Optimizing Smart Factory Operations: A Methodological Approach to Industrial System Implementation based on OPC-UA

Henry O. Velesaca[1,2,*], *Juan A.* Holgado-Terriza[2], and *Jose M.* Gutierrez Guerrero[3]

[1]ESPOL Polytechnic University, Escuela Superior Politécnica del Litoral, ESPOL, Guayaquil-Ecuador
[2]Software Engineering Department, University of Granada, 18014, Granada, Spain
[3]Abbott Laboratories, 18004, Granada, Spain

Abstract. The article presents a comprehensive methodology for deploying OPC-UA models as a standard communication protocol, emphasizing their key role in improving near real-time data exchange and operational efficiency within industrial systems. A case study centered on a continuous flow scale system within a grain factory that handles commodities such as corn, soybeans, and wheat, illustrates how OPC-UA significantly improves speed, precision, and consistency in weight measurements, thereby fostering a smarter and more sustainable agricultural future. The primary objective of the study is to provide a roadmap for the development of industrial system controls leveraging OPC-UA architecture. This involves delineating and implementing control modules based on OPC-UA, utilizing cost-effective solutions and high-level programming languages for creating servers and clients (e.g., Python, Java, Android, Node-RED). By seamlessly integrating UML-based design methodologies with OPC-UA, the article advocates for streamlined and standardized development processes, particularly within the scope of Industry 4.0-driven smart factories. The code is available at GitHub: https://github.com/hvelesaca/OPC-UA-methodology, facilitating further research.

1 Introduction

The era of Industry 4.0 heralds the rise of smart factories, epitomizing modern manufacturing with its fusion of automation, connectivity, and data-driven decision-making [1]. At the core of smart factory success lies seamless information exchange between disparate systems, enabling near real-time monitoring, control, and analysis. Industrial communication protocols, such as field buses (e.g., Profibus, Modbus) or industrial Ethernet, and standards such as classic OPC, OPC DA and newer standards like OPC UA, play a key role in providing standardized frameworks for interoperability and integration across heterogeneous environments [2].

In industrial systems, the application of methodologies is important for the efficient design, construction, and deployment of solutions [3]. These methodologies, spanning activities from requirements analysis to maintenance, provide structured approaches to managing resources, timelines, and budgets, while addressing risk management and ensuring quality

*Corresponding author: hvelesac@espol.edu.ec

[4]. Model-driven development methodologies, such as Model-Driven Architecture (MDA) [5], Domain-Specific Modeling (DSM) [6], and the Unified Modeling Language (UML) [7], foster coherence, traceability, and reuse throughout the development lifecycle. The use of methodologies in industrial systems development is necessary to ensure efficiency, quality, and effective management of the software lifecycle in industrial environments.

Seamless communication among systems, devices, and components is imperative to improve operational efficiency and productivity in modern industrial landscapes [8]. Standardized communication protocols such as OPC-UA [9], Modbus [10], Profibus [11], and MQTT [12] play a crucial role in enabling interoperability, reliability, and scalability. These protocols ensure coherence, compatibility, and clear guidelines for data sharing and error management, empowering organizations to effectively address changing business needs and technological advancements [13].

This article presents a methodology for developing and deploying OPC-UA models to establish a standardized, secure, and interoperable communication framework for data exchange and control between PLCs and vertical IT systems (e.g., SCADA, MES) in industrial environments. Furthermore, the proposed methodology is applied to the development of the OPC-UA platform as a case study for a continuous flow scale system in a grain marketing company.

To address this work in detail, the manuscript is organized as follows. Section 2 introduces some related works of design methodologies based on model-driven industrial systems development and communication protocols in industrial processes. Section 3 presents the proposed methodology for the implementation of OPC-UA in industrial applications. Then, section 4 illustrates how the methodology can be applied to developing the OPC-UA platform using a system of continuous flow scales as a case study. Finally, conclusions are presented in Section 5.

2 Background

The development of industrial systems usually involves a three-layered approach, where OPC-UA serves as a bridge between the machine layer (e.g., PLCs, physical devices, ...) and vertical enterprise-level IT systems. This section summarizes some of the most relevant methodologies and techniques proposed for the development of industrial systems based on model-driven approaches and the critical role of communication protocols in industrial systems such as OPC-UA.

2.1 Design Methodologies for Model-Driven Industrial Systems Development

Model-driven development is used as an approach in the development of industrial systems, offering structured methodologies to streamline the design, implementation, and maintenance of complex systems. These methodologies provide systematic frameworks for modeling various aspects of industrial systems, including requirements, architecture, behavior, and data. Key methodologies in this domain include the Model-Driven Architecture (MDA) [5], Domain-Specific Modeling (DSM) [6], and Unified Modeling Language (UML) [7], each offering specific techniques and tools tailored to industrial system development.

For instance, Binder et al. [14] present a MDA process designed for Industry 4.0 applications, emphasizing the central role of models across the development lifecycle. This systematic approach aims to streamline the design, implementation, and evolution of Industry 4.0 applications by adhering to MDA principles. Through standardized modeling languages and tools, the process promotes collaboration, interoperability, and scalability, ensuring alignment with evolving technologies and business needs.

Teilans et al. [15] propose a method for risk assessment in cyber-physical systems (CPS) using DSM and simulation. By employing DSM and simulations, organizations can better identify and analyze risks linked with CPS deployments. This method facilitates a thorough understanding of the vulnerabilities, dependencies, and potential failure scenarios of the system, allowing proactive risk mitigation strategies. Through the integration of DSM and simulation, the approach provides a structured and systematic framework for the evaluation of CPS risk, ultimately improving system resilience and security.

On the other hand, Bruccoleri et al. [16] propose an object-oriented approach to analyze and design flexible manufacturing control systems (FMCS) using UML. By adopting this approach, organizations can develop a structured framework for modeling FMCS components and interactions, leading to more effective analysis and design processes. Leveraging UML facilitates the representation of FMCS elements, such as machines, processes, and control strategies, in a standardized and intuitive manner. This approach generates the model as a result in a diagram in UML format.

Finally, based on object-oriented principles, Gutierrez et al. [17] propose a metamodel iMMAS for conceptualizing the industrial automation systems with a concrete syntax and specific semantics that simplify the development and deployment of manufacturing control systems. In these systems, the models can be transformed into PLC programs and OPC-UA data models.

