperability and flexibility. A common problem that arises in such systems is the complexity of the configuration task: they usually require each component to be manually configured. Thus, machinery requires different configuration strategies, negatively impacting scalability and increasing the chance of human errors. Furthermore, each time the manufacturing system has to be updated, the entire procedure must be repeated. This paper proposes a software architecture abstracting the complexity of existing production lines by enabling the service-oriented manufacturing paradigm. Then, it presents a strategy to model manufacturing systems, covering the topology of the production plant, machinery, and production recipes. The paper also proposes a model-based methodology to automatically configure the reference software architecture and hence the machines in the system. The application of the contributions to a fully-fledged production line shows the effectiveness of relying on model-based automatic configuration.

INDEX TERMS Smart manufacturing, computer aided manufacturing, flexible manufacturing systems, software architecture, systems modeling.

I. INTRODUCTION

Information technologies in manufacturing enable a wide set of new functionalities in manufacturing systems, making production lines more flexible and turning them into complex Cyber-Physical Production Systems (CPPSs). To maximize the benefits of such a revolution, principles from the Service-oriented Architecture (SoA) paradigm has been introduced in manufacturing [1], generating the concept of Service-oriented Manufacturing (SoM) [2]. This paradigm assumes that the functionalities carried out by pieces of equipment are organized into "*machine services*", hiding internal implementation details, and exposing only the machine's interaction protocols, by means of specific drivers [3].

Exploiting service-oriented principles implies moving complexity from the system design phase to the system configuration phase [4]. However, configuring a manufacturing system is typically a time-consuming and manual task. It consists of setting up machines and the controlling software architecture. Furthermore, each component's configuration is specific and tightly coupled with other component's configurations. This strict relation is typically the most common source of errors. Furthermore, the same procedure occurs each time the configuration must be modified during

The associate editor coordinating the review of this manuscript and approving it for publication was Zhiwu Li.

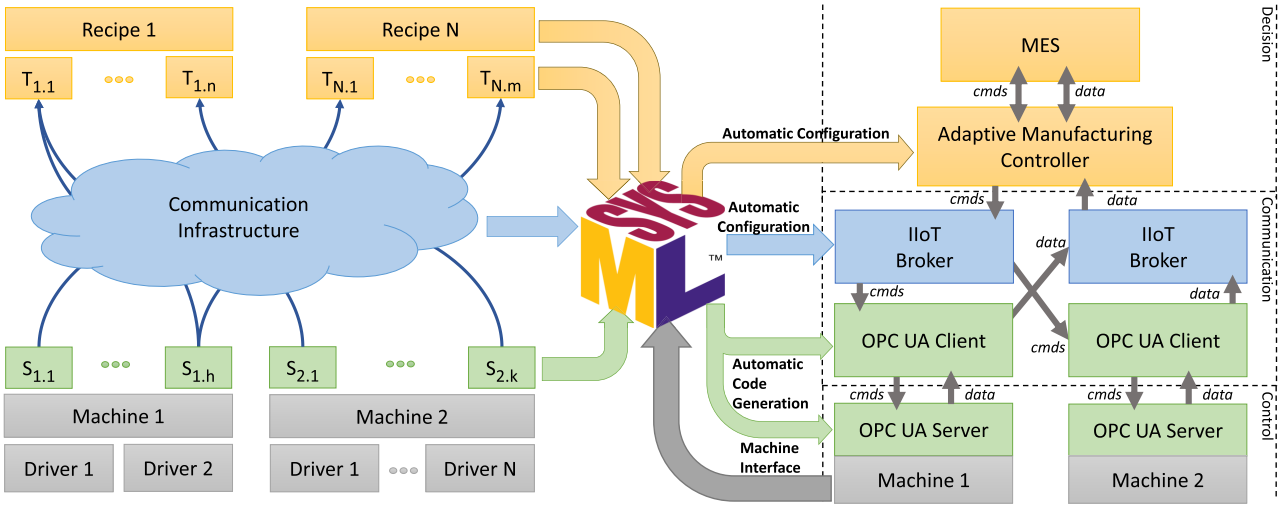


FIGURE 1. Overview of the proposed methodology, centered around System Modeling Language (SysML). The left side depicts the conceptual model represented using the SysML language. The right side depicts the proposed SoM architecture, automatically configurable from models.

the system's life-cycle, or when new machines are introduced or replaced in the system.

For such reasons, we define a software architecture able to support SoM principles, and an automated configuration methodology for SoM production systems, which relies on Model-based System Engineering (MBSE) techniques [5]. An overview of the methodology is depicted in Figure 1. At the core, SysML is used to model both the manufacturing architecture and the desired architecture functionalities. The structure of the proposed SoM architecture is represented in the right portion of Figure 1. It consists of three main components: *decision* through the *Automation Manager*, *communication* through a dedicated factory communication infrastructure, and *control* through OPC Unified Architecture (OPC UA) servers and machine drivers. Starting from the bottom, the machines are connected with the architecture through the communication protocol OPC UA. Industrial Internet of Things (IIoT) message brokers enable communication between the components of the architecture; the choice of the appropriate message broker for the architecture is based on a qualitative and quantitative analysis, which is presented later in this paper. On top, the Adaptive Manufacturing Controller (AMC) handles the interaction with the Manufacturing Execution System (MES) and the machines. The architecture is designed according to the principles of the cloud-native architectures, which are extensively used for SoAs [6]. The left side of Figure 1 reports the architecture functionalities. On the one hand, we propose a methodology to create SysML models of machines and drivers by reusing structural models expressed using different languages (e.g., the Automation Markup Language (AutomationML)). On the other hand, we propose a strategy to model production tasks and recipes. The resulting models encapsulate the information necessary to automatically generate the configuration and code necessary to control the reference SoM architecture.

The main contributions can be summarized as follows:

1) A cloud-native SoM architecture, which introduces advanced scheduling and dynamic production reconfiguration, extending the functionalities of a traditional MES. While the proposed architecture is general, an in-depth analysis of the most common IIoT message brokers will guide the concrete realization of the proposed architecture;
2) A modeling strategy to represent manufacturing systems using SysML, defining the topology of the plant, the production recipes to be executed and the requirements of the system;
3) A methodology relying on the SysML models to generate the configuration files for the proposed SoM architecture and automating its deployment on a Kubernetes cluster;
4) The application of the architecture and its configuration on a full-fledged reconfigurable manufacturing system. The results estimate the necessary effort to add a new machine, comparing it to a manual approach. In addition, we compare the scheduling implemented by a commercial MES with the one implemented on top of our proposed architecture, showcasing its advantages.

The paper is organized as follows: Section II presents the necessary background and analyzes the state-of-the-art software architectures for manufacturing systems and modeling of CPPSs. Section III describes the proposed SoM architecture. Sections IV and V describe the model-based methodology to specify the architecture and the functionalities of a production system. Section VI details the methodology for exploiting the models to automatically configure of the architecture. Section VII presents a qualitative and quantitative analysis of the most common message brokers; the analysis will be used to concretely instantiate the SoM-enabled architecture to control the system used in

Section VIII to experimentally evaluate the contribution in a full-fledged production system. Finally, Section IX will draw some conclusions.

II. BACKGROUND

This section will first summarize the state-of-the-art and practice on *software architectures* for production systems; then, it describes the current literature on *production systems modeling*. Finally, it presents the most used communication protocols used in IIoT systems.

A. SOFTWARE ARCHITECTURES FOR MANUFACTURING SYSTEMS

Software governing manufacturing systems is organized according to the *Automation Pyramid* since 1985 [7], and it is still in use also when looking at the most recent trends, such as the German plan for *Industrie 4.0* [8] or *Made in China 2025*. According to the automation pyramid, systems are organized into five layers, from the bottom up to the top:

- the *Field Level* is made of the sensors and actuators physically interfacing with the machinery on the production floor, as well as the software governing sensors and actuators;
- the *Control Level* is composed by a set of Programmable Logic Controllers (PLCs), each one governing the control algorithms of each machine in the plant;
- the *Supervisory Level* allows human operators to monitor the production processes being carried on by the system. Its main components are the Supervisory Control and Data Acquisition (SCADA) systems and the Human-Machine Interaction (HMI) interfaces;
- the *Planning level* is in charge of deciding which operations has to be carried out and how to allocate the resources already available. The decisions are taken by the MES, the main actor on the level.
- the *Management level* connects the production facilities with the other areas of the company. Its main component is the Enterprise Resource Planning (ERP) software, which takes care of planning the resources supplying, their management, and in general, it plans the medium-term activities of a production system.

The complexity of actual architectures additionally demand the definition of a standard terminology and a unique data representation used throughout all the automation levels. For this reason, the International Society of Automation (ISA) defined the IEC 62264 standard (also known as ISA-95) [9]. The standard defines functionalities, responsibilities, standard terminology, and data exchange within the pyramid and between the production and corporate parts of a manufacturing company.

While the automation pyramid has been guiding the design of manufacturing systems, the strict delimitation among its layers may limit the flexibility of production lines. Hence, this design is being replaced in more recent work with network-structured architectures. However, the ISA-95 standard is still able to capture the aspects of this kind of architectures [10].

As a matter of fact, our proposed architecture is both network-structured and a classic automation pyramid and its design can be fully expressed using the concepts outlined in the ISA-95 standard.

