Exploiting SysML v2 Modeling for Automatic Smart Factories Configuration

Mario Libro[*], Sebastiano Gaiardelli[*], Marco Panato[†], Stefano Spellini[†], Michele Lora[*], Franco Fummi[*]
[*]*Department of Engineering for Innovation Medicine*, University of Verona, Italy, `name.surname@univr.it`
[†]FACTORYAL S.r.l., San Giovanni Lupatoto, Italy, `name.surname@factoryal.it`

***Abstract*—Smart factories are complex environments equipped with both production machinery and computing devices that collect, share, and analyze data. For this reason, the modeling of today's factories can no longer rely on traditional methods, and computer engineering tools, such as SysML, must be employed. At the same time, the current SysML v1.* standard does not provide the rigorousness required to model the complexity and the criticalities of a smart factory.**

Recently, SysML v2 has been proposed and is about to be released as the new version of the standard. Its release candidate version shows the new version aims at providing a more rigorous and complete modeling language, able to fulfill the requirements of the smart factory domain. In this paper, we explore the capabilities of the new SysML v2 standard by building a rigorous modeling strategy, able to capture the aspects of a smart factory related to the production process, the computation and the communication. We apply the proposed strategy to model a fully-fledged smart factory, and we rely on models to automatically configure the different pieces of equipment and software components in the factory.

***Index Terms*—Model-Based System Engineering, SysML v2, Smart Manufacturing.**

I. Introduction

Modeling is a fundamental activity when designing and maintaining a complex advanced manufacturing system, as it is for industrial cyber-physical systems in general [1]. However, due to historical reasons, languages typically used to model industrial plants and manufacturing systems are not well suited to represent the complexity of modern smart factories. Typically, modeling languages in manufacturing lack the capabilities required to represent the computational and communication aspects of the system. Thus, looking at languages like SysML seems a natural choice to take the ongoing digital transformation of factories. Many approaches have been proposed to model smart factories using SysML [2]–[5]. However, the current standard of SysML v1.* is on the one hand, very expressive and intuitive, but on the other hand, it lacks the rigorousness and formality required to model critical systems like smart factories. To tackle this issue, the OMG consortium is working on the next version of SysML, i.e., SysML v2, which is expected to provide more rigorousness while preserving the expressiveness and intuitiveness of the current version of SysML [6]. SysML v2 is undergoing the standardization process, and it is expected to be released by the end of 2024. In this paper, we explore the capabilities of the upcoming SysML v2 standard to model smart factories, and we propose a modeling strategy to capture the necessary information to model the machines' functionalities, the data being collected and exchanged in the system. To evaluate whether the new standard is capable of capturing the necessary information, we use the produced models to automatically generate the configuration files to be deployed in the software monitoring and controlling the production system.

This study was carried out within the MICS (Made in Italy – Circular and Sustainable) Extended Partnership and received funding from Next-Generation EU (Italian PNRR – M4 C2, Invest 1.3 – D.D. 1551.11-10-2022, PE00000004). CUP MICS D43C22003120001 - Cascade funding project CollaborICE. It has been also partially supported by the PRIN 2022T7YSHJ SMART-IC - Next Generation EU project. This manuscript reflects only the Authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

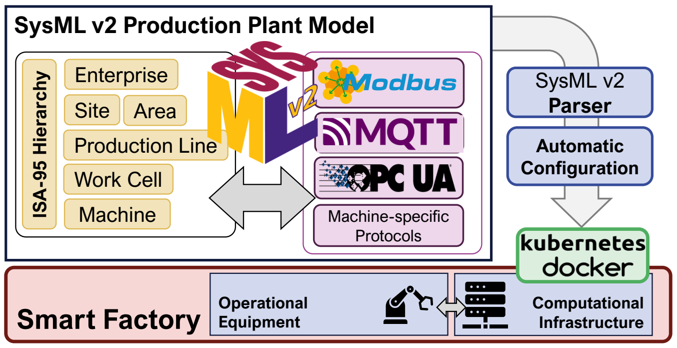

Figure 1. Overview of the contribution: a smart factory and its equipment are modeled in SysML v2 according to the ISA-95 hierarchy, specifying the involved communication protocols. An automatic toolchain exploits the model to automatically configure the modeled smart factory.