2.2 Communication Protocols in Industrial Processes

In industrial processes, effective communication between different systems, devices, and components is crucial to ensure smooth operations and maximize productivity. With the advent of digitalization and Industry 4.0, standardized communication protocols have become indispensable for facilitating seamless data exchange and interoperability across various industrial environments. This section provides an overview of related works focusing on component communications within a factory, particularly in the context of Industry 4.0, and discusses different protocols.

The first protocol presented is Open Platform Communications Unified Architecture (OPC-UA) [9], which is highlighted as a prominent communication protocol for industrial automation and control systems. It enables seamless data exchange between sensors, machines, and quality control systems, facilitating near real-time monitoring and analysis of production processes from supervisory systems. Various studies, including one by Martinov et al. [18], showcase OPC-UA's effectiveness in different manufacturing sectors. Martinov et al. demonstrate the implementation of OPC-UA to monitor equipment with variable kinematics, collecting data from CNC systems without requiring a separate adapter for the OPC server. The "Publisher-Subscriber" model in OPC-UA streamlines client-side data retrieval, enabling efficient modification of the kinematic schema. Nonetheless, challenges may arise in environments with unreliable network connectivity or limited availability of OPC-UA servers.

Although OPC-UA is recognized for its advantages in interoperability and security, alternative communication protocols have also been explored in the context of smart factories. Message Queuing Telemetry Transport (MQTT) [12] has become increasingly popular due to its lightweight and efficient messaging protocol. This protocol supports scalable and reliable data exchange, making it ideal for near-real-time communication and IoT environments. The capability of MQTT to handle intermittent connections, low bandwidth, and constrained devices has made it a preferred choice for industrial and IoT applications, driving its widespread adoption and ongoing evolution. Manowska et al. [19] explored the utilization of the MQTT protocol in energy management systems for measurement, monitoring, and

Table 1. Comparison of main features between OPC-UA and MQTT

Main features	OPC-UA	MQTT
Use	Mainly used in industrial and IIoT environments	Widely used in IoT and messaging applications
Transport protocol	TCP/UDP	TCP
Patterns	Pub/Sub, Client/Server	Pub/Sub
Message format	Text Type (JSON), Binary	Text Type (Plain Text, JSON)
Scalability	Large industrial systems	IoT networks and distributed systems
Session management	Yes	No
QoS	No	Yes
Encryption	Yes	Yes
Semantic Data	Yes	No

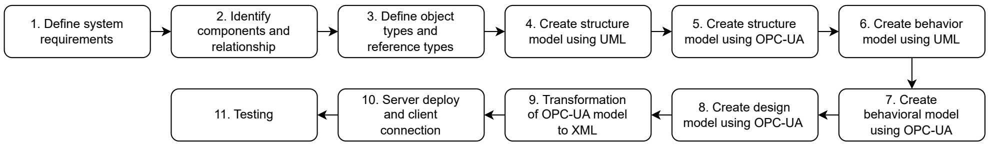

1. Define system requirements
2. Identify components and relationship
3. Define object types and reference types
4. Create structure model using UML
5. Create structure model using OPC-UA
6. Create behavior model using UML
7. Create behavioral model using OPC-UA
8. Create design model using OPC-UA
9. Transformation of OPC-UA model to XML
10. Server deploy and client connection
11. Testing

Figure 1. Overall pipeline of the proposed methodology.

control purposes. Their study delved into the application of MQTT in facilitating efficient communication within industrial systems. Furthermore, Table 1 shows a comparison of the main features between OPC-UA and MQTT.

3 Proposed Approach

This section presents the methodology followed for the design method for Model-Based Industrial Systems Development. OPC-UA will be used for communication with the PLC. It will also be used to control sensors, machines, and other devices in almost real-time, allowing for continuous monitoring of performance and production in the plant. Figure 1 shows the general process of the proposed approach.

3.1 Define system requirements

The initial stage of the design method involves defining the requirements of the system by establishing what the system must accomplish. This process requires consultation with clients and end users to achieve complete, consistent, and well-defined objectives. The definition of requirements does not require excessive formality; a comprehensive list is sufficient. The subsequent steps in the design method will be illustrated with examples tied to a consistent case study throughout the process.

3.2 Identify components and relationships

After defining system requirements, the subsequent step involves identifying system components and their relationships. Two strategies for component identification are proposed: a grammatical analysis of the requirement text, focusing on nouns as potential components, and

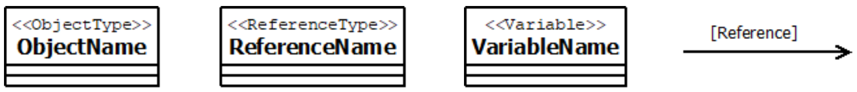


Figure 2. Structure Model elements using UML [20].

expert-based identification leveraging domain knowledge. The identified components should be listed in a specified format (e.g., Components and Attributes).

For relationship identification, two strategies are also proposed. The first involves checking, based on the identified components, if the requirement text specifies relationships between them. The second strategy relies on expert knowledge to identify relationships between components. Both approaches result in a detailed list of relationships following a specified format (e.g., Relationship Name, Source Component, Destination Component).

3.3 Define object types and reference types

After identifying the components and relationships of the system, the next thing is to define the types based on those components and relationships, the type of objects of the components will be defined, and the reference types of the relationships will be defined. The identified object types should be listed in a specified format (e.g., Component, Object type).

Before defining the reference types, it is necessary to mention that within OPC-UA there is a reference type called HasComponent whose semantics represent that the Destination Component is part of the Source Component or, in other words, the Source Component has the Destination component as part. If this generic reference type is used in the following steps, it would no longer be necessary to define it as a new reference type. Identified references should be listed in a specified format (e.g., Relationship Name, Reference Type).

3.4 Create Structure Model using UML

After having defined the Object Types and Reference Types that are present in the system, the next step is to create the Structure Model using the UML notation. As an additional consideration within the Structure Model, the Object Types are linked together using References which are the instances of the Reference Types.

For each Reference Type that is not HasComponent, it is necessary to use UML classes to define the reference type. It is worth mentioning that within the Structure Model, these UML elements do not need to be linked to other UML elements unless it is strictly necessary. The notation to define Reference Types in UML is shown in Fig. 2 (*see column 2*). Within the Structure Model, Object Types will be represented by UML classes, while References will be represented by UML Associations. The Fig. 2 (*see column 1 and 4*) shows this representation. Finally, if a component contains attributes, these will be represented as UML classes, using the Variable stereotype. They will be bound to the object type of the component they belong to using the HasComponent reference. The Figure 2 (*see column 3*) shows this representation. The work done by Lee et al. [20] is taken as a reference.

