



**UNIVERSITÀ DI PARMA**

**UNIVERSITA' DEGLI STUDI DI PARMA**

DOTTORATO DI RICERCA IN  
INGEGNERIA INDUSTRIALE

CICLO XXXIV

**3D Virtual commissioning on IEC61499: methods and  
technologies to streamline the simulation model development**

Coordinatore:

Chiar.mo Prof. Gianni Royer Carfagni

Tutore:

Chiar.mo Prof. Marco Silvestri

Dottorando: Diego Rovere

Anni accademici 2018/2021

## Table of contents

1	Introduction .....	11
1.1	Context and motivation .....	11
1.2	Objectives of this research.....	17
1.3	Executive summary.....	21
2	State of the art .....	23
2.1	IEC 61499 .....	26
3	3D simulation .....	30
3.1	Introduction .....	30
3.2	Structure of a virtual environment.....	31
3.3	Virtual Environment Development process.....	31
3.3.1	3D Models .....	32
3.3.2	Structural Constraints and attributes .....	33
3.4	DDD Platform .....	34
3.4.1	DDD Model Editor .....	34
3.4.2	DDD Simulator .....	35
3.4.3	DDD Machine NC .....	36
3.4.4	Simulation model structure.....	36
3.4.5	Structure of a prototype .....	37
3.4.6	Basic data types .....	43
3.4.7	Authoring and execution of models.....	44
4	Proposed architecture .....	45

4.1	Introduction .....	45
4.2	Overall design.....	47
5	Connecting automation and simulation at runtime.....	52
5.1	Introduction .....	52
5.2	Objectives and requirements.....	52
5.2.1	Requirements list.....	53
5.3	Implementation.....	58
5.3.1	IO data model .....	59
5.3.2	Connection on WebSocket .....	68
5.3.3	Connection on MQTT .....	80
6	Connecting automation and simulation at design time .....	94
6.1	Objectives and requirements.....	94
6.2	Implementation.....	101
6.3	Digital avatar data model.....	102
6.3.1	XML implementation of the data model .....	104
6.4	The integration API.....	109
6.4.1	The gRPC API.....	111
6.4.2	User workflows .....	114
6.4.3	Implementation within target environments.....	118
7	Global validation scenarios.....	123
7.1	Logistics.....	124
7.1.1	Prototypes .....	125
7.1.2	Model Statistics .....	127
7.2	Packaging .....	128

7.2.1	Prototypes .....	129
7.2.2	Model Statistics .....	130
7.3	Requirements fulfillment assessment .....	131
7.3.1	Runtime requirements .....	131
7.3.2	IDE Integration requirements .....	133
8	Conclusions and future developments.....	135
9	Bibliography.....	136

## List of figures

Figure 1 Evolution of digital pyramid with CPS.....	12
Figure 2 Typical development process of discrete manufacturing systems .....	14
Figure 3 Possible scenario of improvement with virtual commissioning.....	14
Figure 4 Full plant 3D simulation.....	30
Figure 5 Steps to build up a 3D simulation.....	32
Figure 6 Simplifying 3D models .....	33
Figure 7 DDD Model Editor IDE application .....	35
Figure 8 DDD Simulator Runtime application.....	35
Figure 9 Model structure.....	36
Figure 10 Structure of a prototype .....	37
Figure 11 Simulation modules connected to form a plant.....	44
Figure 12 Typical virtual commissioning architecture .....	47
Figure 13 Proposed architecture evolution.....	48
Figure 14 Architecture instantiation with DDDMachine engine .....	50
Figure 15 Architecture instantiation with DDD Simulator engine .....	51
Figure 16 IEC61499 Function Block signals .....	60
Figure 17UML Class Diagram of the I/O Data Model.....	61
Figure 18 Complementary direction of the signals between runtimes .....	67
Figure 19 Each signal contained in the payload is an output .....	68
Figure 20 Architecture schema of the WebSocket channel implementation .....	73
Figure 21 List of classes composing the connector and handling the CPS-Protocol over WebSocket .....	74

Figure 22 IEC61499Connector within the simulation engine architecture .....	79
Figure 23 Runtime communication architecture with MQTT.....	82
Figure 24 Scalability of the MQTT approach with multiple brokers .....	83
Figure 25 Software architecture implementation .....	84
Figure 26 Correspondence between Simulation Instance and device Function Block .....	85
Figure 27 UML Class diagram of the Simulation MQTT Proxy .....	86
Figure 28 Integration high level architecture .....	101
Figure 29 Digital Avatar Data Model UML Class Diagram .....	102
Figure 30 gRPC framework .....	110
Figure 31 Synoptics view of the IDE Integration API.....	113
Figure 32 Portion of HTML auto-generated documentation of the SimulationService.proto ..	113
Figure 33 Startup and creation workflow with gRPC API.....	115
Figure 34 Stop and close workflow .....	116
Figure 35 Destroy workflow .....	117
Figure 36 Component view of the implementation .....	118
Figure 37 Internal structure of the GRPCInterface plugin.....	119
Figure 38 Startup of the gRPC server module .....	120
Figure 39 Library of simulation prototypes corresponding to IEC61499 device FBs .....	120
Figure 40 Simulation model composed of 2 instances of conveyor prototype.....	121
Figure 41 Instantiation of the workflow in the target applications .....	122
Figure 42 De-palletization pilot plant model.....	124
Figure 43 Library of prototypes developed for testing the logistics scenario.....	125
Figure 44 Reepack food packaging line.....	128
Figure 45 Library of prototypes developed for testing the packaging scenario.....	129

## List of tables

Table 1 Prototype folder structure .....	38
Table 2 Basic data types .....	43
Table 3 Runtime communication requirements.....	57
Table 4 IEC 61499 impact on IO Data model .....	60
Table 5 CP-Protocol phases and involved messages .....	70
Table 6 MqttProxy details table .....	87
Table 7 MQTTIO details table .....	89
Table 8 AbstractMqttModule details table .....	91
Table 9 IDE integration requirements.....	100

## Abstract

Current discrete manufacturing systems are characterized by an ever-increasing complexity, demanding for innovative technologies that promote the agile development of intelligent and flexible systems, capable of highly optimized performances, but also ready to evolve promptly according to the requirements of production. Industry 4.0 and its whole ecosystem of methodologies and technologies lay their basis on the assumption that no future development is possible without a tight integration between the mechatronic system and its digital counterpart.

With such a background, changing perspective to deal with distributed modular architectures of Cyber Physical Systems is mandatory, and the IEC 61499 standard, its object oriented and event based approaches promote this paradigm shift. The opportunities of this revolution reside in the multi-disciplinary nature of the CPS entities and in the possibility to exploit their digital counterparts to increase the reliability of the control applications and reduce the development times. The adoption of a virtual commissioning (VC) system can effectively support the validation phase of the overall procedures, providing immediate feedback and reducing significantly the amount of time needed to carry on physical tests on the real mechatronic system. However, currently creating a virtual commissioning model is still a complex and potentially expensive process that needs to be carried out by different professionals who must tightly cooperate to generate an effective playground for the automation testing.

The main objective of this PhD is the engineering of a new way to the design and develop virtual commissioning models, improving the efficiency of the overall process of implementing 3D simulation digital twins for complex automated discrete manufacturing systems. The proposed approach, leveraging the synergies between modular simulation and automation technologies, aims at reducing the required interaction between competences, increasing the level of independence of the automation engineer, to increase his productivity. This target requires the study and development of an holistic solution that encompasses all the stages involved in the implementation of a distributed CPS system, following the natural evolution of the control logics, from the early prototyping up to the final commissioning of the whole system and



embracing the problem space from the perspectives of two main domains that compose a virtual commissioning model: the system engineering and its runtime execution.

The most ambitious objective of this PhD is therefore study and implement a proof of concept of an integrated engineering platform composed of software tools instrumented to cooperate for the joint production of virtual-commissioning-ready instances of CPS digital twins. This achievement of such result requires the analysis of the current possibilities of extension of existing IEC 61499 Automation and 3D Simulation IDEs, the design of an open architecture made of a reference data model and a set of software API that, once deployed, manages the communication between the applications and the actual realization of scenarios to validate the approach.

## Acknowledgements

I would like to thank SUPSI and in particular the responsible of the SPS Lab of the ISTePS department Prof. Paolo Pedrazzoli to give me the possibility to carry on this PhD. A great thanks goes to Prof. Marco Silvestri who followed my work and supported my efforts during the whole three years of this experience. I would like to thank also NxtControl and Technology Transfer System people, in particular respectively Hilmo Dzifac, for having shared with me his knowledge of the IEC 61499 and for the mutual support we gave in the development of the virtual commissioning models, and Eng. Giovanni Dal Maso who always provided the right hints to deal with the simulation applications. A special thanks is due to Eng. Franco Cavadini, for his kind support and for being the first one to believe in this new approach.

The activities of this PhD have been partially supported by Daedalus and 1-SWARM, two European research projects focusing on the improvement of the ecosystem of the service and opportunities around IEC 61499 standard, where I had the chance to meet valuable experts from all around Europe.

Particular gratitude is due to my family that has been my fan club especially when mixing work, research, thesis and home was more difficult: to my wife Antonella who always believed I could succeed and relieved me from the family duties to concentrate on the thesis, to my daughter Beatrice because she always pushed me to the desk to keep on writing and never give up, to my daughter Margherita who lent me her desktop that was the warmest place where writing and finally to my son Giovanni who never stopped to root for me.

# 1 Introduction

## 1.1 Context and motivation

Current discrete manufacturing systems are characterized by an ever-increasing complexity, mainly due to the quick change of demand and to the request for highly customized multi-components products [1]. This context demands for innovative technologies that promote the agile development of intelligent and flexible systems, capable of highly optimized performances, but also ready to evolve quickly according to the requirements of production. Device, machine tool and plant builders must promptly react to the modifications of the surrounding environments, adapting their products to meet the customer needs, that more than ever move in the direction of using reliable and multi-purpose automated systems.

If the mechanical and electrical design of these complex systems can be improved and generate benefits in terms of dynamics of the systems, it has been demonstrated that the real breakthrough takes place thanks to the improvement of the automation and, in general, of the digital facets of the industrial products [2].

Industry 4.0 and the whole ecosystem of methodologies and technologies for the fourth industrial revolution lay their basis on the assumption that no future development is possible without a tight integration between the mechatronic system and its digital counterpart [3]. This is demonstrated by the European research agenda that started with Horizon 2020 program to push forward the concept of digital twins as fundamental enablers of the new generation of intelligent systems and by the current national and international research initiatives whose main objective is fostering the adoption of the cyber physical system paradigm in the standard workflow of small, medium, and large production systems manufacturers. The digitalization process in the last years has grown at an exponential speed and nowadays it permeates all the levels and sizes of devices: from the single actuators up to the large scale systems the connectivity has become an essential feature, thanks also to the wide diffusion of the IoT technologies that reached so optimal levels of maturity and security to be allowed to enter the shopfloor.

This evolution changed in a permanent way not only the physical nature of the mechatronic systems, but also the way they are controlled and governed in the, so called, automation pyramid that, as highlighted in Figure 1 [4], stops being a strict hierarchical structure, to become a fluid de-centralized architecture.

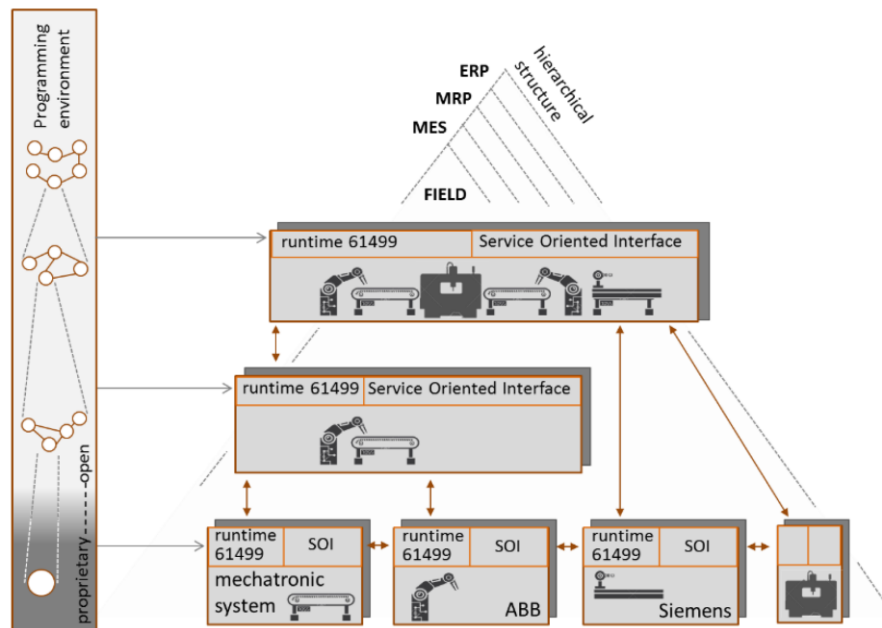


Figure 1 Evolution of digital pyramid with CPS

The interlaced CPS facets represent transversal elements, crossing all the layers from the field up to the ERP, providing different views of the same object and exposing multi-level services, publishing and consuming data, that can be exploited locally, on the edge or on the cloud. Within this context the simulation represents a fundamental component of the new automation pyramid because it exploits at maximum level the digital nature of the CPS bringing a great number of benefits at design and operation time.

Within these boundaries, a key role is played by the automation logics developer who is in charge of formalizing and translating into working code not only the control logics of the device as he always did, but also its multiple interfaces towards the digital world. These connectors include protocols and technologies whose objective is opening as much as possible in a standard way the doors to the internal information of the production environments (e.g., OPC-UA). These new requirements and, the need to implement industrial products with a “first time right”

approach stress the automation development procedures so that if on one side the complexity of the work increases, the time to produce optimized and reliable customized devices decreases. With such a background, changing perspective, from a centralized classical approach to the distributed modular one, is mandatory, in order to exploit tools and methodologies that may help the control engineers in their everyday activities, reducing the overall burden and splitting it on smaller and more comprehensible components. In this way it is possible to better focus on the optimization of particular parameters of interest in order to obtain the improvement of the global performance of the system. This kind of architectural approach is strongly supported by the IEC 61499 standard, which defines function blocks for industrial process measurement and control systems. The IEC 61499 replaces the old concept of monolithic program, written in a language compliant with the standard IEC 61131, with the concept of Application which is composed of hierarchical bricks called Function Blocks (FBs) and that can be dynamically distributed at runtime on multiple Resources.

This design pattern, streamlines the organization and development of the automation logics, modularizing it and improving the abstraction from the underlying executing hardware. Nevertheless, the problem of obtaining a “zero defect” behavior in a short implementation time, particularly for large systems, is attenuated but not removed, the automation solution must be still tested and accurately debugged. Testing the logics behavior of an automated system is a time-consuming task of the design and engineering phase that frequently leads to team racing and causes delays on the delivery to the final customer. Figure 2 shows a typical development process of a discrete manufacturing system, highlighting the critical phase where the inefficiencies can occur.

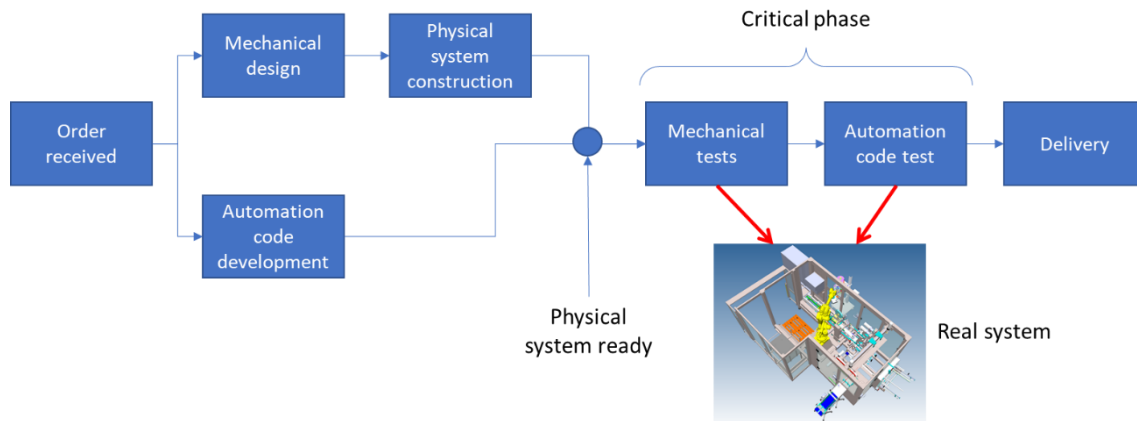


Figure 2 Typical development process of discrete manufacturing systems

To this purpose, the adoption of a virtual commissioning (VC) system can effectively support the validation phase of the overall procedures, providing immediate feedback about the expected results. A virtual commissioning system is based on the connection of the controllers with a simulation model capable of reproducing the reaction of the real environment under actions generated by the automation. Such tools can reduce significantly the amount of time needed to carry on physical tests on the real mechatronic system because they provide the means to perform off-line debugging sessions.

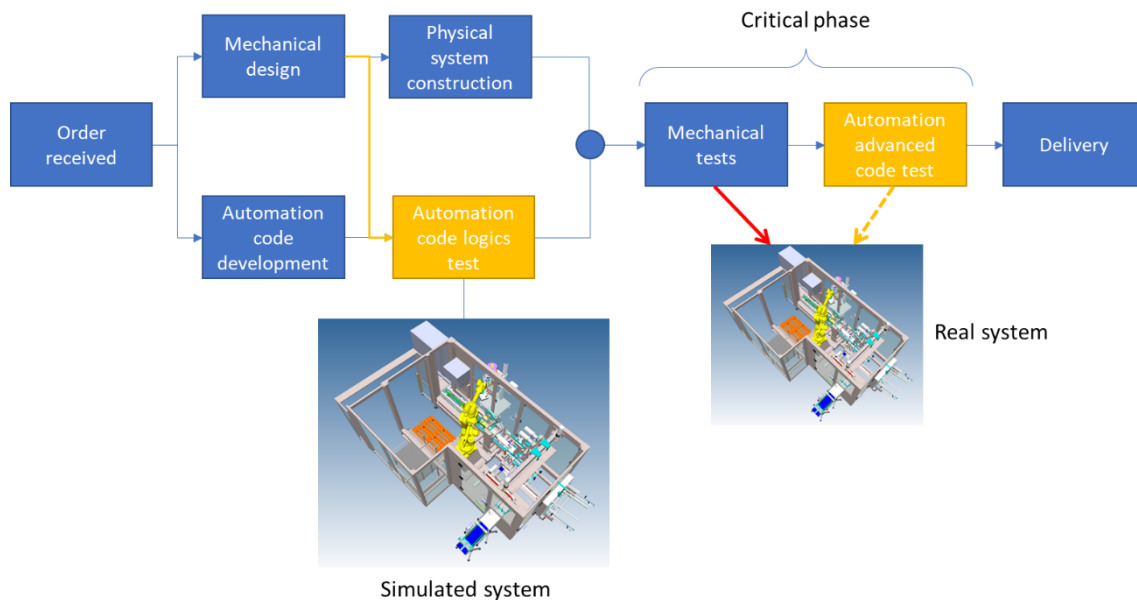


Figure 3 Possible scenario of improvement with virtual commissioning

Figure 3 shows a possible scenario of improvement of the global production process that could be derived from the introduction of the use of virtual commissioning tools. The automation code testing is split into a first extensive phase parallel to the construction of the physical devices and a lighter and faster advanced phase concurrent with the mechanical tests.

The benefits of this approach are multiple, and all go in the direction of increasing the performances of the production:

- extensive debugging sessions can be anticipated without the need to wait for the physical system to be fully assembled;
- the parallelization of the testing procedures, reduces the risk of racing conditions between production departments, because decreases the amount of time spent by the automation engineers on the device before acceptance tests;
- the possibility to test multiple scenarios without facing the risk of system breakage allows the control engineers to evaluate the safety boundaries and verify the correct behavior also in dangerous operating conditions;
- when the virtual commissioning system is capable to simulate also the technological process, the virtual tests imply savings in terms of materials that would become waste, energy and time that would have to be spent in raw parts handling; thus the approach goes in the direction of an improved sustainability of the whole production.

If the benefits of the approach are evident, it is important noting that currently creating a virtual commissioning model is still a complex and potentially expensive process that needs to be carried out by different professionals who must tightly cooperate to generate an effective playground for the automation testing.

The Automation Developer is the person in charge of the development of the device logics, its testing and debugging and becomes the repository of the knowledge of the system rules. For this reason, from a user perspective, he is also the actual consumer of the simulation results, since they provide the needed feedback to identify unpredictable pitfalls in the device operation (e.g., possible collisions, wrong operating sequences, etc.).

On the opposite, the simulation expert is not able to complete the aforementioned activities without a deep understanding of the system, its mechanical structure, its physics and the logics reaction rules. For this reason, the two figures must closely interface; the automation engineer must transfer to the simulation expert the full knowledge of the device, provide the map of input and output signals governing the interaction between the control and the mechatronics and explain how each component is expected to operate.