Figure 1 shows an overview of the proposed methodology. A SysML v2 model is used to represent the machines in the factory. The model of the equipment is organized hierarchically according to the principles defined in the ISA-95 standard. The model also contains the specifications of the communication protocols, and data model of the data exchanged by the different pieces of equipment. The SysML models produced according to the proposed modeling strategy are input to an automatic generation and deploy methodology, inspired by the SysML v1.*-based methodology proposed in [5], but adapted to the new SysML v2 specification. The automatic configuration methodology generates the configuration files for the software being deployed to control and monitor the factory, and it distributes the software using container technology.

The main contribution is threefold:

- an evaluation of the expressiveness of the upcoming SysML v2 standard in the manufacturing context;
- a modeling approach based on this evaluation;

- the adaptation of [5] to accommodate the modeling paradigms shift introduced by SysML v2.

The proposed modeling, automatic generation, and deployment methodology has been applied to a fully operational smart factory, the ICE Laboratory. The current version of SysML v2 proved capable of capturing the necessary information to model the data collection, communication, and processing aspects of a complete factory, while also enabling automation. Thus, showcasing that SysML v2 overcomes the limitations of its predecessor, and its potential for becoming a predominant modeling language for smart factories and, in general, Cyber-Physical Production Systems (CPPSs).

II. Background

Service-oriented Manufacturing (SOM) is a promising paradigm for software architectures in CPPS [7]. It is a revamp of the well-known Service Oriented Architecture (SOA) paradigm for the manufacturing domain. In SOAs, each software component offers a set of *microservices* to the other components within the architecture to enhance the modularity and resilience of the architecture [8]. Similarly, SOM consists of a set of machinery exposing their functionalities as a set of *machine services*, and production processes are composed of sequences of machine services [5].

While the implementation of SOAs relies on REST APIs or message brokers to offer microservices within the architecture, SOM needs to deal with the industrial protocols supported by the machinery as communication interfaces. The most diffused industrial protocols are OPC Unified Architecture (OPC UA), Modbus, EtherCAT, Profinet, and more [9]. Most of these protocols were designed for point-to-point communication. As such they are not easily integrable within SOM architectures. To enable SOM in modern manufacturing systems, a unifying layer is needed with the role of abstracting the communication between high-level control software and the machinery as explored by Gaiardelli et al. in [5]. Indeed, a system based on SOM is harder to model than a classical manufacturing system. For this reason, the methodology proposed in this work aims at modeling SOM-based systems.

This section presents some background, the related work, and the guiding example used to develop the contribution.

A. The System Modeling Language: SysML

CPPSs design often relies on Model-based System Engineering (MBSE) methodologies, which guide system engineers from the definition of the requirements to system implementation [10]. In the last decade, SysML emerged as a powerful modeling language to support the design of many types of systems, including manufacturing systems [11]. The 1.* version of SysML is based on Unified Modeling Language (UML) that allows the representation of the diverse aspects of complex systems through the usage of diagrams.

Currently, a new release of the SysML language, SysML v2 is going through the final standardization steps [12]. SysML v2 overcomes some major limitations of the previous version by increasing interoperability, adaptability, and rigorousness [6], [13]. It is no longer based on UML, but it now lays is foundations in Kernel Modeling Language (KerML): a modeling language containing the core constructs used to implement other modeling languages. KerML is based on the *definition* and *usage* paradigm, where definitions are used to define types of elements, while usages are used to specify a definition in a specific context. As such, SysML v2 is defined as a library of KerML defining the constructs of the language and its semantics, such as *packages*, *parts*, *attributes*, *ports*, and *connections*. The structure of a SysML v2 model is defined by the packages used to organize the model elements. Parts are used to define the components of the systems, which may have a set of attributes and ports representing parts' states and interactions. Moreover, parts may perform actions and exhibit states representing the components' behavior.