3.5 Create Structure Model using OPC-UA

Following the creation of the Structure Model in UML, the next step involves transforming each Object Type, Reference Type, and Reference into the OPC-UA model. A crucial preliminary step is establishing the Main Structure Node, which represents a node containing the

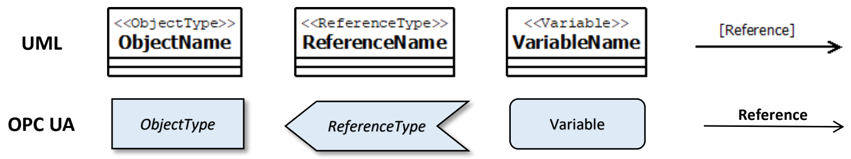


Figure 3. Equivalence of UML elements to OPC-UA [20].

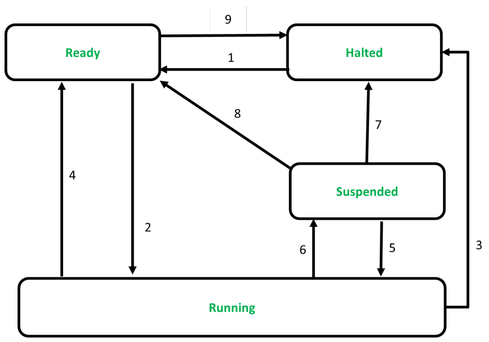


Figure 4. Base model of the FSM extracted and converted from the OPC-UA Program Information Model.

most references to other nodes, often encompassing additional nodes. This node is visually highlighted in yellow for identification purposes.

For all Object Types, excluding the Main Structure Node, it is imperative to define specific instances of each ObjectType. This involves creating an Object from the ObjectType, and the linkage between the created Object and the ObjectType is established using the HasType-Definition reference. Instances of ObjectTypes are intricately linked to components – object types defined in step 3 of the design method. The equivalence between UML elements and their OPC-UA counterparts is described in Fig. 3. The work done by Lee et al. [20] is taken as a reference.

3.6 Create Behavior Model using UML

The next step after establishing the Structure Model in UML involves defining the behavior model using a finite-state machine (FSM) in UML notation. This entails outlining the system's behavior, including states, transitions, causes, and effects, based on the requirements. States represent system conditions, transitions denote state changes, causes trigger transitions, and effects signify actions resulting from causes. A state diagram serves as the foundational model, complemented by a table detailing transitions, source and destination states, causes, and effects. This table guides the creation of different elements for the Behavior Model. The base model situates the states within the running state, as depicted in Figure 4.

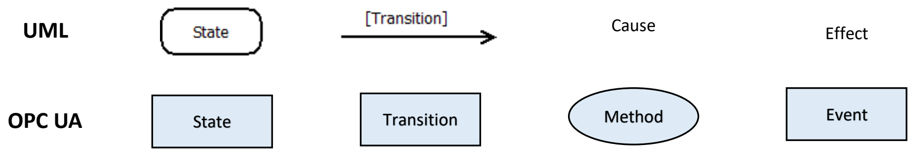


Figure 5. Mapping elements of Finite State Machine in UML notation to OPC-UA.

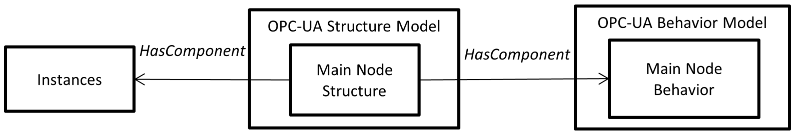


Figure 6. Integration of the Structure and Behavior Models.

3.7 Create Behavior Model using OPC-UA

Based on the Behavior Model defined in the previous step, the next step to perform is the transformation of each of the elements of the FSM in UML notation into OPC-UA elements. Figure 5 shows the equivalence between each of the UML elements with OPC-UA.

When transitioning from UML to OPC-UA, it is crucial to follow a structured approach, considering the transition table. This process involves several steps: creating objects based on states defined in the FSM in UML format, generating objects from transitions, and creating methods using transition causes. Events in OPC-UA are represented by objects using transition effects' names. A detailed review of transitions is then conducted to establish references (FromState, ToState, HasCause, HasEffect) between transitions and corresponding objects/methods, facilitating state representation. If sub-state machines exist, references (HasComponent) between states and sub-state machines are established. Finally, an object of type ProgramStateMachineType [21] is defined to represent the main behavior node (e.g. painted yellow to differentiate it), serving as the root for states, transitions, methods, and events in the OPC-UA model.

3.8 Create Design Model using OPC-UA

To define the Design Model, it's essential to integrate the Structure Model, Behavior Model, and instances of the Structure Model. The initial step involves establishing a HasComponent reference between the Main Structure Node and the Main Behavior Node, with the Main Structure Node as the origin. In the final step, instances are defined using the Main Structure Node as the parent. Instances are represented as objects, and the reference type connecting the instances to the Main Structure Node is HasComponent. The result of applying these steps is visually presented in Figure 6, showcasing the comprehensive integration of the Structure and Behavior Models along with the respective instances.

3.9 Transformation from OPC-UA to XML model

In this step, the options are presented to transform the OPC-UA design model into XML format. Two alternatives are discussed; the first involves manually creating the model based on the transformation rules outlined in SubSec. 3.5, detailing the correspondence between OPC-UA and XML elements. While comprehensive, this option is time-consuming and labor-intensive. The second option proposes using a modeler, which streamlines the process and reduces the workload associated with creating an XML-format model.

3.10 Server deployment and clients connection

Once the OPC-UA model has been obtained, it can be saved in XML format 3.9. This allows us to export to different tools and to deploy in real scenarios. In our case, the Free OPC-UA Modeler [22] is used to export the OPC UA models in XML. The OPC UA information models in XML are then loaded and deployed into an OPC-UA Server as a working system.. To assess the system operation and its status anytime, three OPC-UA clients are created: a Prosys OPC-UA client for desktop and Android mobile devices [23, 24] and a Node-RED client [25]. These clients provide a dashboard to facilitate the supervision of the system.