B. PRODUCTION SYSTEMS MODELING

MBSE techniques [11] allow the encapsulation and abstraction of systems architectures and functionalities into models that are able to capture widely heterogeneous components, dynamics, and behaviors, as well as a large variety of different viewpoints [12]. Thus, the MBSE approach is especially apt for designing advanced manufacturing systems [13], [14].

In particular, SysML and AutomationML are popular modeling languages in this field [15]. SysML is widely used for software and systems model engineering. It provides a set of diagrams over Unified Modeling Language (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. Specializations of SysML have been proposed [16] to aid the development of automation software (i.e., PLC software) for smart manufacturing systems. Models can be used to automatically generate and integrate control software into machines. SysML has also been used to ease the development and the integration of a MES in a production line [17]. AutomationML is an XML-based data format, created to provide a formal exchange model for heterogeneous engineering tools [18]. It is capable of describing plant components from different points of view, from the plant topology to a machine's kinematics. Different standards are strongly intertwined within AutomationML to compose a complete description of the system to be characterized. In particular, the Computer Aided Engineering Exchange (CAEX) (IEC 62424) provides features to represent a topological view of the system, with relations between objects.

C. COMMUNICATION PROTOCOLS

Among the plethora of today's manufacturing standards, OPC UA 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 [19]. The communication is based on a client/server structure, where the server contains and exposes its information model. Due to its versatility, OPC UA allows modeling data transport that is compliant with the ISA-95 standard [20].

Distributed applications need scalable, fault-tolerant, and low-latency communication channels to interact with each other. Publish-subscribe is a popular communication protocol used in SoAs, in which a set of entities (publishers) sends messages to other entities (subscribers) through queues created on a central broker. The broker acts as a router between producers and consumers, and follows a delivery

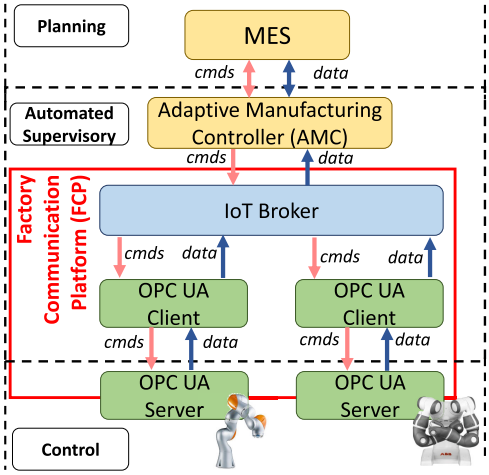


FIGURE 2. General structure of the proposed SoM-enabled architecture.

policy that may include prioritization and filtering. Among the plethora of message brokers available, the most diffused ones are [21], [22]: Apache Kafka [23], RabbitMQ [24], and KubeMQ [25].

Apache Kafka is an open-source message broker designed to process high volumes of event streams. It only supports the asynchronous messaging pattern, in which events are processed independently. Published events are organized into topics and further divided into partitions (the smallest storage unit holding a subset of records owned by a topic) managed by multiple brokers, guaranteeing persistence and resilience. RabbitMQ is an open-source message broker supporting different messaging protocols and messaging patterns. It supports both asynchronous and synchronous messaging patterns enabling Remote Procedure Call (RPC). Messages can be sent directly to queues or to exchanges that filter and forward messages to other queues based on the messages' metadata. KubeMQ is a message broker designed to perform better in a Kubernetes cluster. It supports all the most common messaging patterns, thus being specifically tailored for a microservice environment, offering persistence, scalability, and resilience.

Section VII presents a quantitative and qualitative comparison of the most used brokers. This analysis will be used to choose the most suitable brokers for the proposed architecture.

III. SERVICE-ORIENTED MANUFACTURING ARCHITECTURE

A SoM-enabled software architecture must be able to interface and manage highly different pieces of equipment, while scaling on a large application scenarios. Furthermore, it must guarantee robustness and provide high levels of Quality of Service (QoS). To fulfill these requirements, we propose to rely on containerization and virtualization by implementing the entire software architecture as a Kubernetes cluster. It consists of different Kubernetes containers connected through Internet of Things (IoT) message brokers, also running within Kubernetes containers. Figure 2 depicts a general view of the proposed architecture: it spans from the control level up to the planning level of the classic automation pyramid. It connects the MES to the machinery via two main components: the *Factory Communication Platform (FCP)* in charge of handling the communication in the system, and the *AMC* monitoring and supervising the system. The FCP collects and stores machine data and provides RPC communication to query and command the resources in the infrastructure. The AMC manages the connection with the MES, and it extends its functionalities by introducing dynamic reconfiguration of the production line, autonomous execution of production recipes, and advanced scheduling. The integration of these components, detailed below, injects automation at the supervisory level of the pyramid, thus shaping an *automated supervisory level*.

A. FACTORY COMMUNICATION PLATFORM (FCP)

The FCP monitors the equipment, stores the gathered data, and propagates the commands invoked by the AMC to the machines. It acts as a service platform allowing, among other functionalities, to interact with the other containers through the RPC paradigm. Its structure is highlighted by the red rectangle in Figure 2, and is composed of one IoT broker, a set of OPC UA servers and clients. All the components run on the nodes of a Kubernetes cluster.

All the flows of data are managed by an IoT broker exposing and listening a set of topics. Many communication brokers are suitable to be used to manage the communication in the FCP. Many broker solutions are suitable to implement the proposed architecture. As long as the solution supports publisher/subscriber semantics, data persistence, high availability (e.g., the possibility of being replicated on different nodes in the cluster), and minimum requirements of throughput and security. The most common potentially suitable broker solutions (i.e., Kafka, KubeMQ, RabbitMQ) are analyzed qualitatively and quantitatively in Section VII.

Each machine in the production system is controlled by an OPC UA server. The piece of equipment and its server are strongly intertwined: the data model (e.g., machine status and sensors data) exposed by the server depends on the underlying drivers that communicate with the machine. Machine services are implemented through *OPC UA methods*, which allow exposing functions in a RPC fashion. Then, the OPC UA data model is enriched with state variables that a client may read to get the status of the running operation. The OPC UA method acts as a wrapper on top of the machine-dependent code. In fact, by modifying the data model and the implementation of machine services, it is possible to modify or add functionalities without dealing with the complexity of PLC functions.

The FCP communicates with the equipment through multiple *OPC UA Client* nodes: an instance (and configuration) is active for each *OPC UA server*. This node creates a persistent connection with the machine and creates a *OPC UA subscription* to the *OPC UA variables*, listening for data

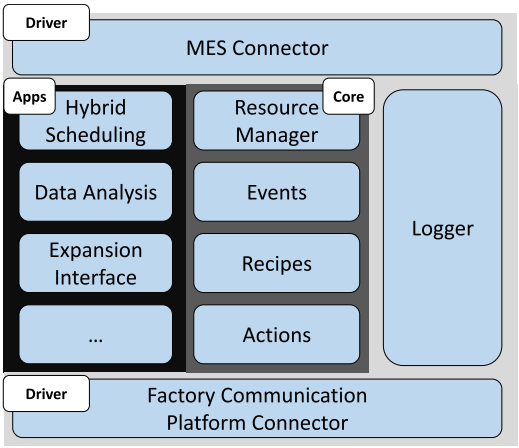


FIGURE 3. The internal structure of the *AMC*.

changes. Each time a variable changes, the client is notified with the new value. The value is written to a topic captured by the IoT broker. The *OPC UA client* nodes are also connected to the broker to listen for RPC requests. The allowed client requests are: *read* a variable, *write* a variable and *invoke* the execution of a *OPC UA method*.

B. ADAPTIVE MANUFACTURING CONTROLLER (AMC)

The AMC handles the communication with the MES and the machines transparently, and its sub-components are organized in three different layers, as depicted in Figure 3. The information gathered from the MES and the machinery is used to continuously make decisions about production adapting it to the current status of the system. It is a refinement and an extension of the Automation Manager software module we originally presented in [26]. The top layer is the *MES connector*: a *driver* interfacing the AMC with the upper layers of the automation pyramid. The bottom layer is the *FCP connector*: a *driver* connecting the manager with the lower layers of the pyramid. The middle layer contains the manager *Core*, a set of *Applications*, and the *Logger* module. The AMC is compliant with the ISA-95 standard and any existing software infrastructure based on the same standard. The AMC takes in input the information coming from both the MES and from a set of configuration files describing the machine capabilities and the implemented recipes. These characteristics are meant to ease its integration within already existing manufacturing plants. Thus, the architecture is meant to be integrated within legacy manufacturing systems to introduce advanced production features.

1) DRIVERS

The *Driver* levels contain the components enabling communication with other pieces of software. The *FCP Connector* exposes the basic functionalities of OPC UA, such as variable read/write, methods call, and subscriptions for data changes. It also communicates with the OPC UA clients connected to the machines in the cluster. The *MES Connector* is implemented as a RPC client, calling functions defined by a RPC server connected directly with the MES. An advantage of using RPC interfaces is that the integration with any other MES only requires to implement the corresponding RPC server. The driver allows navigating through the MES configuration and notifying the actions executed by the architecture, such as the execution of a set of operations or a triggered reconfiguration. The last driver is the *Logger* that publishes log messages on two different topics: one for the debugging messages, while the other includes messages logging actions executed by the architecture (e.g., execution of recipes or machine services). This allows notifying the entire architecture of the status of each level.