B. Modeling of Smart Factories

Several industrial standards have been created over the years aiming to tame the complexity of manufacturing information systems by unifying the knowledge representation among manufacturing software. One of the most widely adopted is the ISA-95 standards, also known as IEC 62264, which defines the widely adopted automation pyramid [14] categorizing into five layers the manufacturing software. The layers range from the lowest level, where sensors and actuators are located to the highest level, where enterprise information systems are located. The ISA-95 standard also defines the terminology and data structures that manufacturing software should use within the different levels.

The encapsulation of these standards within the modeling of manufacturing systems is fundamental to ensure consistency of terminology and compatibility with other existing software [5]. To the best of our knowledge, this is the first comprehensive factory modeling approach compliant with the industrial standards exploiting the new features provided by SysML v2.

C. Guiding Example: 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]. It consists of a fully-fledged production line comprising several machines, such as a robotic workcell for assembly, a quality control cell, 3D printers, a milling machine, a and an electronic functional testing machine. Materials are stored in a vertical warehouse and transported to the working cells by exploiting two Automated Guided Vehicles (AGVs) and a minipallet conveyor belt. Each machine differs in terms of communication protocols, data formats, and control interfaces, making the integration of the production line a challenging task. To overcome this issue, each machine is equipped with an OPC UA server, which uniforms the communication interface with the machinery, hiding the drivers' complexity. The OPC UA servers are connected to a SOM architecture via a central message broker interconnecting all the machinery, equipment and software in the system, such as the control software and the databases.

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

III. Modeling Methodology

A plant consists of production lines, each production line consists of multiple workcells, and each workcell may be composed of one or more machines. The methodology follows the principles of ISA-95 standard, organizing models of the different components and aspects of the production system in a hierarchical structure capturing components from the enterprise level down to the individual machines in workcells.

The methodology starts by defining the main components describing the factory structure. The two main concepts to be captured are the *Machine* and the *Driver*. A *Machine* represents a piece of equipment, detailing the services it exposes by modeling the machine's variables, parameters, and operations. A *Driver* represents the communication protocol that the machine uses to interface with the system. A *Driver* could be machine-proprietary or standardized. Subsequently, the components that establish information exchange channels between machines and drivers must be defined. Information exchange channels are modeled using structural SysML v2 constructs, such as *interfaces*, *ports*, and *connections*.

Once defined, *parts* are instantiated to create the actual model of the factory. This involves specifying real-world *attributes* for the machines and drivers, and linking them through the defined *connections* to represent the actual operational setup of the workcells. This comprehensive modeling approach ensures that all aspects of the workcells, including their machines and communication protocols, are accurately captured and can be effectively monitored and controlled.

We detail the proposed modeling methodology, pairing the explanation with its application to a running example. The running example is based on the subtractive manufacturing workcell in the ICE Laboratory, composed of a EMCO Concept Mill 105 milling machine, and a UR5e collaborative robot. The UR5e robot interacts with the milling machine by handling the loading and unloading of workpieces.

A. Definition of the General Structure

The hierarchical nature of production systems is captured in SysML v2 by utilizing nested *part definitions*, representing the components and sub-components within the system. Each *part definition* follows the hierarchical levels of the ISA-95 model, from the top-level entity down to specific machines, as depicted in Code 1: the top-level component is the `Topology`, which contains nested `Enterprise`, `Site`, `Area`, `ProductionLine`, `Workcell`, and `Machine` parts. The `Machine` is defined as a *referential part*, meaning it references existing machine definitions within the system rather than creating new instances. The $\star$ symbol in `Machine[*]` indicates that the `Workcell` can reference multiple machines, representing a set of machines within the workcell rather than a single instance. This provides flexibility to model dynamic configurations of machines within each workcell, aligning with the modular nature of ISA-95. Additionally, *attributes* are defined at the `ProductionLine` and `Workcell` levels. These attributes serve to monitor aggregated information relevant across the entire production line or work cell, such as performance metrics or overall energy consumption.

```
part def Topology {
    part def Enterprise {
        part def Site {
            part def Area {
                part def ProductionLine {
                attribute def ProductionLineVariables;
                part def Workcell {
                    ref part Machine [*];
                    part def WorkCellVariables;
                } ...
```

Code 1. Hierarchical structure for modeling an industrial system according to the ISA-95 standard. At each hierarchical level, variables can be defined to capture operational information relevant to the specific layer.