3.11 Testing

After completion of server deployment, it is crucial to perform extensive testing of the system to ensure its functionality, reliability, and compliance with requirements. Among the different tests, the integration tests are carried out to validate the integration between different system components (e.g., PLC, OPC UA server, sensors, actuators, and any other connected devices). Another test to perform is the functionality test to evaluate whether all functions specified in the system requirements are implemented correctly. This will include testing data reads and writes, and correct operation of the FSM. Testing addresses issues such as communication between components, requirements, error management, and security. Test results are used to identify and correct potential defects or problems in the system before it is deployed to the production environment.

4 Case Study

A system of continuous flow scales is considered as a case study to assess the validity of the proposed methodology. It is necessary to obtain the accumulated weight of the grain that flows into four silos located at different locations in the plant. There are four continuous-flow weighing scales which consist of the following elements: loading and weighing hopper, loading and discharge gate, hydraulic loading and discharge pistons, electro valves used to operate the loading and unloading pistons, plus a set of three load cells that are responsible for capturing the weight. Figure 7 shows a schematic of the scale system and its layout. The next sections illustrate how the OPC-UA platform can be developed and deployed according to the proposed methodology.

4.1 Define system requirements

Different types of grains move through the plant via conveyor belts and elevators to be stored in different silos. Scales are strategically placed just before each silo, and while deactivated, they act as a bypass, allowing grains to pass without capturing weight. Upon activation, the following steps occur: 1) the discharge gate closes, awaiting corn arrival; 2) once the predetermined batch is complete (e.g., 180 Kg), the loading gate closes, capturing the weight; 3) after weight capture, the discharge gate opens, releasing the corn; 4) once all grain is released, the discharge gate closes, and the loading gate reopens. This cycle repeats until the scale is deactivated or an error occurs.

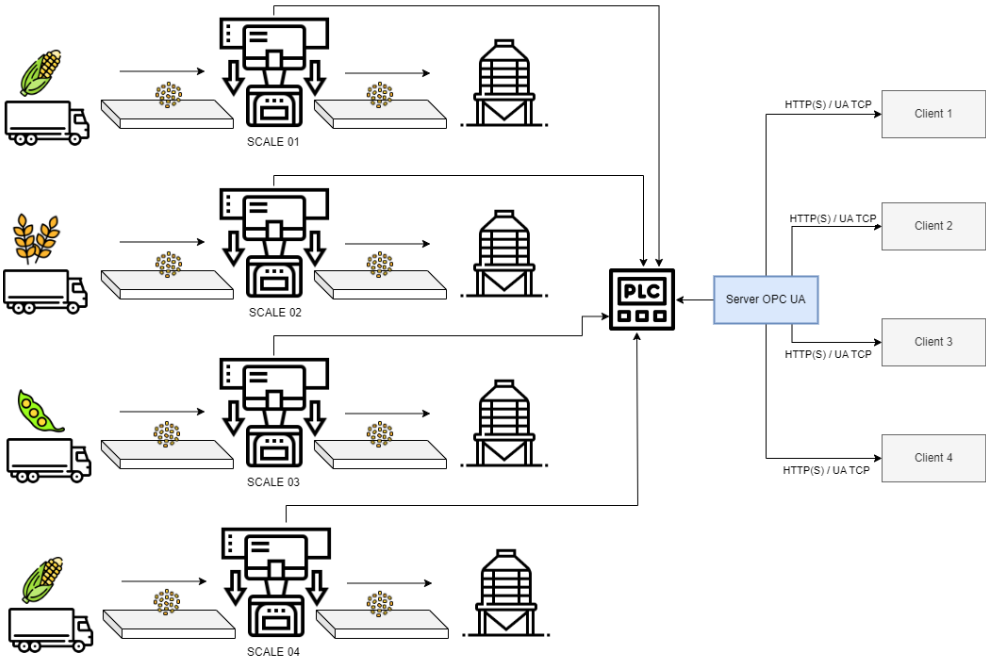


Figure 7. General outline of the Scale System project.

Table 2. Components of the Scale System and their relationships.

Relationship name	Origin Component	Target Component
It has a component	Scale	Loading Hopper
It has a component	Scale	Loading Hopper
It has a component	Loading Hopper	Loading Gate
It has a component	Weighing Hopper	Discharge Gate
It has a component	Loading Hopper	Loading Piston
It has a component	Loading Hopper	Load Electrovalve
It has a component	Weighing Hopper	Discharge Piston
It has a component	Weighing Hopper	Discharge Electrovalve
Control	Loading Piston	Loading Gate
Control	Discharge Piston	Discharge Gate
Control	Loading Electrovalve	Loading Piston
Control	Discharge Electrovalve	Discharge Piston

4.2 Identify components and relationships

Different components are identified, which are listed below: Scale (e.g., Weight, Batch, Accumulated Weight); Loading Hopper; Weighing Hopper; Loading Piston; Discharge Piston; Load Electrovalve; Discharge Electrovalve; Load Cells; Loading Gate; Discharge Gate; Silo; Conveyor Belt; Elevator. From this list of elements, different relationships between them are identified and this correspondence is shown in Table 2.

Table 3. Object types and reference types of the Scale System.

Component	Object Type
Scale	ScaleType
Loading Hopper	HopperType
Weighing Hopper	HopperType
Loading Piston	PistonType
Discharge Piston	PistonType
Loading Gate	GateType
Discharge Gate	GateType
Load Electrovalve	ElectroValveType
Discharge Electrovalve	ElectroValveType

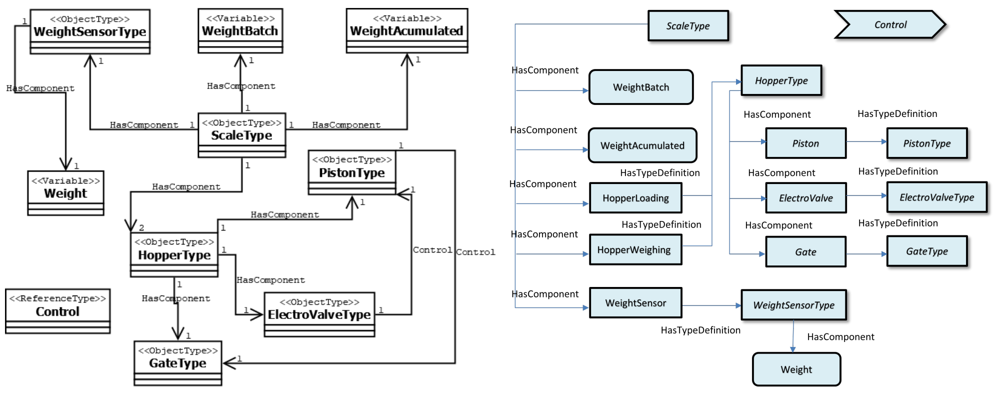


Figure 8. *(left)* Structure Model of Scale System using UML. *(right)* Structure Model of Scale System using OPC-UA.