2) CORE

The second level contains the *Core* components, defining and implementing production processes. The MES represents production recipes as a sequence of dependent tasks, each associated specifically to a *class* of working cells (e.g., a work center). This allows modeling production processes at a higher level of abstraction, hiding the implementation details (e.g., PLC cycles). However, this is not enough to execute tasks without human intervention. Therefore, the recipe representation in the Core of the AMC is extended with a lower-level model describing the implementation of tasks on the working cells. This representation consists of an ordered sequence of actions with input and output parameters, formalized as a directed cyclic graph where actions are nodes connected by directed edges to represent dependencies. Section V will describe how the production recipes, tasks, and services are modeled.

An action can be a service exposed by a piece of equipment or a logical construct (e.g., creation of variables, the sum of variables, if, cycles, etc.) proposed by the Core. This extension allows executing tasks with a simple visit of the graph nodes. Then, the actual execution of tasks is managed by the *Resource Manager* that retrieves the manufacturing structure from the MES and, for each working cell, it connects to the correct machine's client. Thus, ensuring clients' isolation when executing the particular production operation. Furthermore, the resource manager guarantees that a maximum of one operation is executed on a working cell at the same time.

3) APPLICATIONS

The AMC is a modular piece of software with modules (i.e., the AMC apps) implementing advanced functionalities to refine the control of the system. Among these software modules, the most important are the production scheduler, and the applications collecting, organizing, and analyzing the production data to detect anomalies [27], [28].

Optimal scheduling of production processes on different machines in a dynamic environment is still an open problem [29] known as Dynamic Flexible Job Shop Scheduling (DFJSS). Although there are many solutions to the static counterparts of this problem, such as Job Shop Scheduling

(JSS) and Flexible Job Shop Scheduling (FJSS). However, once introduced the dynamic component that characterizes real systems, the solutions for these problems are not applicable. The main 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 becomes infeasible. 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 [30]. The first implementation of the proposed architecture relies on 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 (e.g., new job arrivals, or machine breakdowns) occurs in the system.

Furthermore, we developed a data analysis application receiving timing data about executed production processes to achieve more precise scheduling. 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.

IV. MODELING THE SYSTEM ARCHITECTURE

The methodology hereby described creates the production plant model alongside the computational infrastructure model within the same SysML description. This is a fundamental feature that enables the reuse of models and improves the design process for complex systems. This section details the steps necessary to build the SysML diagrams representing the architecture of the production system. First, it proposes a methodology to build the structural description of the production plant by reusing AutomationML models. Then, it presents a strategy to model the computational infrastructure using SysML.

A. SYSTEM'S STRUCTURE DEFINITION VIA MODELS REUSE

Our model reuse methodology is inspired by the work proposed in [31] and, which presents a mapping of the AutomationML language elements into SysML. AutomationML models describe the hierarchy of components in a production line, their parameters, and relations. Therefore, on the one hand, AutomationML models can be exploited to create the structural part of SysML models, which is necessary to define the components to be configured and their target parameters. On the other hand, AutomationML limitations on representing functionalities, protocols, etc. require exploiting SysML, which is more expressive and includes a wider set of language constructs. Table 1 presents a set of conceptual constructs used by the proposed methodology. It provides a concretization of such constructs on AutomationML language elements, also describing a mapping to appropriate SysML language elements. The Table organizes the modeling strategy by subdividing the

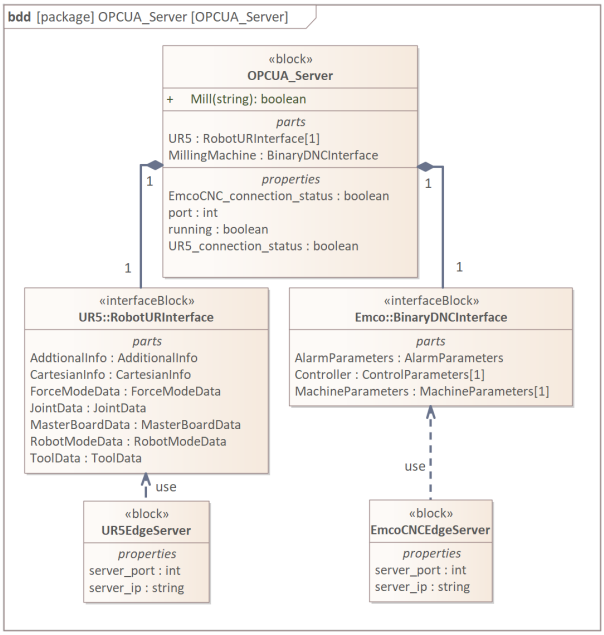


FIGURE 4. SysML Block Definition Diagram of a OPC UA server controlling two machines: a UR5 robot arm and a milling machine.

set of constructs into four categories, from the most basic one to the most complex: *classes*, *Internal Classes Fields*, *Components Instances and Relations* and *Libraries*. The first category describes a means to specify component types, their roles within the production environment and their interfaces to other components. The second category details a way to customize the components through a set of parameters and actual instances. The third category outlines the instantiation of components and the relations between instances, to construct a system as a hierarchy of concretized components. Finally, the third category provides a method to collect different concepts (e.g., components classes) in specific libraries.

Both languages exploit object-oriented principles. Therefore, classes can inherit properties from other classes or can implement abstract classes. As a consequence, different relations can be expressed using different constructs. As an example, in SysML, the *Generalization* connector between blocks represents the inheritance relation of AutomationML, expressed between a superclass (which is the actual generalization) and the sub-class. Furthermore, AutomationML requires that a role class must be associated to a simple class: the *RoleRequirement* is the specific construct that defines the relation between a class and a role. In addition, an element may be associated with multiple roles, expressed by multiple *SupportedRoleClass* AutomationML relationships. Roles are mapped to abstract classes in SysML. Therefore, the association of a role class to a class or an internal element is mapped to the *realization relation* between SysML blocks.

B. COMPUTATIONAL INFRASTRUCTURE MODELING

Manufacturing systems characterize their current state and its evolution through data, typically describing operating

TABLE 1. The set of conceptual constructs of the proposed models reuse strategy, their concretization on AutomationML language elements and their mapping to SysML.

Concept	Description	AutomationML	SysML
Classes			
Component Type	Definition of a specific component (e.g., a PLC), that includes a set of properties, may be composed of different sub-components and can be related to other entities.	*SystemUnitClass*, with attached attributes and interfaces. Each SystemUnitClass can have instances of other SystemUnitClass objects (i.e., sub-components).	Single *Block* in a Block Definition Diagram (BDD)), representing the component and its properties. The Block could also have different contained parts (i.e., other sub-components Blocks).
Role Type	Semantic definition of a production role or functionality (e.g., cutting), to be attached to a piece of equipment.	*RoleClass* with attached attributes and interfaces.	Single *Block* in a BDD, defining the production capabilities of the machine, referenced by component Blocks.
Interface Type	Definitiion of a connection relation between components, detailed through attributes (e.g., baud rate).	*InterfaceClass* with attached attributes.	A single *InterfaceBlock* in a structural diagram(i.e., a BDD) and the set of attached properties.
Internal Classes Fields			
Internal Class Field	Characterization of a specific property of a component.	An *Attribute* contained in a Class, specified by a typed value.	A typed *Property* of a Block belonging to a BDD.
Interface Instance	An actual interface of a component, typed by a class.	An *ExternalInterface* contained in a Class.	A *Port* component belonging to a Block.
Components Instances and Relations			
Instances Structure	The instance hierarchy representing the actual system, composed of typed components.	The *InstanceHierarchy* structure, defining the component of the system, their actual attributes and their typed interfaces.	An *Internal Block Diagram (IBD)* which defines the relations and the physical connections of the system components.
Instances Connection	An actual connection between components in a semantic domain (e.g., physical, digital, etc.).	An *InternalLink*, connecting *ExternalInterfaces* of two different components.	A *Connector* construct between typed Ports of the different components in an IBD.
Libraries			
Library of Components	A collection of components which may be in a semantic or structural relation between each other.	*SystemUnitClassLib*, composition of SystemUnitClasses	A *BDD* composed of Blocks representing the components.
Library of Roles	A collection of abstract component functionalities.	*RoleClassLib*, composition of RoleClasses.	A *BDD* composed of Blocks representing an abstract functionality and referenced by other Blocks in other BDDs.
Library of Interfaces	A collection of interfaces between components.	*InterfaceClassLib*, composition of InterfaceClasses.	A single *BDD*, composed of InterfaceBlocks, representing different component interfaces.

parameters (e.g., the temperature of a component, its rotational velocity, or a possible alarm status). Such data is retrieved through machine drivers: software components responsible for the collection of the parameters' values from PLCs or edge-nodes, and sending them to the plant's computational systems. In SysML, the structure of the data transmitted and received is modeled by BDDs: each component can be represented as a single *block*, while block *properties* are used to specify the details of the component. Figure 4 shows a BDD of the OPC UA server in a multi-system working cell. The main entity is the OPC UA server, which coordinates two independent systems: a UR5 collaborative robot and a milling machine. SysML blocks are used to model both the OPC UA server and the two machinery's edge servers. The interaction between each component is carried out by means of an interface (i.e., machine driver). This detail is represented by a *composition* relation between the machine drivers and the OPC UA server. In SysML, interfaces (i.e., drivers) are represented by *interface blocks*.