Navigating the hierarchy, a workcell is typically composed of machines. A machine is initially represented as *abstract part* and later specialized to model a specific machine. A `Machine` defines the *parts* `MachineData` and `MachineServices` capturing the machine's operational characteristics and the services each instance of the machine provides. The `Machine` part also contains a *referential part* `Driver` defining the communication protocol used by the machine. Separating the definitions of machines and drivers increases the flexibility of modeling the associations between machines and their respective communication protocols.

The `Driver` part is defined as an *abstract part*: it serves as a template that cannot be instantiated on its own, thus forcing the definition of specialized drivers that inherit common properties while adding driver-specific details. The abstract `Driver` definition consists of three main sub-parts: `DriverParameters`, `DriverVariables`, and `DriverMethods`. These sub-parts are essential for configuring and managing the communication between the machines. The communication protocols can either be proprietary or standardized.

For this purpose, the part `Driver` is specialized in two different abstract parts: `GenericDriver`, representing a generalized driver for standardized communication protocols, and `MachineDriver`, representing a machine-specific proprietary driver. Typically, heterogeneous machines must collaborate within the same factory, despite using different communication protocols.

B. Driver Specialization Definition

The defined *abstract parts* `MachineDriver` and `GenericDriver` need to be specialized to represent specific communication protocols, whether standardized or proprietary. Specialization in SysML v2 is expressed via the `specialize` keyword or shorthand `:>`, allowing to extend and refine the abstract structure, tailoring it to the specific communication protocol being modeled. This approach requires the refinement of the internal components, particularly variables, parameters, and methods, to match the specification of the chosen communication protocol.

The `DriverVariables` part is specialized to capture data produced by the machine, such as status updates, sensor readings, and real-time data. These variables, which include indicators like the machine's operational mode, program status, and error states, are essential for monitoring and controlling the machine's performance. Variables can be organized hierarchically based on their functional roles, grouping them through *part definitions*. The specific variables are also tailored to the communication protocol in use.

Each variable is modeled using a *port*. Ports enable the exposure of their *attributes* and provide a mechanism for defining data flow directions (*in*, *out*, or *inout*). Before the ports are instantiated, the SysML v2 *port definition* construct is used to define the internal structure of each port. The *usage* of these ports is detailed in Section III-E. Each *port definition* specifies a set of attributes, including the variable's value, type, and the associated metadata providing essential details describing the context and purpose of the variable within the system. The attribute direction is set to input, as the data originates from the machine.

The `DriverParameters` part is also specialized by defining its internal structure: the parameters represent static configuration data that does not change during runtime; these are modeled as attributes rather than ports. The specified attributes represent static configuration parameters necessary for establishing a communication channel with the machine, like IP address, port, or protocol-specific configuration details.

Building upon this structure, the `DriverMethods` part is specialized and used to allow the execution of all the methods available for controlling or querying the machine via the communication protocol being modeled. Methods can include operations, such as, checking the machine's status or executing a specific machine operation. Similar to the `DriverVariables`, the `DriverMethods` specify a *port definition*. Unlike the ports defined to represent variables, method ports encapsulate method-related functionality. The *port definition* includes metadata attributes describing the method, e.g., purpose, category, identifier.

Additionally, the `action` construct is employed within the port definition to represent the concept of invoking a method. The `action` specifies the method's input arguments and output return parameters, thereby defining the data exchange involved in performing the operation. Using ports allows for the exposure of attributes and actions, promoting modularity and reusability. Ports act as standardized interfaces, facilitating seamless communication between machines and drivers.