This iterative process is evidently expensive and affected by some inefficiencies:

- the transfer of the expected system behavior to the simulation engineer can be really high, especially for complex machines and plants
- the possibility to introduce errors increases at least linearly with the dimension of the simulated system, even if it is an assembly of reusable components, because in a non-object-oriented automation paradigm, each signal must be manually mapped to the right entity of the simulation model
- a change in the physical domain requires the intervention of both engineers, requiring at minimum three steps of description of the differences, their implementation, and their validation

The third aspect in particular makes it really expensive and hard to effectively apply the described paradigm when the underlying mechanical system is evolving quickly. This happens very often, for instance, in the everyday work of the system integrator companies, which need to test several possible alternatives of the same productive layout, with proof of concept control logics, and to further implement the most promising.

In the following chapters the problems here introduced will be examined in detail and split on the functional domains of the virtual commissioning system, leading to possible solutions that have been designed, implemented and validated during the research activities here documented.



## 1.2 Objectives of this research

The main objective of this PhD is the engineering of a new approach to the design and development of virtual commissioning models improving the efficiency of the overall process of implementing virtual commissioning digital twins for complex automated discrete manufacturing systems.

The proposed approach, leveraging the synergies between modular simulation and automation technologies, aims at reducing the required interaction between competences, increasing the level of independence of the automation engineer, and overcoming some of the cited limitations. This target requires the study and development of a holistic solution that encompasses all the stages involved in the implementation of a distributed CPS system, truly following the natural evolution of the control logics, from the early prototyping up to the final commissioning of the whole system.

The foundation of the work is established on the capability of the IEC-61499 standard to orchestrate CPS hierarchies relying on the concepts of object-orientation, so that that the modularity and reconfigurability of the mechatronic product coincides to that of the software governing them. This means that the same organized and scalable methods can be applied in the concurrent design of the digital models mirroring the control system, promoting the development of a new generation of CPS entities whose simulation counterpart not only exist and operate at production time but streamlines the everyday work of the automation developer.

The final tangible expected result of the whole work is the full integration between the behavioral models of CPS devices with their IEC-61499 functional architecture, enabling a complete and seamless connection between the information flows originating through the sensing and acting capabilities of the CPS itself on one hand, and the data structure of the simulation model on the other hand, independently from where they reside and transparently to the engineers' efforts.

To this purpose, the activities of this research start from the exploration of the possibilities to integrate the IEC-61499 platform with third party model design environments and simulation

engines that allow the definition of multi-disciplinary behavioral models, representing complementary aspects of the CPS physical dynamics.

The creation of an infrastructure coupling the intelligence supervising the physical system with the multi-level simulation models can be achieved only embracing the problem space from the perspectives of two main domains that compose a virtual commissioning model: the system engineering and its runtime execution. In fact, both at automation and at simulation side the concept of separation between model in preparation and model in execution is strongly present and mainly arises from the big difference between the requirements of design time formats, that need to retain all the source information for the continuous editing, and the requirements for the runtime artifacts, that usually need to be optimized for performance and resource consumption, i.e. compiled. For the same reason, usually the two different platforms are composed of two main applications: an authoring environment and an execution engine.

The runtime facet of the virtual commissioning domain is based on the live and quasi real time connection between the automation and the simulation engines, and deals with the objectives of ensuring a high-throughput data exchange between the artifacts mimicking the physical exchange of low level I/O signals between the automation solution and the real set of sensors and actuators of the device hardware. At this level, many solutions are already present on the market (see the state of the art of Chapter 2) but almost all of them are based on non-object-oriented automation standards belonging to the IEC 61131 ecosystem. They are characterized by the adoption of low-level standard communication protocols, like Modbus, that require an extremely time consuming mapping process of the I/O signals, or by the usage of high-end data access protocols like OPC-UA that perfectly maintain the semantics of the information but lack in performance when they must be used for testing and debugging of control logics. Therefore, the research activities, at runtime level, aim at developing a high performance and secured I/O channel that leverages the two fundamental features of the IEC 61499 standard: the object-orientation and the event based paradigm. The main expected benefits on this topic are a significant reduction of the configuration burden and the improvement of the support of the communication architecture to distributed cyber physical systems. Maintaining near-real-time bidirectional synchronization with the shop floor is based on the study of extensions of the

platform and plug-ins for commercial software to allow the integration of IEC-61499 approach with any simulation tool wanting to become compliant with the standard.

The engineering aspect, first in terms of production workflow, represents also the most difficult context to deal with. The nature of the software environments involved in the engineering phase of the cyber-commissioning are completely different and share only a similar and really high degree of complexity. An IEC 61499 Automation Integrated Development Environment (IDE) and a 3D Simulation IDE are meant to support their respective end users in the fine-grained authoring of large scale models. They typically provide complete set of tools that allow the definition, programming, compilation and execution of completely different software artifacts and they are designed to be self-contained and self-standing applications, rarely interfaced with other platforms. The most ambitious objective of this PhD is therefore study and implement a proof of concept of an integrated engineering platform composed of software tools instrumented to cooperate for the joint production of virtual-commissioning-ready instances of CPS digital twins. This achievement of such result requires the analysis of the current possibilities of extension of existing IEC 61499 Automation and 3D Simulation IDEs, the design of an open architecture made of a reference data model and a set of software API that, once deployed, manages the communication between the applications and the actual realization of scenarios to validate the approach. The activities originate within the context of Daedalus European Research Project, where the consortium was designed to pursue a wider superset of objectives related to the promotion of IEC61499 standard as key enabler for the distributed control of new generation of CPS systems. The natural follow up of Daedalus is represented the 1-SWARM European Research Project that pushes forward the original targets towards the concept of Cyber Physical System of Systems. 1-SWARM lays its basis on the industrial implementation of IEC-61499 to provide an industrial grade engineering environment supporting the design of control applications and visualization together in one tool; automatic generation of the communication during the distribution process of the application; the debugging and online-monitoring infrastructure, allowing to remotely debug single Function Blocks (that is, one of the key programming formalisms of the standard) as well as fully distributed applications. Thanks to this unique, innovative, and standard-based approach, 1-SWARM aims at obtaining simple-to-deploy aggregation of already existing CPS, each one with its own on-board intelligence, to

compose the articulated “Cyber-Physical Systems of Systems”. The activities are then carried on in continuity among the two research initiative and in collaboration with several academic and industrial partners that provided, not only the domain knowledge, but also the open access to the corresponding software platforms for the runtime and engineering of IEC 16499 and manufacturing system 3D kinematics simulation.

For this reason, the whole present research is based on a twofold approach that starts from the study and design of platform independent solutions, characterized to be openly accessible and applicable virtually to any third-party commercial application, and then goes deep in the implementation of proof of concept prototypes based on the software platforms available in the consortium. This conceptual and pragmatic way of proceeding reflects in the organization of the dissertation, which alternates the documentation of the developed open specifications of data models and interfaces with the presentation of the results of applying them to the commercial software platforms provided in particular by NxtControl GmbH for the IEC 61499 automation side and Technology Transfer System s.r.l. for the simulation side.

In compliance with the desire of realizing not only a theoretical study, but to explore with a real implementation the potentialities of the proposed approach, the third objective of this PhD is developing a set of show cases that, applying the previous results demonstrate the actual usability of the engineered solutions, highlighting benefits and limits, to pave the way for future improvements.

### 1.3 Executive summary

The main objective of this PhD is the design and development of virtual commissioning framework based on the IEC 61499 standard capable to improve the process implementing digital twins for complex automated discrete manufacturing systems. In order to produce a coherent and vendor independent result, the research activity covers both main functional domains of the virtual commissioning system: the engineering phase and the runtime execution. They are characterized by really different requirements and solutions, but the former can't overlook the latter, and the harmonization of the approaches is a key aspect of the activities of this PhD work.

The proposed solutions are meant to increase the capability of automation developers to deal with the advanced testing of IEC 61499 control application for complex and distributed architectures of Cyber Physical Systems, supporting them in exploiting the expressivity of 3D virtual environments simulations for the testing phase. The holistic view on the problem requires to advance in both the two involved knowledge domains, the automation, and the simulation, to study and implement bridges that facilitate their interaction. Such an objective can't be reached without a tight collaboration between the IEC 61499 and 3D simulation professionals that must intervene within their respective contexts to adapt software tools and methodologies to the proposed integration layers. This is the reason why the research has been carried on in the scope of the collaborative initiatives of the Horizon 2020 Daedalus Research Project and of the Horizon 2020 1-SWARM Research Project where reference partners in the field of IEC 61499 like NxtControl cooperated with simulation platform builders to create industry ready prototypes of integrated digital twin frameworks. This PhD in particular focuses on the simulation side, leveraging on the personal expertise matured during a personal path dedicated to 3D kinematics and dynamics simulations, and fully documents the work from the digital twin perspective. The whole approach has been tested extending the DDD simulation platform provided by Technology Transfer System S.r.l. The automation counterpart, which is not reported in this thesis, has been developed by NxtControl that extended its own IDE and its runtime to cope with the respective new communication architectures.

The structure of the document is organized to start from a detailed introduction about the target type of simulation (Chapter 3) and about the software platform and artifacts that stand behind the modular set-up of a 3D kinematics virtual environment. This overview is needed as a background to the comprehension of the data models underpinning the whole proposed infrastructures, since they are internally mapped on the components of the simulation software applications. The report then proceeds with the description of the overall proposed architecture (Chapter 4), discussing the design choices made to intervene on the two major aspects of the engineering and execution of virtual commissioning sessions on IEC 61499. The results obtained for the runtime execution of virtual commissioning models are fully detailed in Chapter 5, that moving from the identified requirements, contains sections dedicated to the I/O Data Model and to the two main implementations, WebSocket based and MQTT based, achieved and tested within the context of this research. Chapter 6 formalizes the solutions studied for the integration of engineering IDE presenting the underlying Digital Avatar Data Model and the integration API based on the GRPC protocol, as well as the impact on the automation developer workflow. Finally, Chapter 0 contains industry ready examples of application of the overall result developed within this PhD, providing evidence of the achieved progresses.

## 2 State of the art

The connection of machines with their digital representation has become an essential aspect for the management of the whole lifecycle of mechatronic systems, from the early prototyping to the optimization of performances [5] [6]. The Digital Twin, a concept borrowed from space programs where simulation of the systems is mandatory to ensure any change produces the desired effect, is indeed becoming a strict requisite providing engineers with the opportunity to address any undesired effect before applying the changes to the system in operation. This concept has been nowadays broadly extended to support products/systems design, virtual commissioning and the optimization of manufacturing lines installation & ramp-up [2] [7].

The virtual commissioning context is intrinsically a shared field of knowledge, methodologies and technologies that cover domains, the automation and simulation, whose differences arise from the nature of the approaches they follow to achieve their objectives: hardware oriented the former and purely digital the latter. The tiny interface layer between them represents the actual subject of an interaction whose great benefits impact on the whole manufacturing systems production process. Therefore, the state of the art in this context is mainly related to the evolution of the communication protocols exposed by the control hardware and software for what concerns the runtime aspects and to the progress of the simulation development platform for what refers to the engineering process.

At runtime, most of the architectures currently available for virtual commissioning rely on widely accepted interfacing standards for signals exchange in order to integrate the controlling logics with the simulation models. The automation solution is treated no more than a black box that generates output signals controlling actuators and receives input signals provided by sensors. It is a quite rigid paradigm, whose quasi-unique variant is represented by the type of connection established, that on its turn, normally is limited by the availability of connectors within the PLC. So typically, a Siemens S7 PLC is interfaced with its proprietary protocol, to access internal memory DBs, while a Schneider PLC can be easily interfaced through Modbus standard messaging to access low level coils, input and holding registers. Each platform provides its own set of connectivity solutions that range from purely proprietary (e.g., the Fanuc Focas2 libraries)

to completely standard ones (e.g. OPC-UA) but none of them tends to cope with the way the automation solution is structured, they only rely on the I/O maps. This approach founds its acceptance mainly on the simplicity of comprehension because it doesn't require any knowledge about the specific automation platform that is governing the system. This being agnostic on the standard running the control is effective when applied at the runtime of the virtual commissioning session because once the input and output are mapped on the two sides of the connection, the signals flow and the simulation model runs independently from the system generating the signal values. However, the possibility to run distributed virtual commissioning sessions is strongly limited by this approach and only few experiences report encouraging results about the possibility to perform multi-sided communication between nodes of the same control solution and all of them are based on the IEC 61499 standard. A good example is provided as an extension to the work of Mazzolini et Al. [8], where the validation of the optimized control of a distributed system is interfaced with a simulation model based on the Simio platform [9]. In this specific case, the adoption of the IEC61499 allows to manage the different nodes and, through the real-time synchronization of the event bus, provide a consistent single entry point to the automation network, through a "signal collection" Function Block in charge of supervising the communication with the digital counterpart. It represents a valid starting point, but it does not yet exploit the object orientation of IEC 61499 to handle the bi-directional mapping of data between simulation entities and the corresponding function blocks. An attempt of exploiting the modularity of the IEC61499 standard and transfer it to the simulation environment is made by J. Cabral et Al. [10]. Their approach is based on the possibility to map between IEC 61499 models and the FMI standard and an implementation of a tool that can export IEC 61499 models into FMUs (Functional Mockup Units – parts of the standard FMI – Functional Mockup Interface [11]), which would allow the co-simulation of physical plants and the PLCs software that controls. More automation oriented research are present in Xavier, Patil, Viatkin [12] where the formal verification of the IEC 61499 solution is performed with simulation in the loop but the cited simulation is constituted by Function Blocks and doesn't work with an external 3D simulation environment. One of the most promising work for performing the collaborative virtual commissioning for PLC validation is represented by the approach of Liu, Atmojo, Vyatkin [13] where they exploit the Software in the Loop (SIL) paradigm to mix both SoftPLC and virtual



device models, deploying them on a containerized docker infrastructure. The virtual devices are mainly implemented by IEC 61499 solutions that react as counterparts, so that the homogeneity of the technology simplifies the interaction. The communication channel in particular coincides with the IEC61499 event bus so there is no need for leveraging different communication technologies for interfacing third party simulation software. This way of managing the Simulation in the Loop is common to other virtual commissioning experiences on IEC 61499, because it does not require the development of exogenous models but the simulation twin of the real factory is implemented using the same Function Blocks and the same programming language [14]. All the reviewed approaches share the focus on the integration at runtime of the deployed control applications, and the same does most of the literature review related to the topic of virtual commissioning on IEC 61499 platform [3], [15].

Considering the limitations aforementioned about the design process and the development of heterogeneous simulation models, almost no solution exists in literature of full attempts of integration between automation and simulation IDE to create multi-disciplinary IEC 61499 enabled Cyber Physical Systems. Only the work of J.Cabral et Al. [10] can be considered an attempt to move in such direction since it relies on the implementation of an automatic tool for the translation of function blocks into FMUs, but this is far from the possibility to guide the automation developer into the building process of a 3D virtual environment capable of a completely different level of realistic and articulated response.

From the commercial point of view, instead, there exist solutions that integrate development environment belonging to different knowledge domains to create multi-perspective engineering platforms. An example is the Siemens Tecnomatix [16] suite of simulation and virtual commissioning tools that are fully compliant with the set of tools dedicated to automation and provided through the TIA portal. In the context of robotic simulation, the possibility to natively operate within vendor specific integrated environments is provided by most of the robot builders, e.g. by Fanuc with Robot Guide [17] or by Delmia with the Robotics Virtual Commissioning application. Historically the robotics field has been one of the first to handle testing on virtualized models because it did not have to cope with a plethora of legacy systems like the traditional discrete automation requires to do. However, all of them behave like closed ecosystems, dedicated to specific hardware and none of them complies with the IEC

61499 standard. This analysis of the state-of-the-art highlights how the field of integration of software tools for the IEC 61499 needs to be explored, looking for solutions that are vendor independent and open for adaptation to multi-disciplinary scenarios to promote the development of real Smart factories. The following chapter reports a short review of the IEC 61499 standard as a quick reference for the reader who is not used to operate in such automation environments.

## 2.1 IEC 61499

The IEC 61499 standard has been designed to facilitate the implementation of distributed automation intelligence [18]. At the beginning the standard could not meet the industrial world acceptance, mainly because of some imperfections of the initial version of the standard, which allowed personal interpretations [1]. Nowadays, with the emergence of professionally made software tools and dozens of hardware platforms, stronger industrial interest to the distributed automation can be expected.

The goal of IEC 61499 is “to offer an encapsulation concept that allows the efficient combination of legacy representation forms with the new object and component-orientation realities”. At the core of the standard there is the concept of Function Block. An IEC 61499 Function Block (FB) represents a system component, which is implemented and controlled by the internal FB's software. The approach based on FBs increases the modularity of the system and promotes the reusability of software components in the system.

### Function Block

The IEC 61499 architecture of a function block derives from the concept of subroutine-like structure in IEC 61131-3, to provide process-like abstraction. A process is an independent computational activity with its own set of variables (context) that communicates with other processes via messages that flow through the event interface. This encapsulation mechanism provides the strength of this architecture because it allows the arbitrary allocation of FBs to distributed domains.

### Hierarchy and internal implementation

The architecture of an IEC 61499 application supports unlimited nesting of composite function block structures, and combination of several diagram types: block-diagrams, state charts, and ladder logic in the same design.

### Encapsulation

A key enabler of the portability of IEC 61499 applications is the strong data encapsulation into components. This is widely recognized as one of the pillars of creating reusable code preventing hidden dependencies between variables of several FBs. This model also reflects the fundamental property of distributed systems where any data exchange can be implemented only via explicit message passing.

### Event-Driven Execution

FBs of IEC 61499 are event-driven, i.e., they remain idle unless an event is sent to one of their event inputs. The main motivation for event-driven execution is portability, i.e., the desire to make the code independent of the sequence of FB invocation in the PLC scan loop. The event-driven execution is the key mechanism enabling transparent modelling of distributed systems. After a FB is activated by an input event, it is assumed that it cannot be re-entered before the previous activation has terminated.

### Execution

By definition, the IEC 61499 system configuration is an executable specification of a distributed automation system. Naturally, to enjoy the benefits of being directly executable (as opposed to more abstract design languages), one needs a tool chain which generates executable machine code from the IEC 61499 design artefacts. The tool chain needs to include the following component software tools:

- Compiler from the source FB format to an intermediate code executed with a virtual machine, or directly to machine code.
- Run-time environment – usually a set of libraries of function blocks implementing service functions akin to device drivers, responsible for scheduling of FB invocation, data, and control flow and interfacing the peripherals.

- Support of device management protocol – the function implementing the load of FB application to a device, creation of new FB instances, or their modification.

#### 2.2.1 Available development software tools

The standard has inspired many researchers to create supporting software tools. The usual implementation tool set includes a workbench for editing function block designs and translating them into executable form, and some kind of run-time environment, which supports the execution of the executable code.

The most developed examples of such research-oriented workbenches are FBDK and 4DIAC-IDE. These have been supported with a consistent development effort until now, with 4DIAC-IDE being an open-source project. These tools have been successfully applied in many automation projects but mostly in academic and research labs.

Currently, the most advanced commercial development is nxtStudio [19].

NxtStudio (developed by an Austrian company NxtControl) integrates distributed control approach based on IEC 61499. It is an industrial grade engineering environment which supports the design of control applications and visualization together in one tool. This approach has great advantages in productivity and reuse of both control and visualization components. Several features of nxtStudio have long been expected from IEC 61499, for example, the debugging and online-monitoring infrastructure, allowing to remotely debug single FBs as well as fully distributed applications. Another feature is the automatic generation of the communication during the distribution process of the application. This greatly reduces the engineering effort when distributed control applications are designed. NxtControl has implemented various CIFB libraries to support communication over popular fieldbuses, such as EtherCAT and Profibus.

#### NxtControl Concept of CAT

The Model-View-Control (MVC) design pattern has motivated NxtControl to invent the composite automation type (CAT) concept. CAT is a function block that combines functions of machines or their parts, with their simulation and visualization.

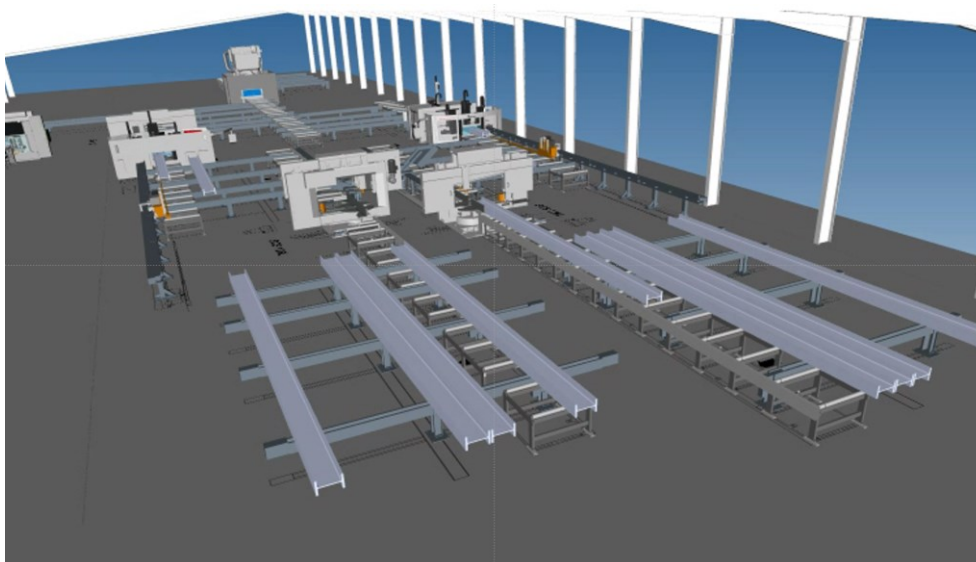
Considering for example a “pick and place” manipulator built of two identical pneumatic cylinders, each of those represented by a CAT in the FB application. Once the application is assembled from instances of such CATs, nxtStudio can automatically deploy the control parts of all CATs to the designated embedded targets, while the View parts will be sent to the device displaying SCADA screens. In the figure, a CAT of a pneumatic cylinder is exemplified. The CAT also includes the (behavioral) model of cylinder’s dynamics. Once executed, the application built of these CATs immediately delivers a complete interactive simulation model of the manipulator. The CAT concept has proven its benefits in a number of industrial projects, where NxtControl tools were used, for example, in building management systems automation.