In Figure 4, the driver blocks (''RobotURInterface'' and ''BinaryDNCINterface'') represents different BDDs, which contain all the properties and *methods* that are exposed by the machine edge server through the drivers. Each driver interface block is the main block of another BDD (not reported in Figure 4) that defines the semantics of the different parameters available in the machine driver's data model. As an example, a parameter can be a driver variable with an elementary data type(e.g., *int*, *float*) or a driver object represented hierarchically by another block. For instance, in Figure 4 the ''EmcoCNCEdgeServer'' driver contains a property for the milling machine parameters (''MachineParameters''). This property refers to a composition of blocks modeling the status of the machine (e.g., ''AxesParameters'', ''WorkingAreaParameters'', and ''ToolSystemParameters''). The blocks identifying a machine edge server (''UR5EdgeServer'' and ''EmcoCNCEdgeServer'' in Figure 4) are connected to the respective interface blocks through a *dependency* relation. In fact, the edge server block *uses* the driver to exchange data with the OPC UA server blocks, and therefore, it depends on the driver. This relation describes the communication channel between the driver interface and the machine edge server, encapsulating the connection parameters.

V. MODELING THE SYSTEM BEHAVIOR

The models of the architecture describe the execution platform where the manufacturing is executed, as well as the computational platform governing the infrastructure. However, they do not describe how tasks are performed, that is the behavior of the system and its components. In the context of SoM manufacturing, modeling the system behavior means modeling the *functionalities* provided by the machines, i.e., the services provided by the infrastructure, as well as

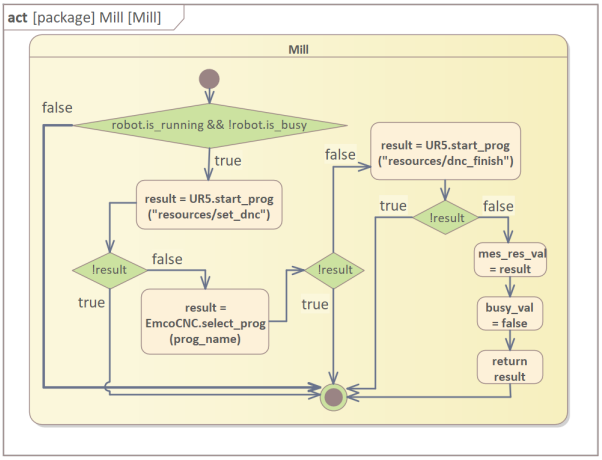


FIGURE 5. SysML Activity Diagram of a OPC UA method, which represents the milling operation flow between two different machines orchestrated by the OPC UA server.

the *recipes* to be implemented, i.e., the services required to produce a product or a set of products. Furthermore, a production system is intrinsically a distributed system, therefore communication is another aspect to be modeled. Indeed, SysML is well suited to model all these aspects. Thus, many modeling strategies are suitable to model SoM systems, as long as the services provided by the infrastructure, and those required by the recipes can be properly identified in the model. In this work, we rely on the modeling strategies we previously introduced in [32] to model the components' behavior, and in [33] to model the production recipes.

A. COMPONENTS BEHAVIOR MODELING

Other than sending data to the external world, a OPC UA server also allows exposing its functionalities through OPC UA commands. Our SoM architecture relies on this principle to expose the services provided by the pieces of equipment available in a manufacturing line. Therefore, it is necessary to represent such functionalities by specifying the operations to be performed, the machine driver interface, and the machine's behavior triggered by a specific condition. In a typical UML fashion, SysML models this aspect with block *methods*, as discussed in the previous section, and with connected *behavioral diagrams* describing the process implemented by the piece of equipment.

In this context, the behavioral diagrams provided by SysML are most suited to describe the services and functionalities of the production plant are *Activity diagrams*, *State Machine diagrams* and *Sequence diagrams*. Activity diagrams and State Machines are well-suited to describe the behavior of the equipment, while Sequence diagrams are best suited to describe the communication services provided by the computational infrastructure. Indeed, following the philosophy of SysML that grants freedom to the designer, a designer may use any of these types of diagrams to model the behavior of a component.

For example, in Figure 4, the method `Mill()` coordinates the robot and a milling machine to pick, cut, and place the piece back on the transportation system.

The method is associated with the Activity Diagram shown in Figure 5, which expresses an operational flow between the two machines. The control procedure interacts with both the robot and the milling machine, checking and enabling control variables necessary to perform the desired functionality.

B. PRODUCTION RECIPES MODELING

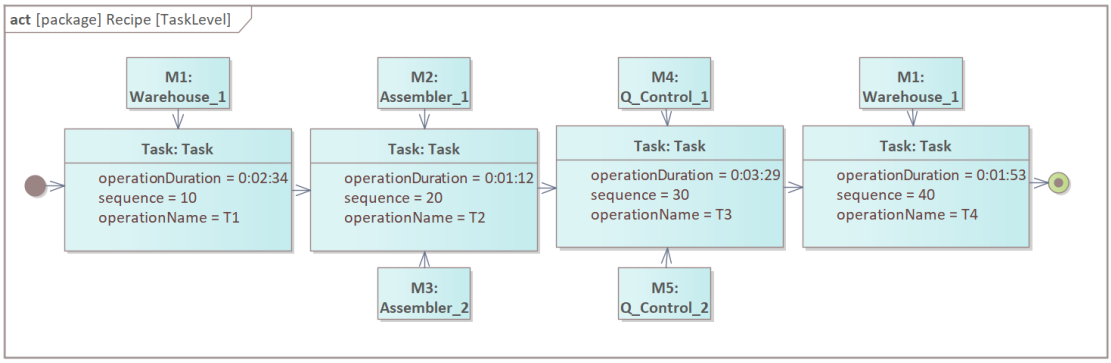
In the context of SoM, production recipes can be modeled with different levels of granularity. Indeed, describing more granular processes enables more schedule optimizations; however, a finer granularity leads to more complex models, harder to be developed by users. For this reason, we adopt a three-level hierarchical modeling approach to model production recipes. Each level depicts a particular abstraction perspective of the production process: from the high-level representation of production recipes as a set of tasks, to a sequence of atomic operations performed by the machines and the infrastructure. Figure 6 provides a simple example of the three levels of a simple four-task recipe. All three levels are described in SysML by using *Activity diagrams*.

1) TASK LEVEL

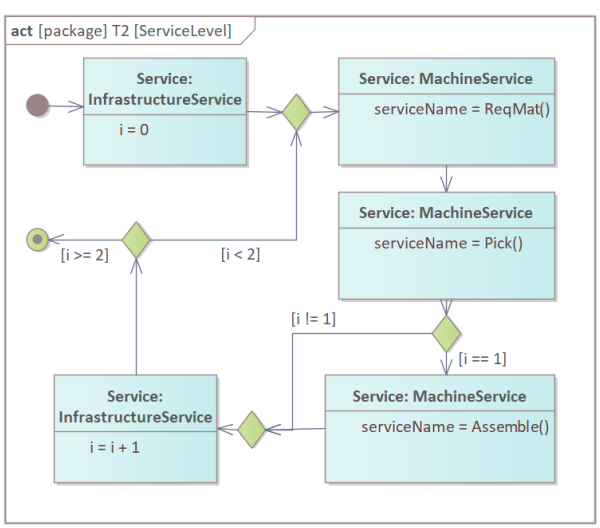
The first level represents at the highest abstraction level, a production recipe as a set of macro-steps (i.e., tasks) executed on a machine. It consists of a task-resource graph, which is similar to a Resource Task Network (RTN) representation. The nodes of the graph represent a task, while the solid edges describe a partial order over the execution of the tasks, as shown in Figure 6. Each task identifies a contiguous production step executed on a machine and is characterized by the required materials and the transformed material states. A task can be allocated and executed on one or more machines with which it is associated (represented by the dashed edges in Section V-B). On these relations, a set of attributes specifies the "cost" associated with the tuple (resource, task). The attributes are the execution time, the hourly cost, the electrical consumption, the estimated efficiency, and the necessary tools. For the sake of clarity, Section V-B reports only a subset of these parameters.

2) SERVICE LEVEL

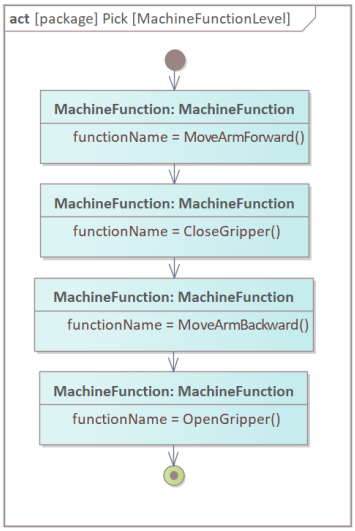
In the second level, each tuple (task, resource) is refined with a more detailed model of the sequence of machine services implementing it (depicted in Section V-B). It consists of a control flow graph, representing the interaction with the machines and sensors through services, and their dependence. This allows identifying the subset of services necessary to carry out the task on the machines. 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



(a) Task level



(b) Service level



(c) Machine function level

FIGURE 6. An example of the proposed three-layer representation for production processes. Figure 6 shows the first layer, which defines tasks and their allocation onto machines. Section V-B specify in a control flow graph the sequence of "services" implementing the task T2. The third layer outlines machines' functions implementing the Pick service (Section V-B).