Code 2 exemplifies the specialization of the `MachineDriver` into `EMCODriver`. `EMCODriver` is the definition of the communication protocol used by the EMCO milling machine. The specialization defines the parameters as attributes to configure the communication channel for the milling machine. The internal structure of the ports `EMCOMethod` and `EMCOVar` is defined. Variables are organized into functional categories such as `AxesPositions` and `SystemStatus`.

```
part def EMCODriver :> MachineDriver {
    part def EMCOParameters :> DriverParameters {
        attribute ip : String;
        attribute ip_port : Integer;
        attribute program_file_path : String;
    }
    part def EMCOVariables :> DriverVariables {
        port def EMCOVar { //input attribute }
        part def AxesPositions;
        part def SystemStatus;
    }
    part def EMCOMethods :> DriverMethods {
        port def EMCOMethod {
            attribute description:String;
            out action operation {
            //arguments and returns attribute
        }...
```

Code 2. *Part definition* of the `EMCODriver` as a specialization of `MachineDriver`. The driver comprehends three parts: `EMCOParameters`, `EMCOVariables`, and `EMCOMethods`.

C. Machine Specialization Definition

The abstract `Machine` part is specialized to represent specific physical machines within the industrial system. The specialization of `Machine` follows a similar process as for `Driver`. The predefined structure of the `Machine` part is refined using *part definitions*, focusing on two key components: `MachineData` and `MachineServices`.

`MachineData` represents all the data produced and exposed by the machine that can be accessed via the driver. Its specialization is similar to the `DriverVariables` specialization. *Part definitions* are used to categorize and group related machine data. Within each *part* representing a data category, a series of *ports* is instantiated, with each port corresponding to a specific machine data point. The ports used are the conjugated versions of those defined for the specialization of `DriverVariables`, retaining the same predefined *attributes*. Ports conjugation, represented by the tilde symbol (~), reverses the direction of a port's attributes. By using the construct `redefines` or its shorthand `:»`, the *value* attribute, which serves as the container for the machine data, is assigned a type. In contrast, the metadata attributes describing the machine data are assigned specific, fixed values, as they serve to consistently define the context and characteristics of the variable throughout the model.

Similarly, the `MachineServices` part is specialized following the same approach used for `MachineData`. This part specialization represents the commands and operations that the machine provides. Each service is modeled using ports. The ports used are conjugated versions of those defined for the specialization of `DriverMethods`, ensuring that the direction of data flow aligns with the driver's expectations.

Within each port, depending on the specific service being modeled, the input (arguments) and output (return values) attributes are assigned to specific types using the `redefines` keyword, ensuring that the correct data types are specified for both the required inputs and the returned outputs.

Code 3 exemplifies the specialization of the `Machine` part into `EMCOMillingMachine`. Two subparts are defined for the `MachineData` of the milling machine: the `AxesPositions` part models the positions of the machine's spindle along its axes; the `SystemStatus` part models the machine's operational status. Two ports get specified: one for `MachineData` and another for `MachineServices`. `EMCOMachineData` uses the predefined `EMCOVar` port type, `EMCOServices` uses the predefined `EMCOMethod` port type. Both ports are conjugated.

```
part def EMCOMillingMachine :> Machine {
    part def EMCOMachineData :> MachineData {
        part def AxesPositions {
            port actual_X_EMCOVar_conj : ~EMCOVar {
                //attributes assignments
            }...
        }
        part def SystemStatus {...}
    }
    part def EMCOServices :> MachineServices {
        port is_ready_EMCOMthd : ~EMCOMethod {
            //attributes assignments
    }...
```

Code 3. Definition of the `EMCOMillingMachine` part as a specialization of `Machine`. The code outlines the structure of the milling machine within the model, focusing on the representation of variables (`EMCOMachineData`) and services (`EMCOServices`) exposed by the machine.