## 3 3D simulation

### 3.1 Introduction

3D simulation of automated systems has born with the idea of enabling the mechanical designers to leave the static world of 3D CAD and move towards the animation of the models. The main objective of a 3D simulation is reproducing the complex behavior of environments composed of several parts, providing the end user with the means to study the evolution of the system over time with multiple and complementary targets among which we can find avoiding collisions of parts sharing the same motions space, optimizing trajectories of active elements (e.g., machine tools) or improving the overall system performances.

Historically, this kind of simulation has been applied to robotic systems simulation to reproduce the complex three-dimensional motion trajectories of the links of anthropomorphic kinematics chains when they are controlled in inverse kinematics mode. In fact, the problem of optimal trajectory planning with collision avoidance for such devices is one of the first playgrounds where the 3D simulation has been exploited and the evolution to include multiple robots and their surrounding devices has been the natural evolution of the engines...



*Figure 4 Full plant 3D simulation*

Currently, 3D simulation engines are capable to reproduce large manufacturing environments, composed of several processing stations and their internal logistics of customized products, with the capability to mimic, with an ever-increasing level of reliability, their complex behavior.

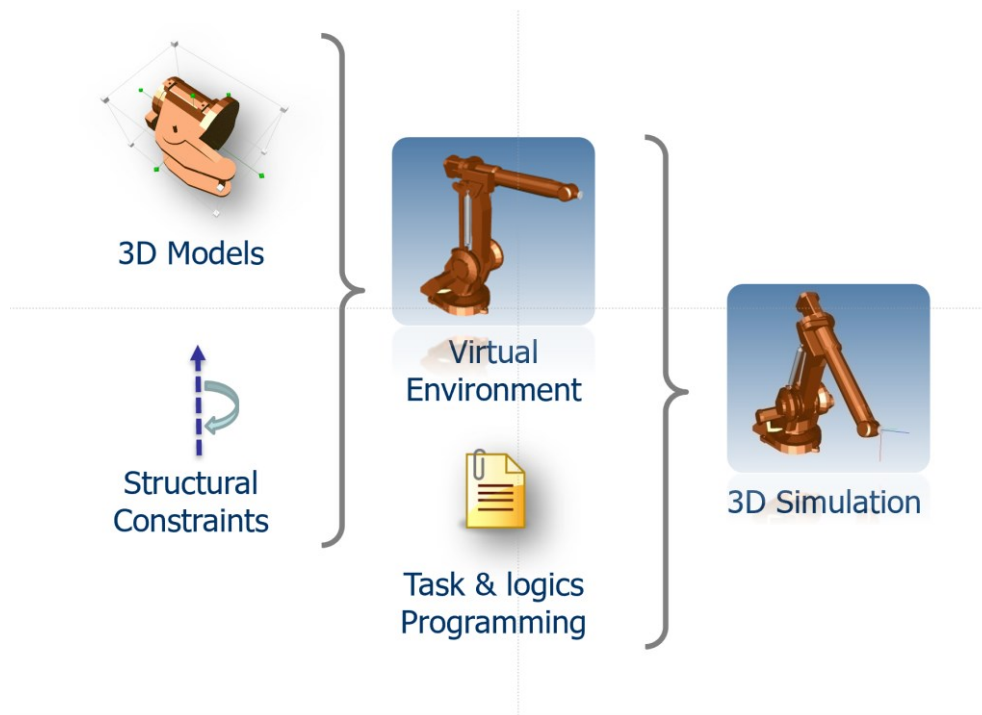
### 3.2 Structure of a virtual environment

A virtual environment can be defined as “a computer animated 3D model of a manufacturing system capable of quasi real-time response”. Each part of this definition can be further developed in the following points:

- Computer animated: a 3D simulation must be able to represent the evolution of the whole system through a series of states along the timeline. The focus of 3D simulation is how the environment modifies according to a series of endogenous or exogenous events. The nature of these events is extensively discussed in the following sections.
- 3D Model: the 3D simulation must work with three dimensional representations of the parts of the system. The representation must be as much as possible realistic
- Manufacturing system: subject of the simulation are mainly multibody discrete manufacturing systems, meaning that it doesn't handle fluid or continuum mechanics, even if in some cases it is possible to approximate them.
- Capable of quasi real time response: a 3D simulation must ensure a high refresh rate (low refresh interval) in order to be able to show in quasi real time what's the current status of the virtual environment and in order to react to exogenous events and adapt the simulation model logics. How low the refresh interval must be is a direct consequence of the dynamics of the phenomena that the model is reproducing (e.g., with high speed and high acceleration motion laws even a refresh rate of 10 milliseconds could lead to some unacceptable approximations)

### 3.3 Virtual Environment Development process

The process to set up a 3D simulation for a manufacturing environment is similar for all the existing commercially available simulation engines and requires a series of steps that, starting from the constructive 3D drawings of the system, lead to an executable model.



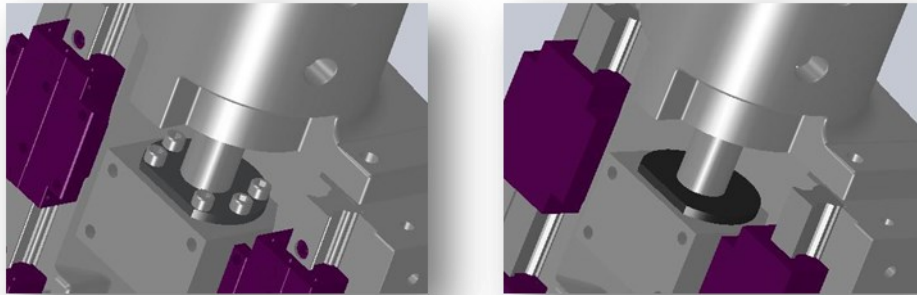
*Figure 5 Steps to build up a 3D simulation*

### 3.3.1 3D Models

The geometrical models at the basis of a 3D simulation of an automated system are generated starting from the constructive 3D drawings available nowadays in every mechanical department. These drawings represent a high value-added asset of the company, whose development cost is mainly allocated on the goods productions. The possibility to reuse such assets in a different lifecycle phase, allows to better amortize their cost while reducing the time needed to build up a simulation model.

Nevertheless, the fact that the constructive drawings contain all the details for the real production, makes them unsuitable, as they are, for a direct application within a large virtual environment. The presence of small components like screws, bolts or threaded holes often is not relevant for the study of the system behavior but it strongly increases the complexity of the models' shapes, leading to an unacceptable load on the graphics.





*Figure 6 Simplifying 3D models*

Once simplified, the drawings are translated from the proprietary format of the CAD platform to a neutral exchange format that can be imported into the simulation engine.

These neutral formats include STEP, IGES, VRML, OBJ....

### 3.3.2 Structural Constraints and attributes

Once the 3D models have been simplified and exported into a suitable neutral format, they must be organized into a multi body system defining their relationships and attributes, which typically include:

- Relative positions of parts
- Level of aggregation into assemblies
- Kinematics joints
- Materials
- Visual appearances

The definition of these features normally takes place in a dedicated editing application of the chosen simulation platform, where, with the aid of visual tools, it is possible to formalize the set-up of the physical aspect of the simulation model: the virtual environment.

### 3.4 DDD Platform

Within the scope of the research project, the DDD suite of software applications has been used as reference 3D simulation platform for the implementation of the new integrated approach to virtual commissioning on IEC 61499. The DDD platform, developed by Technology Transfer System – TTS S.r.l., is a complete set of tools that support whole lifecycle of multi-purpose 3D simulation models, from the initial import of CAD drawings to the runtime for the execution of scenarios and the integration with external control systems.

The suite is composed of three main applications:

- DDD Model Editor
- DDD Simulator
- DDD Machine NC
- DDD Supervisor

During the work of this thesis, only the first three, namely DDD Model Editor, DDD Simulator and DDDMachine NC, have been used and extended for the purposes of virtual commissioning, therefore they will be briefly introduced in the following paragraphs.

The whole architectures of the applications and of the simulation engine are developed on top of Java Platform (version 8), while the rendering engine relies on OpenGL to exploit hardware accelerated graphics, when available. Both technologies ensure a high level of portability on different hardware platforms, so that the suite can be easily installed and run both on Microsoft Windows (7 and upper) and on Linux systems.

#### 3.4.1 DDD Model Editor

The DDD Model editor is the integrated development environment (IDE) for the creation of simulation models. The application supports the end user with visual tools in the three main processes of:

- Building up the 3D virtual environment
- Developing behavioral logics
- Assembly and parametrize the overall simulation model

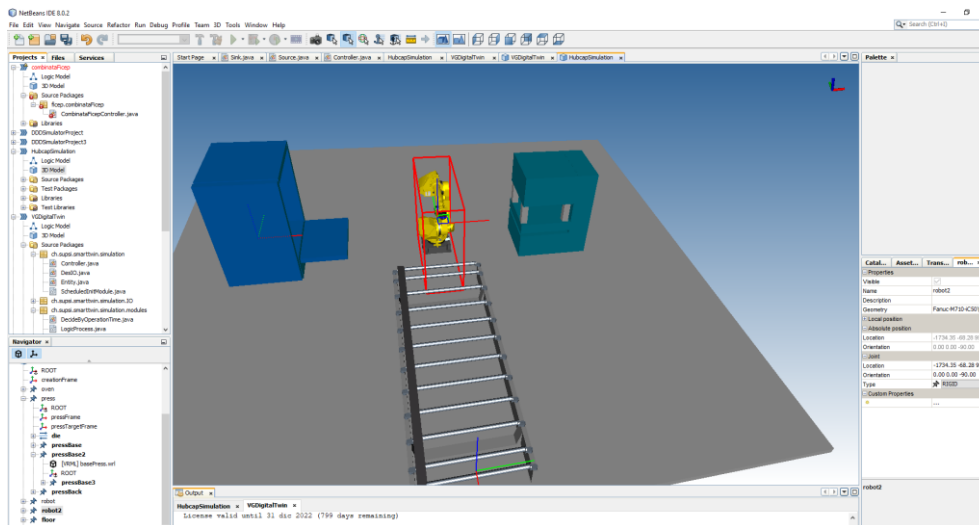


Figure 7 DDD Model Editor IDE application

### 3.4.2 DDD Simulator

DDD Simulator environment is the runtime application that supports the optimized execution of 3D simulation models or virtual commissioning models, enabling the end user to start several executions of the same models with different inputs and collect the output statistics for the evaluation of simulation results.

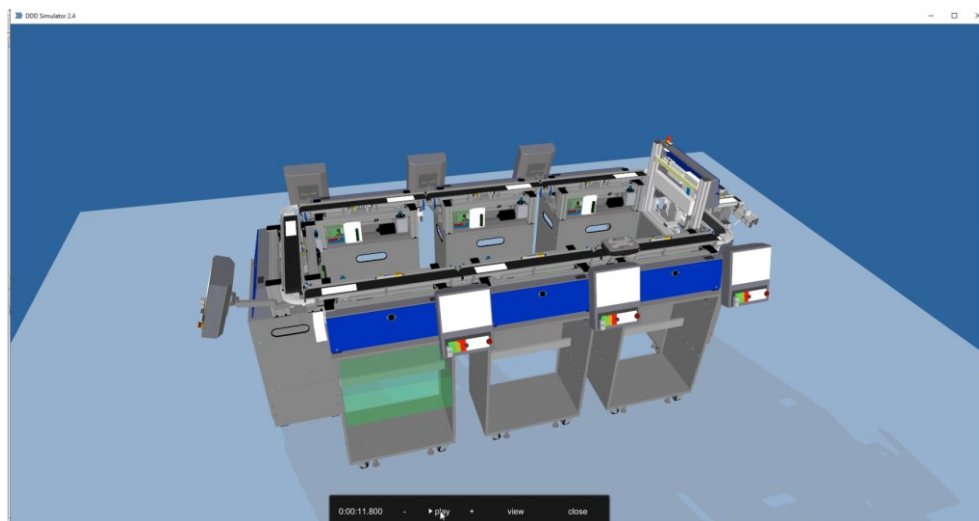


Figure 8 DDD Simulator Runtime application

### 3.4.3 DDD Machine NC

DDD Machine NC is the software application explicitly dedicated to Virtual Commissioning in the DDD platform. Is based on a modified version of the high-performance simulation engine of the DDD Simulator, the Eagle engine, that is equipped with an abstraction layer dedicated to handle different connectors to control hardware for the quasi real-time I/O communication.

### 3.4.4 Simulation model structure

Within the DDD Platform, simulation models are a composition of instances of reusable elements called prototypes, each one representing a component of the simulated system (machine, transporters, logic module, etc.). Each prototype brings two main aspects: a 3D geometrical structure and a behavior whose tight interaction determines the evolution of an instance at runtime.

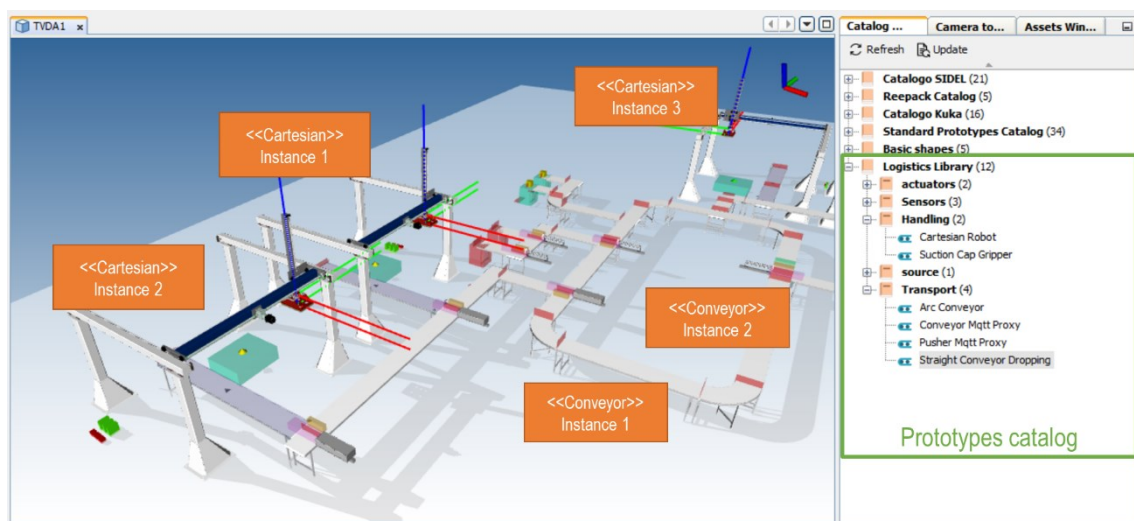


Figure 9 Model structure

This approach, quite common in the ecosystem of simulation platforms, makes it possible to split the development process of 3D simulation models into two main processes, requiring different levels of skills:

1. The development of prototypes for the creation of libraries of reusable objects
2. The configuration of prototype instances and their composition into the final complete simulation models

### 3.4.5 Structure of a prototype

A prototype is a complex simulation object defining both the 3D model and the behavior of a device.

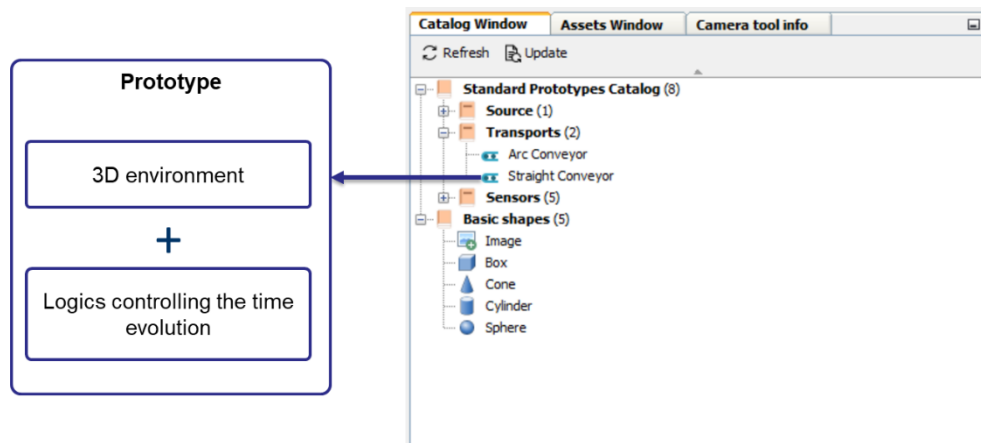


Figure 10 Structure of a prototype

The following table reports the required and optional assets composing the internal structure of a valid prototype definition.

Asset name	Format	Short Description	Use
Meta-descriptor	XML	Contains the meta-data of the prototype and its internal organization	required
3D model	XML	kinematics structure file (an XML) defining the hierarchical aggregation scene-graph of the functional parts	optional
Meshes	Neutral format geometry files (RML, IGES, STEP)	set of 3D geometry files (in neutral format like VRML, STEP or IGES)	optional

Customizer	JavaScript (.js)	JavaScript file containing directives for the preliminary customization of an instance	optional
Builder	JavaScript (.js)	JavaScript file which contains the parametric geometries	required
Logics	Java class (.java)	java class extending the “module” class which contains the code which determines the behavior of a simulation module	optional

*Table 1 Prototype folder structure*

#### *3.4.5.1 The meta-descriptor*

The meta-descriptor file plays a key role in the structure of a prototype because it contains the definition of all its external interfaces and links all the other assets above mentioned. The file must be named “prototype.xml” and must reside in the root folder.

Each prototype meta descriptor defines a large set of data; hereby are reported the most relevant for the comprehension of the adaptation work that has been carried out during the research activities.

#### *<prototype> data structure*

<b>Attributes</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>uuid</b>	<i>string</i>	Unique identifier of the prototype in the form of a UUID, it must be different for all the prototypes	Required
<b>name</b>	<i>string</i>	Name of the prototype: it is the human readable identifier, it should be different for each prototype but this is a light constraint	Required
<b>version</b>	<i>integer</i>	Progressive number identifying the current version of the prototype	Required
<b>Elements</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>

<b>&lt;parameter&gt;</b>	<i>&lt;parameter&gt;*</i>	Set of parameters that can be used to customize the future instances of the prototype when it will be used to assembly whole simulation models	Optional
<b>&lt;input&gt;</b>	<i>&lt;port&gt;*</i>	Communication interfaces that instances can exploit to receive data and events at runtime	Optional
<b>&lt;output&gt;</b>	<i>&lt;port&gt;*</i>	Communication interfaces that instances can exploit to send data and events at runtime	Optional
<b>&lt;property&gt;</b>	<i>&lt;property&gt;*</i>	A variable set of properties that can be used as further meta information of the prototype. Some properties are always present even if not mandatory for the correct instantiation of the prototype.	Optional
<b>&lt;logics&gt;</b>	<i>&lt;logics&gt;</i>	Characterization of the behavior of the prototypes	Optional

The following table reports the list of properties that are currently accepted within the <prototype> element:

<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>displayName</b>	<i>string</i>	Label that will be displayed in the catalog view.	Required
<b>vendor</b>	<i>string</i>	Name of the developer of the prototype	Required
<b>category</b>	<i>string</i>	Name of the group this prototype belongs to.	Optional
<b>generator</b>	<i>string</i>	Name of the JavaScript file capable to instantiate the geometry of the prototype. This file contains the code creating the 3D model of the instances according to the parameter values.	Required
<b>bounds_center</b>	<i>double[]</i>	Array of 3 values representing the position (x, y, z) of the center of the box containing the geometry	Optional
<b>bounds_size</b>	<i>double[]</i>	Array of 3 values representing the size (x, y, z) of the box containing the geometry	Optional

Example of minimal <prototype> structure:

```
<prototype uuid="bad1968f-5607-4f4b-960e-c333ce324ae4" name="Cartesian"
version="1">
```

```

<property name="displayName" value="Cartesian Robot" type="string"/>
<property name="vendor" value="TTS" type="string"/>
<property name="category" value="handling" type="string"/>
<property name="generator" value="builder.js" type="string"/>
</prototype>

```

*<parameter> data structure*

<b>Attributes</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>name</b>	<i>string</i>		Required
<b>type</b>	<i>string</i>		Required
<b>value</b>	<i>string</i>		Required
<b>Elements</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>&lt;property&gt;</b>	<i>&lt;property&gt;*</i>	A variable set of properties that can be used as further meta information of the prototype.	Optional

The following table reports the list of properties that are currently accepted within the *<parameter>* element:

<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>displayName</b>	<i>string</i>	Label that will be used to display the parameter in the prototype customization panel during the editing	Optional
<b>affectGeometry</b>	<i>boolean</i>	Flag that indicates whether the parameters value change determines a recalculation of the instance geometry. It is used to optimized the editing mode.	Optional
<b>constraints</b>	<i>string</i>	String containing validation hints that are provided to the end user at editing time	Optional

Example of *<parameter>* element:

```

<parameter name="zSpeed" type="double" value="10">
  <property name="constraints" value=">0" type="string"/>
  <property name="displayName" value="Z speed [m/min]" type="string"/>
  <property name="affectGeometry" value="false" type="boolean"/>
</parameter>

```



#### *<property> data structure*

<b>Attributes</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>name</b>	<i>string</i>	Name of the property. It must be unique within the context of the element that contains the property definition.	Required
<b>type</b>	<i>string</i>	Simple type of the	Required
<b>value</b>	<i>string</i>		Required

#### *<port> data structure*

<b>Attributes</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>name</b>	<i>string</i>	Name of the connection port. It must be unique within the prototype definition.	Required
<b>type</b>	<i>string</i>	A string representing the type of this connection port. This string can be any valid identifier recognized by the engine or by any of its extensions. Typically, it contains a valid Full Qualified Name (FQN) of the Java type of the connection.	Required
<b>Elements</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>&lt;property&gt;</b>	<i>&lt;property&gt;*</i>	A variable set of properties that can be used as further meta information of the prototype.	Optional

The following table reports the list of properties that are currently accepted within the <input> and <output> elements:

<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>frame</b>	<i>string</i>	Identifier of a reference frame (within the 3D model it represents a named coordinate system)	Optional

#### *<logics> data structure*

<b>Attributes</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>

<b>type</b>	<i>string</i>	Full Qualified Name (FQN) of the Java class implementing the logics of the prototype	Required
<b>Elements</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>&lt;param&gt;</b>	<i>&lt;param&gt;*</i>	A variable set of parameters that can be passed to the logics implementation to customize its behavior	Optional

#### *<param> data structure*

<b>Attributes</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>name</b>	<i>string</i>	Name of the property of the logics Java class to assign value	Required
<b>type</b>	<i>string</i>	Valid primitive type or array type of the parameter to be set. It must match one of the valid primitive types	Required
<b>Content</b>			
<p>A string containing the actual value to be assigned to the logics parameter. This string can be a simple value or it can refer to the prototype parameters using the following addressing syntax:</p> <p><i><math>\\${[name\ of\ the\ parameter]}</math></i></p> <p>Example:  <math>\\${xScale}</math> refers to the runtime value acquired by the prototype parameter called <i>xScale</i>  If the parameter is an array, the values composing the array must be separated with a semi-colon “;”</p>			

Within the meta descriptor the importance of the parameters set and of the input/output ports must be underlined because they implement two fundamental mechanisms of the prototype-based simulations.