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 machine behavior 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.

Each service is characterized by a set of input parameters and output parameters that are used by the control flow statements and as parameters for the machine services. Variables defined within a task have a global scope within the production recipes, allowing to make available the results of machine services to subsequent tasks.

This level, combined, with the upper level, allows to highlight the machine's usage more precisely, enabling to further optimize the machine schedule by developing methodologies implementing tasks interleaving. Regarding Section V-B, the service level models a task first requesting two materials before assembling them with a robotic manipulator. In this case, the robotic manipulator could perform other activities meanwhile, materials are being moved to the robotic cell.

3) MACHINE FUNCTION LEVEL

The machine functions used in the second level is refined in the third level. The purpose of this level is to represent the actual control behavior of machinery at the PLC level as a control flow graph. Unlike the upper level, the variables defined in this level have private scope. Therefore, to access variables defined in the outer scope each function has a list of input and output parameters that can be variables or constants.

This level allows planning the execution of the machine services, taking into consideration when a precise atomic function will take place. In fact, 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. With regard to the example reported in Section V-B, the control function

"Pick" is refined in a sequence of four machine functions: (1) move the robotic arm, (2) close the gripper, (3) move back the arm, and (4) open the gripper to release the material. In addition, the information represented at this level can also be exploited to further optimize the process, to minimize the machines' setup and waiting times.

VI. MODEL-BASED SYSTEM CONFIGURATION

The FCP and the AMC, the main components of the proposed SoM architecture, are generic pieces of software that require to be configured for each specific production system. The models created as described in the previous sections provide all the information required to control the system and its communication flows. This section describes how the information stored in the model can be used to automatically configure the software component in the proposed architecture.

A. OPC UA SERVER CONFIGURATION

The OPC UA servers act as a communication interface between the machine and the AMC, by interacting with OPC UA clients. The automatic generation of a OPC UA server requires a set of mandatory information: the details of the communication channel to the machine, the interaction protocol with the drivers (i.e., how to read data and send commands), and which parameters and methods are offered by the server. The required pieces of information for the automatic generation are contained in the SysML models described in Section Section IV. Such information is extracted from the model and exploited to build a set of configuration files (i.e., the inputs of the OPC UA servers and clients). In particular, the files generated for the OPC UA servers are the *Information Model*, the network connection parameters, the drivers' interface parameters, and the implementation of the OPC UA server methods. The same Information Model and network connection parameters are then used by the OPC UA clients to correctly instantiate the communication with the server. Driver's interface parameters are mapped into OPC UA variables using the value-type specified in the SysML diagrams of the Information Model.

The generated Information Model is composed of the driver's interface parameters of the underlying system components and the functionality modeled on top of them. A piece of code implementing a OPC UA method is also generated for each *Activity Diagram* associated with a OPC UA server block. More specifically, the generated method exploits the driver interfaces' functions and implements the interaction flow described in the SysML Activity Diagram. The generated OPC UA server loads the driver interfaces with the respective communication channel parameters, the Information Model, and imports the code implementing the OPC UA methods. Each driver interface periodically polls the machine to obtain data changes, which are published for the OPC UA clients. Therefore, data coming from the driver interfaces are published directly to the specific OPC UA parameters.

TABLE 2. Configurations complexity in the number and type of messages sent each second on the chosen architectures.

config. type	# clients	# reads/s	# writes/s	# updates/s	# cmds/s	tot/s
small	1	228	228	788	204	1448
	2	256	256	816	208	1536
	5	340	340	900	220	1800
medium	1	256	256	936	208	1656
	2	312	312	992	216	1832
	5	480	480	1160	240	2360
large	1	312	312	1112	216	1952
	2	424	424	1224	232	2304
	5	760	760	1560	280	3360

B. AMC CONFIGURATION

AMC internally represents production recipes following the structure presented in Section V: a multi-hierarchical representation composed of three levels describing different layers of knowledge. By exploiting this model, it is possible to generate the configuration for the AMC. Each production recipe is converted into a JSON file containing its graph represented through SysML. Thus, machine services used within the recipes are correlated with the services offered by the machines. This allows to uniquely identify a machine service and configure all the parameters necessary to call that service at run-time.

VII. EVALUATION OF IIOT BROKERS

The performance of message brokers has been studied in the literatures [21] and [22]. All these comparisons are centered on the transmission of raw data (i.e., throughput), but as far as we know, there are no available comparisons regarding how message brokers perform with respect to their supported specific types of communication paradigms.

In our proposed architecture, the communication between the applications is based on RPC and events subscription (e.g., machine data updates). For this reason, we performed an in-depth comparison between the standard architecture (taken as baseline) based only on OPC UA servers and OPC UA clients and our proposed architecture, where the central IIoT broker is one of the following: Apache Kafka, RabbitMQ, and KubeMQ, for a total of 4 different architectures. The analysis evaluates both the machine message's delay (variable read, variable write, method call, and variable update) and the resource consumption (CPU usage, RAM usage, and network load) of such architectures.

Table 2 reports the tested architecture's configurations, differing in the number and type of messages sent each second, and identified by the tuple configuration type and number of clients. Each configuration type (small, medium, and large) differs in the number of variables in the information model of the OPC UA server and their update frequency. The increasing number of OPC UA clients allows analyzing the scalability of the different architectures by evaluating their behavior under different network loads. In total, we tested on each architecture these 9 configurations (ranging from 1448 messages to 3360 per second) for a total of 36 test cases.

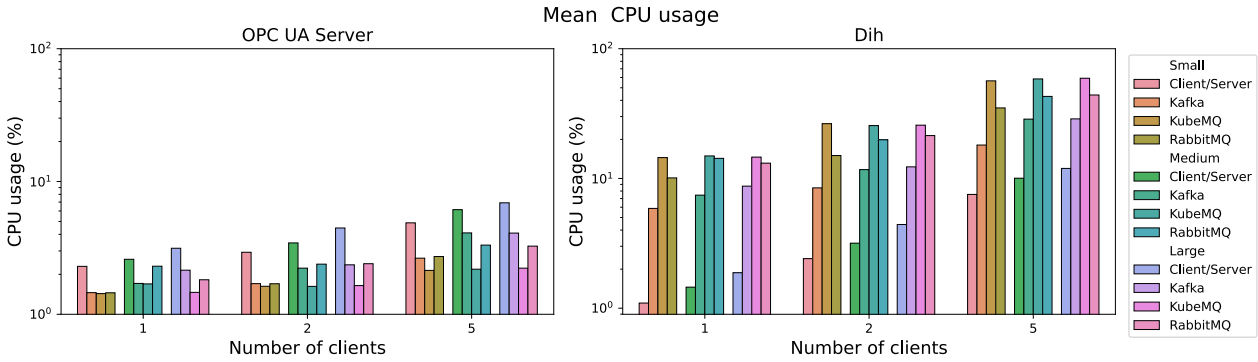


FIGURE 7. Mean CPU usage in percentage, normalized between 0 and 100 of edge devices (on the left) and on the Kubernetes cluster (on the right).

All the tests have been executed on a Kubernetes cluster, while the CPU usage, RAM usage, network traffic has been collected by exploiting Prometheus [34] as metrics server.

A. PERFORMANCE ANALYSIS

Our proposed architecture can be divided based on the hardware in which the software is executed into two parts: 1) machine interface (OPC UA servers) executed on edge devices 2) FCP executed on a Kubernetes cluster. For the sake of clarity, we have divided the subsequent performance analysis in these two categories, to differentiate the impact of our proposed architecture on both, edge devices and Kubernetes cluster (respectively, left and right parts of Figures 7 to 9).

Figure 7 depict the mean CPU usage, normalized between 0 and 100% of the different architectures, with an increasing number of clients reading and writing data from the server. From the left part, we can see that by introducing a SoM architecture (independently of the chosen central message broker) allows moving the CPU load from the edge devices to the Kubernetes cluster. Thus, by increasing the number of clients connected to the OPC UA server, this trend becomes more visible, showing a decrement in CPU usage from 4 up to 10 times with KubeMQ as the message broker. This leverages the edge devices (with limited computational power) from executing non-priority tasks, leaving room for real-time tasks and other on-the-edge applications.

Figure 8 depict the mean RAM usage in MB of the different architectures. In this case, we can see a minor difference between the standard architecture and the proposed one. The only difference is in the FCP, where the configurations using Kafka and RabbitMQ consume more memory. This phenomenon is caused on one side by the preventive allocation of memory, and from the other side to the messages' delay, which requires more time to be consumed, increasing the dimension of the message queues (discussed in Section VII-B).

Figure 9 shows the mean MB/s received and transmitted from the OPC UA server and from the Kubernetes cluster. In the standard architecture, the network load is higher on the OPC UA server, while in the SoM architectures is moved to the Kubernetes cluster. This leads to a lower impact on the Operational Technology networks (usually composed of hardware with limited bandwidth) and a higher load on the cluster network (local communication with higher bandwidth and lower delays). Among the evaluated technologies, KubeMQ stands out as the message broker with better performance.