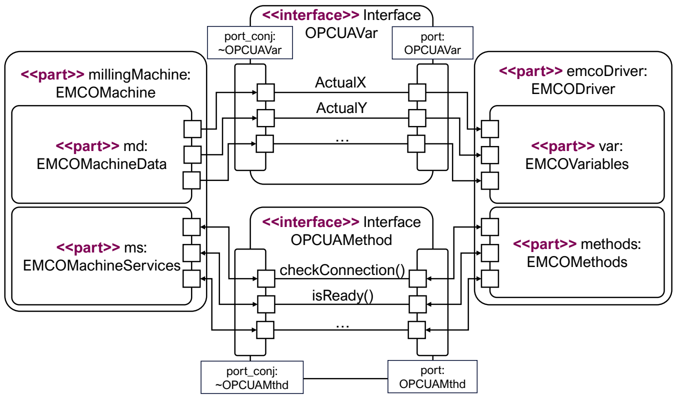

Figure 2. Overview of the communication channel between the milling machine and its driver. The machine consists of `MachineData` and `MachineServices`, each with ports. The driver includes `DriverVariables` and `DriverMethods`, also with ports. Two interfaces connect the machine data ports to the driver variables and the machine services ports to the driver methods.

```
part ICETopology : Topology {
  part UniVR : Enterprise {
    part Verona : Site {
      part ICELab : Area {
        part ICEProductionLine : ProductionLine {
          part workCell02 : Workcell {
            part emco : EMCO {
              ref part emcoDriver;
            part emcoMachineData : EMCOMachineData {
              part emcoAxesPosition : AxesPositions {
                attribute actualX : Double;
                bind actual_X_EMCOVar_conj.value=actualX;
                ...
              part emcoSystemStatus:SystemStatus {...}
              ...
            part emcoServices : EMCOServices {
              action isReady {out ready : Boolean;}
              ...
          //ur5 machine instantiation
          ...
```

Code 4. Instantiation of the `ICETopology` part, representing the system hierarchy of a specific smart factory.

```
part emcoDriver : EMCODriver{
    part emcoParameters : EMCOParameters{
        :>> ip = '10.197.12.11';
        :>> ip_port = 5557;
        :>> program_file_path = 'path/program/file';
    }
    part emcoVariables : EMCOVariables{
        part emcoSystemStatus : SystemStatus;
        part emcoAxesPositions : AxesPositions{
            attribute actualX : Double;
            port pp_actual_X_EMCOVar : EMCOVar;
            bind pp_actual_X_EMCOVar.value = actualX;
            ...
    part emcoMethods : EMCOMethods{
        action call_is_ready {
            out ready : Boolean;
            perform pp_is_ready_EMCOMthd.operation{
                out ready = call_is_ready.ready;
            }...
```

Code 5. Instantiation of the `emcoDriver` part used by the EMCO machine. The driver includes driver-specific parameters, variables and methods.

D. Connections Definition

Connections between parts are defined by specifying how the ports of the machine and the driver interact using *interface* and *connection* definitions. Interfaces are defined to connect the standard ports to their conjugated counterparts, specifying the flow of data between the machine and the driver and binding the attributes of each port.

After defining the *interface*, *connections* are established to link the machine and driver parts. The mapping is defined between the `MachineData` of the machine and the `DriverVariables` of the driver. Similarly, the `DriverMethods` of the driver are connected to the `MachineServices` of the machine.

Interfaces and connections define how data and commands are transmitted between the machine and the driver. Figure 2 provides an overview of ports, interfaces, and connections.

E. Component Instantiation

The final step in the modeling methodology is to instantiate these components to represent the actual machines and drivers within the model of the industrial system, instantiating specific instances of the specialized parts defined above.

As shown in Code 4, the instantiation starts with the top-level part `ICETopology`. In `workCell02`, the EMCO milling machine is instantiated as `emco`, and the UR5e collaborative robot is instantiated as `ur5`. Each machine is associated with its respective driver, `emcoDriver` and `universalRobotDriver`, which are specialized to interface with the EMCO and UR5e machines respectively, as shown in Code 5. Each machine defines its own machine data and services. For each subpart of `MachineData`, such as `emcoAxesPosition`, all *attributes* are defined and bound to corresponding *ports*. A similar approach is applied to `MachineServices`, using *actions* rather than attributes.