The former allows the creation of flexible prototypes, which represent entire families of devices, whose visual appearance and behavior change according to the parametrization assigned at instantiation time. An example of parametrization is the prototype of a conveyor that could have a “speed” parameter which influences the speed of the conveyor and a “length” parameter which scales the length of the conveyor in the 3D environment.

The latter enable the cooperation among the instances belonging to a simulation model, so that the overall evolution of the simulation is the result of a complex interaction of single atomic logics.

### 3.4.6 Basic data types

The following table reports the basic data types that are used throughout the description of the elements composing the prototype. The same types have been applied in the XML descriptors developed during the research activities to enable the virtual commissioning extensions.

Beside the description of the primitive type, the table reports the corresponding array type definition, when available.

<b>Name</b>	<b>Array type</b>	<b>Description</b>
<b>float</b>	<b>float[]</b>	The basic value float is a single precision real value (32 bit). The range of float values is defined by $m * 2^e$ , where $m$ is an integer, whose absolute value is below $2^{24}$ , and $e$ is an integer from -149 to 104. Other possible values are positive and negative infinity and not-a-number (NaN). Examples of valid floats are: -1E4, 1267.43233E12, 12.78e-2, 12, -0, 0, INF and NaN
<b>double</b>	<b>double[]</b>	The basic value space of double consists of the values $m \times 2^e$ , where $m$ is an integer whose absolute value is less than 253, and $e$ is an integer between -1075 and 970, inclusive. In addition to the basic value space described above, the value space of double also contains the following three special values: positive and negative infinity and not-a-number (NaN). Examples of valid double are: -INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN
<b>string</b>	<b>string[]</b>	This data type represents a character strings. The value space of string is the set of finite-length sequences of characters. Examples of valid string are: "sbdkad", "??::O", "", " ", e "s[]{}pdns2793"
<b>URI</b>	<b>n.a.</b>	Base type uri represents a series of characters linking to a resource. URI means <i>Uniform Resource Locator</i> . Examples of valid uri are: "pippo.txt", "../models/a.wrl" and "../model/alfa.xml"
<b>boolean</b>	<b>boolean[]</b>	The basic type boolean represents the mathematic version of the logic value of true or false. The possible values are: true, false, 1, 0

Table 2 Basic data types

### 3.4.7 Authoring and execution of models

Prototypes are arranged in catalogs, and they can be instantiated with a drag and drop interface. The use of the compiled prototypes doesn't require any knowledge of the Java language so prototype catalogs can be distributed to third parties and assembling a full simulation model becomes a straightforward visual process.

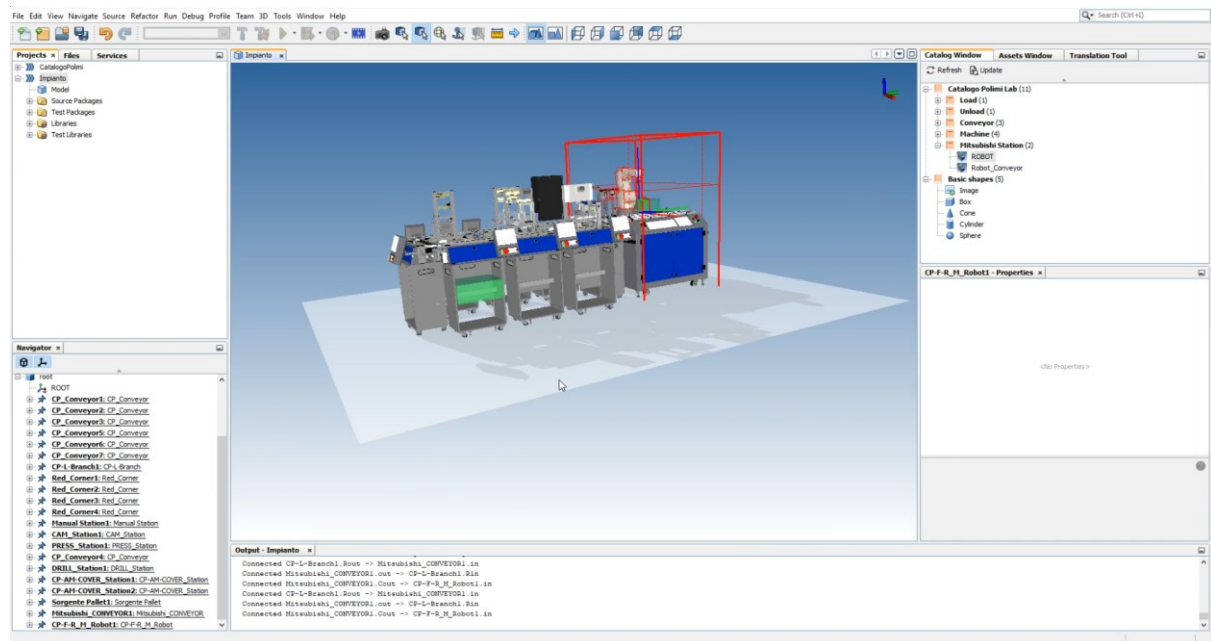


Figure 11 Simulation modules connected to form a plant

Instances of prototypes can be manipulated in the 3D editor translating and rotating them with visual tools and their parameters can be configured. Simulation models can be executed inside the DDD Model Editor, typically during the model development phase, or distributed as self-contained executable .jar, so that simulation models can be executed without the need of a DDD Model Editor license.

Being Java based, there are virtually no limits to the type of data sources and formats that can be read as inputs from DDD Simulator models. Similarly, output data can be written in any format supported by the Java Language.

## 4 Proposed architecture

### 4.1 Introduction

Starting from the analysis of the state of the art and of the current existing approaches to the virtual commissioning problem, it is clear that one of the most evident limitations arises from the fact that the process to set-up a dedicated simulation model and connect it to the automation logics is mainly a manual procedure and involves skills that are very different, yet complementary. The Automation Developer is the person in charge of the development of the device logics, its testing and debugging and becomes the repository of the knowledge of the system rules. For this reason, from a user perspective, he is also the actual consumer of the simulation results, since they provide the needed feedback to identify unpredictable pitfalls in the device operation (e.g., possible collisions, wrong operating sequences, etc.).

With the standard approach, the automation engineer must interact at least with another figure, the simulation expert to:

- build an effective virtual environment, with a balanced level of detail
- implement the right reactions (like motion laws, simulated sensors, etc.)
- map the I/O signals onto the simulation model properties
- identify and formalize the model parameters to enable the desired level of customization of the model

On the opposite, the simulation expert is not able to complete the aforementioned activities without a deep understanding of the system, its mechanical structure, its physics and the logics reaction rules. For this reason, the two figures must closely interface; the automation engineer must transfer to the simulation expert the full knowledge of the device, provide the map of input and output signals governing the interaction between the control and the mechatronics and explain how each component is expected to operate.

This iterative process is evidently expensive and affected by some inefficiencies:

- the transfer of the expected system behavior to the simulation engineer can be really high, especially for complex machines and plants
- the possibility to introduce errors increases at least linearly with the dimension of the simulated system, even if it is an assembly of reusable components, because in a non-object-oriented automation paradigm, each signal must be manually mapped to the right entity of the simulation model
- a change in the physical domain requires the intervention of both engineers, requiring at minimum three steps of description of the differences, their implementation, and their validation

The third aspect in particular makes it really expensive and hard to effectively apply the described paradigm when the underlying mechanical system is evolving quickly. This happens very often, for instance, in the everyday work of the system integrator companies, which need to test several possible alternatives of the same productive layout, with proof of concept control logics, and to further implement the most promising.

The proposed approach aims at reducing the required interaction between competences, increasing the level of independence of the automation engineer, and overcoming some of the cited limitations. The solution relies on the intrinsic object orientation and extensibility of the IEC61499 standard to implement an architecture capable to:

- ease the design phase, supporting a semi-automated set-up of the virtual commissioning model
- ease the runtime phase, supporting an automated mapping of the I/O on the simulation model properties
- apply to the virtual commissioning model the same distributed approach supported by the IEC 61499 standard.

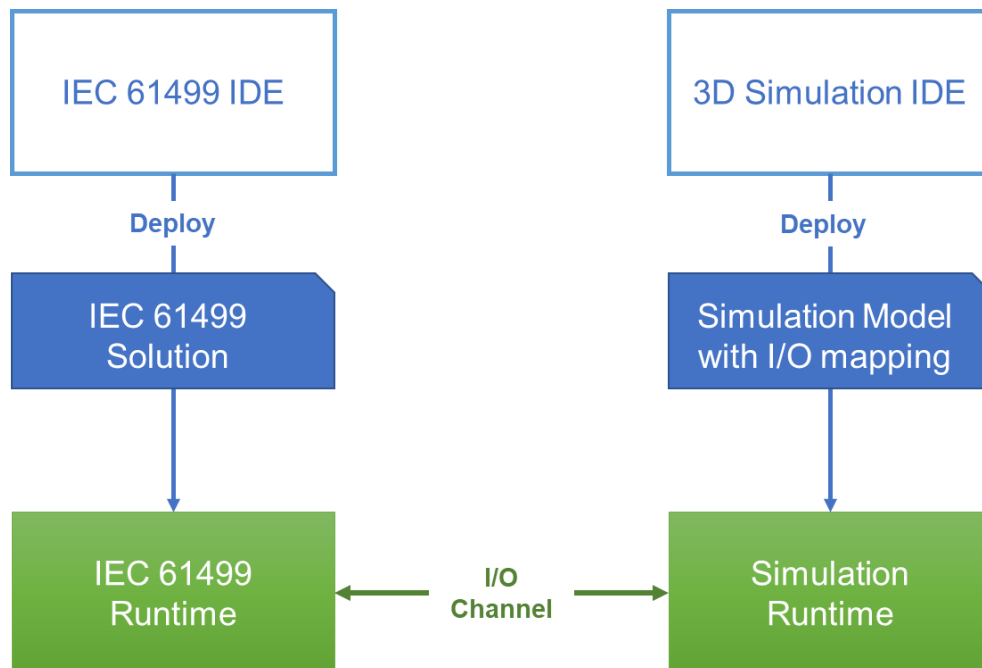
This chapter will present the result of the research activities starting from a description of the high-level architecture to document the implementation details both for the design and for the runtime tiers.

## 4.2 Overall design

The overall architecture has been designed considering the basic distinction between development environments and runtimes which is typical both for the IEC 61499 and for the 3D kinematics simulation ecosystems.

As mentioned in the previous chapters, the implementation of a virtual commissioning model starts with the definition of the automation logics and of a simulation model in dedicated software applications called Integrated Development Environments (IDE). These artifacts are deployed and run in the corresponding execution engines called Runtimes, while a I/O channel manages the data exchange among them.

In a usual scenario, therefore, the communication between the world of automation and simulation is realized only at runtime level, as shown in the following schema.



*Figure 12 Typical virtual commissioning architecture*

Until the execution time there is no relation between the two contexts: the sole data exchange is due to the mapping of the I/O signals of the automation onto the properties of the simulation model.

This approach can be considered quite natural from a conceptual point of view since the I/O list represents the physical interface of the automation with the mechatronic device and implementing a representation that mimics the same signals could be sufficient to test the behavior of the coupled system.

Nevertheless, in this way, the development process tends to become almost vertical, with the result that the conceptual structure of the control logics can differ significantly from the organization of the digital counterpart. Moreover, in order to reduce the costs, from a temporal perspective, very often, the simulation model creation is subsequent to the completion of the automation solution. This silo effect of the artifacts building process mirrors and emphasizes the limitations already mentioned in the chapter introduction.

The architecture proposed in this thesis, mainly aims at providing a possible approach to improve the overall virtual commissioning process.

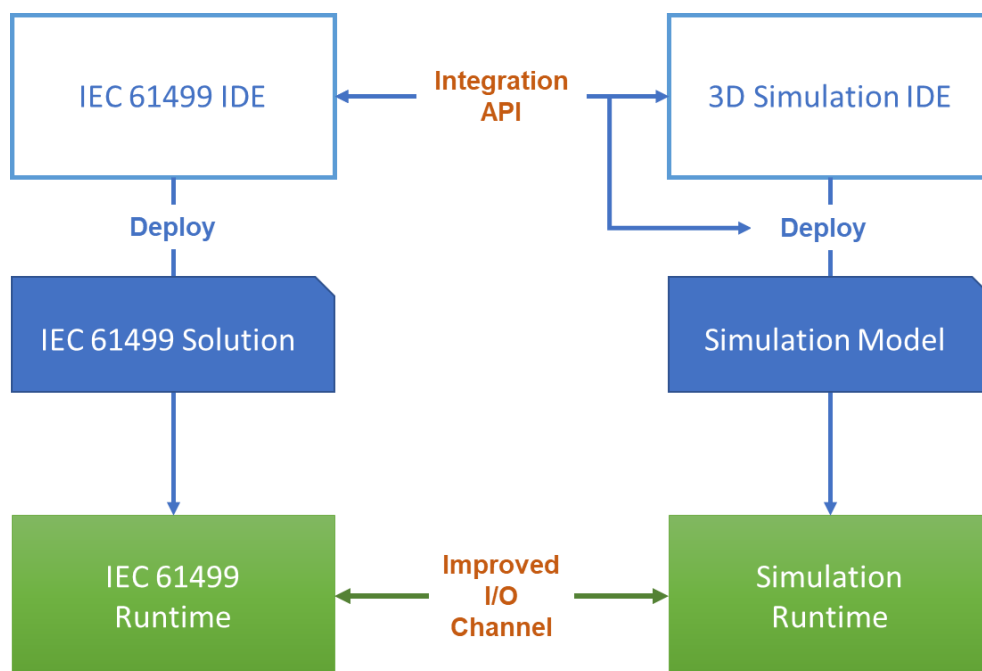


Figure 13 Proposed architecture evolution

Figure 13 shows from high level perspective the points (highlighted in red) where the software components developed during the research activities have brought a progress behind the state of the art.



The main interventions targeted both levels of the architecture:

- *at design level*, filling the gap between the automation and the simulation IDEs, implementing a real integration between the two environments through the definition of an Integration API
- *at runtime level*, improving the I/O runtime communication to comply with the distributed approach of the IEC 61499 and to provide an effective data stream satisfying the requirements to execute reliable virtual commissioning scenarios

The following chapters report all the details related to the formalization, implementation, and tests of each single component. The analysis starts with the runtime tier, providing a description of the improvements made on the I/O communication channel and then it moves to the design tier to document the methods and technologies applied to create a tight interaction between the IDEs.

During the whole research activities carried on in the context of Daedalus Project and 1-Swarm project, the reference platforms for the implementation of all the prototypes and of the use cases have been:

- Automation: nxtStudio IEC 61499 Platform, developed by NxtControl GmbH (Austria)
- Simulation: DDD Platform, developed by TTS - Technology Transfer System (Italy)

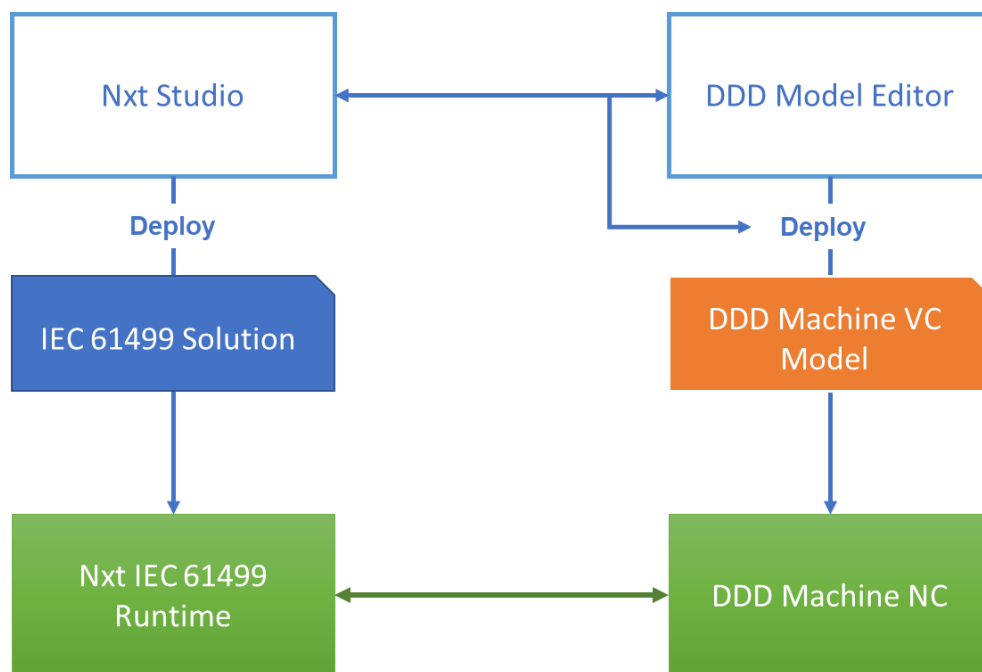
The DDD Platform provides two slightly different simulation engines, DDD Machine dedicated to the virtual commissioning of machine tools and DDD Simulator dedicated to the simulation of small to large production plants.

The two software share a large part of the data model, but the former uses a monolithic model approach optimized for the quasi real time communication with the hardware, while the latter has a higher level of modularity even though at the time of the initial activities was not enabled for I/O with external systems.

The possibility to choose the underlying engine and the peculiarities of each application lead to two different instantiations of the proposed architecture.

A first one, based on DDD Machine engine, as reported in Figure 14: the DDD Model Editor application generates and deploys a DDD Machine compliant virtual commission model, while the DDD Machine NC application acts as runtime environment.