B. DELAY ANALYSIS

Another important parameter for a SoM architecture is the messages' delay. A lower message delay implies a faster reaction from the applications subscribed to the same data source. Table 3 reports the communication delay of our proposed architecture with respect to direct connection with OPC UA of variable read, variable write, method call, variable update, and 50 consecutive variable updates. The results show that a variable read, variable write, and method call have higher delays in our proposed architecture. This is given by the intermediate component in our proposed architecture acting as a bridge between the OPC UA server and all the applications connected to the message broker. The maximum delay introduced is 5 milliseconds (for the architecture configured with KubeMQ), a negligible delay considering that the number of read requests can be reduced with caching mechanisms and complex machine functionalities in a SoM are enclosed into machine services (i.e., method call). Moreover, we can see that with larger configurations and a higher number of clients, where our proposed architecture consumes less resources on the OPC UA server this gap shrinks up and sometimes reverses.

Our proposed architecture reduces the delay of the variable subscription. OPC UA variable subscription relies on server polling to check if there are updates for the subscripted clients, and if so, the clients must perform a variable read. Having a single client connected to the OPC UA server reduces this mechanism's overhead by exploiting the message brokers for the messages' persistence and immediate forwarding of messages to multiple clients. This improvement becomes more evident with an increasing configuration complexity in which the improvement is 10%. Summing up, among the message brokers, KubeMQ performs better in our

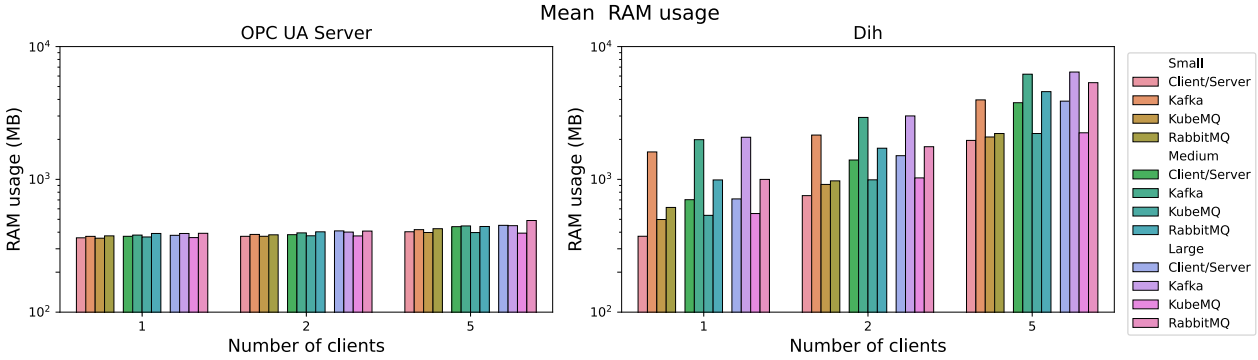


FIGURE 8. Mean RAM usage in MB, of edge devices (on the left) and on the Kubernetes cluster (on the right).

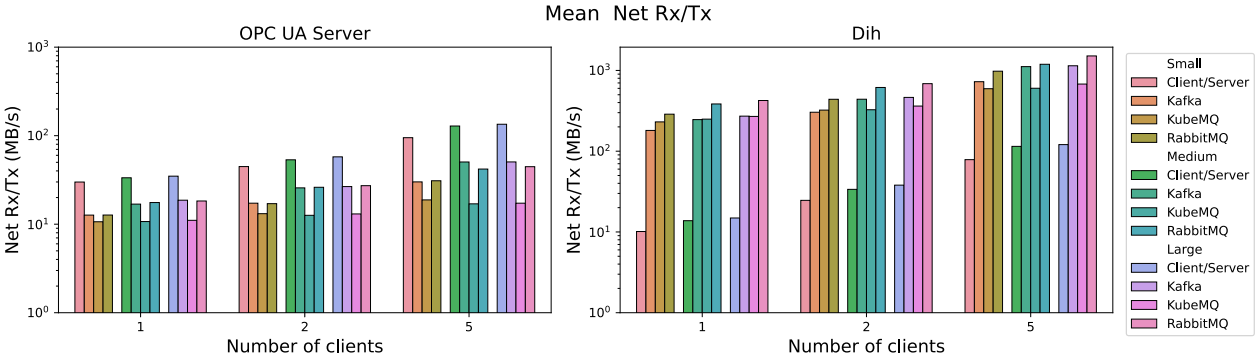


FIGURE 9. Mean MB/s transmitted and received from edge devices (on the left) and from the Kubernetes cluster (on the right).

proposed architecture with regard to Kafka and RabbitMQ, with lower delays and performance usages.

VIII. CASE STUDY: THE ICE LABORATORY

The Industrial Computer Engineering (ICE) Laboratory is a research facility of the University of Verona, meant to serve as a demonstrator for a wide set of computational technologies applied to the industrial manufacturing field.[1] The centerpiece of the laboratory is a complete and reconfigurable manufacturing line. The facility is equipped with a *vertical automated warehouse* storing the materials, semi-finished and finished products. The production line accommodates multiple diverse cells: a *robotic assembly cell* with two collaborative robot arm manipulators equipped with gripper and screwdriver tools; a *Quality Checking (QC) cell* equipped with a set of camera and laser scanners to inspect products for possible defects; an *additive manufacturing cell* equipped with multiple 3D printers; a *subtractive manufacturing cell* equipped with a four-axis Computerized Numerical Control (CNC) milling machine; a *functional control cell* equipped with a flying probe functional tester machine for testing electronic boards. The ICE laboratory logistics is also managed by a very flexible set of technologies: two Autonomous Ground Vehicles (AGVs) to move objects and materials in the shop-floor, for instance from the automated warehouse to the production line and vice versa; a reconfigurable system of conveyors that moves a set of mini-pallets between the different working cells in the line.

Each piece of equipment in the plant was originally equipped with a OPC UA server, which exposes the services provided by the machine. A state-of-the-art commercial MES was governing the production line, scheduling the production to send the work order machines, and tracing the products in the line. While the commercial MES offers the possibility to send commands to the machines for standard production, this possibility cannot be exploited when managing a flexible production line. In fact, reconfiguring the line would require extensive downtime to reconfigure the MES.

The contributions of this paper have been all applied to the ICE laboratory. The ICE laboratory's production line has been modeled in SysML, as described in Section IV, while the production recipes to be realized by the system have been modeled in SysML following the strategy described in Section V. Then, to automatically generate the OPC UA servers and the configuration for all the Kubernetes containers (e.g., the AMC and message brokers containers) we built a software that analyzes the SysML model to extract the represented knowledge. Specifically, the software analyzes the XML Metadata Interchange (XMI) produced by any SysML tool and produces the set of configuration files needed

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

TABLE 3. Comparison between the communication delay with the proposed architecture with respect to a direct connection with OPC UA.

# clients	config. type	architecture type	read (ms)	write (ms)	call method (ms)	subscription (ms)	50 var updates (ms)
1	small	client_server	**1.028**	**1.208**	**1.06**	**3.325**	534.826
		kafka	5.142	5.985	4.56	6.393	**521.262**
		kubemq	2.412	3.258	2.7	4.447	522.892
		rabbitmq	6.157	6.824	5.846	6.429	522.465
	medium	client_server	**1.055**	**1.362**	**1.146**	**3.279**	530.388
		kafka	5.74	6.194	4.777	6.98	**521.522**
		kubemq	3.92	4.236	3.799	6.323	532.28
		rabbitmq	10.328	10.751	8.94	13.916	528.842
	large	client_server	**1.242**	**1.555**	**1.141**	**4.099**	540.615
		kafka	6.596	8.088	6.636	8.985	529.06
		kubemq	2.951	3.369	2.943	5.158	523.083
		rabbitmq	9.492	10.148	9.359	11.609	**522.559**
2	small	client_server	**1.085**	**1.437**	**1.07**	**3.625**	529.377
		kafka	5.374	6.065	5.207	6.75	**521.48**
		kubemq	2.99	3.734	3.219	5.075	527.424
		rabbitmq	6.972	7.341	6.946	8.309	524.913
	medium	client_server	**1.216**	**1.42**	**1.179**	**4.18**	530.906
		kafka	5.868	6.764	6.452	7.586	526.319
		kubemq	3.522	3.751	3.483	5.38	529.434
		rabbitmq	11.583	10.909	8.385	11.872	**526.068**
	large	client_server	1.494	2.728	2.224	10.812	540.623
		kafka	6.112	7.585	5.895	7.695	525.722
		kubemq	3.461	4.04	3.891	5.671	527.545
		rabbitmq	13.251	12.428	12.2	16.886	527.185
5	small	client_server	**1.273**	**2.321**	**1.516**	**6.489**	539.702
		kafka	6.726	7.859	6.298	8.56	**524.906**
		kubemq	6.737	6.868	6.121	8.015	532.246
		rabbitmq	12.496	12.648	11.876	12.759	526.633
	medium	client_server	**1.578**	**2.695**	**1.652**	8.394	542.265
		kafka	11.022	11.998	10.893	13.605	**528.334**
		kubemq	6.167	6.163	4.622	**7.514**	531.731
		rabbitmq	23.796	31.43	21.742	20.272	533.82
	large	client_server	**2.013**	**3.533**	**2.168**	19.402	551.482
		kafka	12.134	13.406	11.964	15.39	**527.875**
		kubemq	6.254	6.369	4.631	**8.732**	532.975
		rabbitmq	25.666	27.542	28.446	182.012	537.747

TABLE 4. Size of the generated files with different recipes and machine configurations.