4.3 Define object types and reference types

From the components obtained in the previous step, different types of objects are identified, which are listed in Table 3. Also, based on the relationships between components, several types of references are identified, which are listed below: It has a component (HasComponent) and control (Control).

4.4 Create Structure Model using UML

The next step is to create the structure model in UML format, and the transformation rules established in Section 3.4 are taken into consideration. Figure 8 *(left)* shows the result of generating the Structure Model in UML notation.

4.5 Create Structure Model using OPC-UA

After having obtained the Structure Model in UML format in the previous step, the next task is applying the transformation rules established in Sec. 3.5, the transition from UML to OPC-UA is carried out, generating the Structure Model with OPC-UA notation shown in Figure 8 *(right)*.

No	Name Transition	Cause	Origin State	Destination State	Effect
1	HaltedToReady	Reset Method	Halted	Ready	Report Transition 1 Event/Result
2	ReadyToRunning	Start Method	Ready	Running	Report Transition 2 Event/Result
3	RunningToHalted	Halt Method or Internal (Error)	Running	Halted	Report Transition 3 Event/Result
4	RunningToReady	Internal	Running	Ready	Report Transition 4 Event/Result
5	RunningToSuspended	Suspend Method	Running	Suspended	Report Transition 5 Event/Result
6	SuspendedToRunning	Resume Method	Suspended	Running	Report Transition 6 Event/Result
7	SuspendedToHalted	Halt Method	Suspended	Halted	Report Transition 7 Event/Result
8	SuspendedToReady	Internal	Suspended	Ready	Report Transition 8 Event/Result
9	ReadyToHalted	Halt Method	Ready	Halted	Report Transition 9 Event/Result
10	FillingToFilling	Internal	Filling	Filling	Report Transition 10 Event/Result
11	FillingToDumping	Internal	Filling	Dumping	Report Transition 11 Event/Result
12	DumpingToDumping	Internal	Dumping	Dumping	Report Transition 12 Event/Result
13	DumpingToFilling	Internal	Dumping	Filling	Report Transition 13 Event/Result

Table 4. Finite State Machine transitions of Scale System.

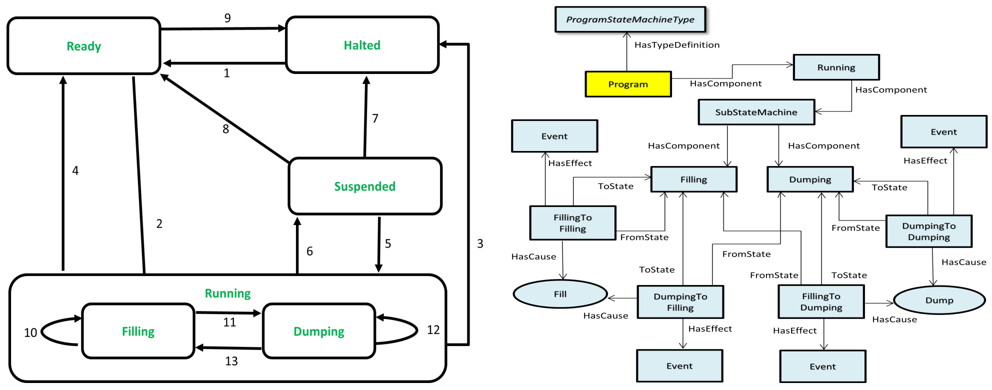


Figure 9. *(left)* Finite State Machine diagram using UML. *(right)* Behavior Model using OPC-UA.

4.6 Create Behavior Model using UML

The next step in the pipeline is to create the behavioral model using the UML format. For which the model proposed in SubSec. 3.6 is used as a basis and the Filling and Dumping states are included within the Running state, which is part of the behavior of the Scale System. Figure 9 *(left)* shows the result obtained from the UML diagram. Additionally, Table 4 shows the transitions of each of the states which are numbered next to the arrows that represent the transitions.

4.7 Create Behavior Model using OPC-UA

The next step is to take the elements defined in the previous step and apply the equivalence rules defined in SubSec. 3.5 to convert each UML element to OPC-UA. Figure 9 *(right)* shows the result obtained after applying the mapping rules. Additionally, the transitions established in Table 4 are taken into consideration.

4.8 Create Design Model using OPC-UA

From the integration of the Structure Model, Behavior Model, and Structure Model instances, Figure 10 shows the Design Model obtained.

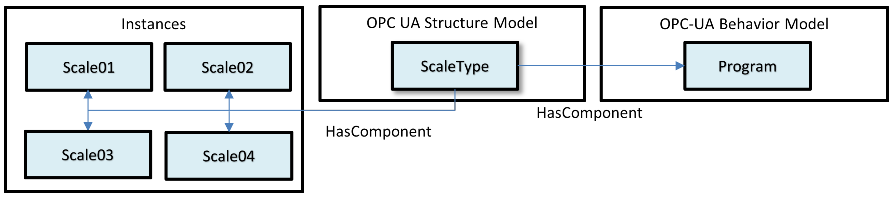


Figure 10. Design Model using OPC-UA.

4.9 Transformation from OPC-UA to XML model

The next step and based on the Design Model proposed in the previous step and with the help of the Free OPC-UA Modeler [22], the Design Model is exported to XML format.

4.10 OPC-UA Server deployment

After obtaining the model in XML format, the next step is to deploy the server. The language used is Python, so to carry out the deployment of the OPC-UA server, three files are created: *Server.py*, *Scale.py* and *Utils.py*. The file *Server.py* contains the general structure of the server. The file *Scale.py* defines a class that represents OPC-UA programs, specifically implements the methods that contain the behavior of the FSM, and uses the semantics offered by OPC-UA to validate transitions between states. *Utils.py* file contains additional functions for server creation. The last step is the execution of the server.

The OPC UA server uses certificates and private keys to enable signed and encrypted endpoints. Additionally, the server exposes nodes that represent each instance of a scale, the FSM for each scale, and programs [21] that control the behavior of the FSM. Finally, subscriptions are allowed for data exchange between client-server through an interactive environment. Figure 12 *(right col)* shows the logs of the server execution.