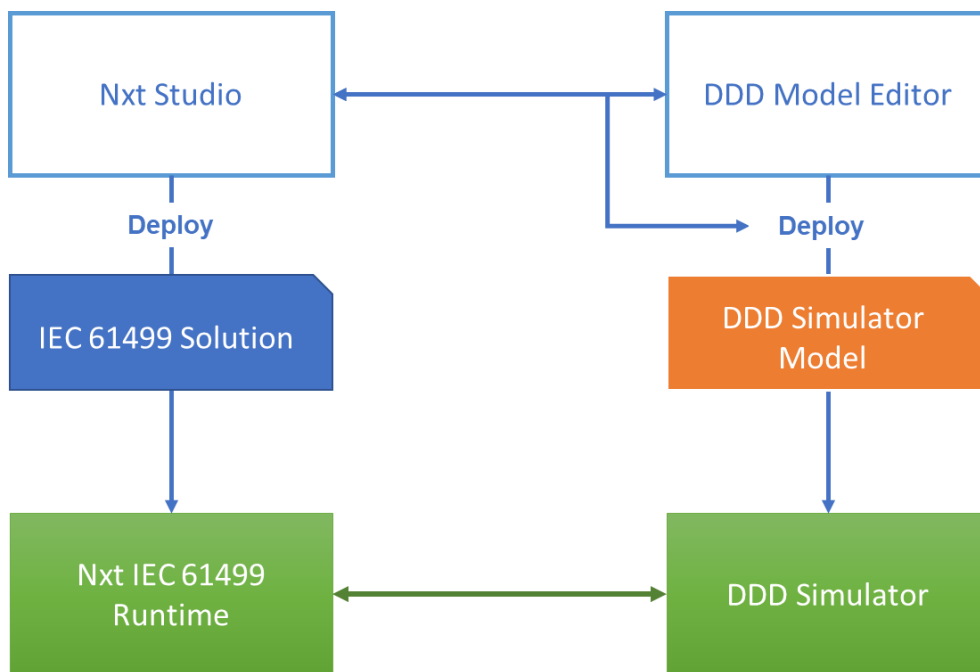
This architecture, which has been also the first one to be tested in terms of time, exploited the benefits to rely on an engine already designed for virtual commissioning, speeding up the early prototyping phase.



*Figure 14 Architecture instantiation with DDDMachine engine*

A second one, based on DDD Simulator and reported in Figure 15; the DDD Model Editor generates and deploys a prototype-based simulation model compliant with the hybrid event based engine of the DDD Simulator runtime.

This architecture is an evolution of the former, of which it exploits a large set of technical solutions, but moves forward the concepts of system of system, exploiting to a higher degree the possibility to reuse library components and create multi-level structures that are more suitable to perform virtual commissioning on large scale automated systems (e.g., internal logistics).



*Figure 15 Architecture instantiation with DDD Simulator engine*

## 5 Connecting automation and simulation at runtime

### 5.1 Introduction

This chapter describes in detail the technical solutions applied at runtime level to improve the I/O data exchange between the IEC 61499 automation and the simulation artifacts.

### 5.2 Objectives and requirements

The main objective of the work carried on at runtime level is creating an open and efficient integration layer:

- exploiting the intrinsic Object-Oriented nature of the IEC 61499 standard to move forward respect to the mapping of simple plain lists of I/O signals exposed by other protocols like Modbus, and create a formalized structure implementing the concept of “Digital Avatar”;
- exploiting the event-driven paradigm of the IEC 61499 standard to avoid, whenever possible the brute force polling of the full set of signals, optimizing the data transfer;
- ensuring a high-performance data transfer between heterogeneous and distributed runtimes avoiding the limitations of more complex infrastructures like OPC-UA.

The joint work with IEC 61499 runtime developers of NxtControl lead to identify the functional and non- functional requirements for the communication channel.

The following table reports the specifications of the requirements. Each requirement is defined by:

- an ID label that has been used to track the progress during the development and to fill a validation report during the functional tests
- a priority level that allowed to schedule the implementation activities; the priority has been formalized according the following three levels (mediated from the common keywords applied in requirements elicitation and quality management):

- SHALL: the final result must completely satisfy the requirement to be positively evaluated;
  - SHOULD: (equivalent to the keyword RECOMMENDED) the requirement is important and should be satisfied by the final result, but it can be accepted also a final result that meets the specification only partially: in this case the implications must be understood and justified;
  - MAY: the requirement is non mandatory and meeting it would represent an enhancement respect to the optimal baseline.
- A description of the desired behavior containing a base indication of possible acceptance criteria to be verified during the validation phase

#### 5.2.1 Requirements list

<b><i>Name</i></b>	<b><i>Description</i></b>	<b><i>Priority</i></b>
R001	<p>The communication channel must ensure a large bandwidth to handle high sampling frequencies of a large set of I/O signals.</p> <p>Acceptance criteria:</p> <p>the communication channel must be able to transfer the Input and output events of 200 IEC 61499 FBs (Function Blocks) each 10 milliseconds when both automation and simulation runtime operate on the same hardware (PC equipped with Soft PLC) or on a cabled LAN (Physical PLC operating the IEC 61499 solution and a PC operating the simulation model)</p>	SHALL

R002	<p>The communication channel, once activated on the deployed system, must have a minimal footprint on the infrastructure.</p> <p>Acceptance Criteria:</p> <p>the difference of processor and memory occupation of the communication channel on a IEC61499 solution deployed on Physical PLC must never interfere with the high priority control cycle</p> <p>The constraints could be relaxed, but not completely dropped, if trying to satisfy them, R001 is not met.</p>	SHOULD
R003	<p>The information (signal values) must be delivered in both directions (from automation to simulation and vice-versa) assuring the packet ordering</p> <p>Acceptance criteria:</p> <p>during the execution of a full virtual commissioning test, all the packets sent by the</p>	SHALL
R004	<p>The information ordering must be ensured without any loss of packet data in both directions.</p> <p>Acceptance criteria:</p> <p>during the execution of a full virtual commissioning test none of the data packets sent by one of the runtimes participating to the communication must be lost</p>	SHALL

R005	<p>The communication channel must accept, at automation level, multiple incoming connection from several simulation clients, thus supporting the unidirectional multicasting of packets generated by the automation runtime towards several simulation clients, even when deployed on a distributed environment</p> <p>Acceptance criteria:</p> <p>multiple (at least 2) virtual commissioning models (Digital Twins), possibly running on different PCs must be able to connect to the same master IEC 61499 automation solution and receive the same data packets</p>	SHALL
R006	<p>The number of sockets opened by each connected client simulation model must be minimized.</p> <p>Acceptance criteria:</p> <p>each client simulation model handles the communication with only one physical socket; subordinately, two physical sockets per client could be considered and acceptable solution.</p> <p>The constraint could be further relaxed if, satisfying it, the requirements R001, R003 and R004 are not met.</p>	SHOULD

R007	<p>The communication channel allows an initial synchronous (request-response) setup phase to select the signals of interest that should be transferred during the virtual commissioning session and the corresponding maximum update frequencies.</p> <p>Acceptance criteria:</p> <p>A client simulation model initiating a virtual commissioning session is capable to select a subset of the whole I/Os exposed by the IEC 61499 FBs, imposing a desired maximum refresh interval for the subset.</p> <p>The constraints could be relaxed if the management of the different subsets for multiple clients causes the system to miss R002</p>	SHOULD
R008	<p>The same communication channel (physical socket) must support the multiple flow of asynchronous signal messages, corresponding to the events governing the IEC 61499 FBs.</p> <p>Acceptance criteria:</p> <p>On the same socket the simulation client and the control logics exchange bidirectional message exactly when they are generated by the corresponding runtimes, without a sticking to a cycle time.</p>	SHALL
R009	<p>The communication channel must be based on widely accepted open standards both at transport layer and at payload level</p> <p>Acceptance criteria:</p> <p>both the technology chosen for the transport layer and the supporting payload definition are documented open standards</p>	SHALL
R010	<p>The communication channel must be natively cross-platform in order to easily deployed on multiple different hardware and operating system platforms</p>	SHALL



R011	<p>The chosen transport layer must be compliant with industrial and shopfloor network setups.</p> <p>Acceptance criteria.</p> <p>a virtual commissioning session can be executed connecting a simulation client located outside a shop floor network with a Physical PLC (or a Soft PLC running on a PC) located inside the shop floor network using a VPN for tunneling.</p>	SHALL
R012	<p>The communication channel can be secured, preventing possible exploitations for cyber-attacks to the control hardware</p> <p>Acceptance criteria:</p> <p>the chosen transport layer supports data encryption, authentication, and authorization mechanisms.</p>	MAY

*Table 3 Runtime communication requirements*

### 5.3 Implementation

The development of a runtime connection satisfying the aforementioned requirements required the completion of the following main activities:

1. Identification and choice of suitable communication standard architecture for the transport layer;
2. Design of a supporting payload data model and choice of a data representation standard;
3. Implementation of transport and data layers at IEC 61499 runtime level
4. Implementation of transport and data layers at simulation runtime level

The whole process has been carried out using an iterative approach, starting from initial prototypes that have been validated and evolved towards a more mature solution.

The final result of the work in this field is then represented by two main implementations, based on different transport technologies:

1. WebSocket
2. MQTT

The reason for existence of two solutions is the fact that the former, capable to provide most of the expected functionalities, when applied to real cases, highlighted some architectural limitations (that will be extensively described in the dedicated paragraphs). At the same time the simulation platform, during the second half of the research, evolved and provided improved functionalities that could be exploited to come to the second and more refined implementation, providing the same performances but overcoming the architectural constraints.

The two implementations correspond respectively to each one of the proposed architectures described in the previous sections:

1. WebSocket based implementation has been applied in conjunction with the DDD Machine virtual commissioning engine (Ref. to Figure 14)

2. MQTT bases implementation has been applied in conjunction with the DDD Simulator engine (Ref. to Figure 15)

Even though the two approaches are based on a very different low-level socket architecture, they are both generic enough to support custom messages so that it was possible to employ the same payload data model (described in §5.3.1), minimizing the impact on the internal runtimes I/O processing.

### 5.3.1 IO data model

The IO data model formalizes the data structures that organize the exchange of input and output signals among the automation and the simulation runtimes. This section describes the data model from a conceptual point of view and documents its JSON implementation, which has been applied in both channel implementations.

The data model has been organized with the aim to preserve as much as possible the object-oriented approach which is a fundamental aspect of the IEC 61499 standard. In fact, in almost all the standard PLC data exchange and in the large part of the virtual commissioning applications, low level I/O signals represent global lists of incoming and outgoing pieces of data flowing through the terminal blocks without any relationships with the internal architecture of the automation. In IEC 61499, instead, each signal is related to well defined events of a well identified Function Blocks, that typically represent logics dedicated to handle a device or a sub assembly of it. Another key aspect that has been considered during the data model design is the event-based nature of the interaction among function blocks, that is the same mechanism that the IEC 61499 standard promotes to enable the possibility to distribute the runtime execution on different devices.

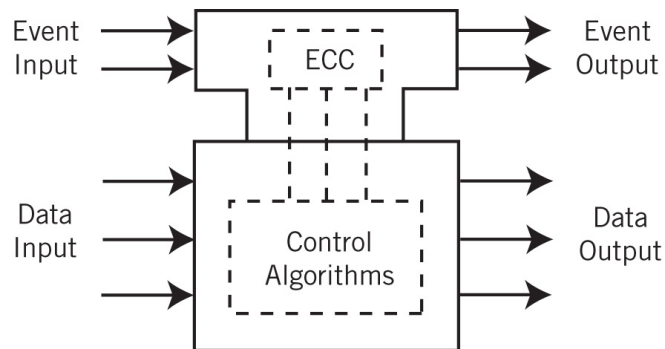


Figure 16 IEC61499 Function Block signals

The following table reports how these two main features of the IEC61499 standard have been reflected on the data model.

<b><i>IEC 61499 feature</i></b>	<b><i>Data Model feature</i></b>
Object Oriented Function Block	<p>Each message organizes signals maintaining the relationship between the origin/destination function block and output/input signal.</p> <p>Each signals name is unique within the function block and similar signals of different function blocks can use the same name.</p>
Event based	<p>The transmission of the I/O signals of a function block, happens only when a triggering event is generated at automation or simulation side. Therefore, each message can contain only the signals of the function block that generated the event.</p>

Table 4 IEC 61499 impact on IO Data model

The overall architecture of the data model is reported using the Class Diagram view of the UML (Unified Modelling Language) standard, while the meaning of each data structure and of its properties is reported in the following tables.

The developed data model can be considered only and information: the classes don't define any method (function) since the objects contained in the payload of the I/O messages must be

treated as exchange data, thus not active elements. Therefore, the UML diagram and the documentation tables contain only properties definition.

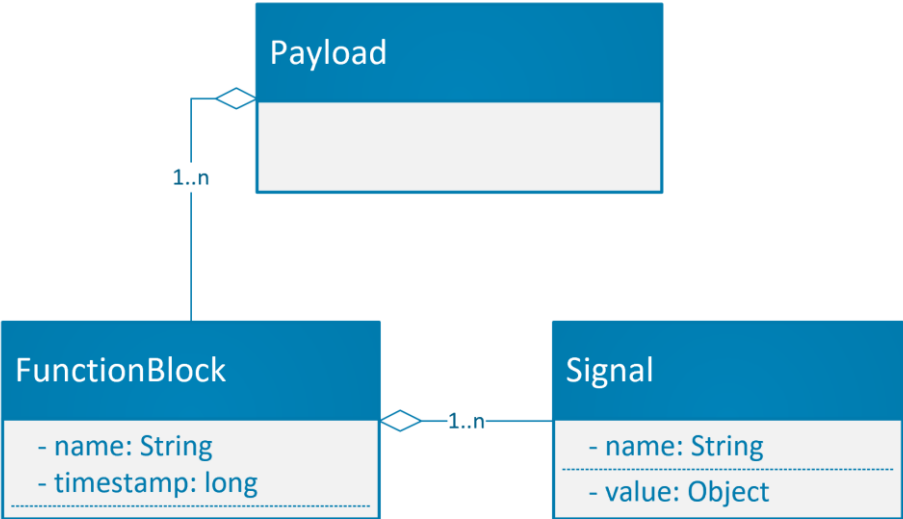


Figure 17UML Class Diagram of the I/O Data Model

*Payload class*

The Payload class represents the root element of the I/O message payload, referencing a set of Function Block instances that represent the function blocks whose events have been generated, collected, and packed together at automation or simulation side.

Properties			
Name	Type	Description	Use
<b>functionBlocks</b>	<i>FunctionBlock[]</i>	Array of instances of function block that generated an event at automation and simulation side  <i>Constraints:</i> The array must have size > 0	Required

*FunctionBlock class*

Represents an instance of Function Block that triggered an event o signal change both at automation and simulation side. It aggregates the signals of a specific Function Block instance.

All the instances of FunctionBlock belonging to the same virtual commissioning model must be identified by a unique string name.

<b>Properties</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>name</b>	<i>string</i>	Identifier of the function block within the virtual commissioning model.  <i>Constraints:</i> The name must correspond to a specific entity both at automation and simulation side.	Required
<b>timestamp</b>	<i>long</i>	Represents the time of generation of the event and it is set by the sending runtime and it is expressed as the value of the system clock expressed in milliseconds elapsed since the epoch time (1 <sup>st</sup> Jan 1970, 00:00).  The runtime receiving the message can use this value to check its reliability and verify if the virtual commissioning model is facing problems of high latency.  <i>Constraints:</i> The value must positive, and also a value of 0 is acceptable.	Required
<b>signals</b>	<i>Signal[]</i>	Array of instances of signals belonging to the instance of FunctionBlock. All the signals must be identified by a unique name within the FunctionBlock instance  <i>Constraints:</i> The array must have size > 0	Required

#### *Signal class*

Represents an instance of Signal within a FunctionBlock.

<b>Properties</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>name</b>	<i>string</i>	Name of the signal. It must be unique within the FunctionBlock instance.	Required

<b>value</b>	<i>object</i>	<p>Current value of the signal.</p> <p><i>Constraints:</i> All the signals belonging to the same message must be coherent among them, meaning that a single change event affecting two or more signal must generate only a single message containing the at least the current value of all the changed signals<sup>1</sup>.</p> <p>The value property type is generic because each signal is defined with a specific data type.</p>	Required
--------------	---------------	---	----------

It is important noting that the Signal class does not contain any definition of the signal data type, but only a key/value pair to report the current valid value of the physical signal. This choice depends on the fact that the I/O data model is applied at runtime when the automation solution and the simulation model are complete and correctly configured to handle to expected signals on both sides<sup>2</sup>. In this way, the amount of redundant information that is transferred for each message is reduced to the minimum.

#### 5.3.1.1 JSON Data Model implementation

The data model presented from a formal point of view has been transformed into JSON (JavaScript Object Notation) format to serve as a reference implementation of the message content.

JSON is an open standard (formalized also by ISO/IEC<sup>3</sup>) format for data exchange using human readable text to represent data objects consisting of attribute–value pairs, arrays, and other serializable values. It is widely accepted in socket communication for the encoding of the payloads, especially in web applications, service-oriented architectures (SOA) and web REST API. Its main benefits can be summarized in:

---

<sup>1</sup> This constraint is especially important because it ensures the coherence of the state of the function block and then the reliability of a payload message.

<sup>2</sup> This aspect is will be better discusses in the chapter dedicated to the integration of the editors

<sup>3</sup> JSON is formalized in ISO/IEC 21778:2017

- Syntax simplicity capable to represent also complex data structures using a reduced number of rules
- Low overhead if compared with other text-based data exchange formats like XML
- Wide availability of libraries supporting parsing and serialization from and to JSON
- Support to data schema formalization with JSON Schema format<sup>4</sup>

The natural conversion of the designed data model into JSON format would have resulted into a structure similar to the following one:

```
{
  "fbName": [FunctionBlock_ID],
  "timestamp": [clock_value],
  "signals": [
    {
      "name": [Signal_ID],
      "value": [Signal value],
    },
    {
      "name": [Signal_ID],
      "value": [Signal value],
    },
    ...
  ]
}
```

Nevertheless, this message structure is affected by some inefficiencies that could be negligible when considering a single message, but that can become a significant amount of redundant data when multiplied for all the possible messages flowing in a real-case virtual commissioning session at high frequency.

For this reason, the data structures defined in the previous chapter have been adapted and optimized to limit as much as possible the size of the messages, reducing the footprint of the communication protocol on the network.

The final format of the payload, expressed in JSON is the following:

---

<sup>4</sup> This feature, which is derived by the similar approach in XML with XSD, is currently rarely adopted



```

{
  [FunctionBlock_ID]:{
    "ts": [clock_value],
    "Param":{
      [Signal_ID]: [Signal value],
      [Signal_ID]: [Signal value],
      ...
    },
  },
}

```

Each placeholder and its mapping on the designed data model is documented in the following table.

<b>Placeholder</b>	<b>Type</b>	<b>Description</b>
<b>[FunctionBlock_ID]</b>	<i>JSON Attribute name</i>	<p>Identifier of the IEC 61499 FunctionBlock within the automation solution.</p> <p>This ID corresponds to a digital counterpart of the FunctionBlock in the simulation runtime. In the proposed architecture, the ID has been modelled as a string containing the fully qualified name of the FB (full path of dot - "." separated unique names of the containers)</p> <p><i>Corresponding property within UML:</i> FunctionBlock.name</p>
<b>[clock_value]</b>	<i>Number</i>	<p>Timestamp of the events that triggered the message communication.</p> <p><i>Corresponding property within UML:</i> FunctionBlock.timestamp</p>
<b>[Signal_ID]</b>	<i>JSON Attribute name</i>	<p>Name of the signal of the function block.</p> <p><i>Corresponding property within UML:</i> Signal.name</p>

<b>[Signal value]</b>	<i>Number</i> <i>Boolean</i> <i>String</i>	Value of the corresponding signal. The content can be: <ul style="list-style-type: none"> <li>• a number, covering all the floating point and integer formats, i.e. INT, REAL, etc.</li> <li>• a boolean covering the corresponding to the BOOL IEC 61499 type</li> <li>• a string corresponding to the IEC 61499 STRING type</li> </ul> <i>Corresponding property within UML:</i> Signal.value
-----------------------	--	--

The following code blocks report few examples of payload messages containing one or several signals for a single function block.

Example 1: payload for a FunctionBlock called “Field\_QSS.PusherA” generating one output data signal called “command” which is assigned with integer (INT) value 1.

```
{
  "Field_QSS.PusherA": {
    "ts": 0,
    "Param": {
      "command": 1
    }
  }
}
```

Example 1: payload for a FunctionBlock called “FI\_QDP.FieldInterface\_Cartesia” generating four output data signals called “XTarget”, “YTarget”, “ZTarget” and “RTarget” which are assigned respectively with floating point (REAL) values 900.0, 800.0, 300.0 and 0.0.

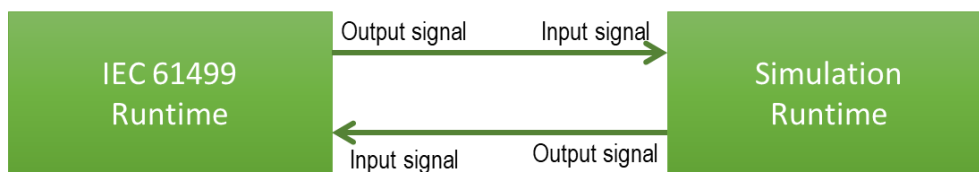
```
{
  "FI_QDP.FieldInterface_Cartesia": {
    "ts": 0,
    "Param": {
      "XTarget": 900.0,
      "YTarget": 800.0,
      "ZTarget": 300.0,
      "RTarget": 0.0
    }
  }
}
```

### 5.3.1.2 Signal direction

Both in the formal data model and in its JSON conversion, there is not any indication about the direction of the signals; there is no property neither dedicated section of the data structures telling whether a signal is an input or an output.