# Recipes	# Machines	Nodeset Size (KB)	Recipes Size (KB)	Container Config Size (KB)	Total Size (KB)
1	1	16451	108	19880	36439
3	2	20323	153	31970	52446
5	3	23276	239	52100	75615

TABLE 5. Comparison of the communication delay introduced by the proposed SoM architecture, against a direct OPC UA connection.

Transport Type	Read (μs)	Write (μs)	Methods (μs)	Subscription Update (μs)
OPC-UA	629.60	729.00	859.80	51124.80
SoA	4020.20	4223.90	3300.40	49560.80
Overhead	538.53%	579.41%	283.86%	-3.06%

for the proposed architecture. Thus, the original MES-centric architecture has been replaced by implementing the software architecture described in Section III, while the protocols choice has been guided by the results in Section VII. Lastly, we showcase the advantages of our proposed SoM over the commercial MES by comparing their scheduling capabilities.

In the following of this Section, we evaluate the complexity of the proposed methodology, by showing how the size of configuration files increases with respect to different scenarios. Then, we evaluate the overhead introduced by our proposed architecture. All experiments have been executed on a 3.60 GHz Intel Core i7 with 32 GB of RAM.

A. AUTOMATIC GENERATION ANALYSIS

The configuration complexity of a production line is directly proportional to the number of machines composing the plant. Therefore, the complexity of the SysML model representing the entire manufacturing system increases linearly with the number of components in the system. Table 4 reports the size of the configuration files automatically generated by our software, applied to three different portions of the laboratory machines. Each test consists of a SysML model containing various combinations of recipes and machines. Specifically, we divide the generated files into three main categories: (1) OPC UA nodeset, (2) recipes, (3) container configuration. The last column shows that the total size increases with the number of machines and recipes. Without the automatic generation provided by the presented methodology, the configuration files would have been written manually by the system installer or maintainer, in a time-consuming and error-prone process. Therefore, the advantages of the proposed methodology increase with the size of the system

TABLE 6. Execution time when using the state-of-the-art versus the proposed architecture to govern three different production recipes.

Recipe	# Tasks	# Service Calls	OPC UA Time (s)	SoM Time (s)	SoM Overhead
1	4	54	140.04	140.34	0.21%
2	5	44	120.03	120.18	0.12%
3	11	132	300.10	300.69	0.20%

to configure. The categories that impact the most on the total configuration files sizes are the container configuration and the OPC UA nodeset. This is due to the fact that, for each container, each software parameter enabling the communication between the different components of the proposed architecture must be set. Meanwhile, the OPC UA nodeset contains the definition of all the nodes.

For all tests, our software generated the configuration in less than 1 second. The time increases linearly with respect to the number of machines. The generated configuration has been tested on the real architecture to verify its correctness.

B. OVERHEAD ANALYSIS

Additional features of the architecture come at a price in terms of computational overhead. Table 5 reports the overhead induced by the architecture in calling machine services through OPC UA functions, in comparison with a direct connection to the machines. It compares the delay of different services, such as the read/write of variables, the method calls, and the subscription to variables. For each operation, the last row reports the additional overhead (in percentage) of the proposed architecture.

Comparing the results we find the maximum delay of around 100ms in the variables subscription. This is due to the structure of the OPC UA protocol which notifies clients when data changes only at certain specific intervals.

The additional overhead implies a significant communication delay for the read, the write, and the methods call, but is negligible for the subscription updates. However, this communication delay is in the context of complex physical processes, typically involved in a manufacturing line. For this reason, we also evaluated the behavior of the proposed architecture to coordinate different manufacturing processes. Table 6 reports the total execution time for three production recipes of different sizes. The table compares the time required to execute the recipes using a state-of-the-practice architecture against the proposed solution. The total execution times do not consider the time required to transport the materials through the conveyor belts. This is because transportation data is highly variable and influenced by many physical factors that may not be controllable by the 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 for each recipe, the delay introduced is negligible with respect to the total execution time.

TABLE 7. Comparison between the scheduling of 500 production recipes, using a classical RTN-based representation against the proposed hierarchical modeling approach.

Param	RTN Representation	Hierarchical Representation	Diff. (%)
Cycle Time	16:38:14 (h)	16:04:17 (h)	-3.52%
Avg. Machines Downtimes	4:43:25 (h)	4:08:15 (h)	-14.17%
Avg. Machines Utilization	70.61%	75.06%	+4.45%
Throughput	349.11 (u)	369.99 (u)	+5.98%
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%

Furthermore, comparing Tables 5 and 6, 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 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 delays are measured in tens rather than tenths of seconds.

C. SCHEDULING ANALYSIS

To demonstrate the advantages of our proposed architecture, we compare the results obtained by the scheduler implemented by the commercial MES, which relies on a classical RTN-based task representation, against the service-based scheduler implemented on top of the AMC. The implemented service-based scheduler, which has been initially proposed in [33] allows exploiting the proposed hierarchical representation of production recipes to optimize the production schedule. Either schedulers have been tested on 500 production orders, randomly sampled from a pool of the 4 production recipes available within the ICE Laboratory. Among these production orders, 450 are high-priority (HP) orders, and 50 are low-priority (LP) orders. While the first ones are considered available from time $t = 0$ and can be scheduled at any time, the second ones are considered available after time t', such that $t' > 0$, identifying new production orders released at runtime. The obtained results are reported in Table 7. The first row compares the total makespan obtained with the scheduler. Our service-based scheduler is able to reduce the total makespan by almost 40 minutes (equal to the 3.52%). The next rows compare the average machines' downtimes, utilization, and throughput. The service-based scheduler reduces the average machine downtimes by 14.17% and increases their utilization and throughput by 4.45% and 5.98% respectively.

By having in-depth knowledge regarding the implementation of production recipes on the production plant, the service-based scheduler is able to better exploit machine downtimes. It does so by splitting low-priority tasks into a sequence of sub-tasks (represented within the SysML model).

This allows the scheduling of a smaller portion of production tasks, enabling a more precise interleaving with respect to the state-of-the-art approach. By doing so, the scheduler is free to fill the available gaps within the schedule with a portion of low-priority production tasks. As a consequence, this reduces the total makespan and machine downtimes and increases the machine's utilization and throughput. Lastly, the last two rows compare the average time to complete production recipes at high priority (HP) and low priority (LP). We compute the completion time for a recipe as the difference between the time instant in which the last task ends and the time instant in which its first task starts. Our service-based scheduler is able to handle the arrival of new production orders more efficiently, reducing the completion time for both new high and low priority orders. In general, the scheduler routine in charge of managing the priorities of the tasks is able to handle the allocation of sub-tasks more accurately whether certain conditions are met (e.g., required materials availability).

IX. CONCLUSION

In this paper, we presented a cloud-native architecture to support flexible manufacturing. We also proposed a modeling approach to specify the information required to automatically deploy, configure, and control the proposed architecture. Then, we benchmarked the main communication protocols suitable to implement the proposed architecture. These three main contributions create a unique framework centered around the model, where the production system acts as a service provider and production evolves by invoking the infrastructure services.

The proposed software architecture has been implemented to monitor and control a fully-fledged reconfigurable manufacturing line; the system has been modeled in SysML to automatically deploy and configure the architecture components. Finally, the production requirements have been also modeled in SysML as have also been the production recipes. The proposed architecture has been validated on a fully-fledged production line using different workloads. The results show the architecture introducing very limited performance degradation while enabling a wide range of production optimizations, thus increasing overall productivity.

In the future, we plan to extend the proposed methodology to automatically generate a digital twin from the SysML model. This will enable the development of optimization strategies exploiting the generated digital twin as a testbed.

ACKNOWLEDGMENT

This manuscript reflects only the authors' views and opinions, neither European Union nor European Commission can be considered responsible for them.

REFERENCES

[1] U. D. Atmojo, Z. Salcic, K. I. Wang, and V. Vyatkin, "A service-oriented programming approach for dynamic distributed manufacturing systems," *IEEE Trans. Ind. Informat.*, vol. 16, no. 1, pp. 151–160, Jan. 2020.

[2] F. Li, T. W. Liao, W. Cai, and L. Zhang, "Multitask scheduling in consideration of fuzzy uncertainty of multiple criteria in service-oriented manufacturing," *IEEE Trans. Fuzzy Syst.*, vol. 28, no. 11, pp. 2759–2771, Nov. 2020.

[3] M. Wang, S. Pang, S. Yu, S. Qiao, X. Zhai, and H. Yue, "An optimal production scheme for reconfigurable cloud manufacturing service system," *IEEE Trans. Ind. Informat.*, vol. 18, no. 12, pp. 9037–9046, Dec. 2022.

[4] C. Pu, X. Ding, P. Wang, and Y. Yang, "Practical implementation of an OPC UA multi-server aggregation and management architecture for IIoT," in *Proc. IEEE Int. Conferences Internet Things, IEEE Green Comput. Commun., IEEE Cyber, Phys. Social Comput., IEEE Smart Data, IEEE Congr. Cybermatics*, Aug. 2022, pp. 476–481.

[5] A. L. Ramos, J. V. Ferreira, and J. Barceló, "Model-based systems engineering: An emerging approach for modern systems," *IEEE Trans. Syst. Man, Cybern., Part C*, vol. 42, no. 1, pp. 101–111, Jan. 2012.

[6] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing–A systematic mapping study," *J. Syst. Softw.*, vol. 126, pp. 1–16, Apr. 2017.