4.11 OPC-UA Clients

To visualize the information exposed by the server, three clients are implemented using different programming languages. Figure 11 shows *(top-lef)* Prosys OPC-UA Client for Desktop, *(top-middle)* Prosys OPC-UA Client for Android Mobile, *(top-right)* Node-RED client created in a custom way and *(bottom-left)* shows the execution of the Node-RED client where the interaction of the proposed scale system is shown.

4.12 Testing

Finally, the testing stage is carried out for which integration tests are verified and the correct communication of the OPC-UA server with the PLC, scales, sensors, and actuators is validated. On the other hand, functionality tests are carried out and the correct reading and writing of the variables exposed in the OPC-UA server is verified. In addition, the correct functioning of the FSM is verified and the transitions between states and the correct change of the variables are validated. Figure 13 shows the Node-RED client that has been connected to the OPC-UA server, and the correct status of each of the nodes can be displayed in each of the flows. Furthermore, Figure 12 shows the execution of the four scales at two different moments in time, and the correct integration and operation of the variables, functions, and FSM can be visualized.

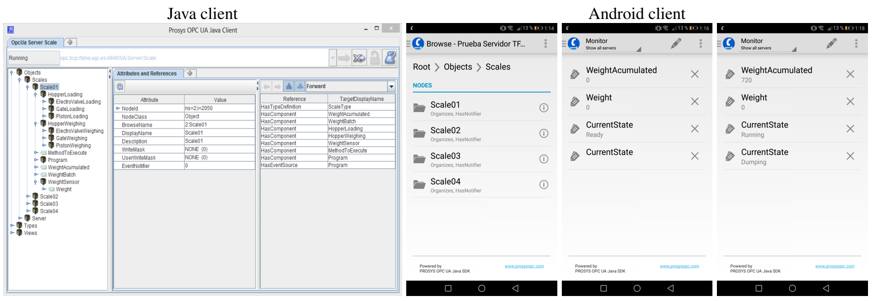


Figure 11. Visualization of the OPC-UA clients in JAVA and Android languages.

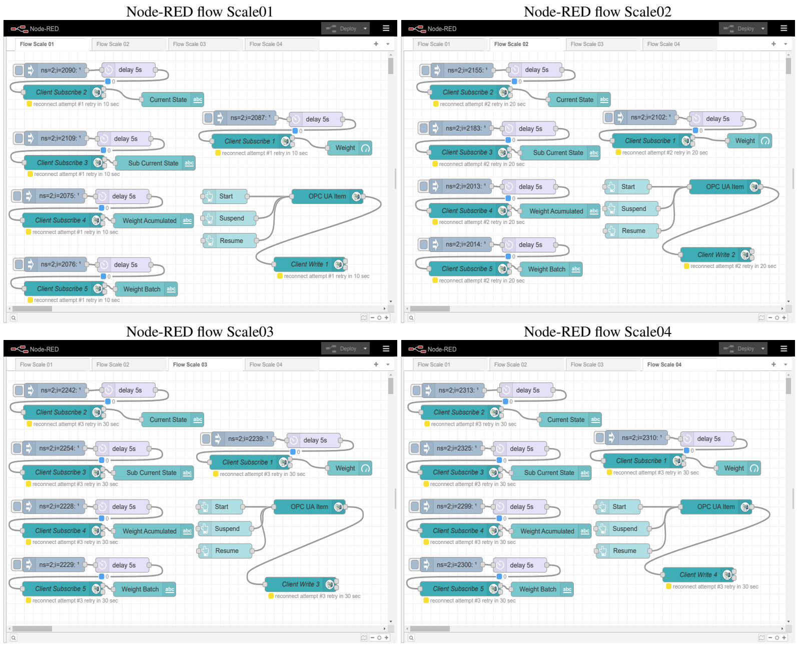
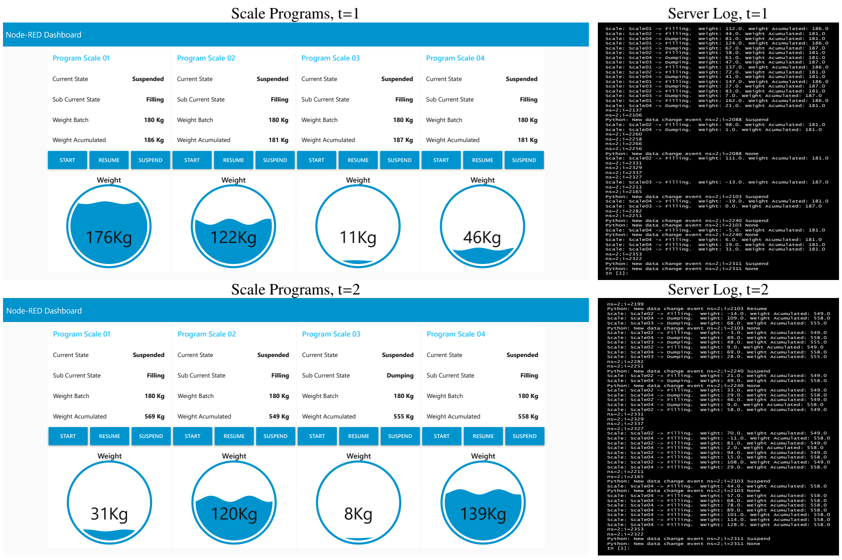


Figure 12. Visualization of variables and nodes in Node-RED flows for each scale.

5 Conclusions

The present study proposes an eleven-step design methodology for the development of industrial systems and it highlights the importance of structured design methodologies in the implementation of OPC-UA within the context of industrial systems. The system generates as a result a model with elements of OPC-UA in XML format that is published through a server in any programming language. For this work, the Python implementation is used. Furthermore, the use of a finite state machine is proposed to control the signals by changing states. A case study is proposed using a continuous flow scale system. To visualize and control the system, high-level programming languages such as Python, Android, and Node-RED are used. One of the most relevant characteristics that can be observed is the flexibility, scalability, and availability of information, which are key elements in an industrial environment.