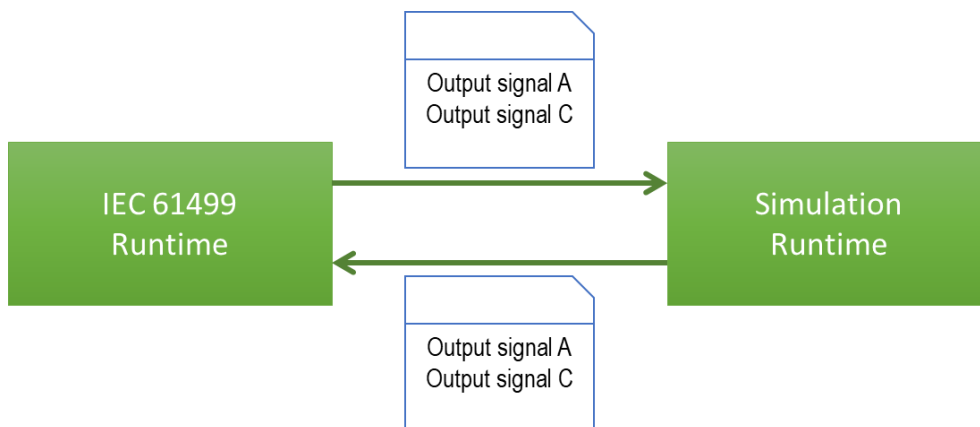
The reason for such an approach is that, in fact, input and output definitions are related to a certain subject system; considering the perspective of the automation runtime, the output signals are the data exiting the Function Blocks and going towards the physical devices in the real world and towards their digital counterpart in the virtual commissioning world, while the input signals are the ones flowing the opposite direction.

Nevertheless, the definition is perfectly specular when considering the perspective of the simulation runtime, and the Figure 18 highlights this symmetry.



*Figure 18 Complementary direction of the signals between runtimes*

A definition of Input and Output concepts within the data models would have possibly led to ambiguity. The solution accepted to overcome the problem is reporting within the payload only the signals outgoing the sender runtime. In fact, a message is generated by one of the two runtimes asynchronously when an event is triggered by the internal logics (be it automation or simulation logics) and this message flows from the publishing runtime to the consumer one.



*Figure 19 Each signal contained in the payload is an output*

Therefore, the only signals that makes sense reporting within a message are the ones that exit the system that generates them. In this way every ambiguity is avoided because, the receiving system knows that the data contained in the payload must be treated as input, then parsed and routed to the correct destinations. This approach further lightens the communications removing the burden of parsing and interpreting useless information.

### 5.3.2 Connection on WebSocket

The first implementation of the communication channel has been developed basing on the WebSocket technology.

WebSocket is a communication protocol that provides full-duplex data transfer channels over a single TCP connection. It is a standard formalized as RFC 6455 in 2011 by IETF and then accepted as WebSocket API standard from the W3C consortium.

WebSocket has been designed to be compatible with HTTP to exploit the capability of HTTP be to easily compliant with the security policies of the network infrastructure. In fact, it initializes the connection with a standard HTTP handshake running on ports 80 or 443 and then it exploits the HTTP Upgrade Header to change the protocol from HTTP to WebSocket.

WebSocket is located at Level 7 of OSI layer, as the HTTP protocol, and manages the full bidirectional continuous data exchange with a very low overhead, making it suitable for quasi real-time communications with low latency.

Nevertheless, due to its low-level nature, WebSocket does not satisfy the requirements defined for the communication channel. In particular, the sender provides an arbitrary UTF-8 or binary payload, and the receiver is notified of its delivery when the entire message is available. WebSocket defines a Binary Framing Layer that splits application binary or text messages produced by a sender into frames with a really small overhead, transports them to destination and reconstructs the original message, notifying the receiver. The content of the message is completely defined by the application.

The payload of the message has been defined according to a preliminary version of the I/O data model, called CPS-Protocol (see 5.3.2.1), implemented in JSON exploiting:

- the ease of encoding/decoding of the function block and digital counterparts' signals
- the platform independence thanks to the JSON text format, which does not imply any binary adaptation that could be related to the differences in the processor architecture<sup>5</sup>

#### 5.3.2.1 The CPS Protocol

The CPS Protocol represents historically the first translation of the IO Data model presented in the previous chapter, which has been developed mainly by NxtControl and evolved during the thesis research activities with the MQTT approach presented in the following sections.

From a structural point of view, it is based on the definition an extended set of messages that are functional to the server-client approach of the WebSocket, handling not only the continuous flow of I/O signals but also the initialization and the conclusion of the communication sessions. In the WebSocket instantiation of the runtime channels, the automation solution acts as the server while the simulation engine represents the client. The following table summarizes the phases supported by the protocol and the flow of the involved messages.

<b><i>Phase</i></b>	<b><i>Messages flow</i></b>
<i>Start communication</i>	Client: sends a "Connect" message with enable=true  Server: replies with a "ConnectionReply" message

---

<sup>5</sup> Typically the simulation engine runs on a Intel 64 bit architecture of a PC, while the automation solution is run a 32 bit PLC hardware

<i>I/O exchange</i>	Client/Server: "SendValue" message flows in both directions with multiple {"name", "value"} entries for each transferred variable. A reply is provided only if an error occurs (see Error phase)
<i>Disconnect</i>	Client: send a "Connect" message with enable=false  Server: replies with a "ConnectionReply" message
<i>Error</i>	Client/Server: send "SendValueReply" messages to report errors for unknown variables, wrong types, or wrong values.
<i>Query</i>	This phase can be applied during the engineering process to check if the interfaces between client and server match.  Client: sends a "QueryAll" message  Server: replies with "QueryAllReply"

*Table 5 CP-Protocol phases and involved messages*

The structure of each message type is briefly documented in the following paragraphs using some examples for reference.

### **Connect**

Message used by the client to initialize or conclude a virtual commissioning session, depending on the value assigned to the "enable" field.

Example of message for initializing a connection:

```
{
  "Connect" : {
    "Id" : "U2ltQ2xpZx50IHFwNDEyIHRlc3RTaW1DJ25BcHAgcjEwMjY=",
    "enable" : true
  }
}
```

The "Id" is the base64 encoded string representing a unique identifier of the Client initialization the connection.

Example of a message used to close a session:

```
{
  "Connect" : {
    "Id" : "U2ltQ2xpZx50IHFwNDEyIHRlc3RTaW1DJ25BcHAgcjEwMjY=",
    "enable" : false
  }
}
```

The "Id" reported when closing a session must be recognized as an alive client by the server.

### *ConnectReply*

Message sent in reply to a "Connect" during the initialization phase:

```
{
  "ConnectReply" : {
    "Id" : "U2ltQ2xpZx50IHFwNDEyIHRlc3RTaW1DJ25BcHAgcjEwMjY=",
    "enable" : true,
    "ServerName" : "VGVzdEpzb25Xc1NpbUNvbm5lY3Rpb24=",
    "result" : 200
  }
}
```

The "Id" is an echo of the client identifier contained in the "Connect" message, while the "ServerName" property identifies the responding automation server.

If the connection has been accepted by the server, the error code is 200 (HTTP OK code to indicate no errors occurred), otherwise other error codes are reported together with an error message. Error codes of type 400 represent temporary failures that could be recovered issuing a retry message, while 500 type results represent permanent failures of the server.

Example of reply message sent during the finalization of a session:

```
{
  "ConnectReply" : {
    "Id" : "U2ltQ2xpZx50IHFwNDEyIHRlc3RTaW1DJ25BcHAgcjEwMjY=",
    "enable" : false,
    "ServerName" : "VGVzdEpzb25Xc1NpbUNvbm5lY3Rpb24=",
    "result" : 200
  }
}
```

```
}
```

### *SendValue*

SendValue messages are used in both directions, to and from the NxtControl automation runtime, and continuously flow after connection initialization. The protocol does not force any constraint on the number or priorities of the messages, neither on the system that should send the first message.

Example of a SendValue message:

```
{
  "SendValue" : [
    {
      "name" : "TC1.Position",
      "value" : 0.5776
    },
    {
      "name" : "TC1.H1matrix",
      "value" : [ 9, 1, -23.4, 1e-14 ]
    },
    {
      "name" : "TC1.status",
      "value" : "T0s="
    }
  ]
}
```

It is possible to identify the similarities with the JSON implementation of the I/O data model described in §5.3.1.1, in particular the fact that each signal contains only the identifier and the current value, because the type has been checked at engineering stage. A limitation of the CPS protocol, which has been overcome with the IO Data Model is represented by the lack of the FB structure within the SendValue message. All signals are listed at the same level and their mapping on the corresponding Function Blocks require both systems (automation and simulation) to either parse the IDs and route them to the correct FB instance or to keep a lookup map to handle the correspondence of the internal signals with the external IDs. Therefore, this approach does not fully exploit the object orientation of the IEC 61499 as the proposed evolution applied in the MQTT approach.



### 5.3.2.2 Software Implementation

The implementation of the I/O communication channel over WebSocket required the development of dedicated extensions for the automation runtime and for the simulation engines, capable to handle de serialization and deserialization of the internal signals into CPS-Protocol compliant messages.

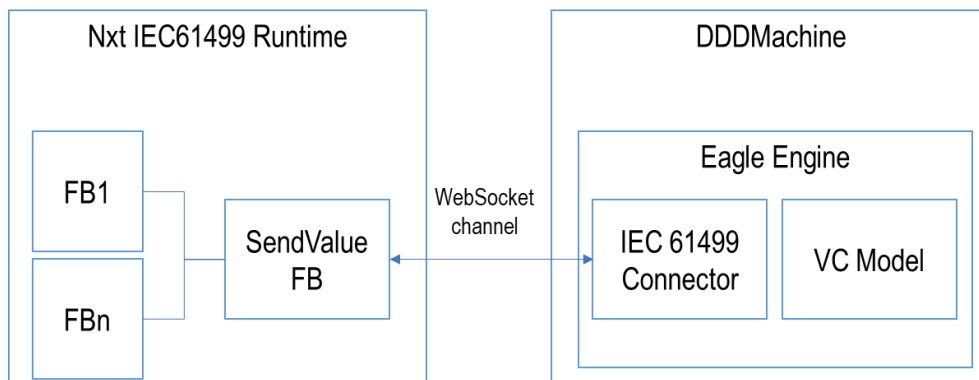


Figure 20 Architecture schema of the WebSocket channel implementation

Within the architecture reported in Figure 20 it is possible to identify the two key components that support the I/O signals exchange over WebSocket:

1. SendValue FB at automation side
2. IEC 61499 Connector at simulation side

This thesis focuses and activities carried on for the adaptation of the simulation engine while the automation architecture has been developed by NxtControl company during the research projects.

### 5.3.2.3 The IEC 61499 Connector

This component has been designed and implemented as an extension plugin of the DDDMachine virtual commissioning platform and in particular of its runtime core which is represented by the Eagle Engine component.

The Eagle Engine is a simulation model execution module whose main functionalities are:

- interfacing external systems through an abstraction layer (Adapter Layer API) dedicated to handle specific transport and applications protocols














- managing the routing of the I/O signals to the behavioral logics that control the simulated entities.

The IEC 61499 Connector module complies with the Adapter Layer API and provides to the Eagle Engine the capability to correctly cope with the above described CPS-Protocol over WebSocket.

The component therefore is responsible for:

- the initialization of the WebSocket channel: the simulation acts as client of the communication, therefore the module establishes the connection with the Automation Runtime that plays the server role;
- the management of the message format: the module encodes and decodes to/from JSON the I/O signals of the IEC 61499 function blocks according to the CPS-Protocol and the data model documented in the previous sections.

The IEC 61499 Connector module developed during the initial phase of the research activities has been coded in Java and it is composed by a set of back-end classes with well-defined and specialized tasks that collaborates to transfer data to/from the simulation.

 Bundle.properties	02/11/2020 09:14	File PROPERTIES
 ConnectInfo.java	02/11/2020 09:14	File JAVA
 ConnectionMessage.java	02/11/2020 09:14	File JAVA
 ConnectionReply.java	02/11/2020 09:14	File JAVA
 ConnectionReplyMessage.java	02/11/2020 09:14	File JAVA
 IEC61499AdapterFactory.java	02/11/2020 09:14	File JAVA
 IEC61499Connector.java	02/11/2020 09:14	File JAVA
 IEC61499Signal.java	02/11/2020 09:14	File JAVA
 Signal.java	02/11/2020 09:14	File JAVA
 SignalMessage.java	02/11/2020 09:14	File JAVA
 SignalsMap.java	02/11/2020 09:14	File JAVA
 SignalValue.java	02/11/2020 09:14	File JAVA
 WSHandler.java	02/11/2020 09:14	File JAVA

*Figure 21 List of classes composing the connector and handling the CPS-Protocol over WebSocket*

The most important object that adheres to the Adapter Layer Specification is the IEC61499Connector class, whose structure is hereby reported. All the other ones are functional to the correct operation of this data adapter.

```
/**
 * IEC 61499 Adapter Handles the Websocket communication protocol developed
in
 * Daedalus to interface IEC61499 applications.
 */
public class IEC61499Connector extends CncAdapter {

    @Override
    public boolean connect() {...}

    @Override
    public void disconnect() {...}

    @Override
    public void dispose() {...}

    @Override
    public void hfIO() throws Exception {...}

    @Override
    public void lfIO() throws Exception {...}

    @Override
    public void hfMemoryUpdate(SharedMemory memory) {...}

    @Override
    public void lfMemoryUpdate(SharedMemory memory) {...}

    @Override
    public void init(SharedMemory memory) {...}

    @Override
    public void configure(Map<String, Object> configProperties) {...}

    @Override
    public boolean isConnected() {...}
}
```

The relevant IEC61499connector methods that handle the communication are listed in the following table.

<b><i>Method</i></b>	<b><i>Description</i></b>
<i>configure</i>	<p>Allows the parametrization of the behavior of the class through a map of key-value pairs that control how the controller will establish the communication with the server. The configuration map contains several general-purpose properties that will not be documented here because they are defined by the underlying DDDMachine engine, and a set of custom entries that are specific to this adapter class that are hereby shortly documented:</p> <ul style="list-style-type: none"> <li>• host: ip address or name of the automation runtime endpoint</li> <li>• port: network port number of the server endpoint</li> <li>• timeout: milliseconds to drop the connection attempts</li> <li>• signals-map: reference to the XML file containing the whole list of I/O signals that the connector must map to/from the automation; this map is auto-generated during the design time thanks to the developed interfaces between the two engineering software applications of NxtStudio and DDD Model Editor that produce the respective automation and simulation artifacts (see §6)</li> </ul>
<i>init</i>	<p>Applies the configuration loaded by the configure method, initializing the status of the virtual commissioning model.</p> <p>During this phase all the signals defined by the signals map are resolved within the internal tasks of the engine, and their corresponding registers are created on the Shared Memory which is a runtime object maintaining a set of data registers in common between the communication cycle and the internal processing logics implemented within the simulation Task instances.</p>

	<p>The consistency of the virtual commissioning model is checked during this phase and the lookup tables to map the incoming signals to the task properties are created.</p>
<i>connect</i>	<p>If the configuration and initialization method succeeded, this method establish the connection with the automation engine opening a WebSocket channel and issuing to the server a “Connect” message to enable the I/O session.</p>
<i>hfIO</i>	<p>Each adapter, according to the Adapter Layer API, must be capable to handle high frequency and low frequency input/output operations, performed respectively by the hfIO and lfIO methods.</p> <p>The reason for such division is providing the developers with a built-in mechanism to manage at least two types of data transfers between simulation and external systems: a fast continuous flow and a slower update of less critical data. This is useful when the underlying communication libraries provide different functions to access the CNC/PLC data<sup>6</sup>.</p> <p>The IEC61499Connector is implemented to perform only the high frequency IO, because in the CPS-Protocol, there is no distinction between signals access. The method executes a two-step process:</p> <ul style="list-style-type: none"> <li>• Reads the incoming “SendValue” messages, parses them and routes the values on the local IEC61499Signal instances</li> <li>• Writes the current values of the output signals in a “SendValue” message and sends it on the output channel of the WebSocket</li> </ul> <p>It is important noting that in this phase, the incoming data values have not been already transferred to the corresponding shared memory registers.</p>

---

<sup>6</sup> An example is represented by the Fanuc FOCAS2 libraries that expose different functions to

	<p>It is possible to control the frequency of execution of this method with a dedicated parameter. Typically, the virtual commissioning models developed during validation have been configured with a refresh interval of 20 milliseconds (i.e. 50 Hz), compliant with the application types. Normally, the refresh interval should not be brought under 10 ms when running on normal PC platforms without real-time extensions of the operating system (e.g., normal Windows 10 or 11) because this value corresponds to a stable and reliable resolution of the system clock.</p>
<i>hfMemoryUpdate</i>	<p>As reported in the description of the hfIO method, the low-level signals exchange on the web socket doesn't affect the actual register instances of the SharedMemory until the hfMemoryUpdate is executed. In this way, the SharedMemory decouples the I/O thread with the simulation thread in order to avoid simulation blocks due to communication failures and vice-versa.</p> <p>This method transfers the data cached by the input IEC61499Signal instances to the memory registers and caches inside the output IEC61499Signal instances the values of the corresponding register, in order to prepare them for the next hfIO execution.</p>
<i>disconnect</i>	<p>When the user completes the virtual commissioning session, the simulation platform calls this method to interrupt the I/O session. The method sends a "Connect" message with enable=false complaint with the CPS-Protocol, waits for a reply from the server and closes the WebSocket channel. If the server does not reply to the disconnection message, the socket is closed after the expiration of the timeout.</p>

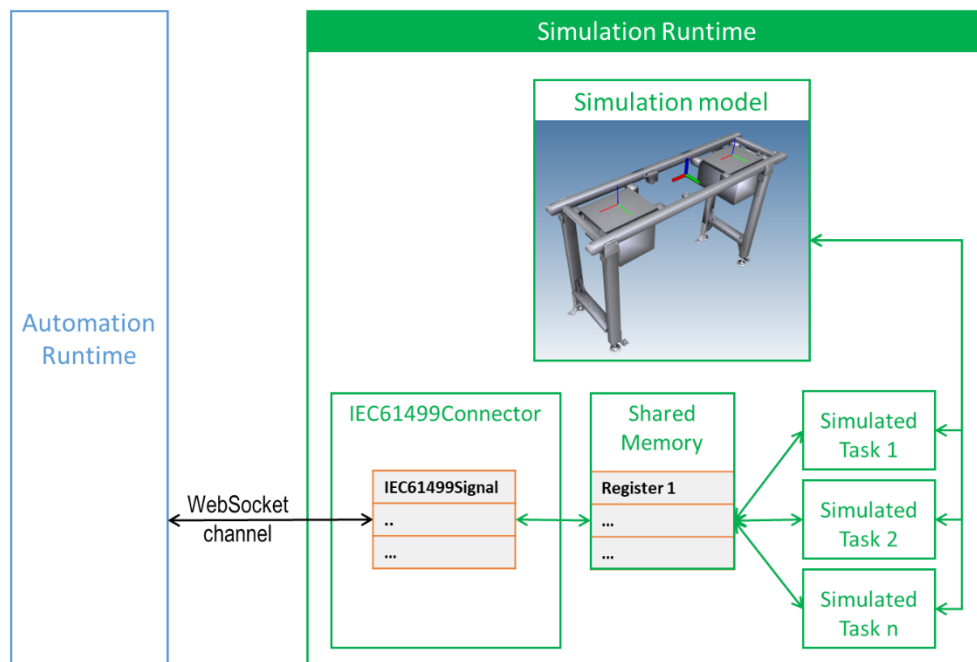


Figure 22 IEC61499Connector within the simulation engine architecture

The schema reported in Figure 22 shows how the IEC61499Connector interacts with the other components of the simulation engine of the DDDMachine NC application.

### 5.3.3 Connection on MQTT

The evolution of the runtime communication has been implemented using MQTT as transport infrastructure, to overcome the limitations of the WebSocket solution and improve the compliance level to the defined system requirements.

MQTT (MQ Telemetry Transport or Message Queue Telemetry Transport) is an ISO messaging standard [20] based on publish-subscribe paradigm that runs typically over TCP/IP. The protocol has been designed to be a lightweight and therefore suitable to support fast communication with limited resources devices. A MQTT infrastructures relies on the presence of a Broker that is a server receiving messages from the and dispatching them to the appropriate clients. A MQTT Client is any application that, relying on suitable MQTT libraries, connect to the broker over the network and either publishes or consumes messages.

The messages of an MQTT communication are organized in hierarchical topics: each publisher client declared the topic on which it wants the message to be dispatched, while each subscriber declares to the broker the set of topics it is interested in. In this way each subscriber receives only the data it considers relevant.

From the architecture perspective, MQTT promotes a complete decoupling among the actors of the communication, in fact the publisher doesn't need to know anything about the subscribers, their presence, location or number and vice-versa. This approach simplifies significantly the management of the low-level sockets, that can be turned on and off without affecting the integrity of the other components connected to the same broker.

This latter feature, together with the small footprint and the capability to support large bandwidth, makes the MQTT protocol a suitable technology to implement an evolution of the Virtual Commissioning runtime communication overcoming the limitation of the WebSocket approach, which can be identified in the following points:

- Single endpoint: with WebSocket, in order to reduce the amount of used resources, the automation solution, which acted as a server, had to limit the incoming connections as much as possible, resulting typically in a single server/single client execution of the virtual commissioning session.