[7] W. Babel, "Automation pyramid and solutions business," in *Industry 4.0, China 2025, IoT*. Cham, Switzerland: Springer, 2022, pp. 75–147.

[8] R. Drath and A. Horch, "Industrie 4.0: Hit or hype?" *IEEE Ind. Electron. Mag.*, vol. 8, no. 2, pp. 56–58, Jun. 2014.

[9] *IKnternational Society of Automation*, Standard ISA-95, 2000.

[10] C. D. Brandl and C. Johnsson, *Beyond the Pyramid: Using ISA95 for Industry 4.0 and Smart Manufacturing*. Rijeka, Croatia: InTech, Oct. 2021, pp. 14–20. [Online]. Available: https://www.isa.org/intech-home/2021/october-2021/features/beyond-the-pyramid-using-isa95-for-industry-4-0-an

[11] P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli, "Modeling cyber–physical systems," *Proc. IEEE*, vol. 100, no. 1, pp. 13–28, Jan. 2012.

[12] C.-W. Yang, V. Dubinin, and V. Vyatkin, "Automatic generation of control flow from requirements for distributed smart grid automation control," *IEEE Trans. Ind. Informat.*, vol. 16, no. 1, pp. 403–413, Jan. 2020.

[13] M. Obermeier, S. Braun, and B. Vogel-Heuser, "A model-driven approach on object-oriented PLC programming for manufacturing systems with regard to usability," *IEEE Trans. Ind. Informat.*, vol. 11, no. 3, pp. 790–800, Jun. 2015.

[14] A. Köcher, A. Hayward, and A. Fay, "Model-based engineering of CPPS functions and code generation for skills," in *Proc. IEEE 5th Int. Conf. Ind. Cyber-Phys. Syst. (ICPS)*, May 2022, pp. 1–8, doi: 10.1109/ICPS51978.2022.9816919.

[15] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer, "Modeling languages in industry 4.0: An extended systematic mapping study," *Softw. Syst. Model.*, vol. 19, no. 1, pp. 67–94, Jan. 2020, doi: 10.1007/s10270-019-00757-6.

[16] 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, Oct. 2014.

[17] L. Piétrac, A. Lelevé, and S. Henry, "On the use of SysML for manufacturing execution system design," in *Proc. ETFA*, Sep. 2011, pp. 1–8.

[18] R. Drath, "Let's talk AutomationML what is the effort of AutomationML programming?" in *Proc. IEEE 17th Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2012, pp. 1–8.

[19] *OPC Unified Architecture Specification—Part 1: Overview and Concepts Release 1.04 OPC Foundation*, 2017. [Online]. Available: https://reference.opcfoundation.org/Core/Part1/v104/docs/

[20] M. V. Garcia, E. Irisarri, F. Perez, M. Marcos, and E. Estevez, "From ISA 88/95 meta-models to an OPC UA-based development tool for CPPS under IEC 61499," in *Proc. IEEE WFCS*, 2013, pp. 1–9.

[21] B. Ayaz, N. Slamnik-Kriještorac, and J. Marquez-Barja, "Data management platform for smart orchestration of decentralized and heterogeneous vehicular edge networks," in *Proc. ACM Conf. Inf. Technol. Social Good*, Sep. 2022, pp. 118–124.

[22] V. John and X. Liu, "A survey of distributed message broker queues," 2017, *arXiv:1704.00411*.

[23] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, vol. 11, 2011, pp. 1–7.

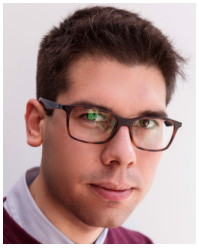
[24] *RabbitMQ*. Accessed: Mar. 14, 2024. [Online]. Available: https://www.rabbitmq.com/

[25] *KubeMQ*. [Online]. Available: https://kubemq.io/
[26] S. Gaiardelli, S. Spellini, M. Panato, M. Lora, and F. Fummi, "A software architecture to control service-oriented manufacturing systems," in *Proc. Design, Autom. Test Eur. Conf. Exhibition (DATE)*, Mar. 2022, pp. 40–43.
[27] Y. Liu, S. Garg, J. Nie, Y. Zhang, Z. Xiong, J. Kang, and M. S. Hossain, "Deep anomaly detection for time-series data in industrial IoT: A communication-efficient on-device federated learning approach," *IEEE Internet Things J.*, vol. 8, no. 8, pp. 6348–6358, Apr. 2021.
[28] Y. Wu, H.-N. Dai, and H. Tang, "Graph neural networks for anomaly detection in industrial Internet of Things," *IEEE Internet Things J.*, vol. 9, no. 12, pp. 9214–9231, Jun. 2022.
[29] L. Zhou, Z. Jiang, N. Geng, Y. Niu, F. Cui, K. Liu, and N. Qi, "Production and operations management for intelligent manufacturing: A systematic literature review," *Int. J. Prod. Res.*, vol. 60, no. 2, pp. 808–846, Jan. 2022, doi: 10.1080/00207543.2021.2017055.
[30] 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," *J. Intell. Manuf.*, vol. 28, no. 7, pp. 1503–1517, Oct. 2017.
[31] 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, no. 4, pp. 253–269, Apr. 2016.
[32] S. Spellini, S. Gaiardelli, M. Lora, and F. Fummi, "Enabling component reuse in model-based system engineering of cyber-physical production systems," in *Proc. 26th IEEE Int. Conf. Emerg. Technol. Factory Autom.*, Sep. 2021, pp. 1–8.
[33] S. Gaiardelli, S. Spellini, M. Lora, and F. Fummi, "A hierarchical modeling approach to improve scheduling of manufacturing processes," in *Proc. IEEE 31st Int. Symp. Ind. Electron. (ISIE)*, Jun. 2022, pp. 226–232.
[34] B. Rabenstein and J. Volz, *Prometheus: A Next-Generation Monitoring System*. Dublin, Ireland: USENIX Association, May 2015.

SEBASTIANO GAIARDELLI (Student Member, IEEE) received the master's degree in computer science and engineering from the University of Verona, Verona, Italy, in 2021, where he is currently pursuing the Ph.D. degree in computer science. He is involved as the Co-Founder and the Scientific Advisor with FACTORYAL S.r.l., a startup specializing in factory automation software that originated as a spin-off from the University of Verona. His research interests include the development of new methodologies for the optimization, reconfiguration, and verification of cyber-physical production systems.

STEFANO SPELLINI received the Ph.D. degree in computer science from the University of Verona, Italy, in 2022, with a thesis proposing a unifying framework to model, verify and optimize production systems. He is currently the Development Team Leader and the Co-Founder of FACTORYAL S.r.l., a spin-off company from the University of Verona. He is involved in the development of modeling methodologies and tools for cyber-physical production systems.

MARCO PANATO received the B.S. and M.E. degrees in computer science and engineering from the University of Verona, Italy, in 2016 and 2018, respectively. He is currently a Software Engineer and the Co-Founder with FACTORYAL S.r.l., San Giovanni Lupatoto, Verona, Italy, a spinoff of the University of Verona. Previously, he was a Research Assistant with the University of Verona.



CARLO TADIELLO received the B.S. and M.E. degrees in computer science and engineering from the University of Verona, Italy, in 2015 and 2018, respectively. He is currently a Software Engineer and the Co-Founder with FACTORYAL S.r.l., San Giovanni Lupatoto, Verona, Italy. Previously, he was a Research Assistant with the ICE Laboratory, University of Verona.

MICHELE LORA (Member, IEEE) received the Ph.D. degree in computer science from the University of Verona, Italy, in 2016. From 2020 to 2023, he held a Marie Skłodowska-Curie Global Fellowship with dual appointments at the University of Verona and the University of Southern California. Previously, he held different research positions in Sweden, the USA, and Singapore. He is currently a Researcher with the University of Verona. He is also involved as the Co-Founder and the Scientific Advisor with FACTORYAL S.r.l., a startup specializing in factory automation software that originated as a spin-off from the University of Verona. His research interests include modeling, simulation, and verification of cyber-physical systems.

DONG SEON CHENG received the Laurea and Ph.D. degrees in computer science from the University of Verona, in 2003 and 2008, respectively, with a focus on computer vision and pattern recognition. From 2012 to 2017, he was an Assistant Professor with the Department of Computer Science and Engineering, Hankuk University of Foreign Studies, South Korea, teaching undergraduate and graduate courses. From 2019 to 2022, he was with SETECNA EPC S.r.l., an electronics company in the HVAC Industry. He has been a Research Associate with the University of Verona, since 2023. His research interest includes machine learning. He has published several journals and conference papers in his research field.

FRANCO FUMMI (Member, IEEE) received the Laurea degree in electronic engineering and the Ph.D. degree in electronic and communication engineering from the Polytechnic of Milan, in 1990 and 1994, respectively. Since March 2001, he has been a Full Professor of computer architecture with Università di Verona. He is also leading the Cyber-Physical and IoT Systems Design (CISD) Group, Università di Verona, where he is composed of more than 20 people and working on hardware description languages and electronic design automation methodologies for modeling, verification, testing, and optimization of cyber-physical systems. He is also the Co-Founder of two spin-off companies: EDALab, focused on networked embedded systems design; and the automation control software company FACTORYAL.

Open Access funding provided by 'Università degli Studi di Verona' within the CRUI CARE Agreement