Scale Programs, t=1

	Program Scale 01	Program Scale 02	Program Scale 03	Program Scale 04
Current State	Suspended	Suspended	Suspended	Suspended
Sub Current State	Filling	Filling	Filling	Filling
Weight Batch	180 Kg	180 Kg	180 Kg	180 Kg
Weight Acumulated	186 Kg	181 Kg	187 Kg	181 Kg
Weight	176Kg	122Kg	11Kg	46Kg

Server Log, t=1

```
Scale: Scale01 -> Filling.  Weight: 112.0. Weight Acumulated: 186.0
Scale: Scale02 -> Filling.  Weight: 44.0. Weight Acumulated: 181.0
Scale: Scale04 -> Dumping.  Weight: 81.0. Weight Acumulated: 181.0
Scale: Scale01 -> Filling.  Weight: 124.0. Weight Acumulated: 186.0
Scale: Scale03 -> Dumping.  Weight: 67.0. Weight Acumulated: 187.0
Scale: Scale02 -> Filling.  Weight: 58.0. Weight Acumulated: 181.0
Scale: Scale04 -> Dumping.  Weight: 61.0. Weight Acumulated: 181.0
Scale: Scale03 -> Dumping.  Weight: 47.0. Weight Acumulated: 187.0
Scale: Scale01 -> Filling.  Weight: 137.0. Weight Acumulated: 186.0
Scale: Scale02 -> Filling.  Weight: 72.0. Weight Acumulated: 181.0
Scale: Scale04 -> Dumping.  Weight: 41.0. Weight Acumulated: 181.0
Scale: Scale01 -> Filling.  Weight: 147.0. Weight Acumulated: 186.0
Scale: Scale03 -> Dumping.  Weight: 27.0. Weight Acumulated: 187.0
Scale: Scale02 -> Filling.  Weight: 83.0. Weight Acumulated: 181.0
Scale: Scale03 -> Dumping.  Weight: 7.0. Weight Acumulated: 187.0
Scale: Scale01 -> Filling.  Weight: 162.0. Weight Acumulated: 186.0
Scale: Scale04 -> Dumping.  Weight: 21.0. Weight Acumulated: 181.0
ns=2;i=2137
ns=2;i=2106
Python: New data change event ns=2;i=2088 Suspend
Scale: Scale02 -> Filling.  Weight: 98.0. Weight Acumulated: 181.0
Scale: Scale04 -> Dumping.  Weight: 1.0. Weight Acumulated: 181.0
ns=2;i=2260
ns=2;i=2258
ns=2;i=2266
ns=2;i=2256
Python: New data change event ns=2;i=2088 None
Scale: Scale02 -> Filling.  Weight: 111.0. Weight Acumulated: 181.0
ns=2;i=2331
ns=2;i=2329
ns=2;i=2337
ns=2;i=2327
Scale: Scale03 -> Filling.  Weight: -13.0. Weight Acumulated: 187.0
ns=2;i=2211
ns=2;i=2165
Python: New data change event ns=2;i=2103 Suspend
Scale: Scale04 -> Filling.  Weight: -19.0. Weight Acumulated: 181.0
Scale: Scale03 -> Filling.  Weight: 0.0. Weight Acumulated: 187.0
ns=2;i=2282
ns=2;i=2251
Python: New data change event ns=2;i=2240 Suspend
Python: New data change event ns=2;i=2103 None
Scale: Scale04 -> Filling.  Weight: -5.0. Weight Acumulated: 181.0
Python: New data change event ns=2;i=2240 None
Scale: Scale04 -> Filling.  Weight: 6.0. Weight Acumulated: 181.0
Scale: Scale04 -> Filling.  Weight: 19.0. Weight Acumulated: 181.0
Scale: Scale04 -> Filling.  Weight: 31.0. Weight Acumulated: 181.0
ns=2;i=2353
ns=2;i=2322
Python: New data change event ns=2;i=2311 Suspend
Python: New data change event ns=2;i=2311 None
In [1]:
```

Scale Programs, t=2

	Program Scale 01	Program Scale 02	Program Scale 03	Program Scale 04
Current State	Suspended	Suspended	Suspended	Suspended
Sub Current State	Filling	Filling	Dumping	Filling
Weight Batch	180 Kg	180 Kg	180 Kg	180 Kg
Weight Acumulated	569 Kg	549 Kg	555 Kg	558 Kg
Weight	31Kg	120Kg	8Kg	139Kg

Server Log, t=2

```
ns=2;i=2199
Python: New data change event ns=2;i=2103 Resume
Scale: Scale02 -> Filling.  Weight: -14.0. Weight Acumulated: 549.0
Scale: Scale04 -> Dumping.  Weight: 109.0. Weight Acumulated: 558.0
Scale: Scale03 -> Dumping.  Weight: 68.0. Weight Acumulated: 555.0
Python: New data change event ns=2;i=2103 None
Scale: Scale02 -> Filling.  Weight: -3.0. Weight Acumulated: 549.0
Scale: Scale04 -> Dumping.  Weight: 89.0. Weight Acumulated: 558.0
Scale: Scale03 -> Dumping.  Weight: 48.0. Weight Acumulated: 555.0
Scale: Scale02 -> Filling.  Weight: 9.0. Weight Acumulated: 549.0
Scale: Scale04 -> Dumping.  Weight: 69.0. Weight Acumulated: 558.0
Scale: Scale03 -> Dumping.  Weight: 28.0. Weight Acumulated: 555.0
ns=2;i=2282
ns=2;i=2251
Python: New data change event ns=2;i=2240 Suspend
Scale: Scale02 -> Filling.  Weight: 21.0. Weight Acumulated: 549.0
Scale: Scale04 -> Dumping.  Weight: 49.0. Weight Acumulated: 558.0
Python: New data change event ns=2;i=2240 None
Scale: Scale02 -> Filling.  Weight: 33.0. Weight Acumulated: 549.0
Scale: Scale04 -> Dumping.  Weight: 29.0. Weight Acumulated: 558.0
Scale: Scale02 -> Filling.  Weight: 46.0. Weight Acumulated: 549.0
Scale: Scale04 -> Dumping.  Weight: 9.0. Weight Acumulated: 558.0
Scale: Scale02 -> Filling.  Weight: 58.0. Weight Acumulated: 549.0
ns=2;i=2331
ns=2;i=2329
ns=2;i=2337
ns=2;i=2327
Scale: Scale02 -> Filling.  Weight: 70.0. Weight Acumulated: 549.0
Scale: Scale04 -> Filling.  Weight: -11.0. Weight Acumulated: 558.0
Scale: Scale02 -> Filling.  Weight: 81.0. Weight Acumulated: 549.0
Scale: Scale04 -> Filling.  Weight: 2.0. Weight Acumulated: 558.0
Scale: Scale02 -> Filling.  Weight: 94.0. Weight Acumulated: 549.0
Scale: Scale04 -> Filling.  Weight: 15.0. Weight Acumulated: 558.0
Scale: Scale02 -> Filling.  Weight: 108.0. Weight Acumulated: 549.0
Scale: Scale04 -> Filling.  Weight: 29.0. Weight Acumulated: 558.0
ns=2;i=2211
ns=2;i=2165
Python: New data change event ns=2;i=2103 Suspend
Scale: Scale04 -> Filling.  Weight: 44.0. Weight Acumulated: 558.0
Python: New data change event ns=2;i=2103 None
Scale: Scale04 -> Filling.  Weight: 57.0. Weight Acumulated: 558.0
Scale: Scale04 -> Filling.  Weight: 68.0. Weight Acumulated: 558.0
Scale: Scale04 -> Filling.  Weight: 78.0. Weight Acumulated: 558.0
Scale: Scale04 -> Filling.  Weight: 89.0. Weight Acumulated: 558.0
Scale: Scale04 -> Filling.  Weight: 101.0. Weight Acumulated: 558.0
Scale: Scale04 -> Filling.  Weight: 114.0. Weight Acumulated: 558.0
Scale: Scale04 -> Filling.  Weight: 128.0. Weight Acumulated: 558.0
ns=2;i=2353
ns=2;i=2322
Python: New data change event ns=2;i=2311 Suspend
Python: New data change event ns=2;i=2311 None
In [1]:
```