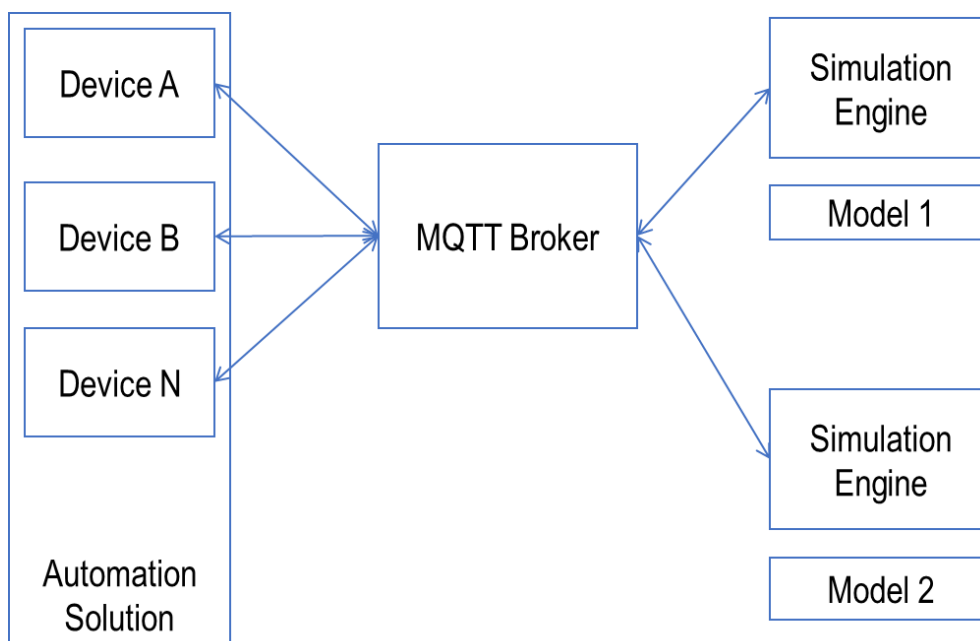
- Components dependence: the need for the simulation engine to establish a direct connection with the automation engine, required the configuration of the complete address of the device running the IEC61499 solution. Considering the fact that an IEC61499 solution is thought to be possibly distributed on several devices in a transparent way, the need for the simulation engine to identify each device server address over the network, manage a dedicated connection with it and locate the contained target function blocks, makes the set-up of complex scenarios quite hard to maintain.
- Order of initialization: the fact that, in the WebSocket implementation, the automation solution is the server, implies that it must be up and running before initializing the simulation counterpart, requiring the control over the sequence of activation of the interacting systems.

Each one of the presented issues finds a valid solution in the MQTT protocol and architecture:

- Single endpoint: with MQTT, each system participating to a communication session is a client independently of its role within the virtual commissioning scenario. Therefore, the automation solution doesn't have to allocate the resources to handle and keep alive the incoming connection requests.
- Components dependence: the complete transparency of each client to the other ones connected to the same broker, allows each engine, automation or simulation, even in multiple instances to be configured to address only the broker, produce messages that will be automatically broadcasted to different endpoints and consume information flowing from different devices, whose location can remain completely unknown. This solution is particularly important considering the distributed nature of IEC61499.
- Order of initialization: the only one component that must be up and running to initiate a virtual commissioning session based on MQTT is the broker, while the other engines, automation, or simulation, can be connected and start publishing and consuming messages in a completely asynchronous way. Moreover, the implementations of the MQTT brokers are typically characterized by a very high level of resilience, standing

continuous connections and disconnections of clients without impacts on the quality of service. This makes the overall architecture not only more fault tolerant but also simpler to control during the execution of the tests.

The following schema shows how a possible distributed automation solution could interact with many simulation models. It is important noting how the system is easily scalable to different deployment scenarios on both sides of the virtual commissioning actors.

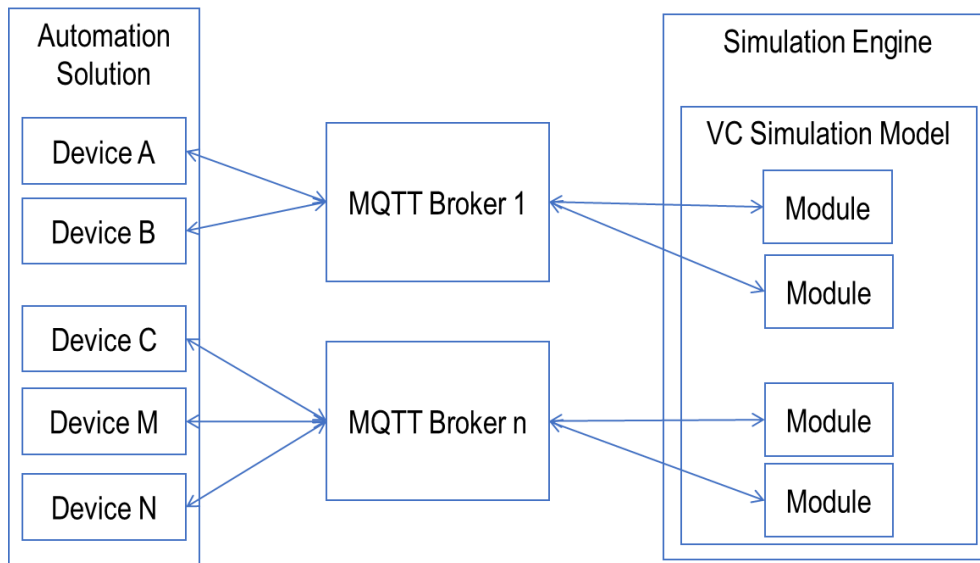


*Figure 23 Runtime communication architecture with MQTT*

The scalability of the solution could be further exploited if the system considers the deployment of multiple MQTT Brokers, which could collect and distribute messages from different portions of the automation and simulation artifacts. Figure 24 shows a possible example of deployment using two MQTT brokers managing the topics and messages flowing among portions of the two artifacts.

This possibility provided by the MQTT architecture It is extremely interesting also from a load balancing perspective because in this way it is possible to select the correct distribution of the workload on the supporting hardware network.

This approach could be leveraged in large plants where the amount of function blocks and simulation modules interacting during the same virtual commissioning session is particularly high. This scenario has been validated theoretically during the development of the engines extensions, creating tests specifically designed to evaluate the readiness of the developed libraries to support it.



*Figure 24 Scalability of the MQTT approach with multiple brokers*

#### 5.3.3.1 Software Implementation

As for the corresponding WebSocket solution, the implementation of the runtime communication architecture over MQTT required the development of dedicated extensions for the automation runtime and for the simulation engines.

From the software point of view this approach is not only more scalable and flexible, but it is also simpler because each engine connects to the MQTT broker as a client, and all the handshaking details can be dropped from the protocol because they are completely handled by the broker.

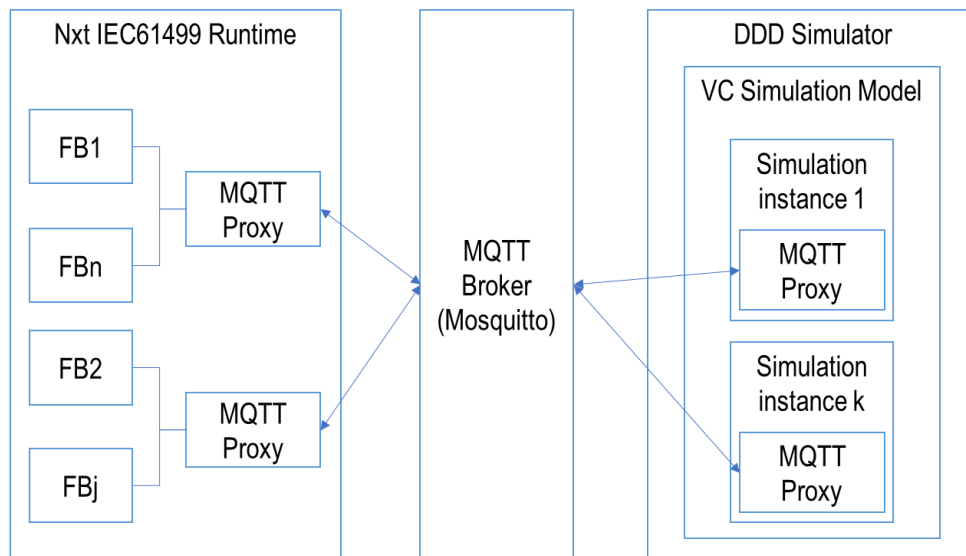


Figure 25 Software architecture implementation

The schema reported in Figure 25 shows the main components of the reference automation and simulation environments that have been adapted and extended in order to test the proposed infrastructure:

- The MQTT broker
- The MQTT Proxy at automation side
- The MQTT Proxy at simulation side

The architecture is considerably different if compared to the one presented in the WebSocket implementation, not only because of the presence of the MQTT broker, but also because the solution, at simulation side, is based on a different engine, the DDD Simulator, more suitable respect to the DDD Machine, to exploit the scalability of the system, thanks to its built-in modularity.

This research work focuses on the simulation side, therefore the following paragraphs document the extensions at simulation engine side, while the corresponding components at automation side have been implemented and provided by NxtControl.

#### 5.3.3.1.1 The MQTT Broker

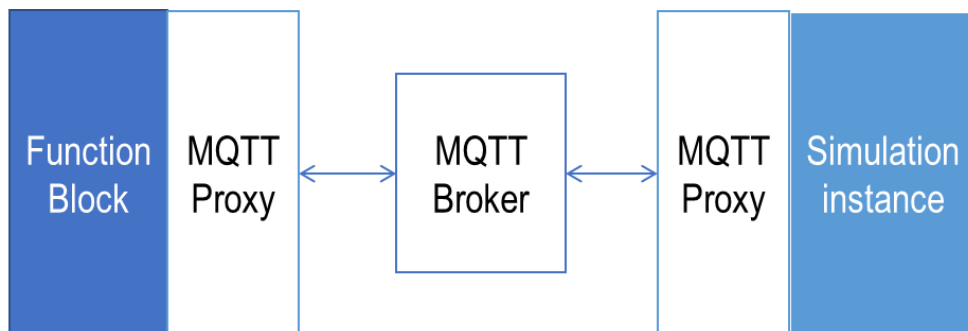
From a deployment point of view, the selection of the MQTT Broker component doesn't affect in any way the operating principle of the system, providing that the broker supports at least the MQTT 3 specification. There exist several open-source implementations of MQTT brokering systems whose licenses adapts not only to research activities but also to possible commercial scenarios.

During the implementation and validation phase, the deployments have been based on the Eclipse Mosquitto™ which is an open source (EPL/EDL licensed) message broker implementing the MQTT protocol versions 5.0, 3.1.1 and 3.1; it is lightweight and suitable for use on all devices from low power single board computers to full servers [21].

#### 5.3.3.1.2 The Simulation MQTT Proxy

As presented in section §3.4, a DDD Simulator model is based on the composition of several simulation instances whose behavioral logics, coded in Java, is capable to perform complex actions on the underlying 3D kinematics entities controlling their evolution over time.

The MQTT Proxy at simulation side has been designed to work as an extension of the behavioral logics of the instances in order to provide each one of them to interact independently with the communication runtime and exchange data with a portion of the automation code.



*Figure 26 Correspondence between Simulation Instance and device Function Block*

In order to exploit at maximum, the object oriented approach supported on both side of the virtual commissioning components, the coupling of simulation and automation functional units has been founded on the assumption that each physical device, represented by a simulation

instance, is governed by a dedicated IEC 61499 Function Block, controlling a set of I/O signals compatible with the real system. This correspondence tends to align the organization of the simulation model according to the structure of the automation code, easing the development process of the virtual commissioning sessions. Moreover, this is a key element to reach one of the most important objectives of the whole thesis work: the automatic synchronization between the environments at design time (described in Chapter 6).

The Simulation MQTT Proxy is constituted by a set of library Java classes that conform to the Heron Simulation API of the DDD Simulator platform.

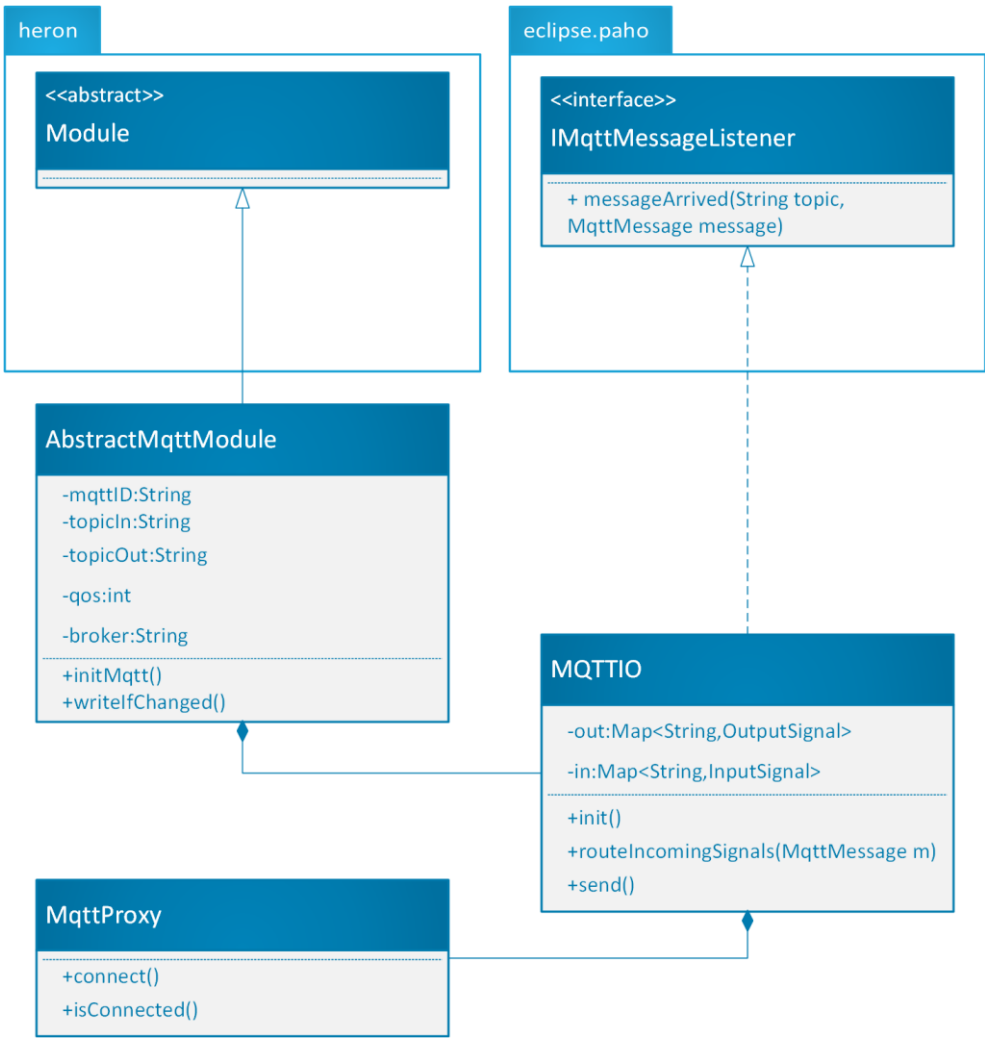


Figure 27 UML Class diagram of the Simulation MQTT Proxy

Eclipse Paho project [22] has been chosen as MQTT client library: it provides open source implementations of MQTT V3 and V5 in a variety of programming languages, Java included and it represents a reference for working with IoT and MQTT because of its reliability, maturity and availability of documentation.

Figure 27 reproduces with UML Class Diagram notation the structure of the classes and their relationship with the reference packages that are:

- Heron Simulation API: com.ttsnetwork.heron.v5
- Eclipse Paho: org.eclipse.paho.client.mqttv3

The following paragraphs document the classes and their responsibilities within the infrastructure.

### *MqttProxy*

Represents the low-level access to the MQTT interfaces, wrapping the basic mechanisms of the Eclipse Paho library for connecting to the broker. This class has been developed to act as an abstraction with the underlying MQTT client library, decoupling the higher-level classes like MQTTIO from the specific implementation. The following table contains the main methods and their description.

<b>Method signature</b>	<b>Description</b>
<b>connect():void</b>	Initiates the connection with the broker, opening the channels, identifying the client and with a unique UUID and declaring the desired protocol version.  The parameters of the connections are defined by the internal attributes set at construction time by the containing classes.
<b>isConnected():boolean</b>	Provides a way to test the current connection status of the proxy.

*Table 6 MqttProxy details table*

## MQTTIO

This class represents the core of the MQTT runtime communication at simulation side and it is responsible for:

- The management of the MqttProxy instance, starting the connection, declaring the topics of interest, and registering as a listener on the message queue, in fact it is an implementation of the IMqttMessageListener interface of the Eclipse Paho client library;
- the management of the application communication protocol, serializing and deserializing the I/O signals into JSON messages compliant with the IO Data Model described in § 5.3.1;
- auto-wire of module signals: using the Java reflection API, this class inspects a module structure identifying input and output signals and creates an internal cache of signal references (represented by the “in” and “out” maps in the UML class diagram);

The following table contains the description of the main methods.

<b>Method signature</b>	<b>Description</b>
<b>init():void</b>	Initializes the MqttProxy instance and, if the connection has been established, registers itself as a listener on the message queue of the specified topic containing the signals flowing from automation to simulation.
<b>messageArrived(String topic, MqttMessage message):void</b>	This method is the realization of the IMqttMessageListener interface and is the endpoint receiving the incoming JSON Messages.
<b>routeIncomingSignals(MqttMessage m):void</b>	Deserializes the incoming MqttMessage JSON payloads, identifying the InputSignals involved. Then it translates the values in the proper data type and routes them to the destination module.



<b>send():void</b>	The class register for the internal notification of changes in the OutputSignal instances of the module it belongs to. When an output value changed, the MQTTIO serialize the new signal values into a proper JSON payload and sends it through the MqttClient provided by the MqttProxy instance.
--------------------	--

Table 7 MQTTIO details table

### **AbstractMqttModule**

This class represents an extension of the base Module class that all simulation module instance inherits from. It is an abstract class, meaning that it cannot be directly instantiated but must be extended by the module classes that need to interface with the automation.

It exposes to the extending classes the infrastructure to cope with the MQTT communication runtime without having to deal with all the low-level implementation details needed to handle the connections and manage the application messaging protocol. In this way the instance module developers, when creating the digital twin representations of the real devices, can concentrate on the high-level coding the behavioral logics.

This class delegates to an internal instance of MQTTIO the management of the protocol and defines the module parameters that must be specified in order to correctly configure the low-level channel. This is the reason way the MUL class diagram reports, for the AbstractMqttModule not only the main operations, but also a set of relevant configuration attributes. Both attributes and operations are described in the table below.

<b>Attribute</b>	<b>Description</b>
<b>mqttID:String</b>	Client identifier, it is used by the MQTT broker to distinguish the clients. Each client must provide a unique ID in order to be accepted by the broker.

	Typically this ID is set to the module id string which is unique within a simulation.
<b>topicIn:String</b>	Name of the topic on which the client expects to receive the messages sent by the automation. This string conforms to the hierarchical format of the MQTT topics.
<b>topicOut:String</b>	Name of the topic the client uses to send the outgoing signal values to the automation. This string conforms to the hierarchical format of the MQTT topics.
<b>broker:String</b>	<p>URL of the broker. This string must contain the chosen transport protocol, the host, and the port on which the connection should be established.</p> <p>Typically the communication is configured to work on TCP, therefore the broker attribute has the following shape:</p> <pre>tcp://[broker_host]:[broker_port]</pre> <p>The standard port is 1883.</p>
<b>qos:int</b>	<p>Sets the quality of service of the communication according to the three levels defined by the protocol to control the delivery of messages:</p> <p>0: At most once (fire and forget, fastest)</p> <p>1: At least once (slower, with cache until delivery)</p> <p>2: Exactly once (secure but the slowest)</p>
<b>Method</b>	<b>Description</b>

<b>initMqtt():void</b>	Initializes the full proxy infrastructure for the module instance, wiring the signals to the MQTT queues and starting the client instance using the specified parameters
<b>writelfChanged(OutputSignal out, T value):void</b>	Writes the values of an output signal if the value has been actually changed, otherwise avoids calling the output signal setValue operation.  This method aims at reducing the number of redundant events that could generate useless MQTT traffic towards automation.

*Table 8 AbstractMqttModule details table*

When implementing an extension of the AbstractMqttModule, it is important that the specialization class follows few directives in order to ensure the correct initialization of the communication and the in and out data flows:

- 1- The class must be a direct or indirect extension of AbstractMqttModule
- 2- The class declares input signals as public access class attributes of type InputSignal and these attributes must be initialized before initializing the MQTT
- 3- The class declares output signals as public access class attributes of type OutputSignal and these attributes must be initialized before initializing the MQTT
- 4- The class implements the init() method as all the simulation logics modules
- 5- The init() method, after initializing all other internal structures calls the initMqtt method of the super class
- 6- If the class needs to reduce the amount of events generated by output signals, instead of directly writing the OutputSignal values, it must call the writelfChanged method of the super class.