In particular, within the `emcoMachineData` part, an attribute is defined for each variable exposed by the machine. Attributes are bound to their corresponding ports to ensure their values are accessible to the driver. For example, the attribute `actualX` is bound to the port `actual_X_EMCOVar_conj.value`, facilitating the transmission of data between the machine and the driver.

The actions of the `emcoServices` part represent machine's services, i.e., `isReady` is an action checking whether the machine is either ready or busy. The action is exposed through ports, enabling it to be invoked directly by the driver.

Similarly, the driver's components are instantiated as `emcoDriver` and `urDriver`, along with their parameters, variables, and methods. Variables and methods are specified and bound to the corresponding ports, as shown in Code 5.

Finally, connections defined in Section III-D are instantiated by linking and enabling exchange of information between the driver and the machine.

Table I
FEATURES OF THE ICE LAB SYSML V2 MODEL AND OF THE RESULTING CONFIGURATION FILES.

WC	Machine	Driver	Part		Attributes Inst.	Ports Inst.	Machine Variables	Machine Services
			Def.	Inst.				
01	SPEA ATE	OPC UA	9	8	48	16	3	5
02	EMCO Milling	Machine Driver	12	17	238	106	34	19
	UR5e Cobot	Machine Driver	23	17	611	206	99	4
03	Siemens PLC	OPC UA	31	82	194	68	26	8
	Fiam eTensil	OPC UA	11	28	82	24	12	3
04	Quality Control PC	OPC UA	10	9	85	30	13	2
05	Vertical Warehouse	OPC UA	10	9	44	16	5	3
06	Conveyor Line	OPC UA	144	143	1220	612	296	10
	RB-Kairos	OPC UA	11	18	48	14	5	6
	RB-Kairos	OPC UA	11	18	48	14	5	6

Generation Time (s)	# OPC UA Server	# OPC UA Clients	Config. Size (KB)
3.19	6	4	697

IV. MODELS APPLICATION AND EXPERIMENTAL RESULTS

We used SysML models to automatically generate the configuration of the software stack controlling the ICE Laboratory. To do so, *FACTORYAL S.r.l.*, developed a proof-of-concept tool to automatically generate the configuration files required when deploying the software stack. The software stack is in charge of collecting data from the machinery, exposing their machine services within the architecture, and storing the data within the databases. It comprises the OPC UA servers for each machinery, the OPC UA clients connecting the OPC UA servers to the message broker, and the software components storing the machinery data within the databases.

The automatic generation process is based on the one proposed in [5], but it is adapted to support SysML v2 instead of SysML v1.*. The process consists of two steps:

1) A set of intermediate JSON files is produced from the SysML model of the factory. The tool explores the represented ISA-95 topology of the manufacturing system, and generates a JSON file for each `Machine`. The JSON file contains the information needed to configure their respective OPC UA server and the connection parameters with the machine drivers. To avoid wasting resources of the Kubernetes cluster the configuration of the OPC UA clients connecting the machines to the message broker follows a different approach. The number of OPC UA clients connecting the machinery to the architecture is minimized by connecting multiple machines to the same client. This is done by grouping multiple machines by considering the maximum number of variables and methods supported by each OPC UA client module. For each group of machines, the tool generates two JSON files containing the information to configure the OPC UA client and the software component storing the data in the databases.
2) The JSON files generated by the first step are used to produce the configuration files for the Kubernetes cluster. For each JSON file, the tool generates a YAML file containing the definition of all the resources required by the software component in the Kubernetes cluster. This is done by using template files rendered according to the information contained in the JSON files.