Figure 13. Visualization of execution of programs for each scale and server log in two instants of time.

Additionally, the importance of having standardized development processes is highlighted, particularly within the context of Industry 4.0-driven smart factories.

In future work, an OPC UA server will be deployed in real industrial environments to evaluate the potential reduction in costs and time-related to industrial systems development. This endeavor seeks to validate the practical impact of the proposed methodology, providing concrete insights into its ability to streamline industrial processes and improve operational efficiency.

Acknowledgment

This research has been partially supported by the ESPOL project "Automatización del proceso de detección de fallas en piezas de hojalata usando visión por computador" (CIDIS-004-2023), and by University of Granada.

References

1. Z. Shi, Y. Xie, W. Xue, Y. Chen, L. Fu, X. Xu, *Smart factory in Industry 4.0*, Systems Research and Behavioral Science **37**, 607 (2020)
2. C. Zunino, A. Valenzano, R. Obermaisser, S. Petersen, *Factory communications at the dawn of the fourth industrial revolution*, Computer Standards & Interfaces **71**, 103433 (2020)
3. S. Liu, A.J. Offutt, C. Ho-Stuart, Y. Sun, M. Ohba, *SOFL: A formal engineering methodology for industrial applications*, IEEE Transactions on Software Engineering **24**, 24 (1998)
4. Z. Ge, *Review on data-driven modeling and monitoring for plant-wide industrial processes*, Chemometrics and Intelligent Laboratory Systems **171**, 16 (2017)
5. R. Soley et al., *Model driven architecture*, OMG white paper **308**, 5 (2000)

6. M. Moser, M. Pfeiffer, J. Pichler, *Domain-specific modeling in industrial automation: Challenges and experiences*, in *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation* (2014), pp. 42–51
7. P. Pinheiro da Silva, N.W. Paton, *UML i: The Unified Modeling Language for Interactive Applications*, in *International Conference on the Unified Modeling Language* (Springer, 2000), pp. 117–132
8. C.E. Pereira, P. Neumann, *Industrial Communication Protocols* (Springer Berlin Heidelberg, 2009), pp. 981–999
9. OPC Foundation, *OPC Unified Architecture* (2006), accessed on May 2024, `https://opcfoundation.org/about/opc-technologies/opc-ua/`
10. Modbus Organization, *Modbus Protocol Specification* (2006), accessed on May 2024, `http://www.modbus.org/docs/`
11. Profibus International, *Profibus: The fieldbus for industrial automation* (2000), accessed on May 2024, `https://www.profibus.com`
12. OASIS Standard, *MQTT Version 5.0* (2019), accessed on May 2024, `https://docs.oasis-open.org/mqtt/mqtt/v5.0/`
13. S. Jaloudi, *Communication Protocols of an Industrial Internet of Things Environment: A Comparative Study*, Future Internet **11**, 66 (2019)
14. C. Binder, C. Neureiter, G. Lastro, *Towards a model-driven architecture process for developing Industry 4.0 applications*, International Journal of Modeling and Optimization **9**, 1 (2019)
15. A.A. Teilans, A.V. Romanovs, Y.A. Merkuryev, P.P. Dorogovs, A.Y. Kleins, S.A. Potryasaev, *Assessment of cyber physical system risks with domain specific modelling and simulation*, **4**, 115 (2018)
16. M. Bruccoleri, S.N. La Diega, G. Perrone, *An object-oriented approach for flexible manufacturing control systems analysis and design using the unified modeling language*, International Journal of Flexible Manufacturing Systems **15**, 195 (2003)
17. J.M. Gutierrez-Guerrero, J.A. Holgado-Terriza, *iMMAS an Industrial Meta-Model for Automation System Using OPC UA*, Elektronika Ir Elektrotechnika **23**, 3 (2017)
18. Z. Luo, S. Hong, R. Lu, Y. Li, X. Zhang, J. Kim, T. Park, M. Zheng, W. Liang, *OPC UA-Based Smart Manufacturing: System Architecture, Implementation, and Execution*, in *Int. Conf. on Enterprise Systems* (2017), pp. 281–286
19. A. Manowska, A. Wycisk, A. Nowrot, J. Pielot, *The Use of the MQTT Protocol in Measurement, Monitoring and Control Systems as Part of the Implementation of Energy Management Systems*, Electronics **12**, 17 (2022)
20. B. Lee, D.K. Kim, H. Yang, S. Oh, *Model transformation between OPC UA and UML*, Computer Standards & Interfaces **50**, 236 (2017)
21. Tech. rep., OPC Foundation (2024), `https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-10-programs/`
22. FreeOpcUa, *FreeOpcUa Modeler* (2016), accessed on May 2024, `https://github.com/FreeOpcUa/opcua-modeler`
23. Prosys OPC, *Prosys OPC UA Java* (2017), accessed on May 2024, `https://www.prosysopc.com/products/opc-ua-java-sdk/`
24. Prosys OPC, *Prosys OPC UA Client for Android* (2017), accessed on May 2024, `https://www.prosysopc.com/products/opc-ua-android-client/`
25. Node-RED contributors, *Node-RED OPC UA* (2016), accessed on May 2024, `https://flows.nodered.org/node/node-red-contrib-opcua`