A. Experimental Results

The proposed modeling methodology has been applied to create a complete model of the ICE Laboratory's production system. Then, the model has been used to automatically configure the equipment and machinery of the production system. Table I provides a comprehensive summary of the SysML v2 elements used in the model. Each line corresponds to a specific machine and its associated driver, grouped by the workcell they belong to. The *WC* column identifies the workcell, the *Machine* and *Driver* columns specify the machines being modeled and their corresponding communication protocols; the *Part* column reports the number of *part definitions* and *part instances* used to model each machine and driver; the *Attribute Instances* and *Port Instances* columns report the number of instantiated *attributes* and *port usages* that define the machine and driver's interactions; the *Machine Variables* and *Machine Services* columns the number of variables and services exposed by each machine. The automatic generation process yielded the results shown in the last row of Table I. The total time required to generate the configuration files for the entire system was 3.19 seconds. This process involved creating an OPC UA server for each workcell in the system, resulting in 6 OPC UA servers in total. Additionally, 4 OPC UA clients were generated to handle connections between the machinery and the architecture, optimizing resource usage by grouping multiple machines per client. The final size of the configuration files produced was 697 KB, which would have been manually written by engineers if not automatically generated from the SysML models. The deployment on the production system was successful and the automatically generated configuration enables all the functionalities of the production line. Thus, SysML v2 proved itself being capable of capturing all the aspects of the used smart manufacturing system.

V. CONCLUSIONS

In this paper, we explored the capabilities of the soon-to-be standard SysML v2 in capturing the information required to model a smart manufacturing system. We did so by proposing an approach for modeling SOM architectures. The proposed methodology supports modeling heterogeneous machines and machine drivers. Then, we showed how the information within the model can be exploited to automatically generate the configuration files needed to deploy the software components necessary to integrate the machines within a SOM architecture.

The methodology has been validated on a real manufacturing system, comprising several heterogeneous machinery. The results demonstrate the effectiveness of the proposed methodology in reducing the time required to configure the entire software stack and ensuring consistency between the SysML model and the actual implementation.

References

[1] 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.

[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 *Proceedings of 2021 Forum on specification & Design Languages (FDL)*, 2021, pp. 01–08.

[3] P. Bareiß, D. Schütz, R. Priego, M. Marcos, and B. Vogel-Heuser, “A model-based failure recovery approach for automated production systems combining SysML and industrial standards,” in *Proceedings of 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–7.

[4] S. Spellini, S. Gaiardelli, M. Lora, and F. Fummi, “Enabling Component Reuse in Model-based System Engineering of Cyber-Physical Production Systems,” in *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021, pp. 1–8.

[5] S. Gaiardelli, S. Spellini, M. Panato, C. Tadiello, M. Lora, D. S. Cheng, and F. Fummi, “Enabling Service-oriented Manufacturing through Architectures, Models and Protocols,” *IEEE Access*, 2024.

[6] S. Friedenthal, “Future Directions for MBSE with SysML v2,” in *Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering - MODELSWARD*, INSTICC. SciTePress, 2023, pp. 5–9.

[7] F. Tao and Q. Qi, “New IT Driven Service-Oriented Smart Manufacturing: Framework and Characteristics,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 1, pp. 81–91, 2019.

[8] N. Niknejad, W. Ismail, I. Ghani, B. Nazari, M. Bahari, and A. R. B. C. Hussin, “Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation,” *Information Systems*, vol. 91, p. 101491, 2020.

[9] “OPC Unified Architecture specification – Part 1: Overview and concepts release 1.04 OPC Foundation,” 2017.

[10] A. L. Ramos, J. V. Ferreira, and J. Barceló, “Model-Based Systems Engineering: An Emerging Approach for Modern Systems,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 1, pp. 101–111, 2012.

[11] 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.

[12] OMG, “System Modeling Language v2.0 beta 2,” Apr. 2024, accessed: 2024-09-20. [Online]. Available: https://www.omg.org/spec/SysML

[13] M. Bajaj, S. Friedenthal, and E. Seidewitz, “Systems modeling language (SysML v2) support for digital engineering,” *INSIGHT*, vol. 25, no. 1, pp. 19–24, 2022.

[14] International Society of Automation, “ISA95, Enterprise-Control System Integration,” 2013, accessed: 2024-09-20. [Online]. Available: https://www.isa.org/standards-and-publications/isa-standards/isa-standards-committees/isa95