The code below reports a possible example of extension following the aforementioned rules:

```

/** Module class to be interfaced with automaton through MQTT */
public class Extension1 extends AbstractMqttModule {

    public InputSignal input1 = new InputSignal(PropertyType.DOUBLE);
    public InputSignal input2 = new InputSignal(PropertyType.INTEGER);

    public OutputSignal output1 = new OutputSignal(PropertyType.DOUBLE);
    public OutputSignal output2 = new OutputSignal(PropertyType.INTEGER);

    @Override
    protected void init() {
        if (broker != null && !broker.isEmpty()) {
            // perform initialization of the internal logics
            initMqtt();
        }
    }
}

```

#### 5.3.3.1.3 MQTT Topics structure

As documented in the IO Data Model section, the JSON implementation does not contain any specification of the signal direction: no distinction between input and output signal is done at message payload level. In the WebSocket, since it is an end to end communication, the direction of the messages is immediately clear because they flow from the sender to the receiver without any intermediary.

With the MQTT solution instead, the presence of the MQTT Broker represents a point of distribution of data that can create confusion on the direction of the messages. The same client in fact can be publisher and subscriber on the same topic creating a loopback on it won messages. For this reason, in order to distinguish the semantics of the signals, the hierarchical structure of the topics within the MQTT broker has been exploited.

Therefore, the topics has been organized to reflect:

- the flow direction of the signal messages;
- the identifiers of the connected simulated and controlled devices.

In particular the identifier of the topic is composed as follows:

**VirtualCommissioning/{System\_name}/{module\_id}**

The “**VirtualCommissioning**” root of the topic URL represents a common namespace for all the topics.

**{System\_name}** can be assigned with “Automation” or “Simulation” depending if the topic contains messages generated respectively by the Automation Solution or from the Simulation Model.

**{module\_id}** is the unique identifier of the entity belonging to the simulation and automation artifacts that is generating the signals. This choice has been based on the assumption, already discussed in the previous paragraphs, that each IEC 61499 Function Block exchanging signals with the external world, has a corresponding digital twin on within the Simulation Model, identified with the same name.

Therefore, considering the existence of an IEC 61499 Function Block called “ConveyorA” in the NxtControl Runtime and a corresponding simulation instance in the DDD Simulator model:

- The IEC 61499 FB:
  - Publishes on topic “VirtualCommissioning/Automation/ConveyorA”
  - Subscribes to topic “VirtualCommissioning/Simulation/ConveyorA”
- The Simulation instance:
  - Publishes on topic “VirtualCommissioning/Simulation/ConveyorA”
  - Subscribes to topic “VirtualCommissioning/Automation/ConveyorA”

It is important underlining that this possibility of publishing and subscribing on named topics for each virtual commissioning entity depends on the capability of the MQTT Broker to dynamically create topics whenever at least one client declares the intention to publish to it or to subscribe to it, even if nobody is going to send/receive on that particular channel.

## 6 Connecting automation and simulation at design time

If the possibility to connect the automation and simulation runtimes with high performance communication channels represents the key enabler for each virtual commissioning session, it is not yet the most important breakthrough carried out by the research activities of this work. The proposed approaches at runtime, documented in the previous chapter are really effective because they exploit the capability of the IEC 61499 standard to execute event based and object-oriented logics coping perfectly with a modular structure of the simulation model.

Nevertheless, they don't fill the wider technological gap that make the creation of virtual commissioning models a complex task that often prevents automation engineers to adopt it for the everyday work. In this chapter, an architecture capable to support the design process of virtual commissioning sessions is presented, starting from the presentation of the objectives and the main requirements until the documentation of the developed infrastructure, whose validity has been proved during the testing phase.

### 6.1 Objectives and requirements

When a virtual commissioning model is defined typically two professional roles are involved in the development process:

- The automation expert who is responsible for the mechatronic system logics and who is expert in process control
- The simulation expert who is responsible for the realization of a digital avatar of the real world capable to react to the same signals that the automation sends to the physical devices and provide reliable and realistic feedback signals

The two characters use different development environments: the former works with an automation-oriented IDE (like `nxtStudio` for IEC 61499) conceived to support the coding of function blocks, of their internal state machines and their internal and external wiring, while the latter is used to operate in a 3D environment, dealing with the programming of geometric and kinematics models.

The exchange of knowledge between the two actors can be really slow and require several hours of work to align the behavior of the two models. This affects the effectiveness of the approach, in particular when the automation logics is under development so that its incremental growth becomes too fast to be synchronized with a digital counterpart.

The main objective of this second part of the research is implementing an infrastructure capable to integrate the automation and simulation IDE applications, using a common data model that automatizes the concurrent evolution of the two artifacts, making it possible to exploit virtual commissioning from the early design down to the use phase of the industrial systems.

During the initial activities, the main requirements for the system have been elicited together with a consortium of international partners working with automation and simulation.

The following table reports the specifications of the requirements. As for the runtime, each requirement is defined by:

- an ID label that has been used to track the progress during the development and to fill a validation report during the functional tests
- a priority level that allowed to schedule the implementation activities; the priority has been formalized according the following three levels (mediated from the common keywords applied in requirements elicitation and quality management):
  - SHALL: the final result must completely satisfy the requirement to be positively evaluated;
  - SHOULD: (equivalent to the keyword RECOMMENDED) the requirement is important and should be satisfied by the final result, but it can be accepted also a final result that meets the specification only partially: in this case the implications must be understood and justified;
  - MAY: the requirement is non mandatory and meeting it would represent an enhancement respect to the optimal baseline.
- A description of the desired behavior containing a base indication of possible acceptance criteria to be verified during the validation phase

<b>Requirement ID</b>	<b>Description</b>	<b>Priority</b>
D001	<p>The integration layer must allow the user to automatically create digital counterparts of the automation code written in IEC 61499.</p> <p>Acceptance criteria:</p> <p>The automation IDE succeed in triggering the instantiation of simulation prototypes in the simulation IDE</p>	SHALL
D002	<p>The integration layer must define automatically the signal mapping between function blocks and simulation entities.</p> <p>Acceptance criteria:</p> <p>The simulation instances created through the integration interface are correctly configured with the I/O signals compatible with the corresponding Function Blocks. The end user should not tweak in any way the signals mapping to run the virtual commissioning session.</p>	SHALL
D003	<p>The simulation model must be a synchronized representation of the automation function blocks.</p> <p>Acceptance criteria:</p> <p>The automation IDE succeeds in creating, deleting, and updating digital twins of the Function Blocks through the integration interface without the need to access the simulation IDE.</p>	SHALL



D004	<p>The integration layer must support a modular object-oriented approach.</p> <p>Acceptance criteria:</p> <p>The integration works with units that have the granularity of the Function Blocks</p>	SHOULD
D005	<p>The integration layer must be based on an open Digital Avatar Data model compatible with the runtime IO Data Model and its implementations.</p> <p>Acceptance criteria:</p> <p>At the end of the development exist an open API and a supporting data model that can be mapped to the runtime JSON implementation of the IO Data Model.</p>	SHALL
D006	<p>The integration architecture must be platform independent; it must be possible to apply the same approach to automation and simulation platforms different from the ones used during the validation.</p> <p>Acceptance criteria:</p> <p>...</p>	SHOULD

D007	<p>The integration must be based on a cross platform technology. The IDE applications are typically developed on different software platforms (Java, C++, C#, etc.), the chosen technology must ease the implementation of the communication stubs.</p> <p>Acceptance criteria:</p> <p>The integration layer can be interfaces independently from the underlying language.</p>	SHOULD
D008	<p>The integration layer must support the compilation and deployment phase of the artifacts at simulation phase.</p> <p>Acceptance criteria:</p> <p>At the end of the development the automation developer, within the automation IDE user interface, without accessing the simulation IDE, compiles the simulation model and creates the corresponding runtime artifact.</p>	SHALL
D009	<p>The integration layer must support the control (start/stop) of the execution of the virtual commissioning sessions, running the automation and simulation engine and connecting them without the need for the automation developer to interacts with the simulation IDE.</p> <p>Acceptance criteria:</p> <p>At the end of the development the automation developer, within the automation IDE user interface, without accessing the simulation IDE and without any further configuration, starts and stops the virtual commissioning session.</p>	SHOULD

D010	<p>The integration layer must be bidirectional, supporting not only the control flow from automation to simulation IDE but also the notifications in the opposite direction.</p> <p>Acceptance criteria:</p> <p>The automation IDE is notified when the requested operations are completed in the simulation IDE.</p>	SHOULD
D011	<p>The integration layer must be resilient to modifications of the simulation model that affect the positioning and parametrization of the digital twins of the Function Blocks.</p> <p>Acceptance criteria:</p> <p>The automation developer modifies the simulation model created by the integration layer directly in the simulation IDE, changing the position (translation and rotation) of the instances and their parameters (speeds, accelerations, etc.) without affecting the synchronization between the two applications.</p>	SHALL

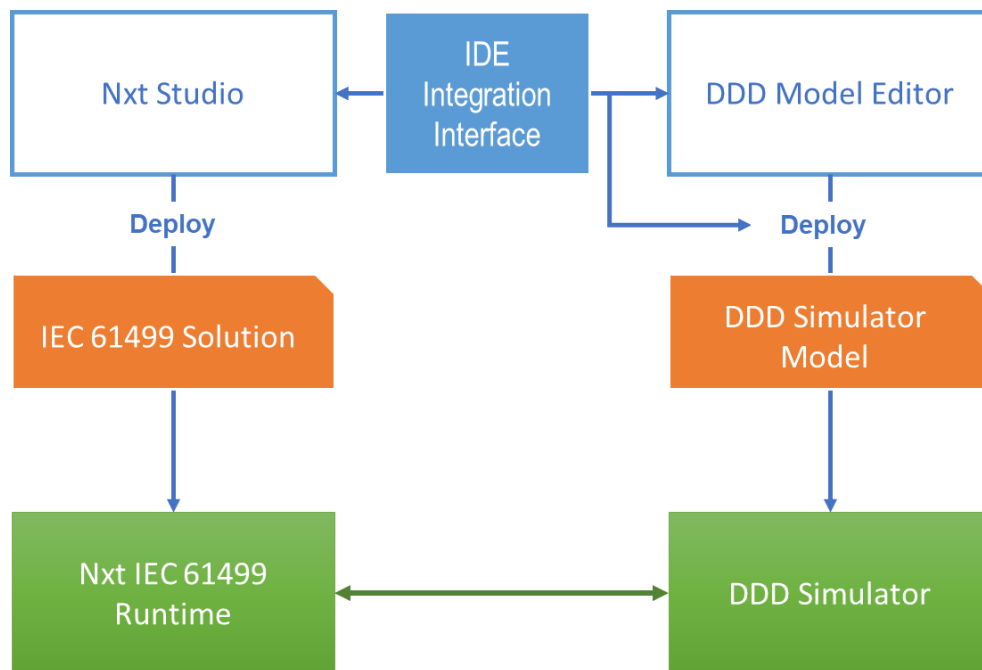
D012	<p>The integration layer should have a small footprint. The concurrent execution of two complex IDEs like the automation and simulation can require a significant amount of system resources, therefore it is important to avoid that a further increase of this consumption due to the communication between the applications causes instabilities or even system crashes.</p> <p>Acceptance criteria:</p> <p>The integrated IDEs running on a middle level laptop (e.g., PC with 16 Gb of RAM, Intel i5 processor and discrete NVIDIA Graphic Card) works smoothly without causing system blocks.</p>	MAY
------	---	-----

*Table 9 IDE integration requirements*

The verification of the acceptance criteria for each requirement is documented in Chapter 0 and is based on global validation scenarios demonstrating the whole virtual commissioning process from design to execution.

## 6.2 Implementation

The implementation of the integration at IDE level is based on the high-level architecture proposed in section §4.2 and whose schema is reported here in for convenience, highlighting the position of the interface.



*Figure 28 Integration high level architecture*

Even though the development has been carried on using the two selected environments `nxtStudio` and `DDD Model Editor`, the proposed solution can be adopted to any IEC 61499 automation IDE and any modular simulation engine.

The realization of the interface followed an iterative spiral process with the preparation of early prototypes to be validated and evolved towards the final released version.

The interface is composed of two main complementary components that concur to enable the communication between the two environments:

1. The Digital Avatar Data Model
2. The integration API

### 6.3 Digital avatar data model

When configuring the simulation entities to behave like the real devices, it is necessary to wrap a pure digital behavior into a container capable to mimic the interfaces of the physical units. This means defining artifacts capable of consuming the same signals of the physical systems, elaborating a reaction and generate the same responses.

The digital avatar data model aims at defining the structure of the digital counterparts of the IEC 61499 Function Blocks, composing its interface with its internal logics, supporting the parametrization of the whole entity.

From an architectural perspective, these objects are similar to the simulation prototypes adopted by the DDD Simulator platform, because they must formalize:

1. A set of configuration parameters
2. A set of I/O endpoints corresponding to the automation I/Os
3. A set of logical tasks that must be executed to react to external or internal events

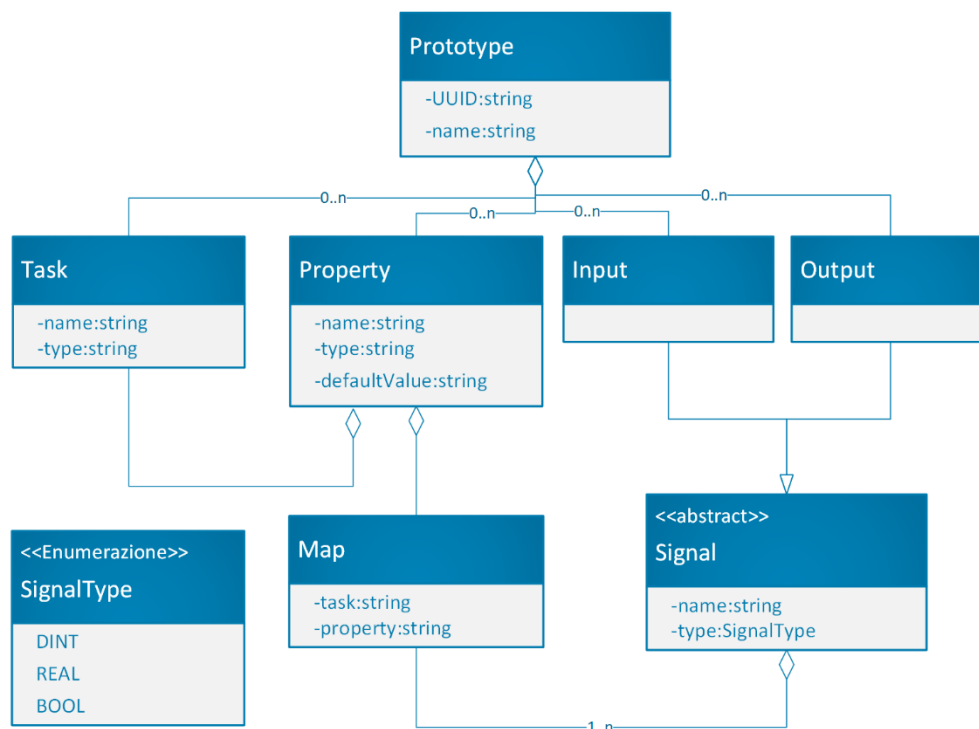


Figure 29 Digital Avatar Data Model UML Class Diagram

The UML Class Diagram reported in Figure 29 shows the organization of the classes that compose the Digital Avatar Data Model. The following section provides a brief description of the role of each class.

### *Prototype*

Represents the root of the data model, acts as a container for the description of an entity mapping to virtual commissioning. The role of this container is mainly to behave as a model glue that allows different concepts like signals, parameters and task to be cross linked and maintain the consistency needed to manage them with an automatic generation system.

The prototype is a blueprint, a structure that is meant to be replicated by the corresponding instances with different parameter values.

### *Input and Output*

Represent the physical signals that will be exchanged, at runtime, through the communication channel, with the IEC 61499 Function Blocks. Their role is not only defining the external interface of the prototype, but also to route the incoming data to the correct internal endpoints of the tasks and localize the task properties that provide the values to be published outside.

### *Signal*

It is a convenience super class for Input and Output classes, providing common attributes and allowing the management systems to handle the I/Os in a consistent way.

### *Property*

The property is used at two different levels, as a configuration parameter of the prototype and as a property of a specific logs task. It always represents a customization parameter that can be controlled from outside the prototype and whose actual value concur to determine the features of a particular instance of the prototype.

### 6.3.1 XML implementation of the data model

The presented model has been transformed into a file format to be used by the simulation and automation IDE applications to share prototype definitions. Collection of prototypes are organized into libraries called catalogue that the two environments share as a common modelling layer.

The file format chosen for the prototype files is the XML, coherently with the file format defining the structure of the simulation entities in the DDD Model Editor platform (see §3.4.5).

The following paragraphs document the XML data types of the Digital Avatar Data Model implementation and their relationships.

#### <prototype>

<b>Attributes</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>uuid</b>	<i>string</i>	Unique identifier of the prototype in the form of a UUID, it must be different for all the prototypes	Required
<b>name</b>	<i>string</i>	Name of the prototype: it is the human readable identifier	Required
<b>Elements</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>&lt;input&gt;</b>	<i>&lt;input&gt;<sup>*7</sup></i>	Set of input signals that the prototype can receive and that is capable to handle.	Optional
<b>&lt;output&gt;</b>	<i>&lt;output&gt;*</i>	Set of output signals that the prototype is capable to publish through the runtime communication channel	Optional
<b>&lt;property&gt;</b>	<i>&lt;property&gt;*</i>	A variable set of properties that can be used as configuration parameters of the prototype. Unlike the simulation entity prototype, there isn't a predefined set of configuration properties, the list changes	Optional

---

<sup>7</sup> The \* next to a type definition indicates that the current structure contains a collection with 0 or multiple definitions of the same type-



		according to the IEC 61499 Function Block that must be interfaced. These properties are mapped internally to the configuration properties of the tasks.	
<b>&lt;task&gt;</b>	<i>&lt;task&gt;*</i>	Characterization of the behavior of the prototypes	Optional

#### **<input> and <output>**

The <input> and <output> XML elements share exactly the same type definition because semantically they are a representation of the same type of objects.

<b>Attributes</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>name</b>	<i>string</i>	Name of the signal; it is especially important because this name is the same identifier used by the runtime communication to compose the JSON payload of the messages exchanged by automation and simulation engines.  This name must unique locally to the prototype in the same way the names of the events of the Function Block are unique. The automatic mapping of the signals in two direction is based on this correspondence.	Required
<b>type</b>	<i>string</i>	String identifying the IEC 61499 type of the signal to be exchanged. This information, in conjunction with the unique name, ensure the correct coupling between the artifacts and the needed low level type adaptations. The values that can be assigned to this attribute are defined in the SignalType enumeration.	Required
<b>Elements</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>&lt;map&gt;</b>	<i>&lt;map&gt;*</i>	Set of mapping elements that wire the signals internally to the input registers of the logics tasks. The presence of these mapping element is optional but, actually at least one <map> should be modelled; otherwise, an	Optional

		input signal, even if received by the instance of a prototype, does not trigger any simulation event and analogously an output signal would never publish value variations because it would be never updated by internal events.	
--	--	--	--

#### <map>

Convenience element needed to map concepts within the prototype definition to a property of a task.

<b>Attributes</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>task</b>	<i>string</i>	Identifier of the task addressed by this mapping.	Required
<b>property</b>	<i>string</i>	Identifier of the target property whose value will be set controlled by the mapped element value (signal or prototype property)	Required

#### <properties>

Convenience element grouping together the configuration properties of the prototype.

#### <property>

Within the defaultValue attribute it is possible to reference the current instance of the prototype using the \$ symbol that, at runtime is substituted with the ID of the instance. This is useful when referencing parts of the model like frames, joints, and sub-assemblies.

<b>Attributes</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>name</b>	<i>string</i>	Name of the property, must be unique within the prototype.	Required
<b>type</b>	<i>string</i>	Low level data type of the property. The accepted values are compliant with the basic data types listed in §3.4.6	Required

<b>defaultValue</b>	<i>string</i>	String representation of the default value assigned to the property at instantiation time. The value must be compliant with the type attribute.	Required
<b>Elements</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>&lt;map&gt;</b>	<i>&lt;map&gt;*</i>	Set of mapping elements that wire the properties internally to the configuration properties of the logics tasks. The presence of these mapping element is optional and typically: <ul style="list-style-type: none"> <li>it is present when the &lt;property&gt; element is used at prototype level;</li> <li>it is missing when the property is used internally to a &lt;task&gt; element.</li> </ul> In the former case it is important that at least one <map> is modelled; otherwise a property would not affect any behavior of the prototype instance and therefore it is useless.	Optional

#### **<task>**

A task is a piece of logics that instance that, conveniently configured, controls the evolution of a part of the simulation entity kinematics model, modifying the position of the elements, the values of the joints, the visibility of the parts, and all the runtime properties exposed by the simulation engine. A prototype can contain several task definitions, each one dedicated to mimic a single aspect of the whole device.

<b>Attributes</b>			
<b>Name</b>	<b>Type</b>	<b>Description</b>	<b>Use</b>
<b>name</b>	<i>string</i>	Unique identifier of the task within the prototype.	Required
<b>type</b>	<i>string</i>	Fully qualified name of the Java type implementing the behavioral logics of the prototype. This class must be present within the catalogue JAR libraries so that it can be	Required

		resolved at runtime by the DDD Simulator application.	
<i>Elements</i>			
<i>Name</i>	<i>Type</i>	<i>Description</i>	<i>Use</i>
<code>&lt;property&gt;</code>	<code>&lt;property&gt;*</code>	Set of configuration properties that can be modified at runtime after the instantiation of the prototype. These properties, internally are translated into values of the fields of the task class.	Optional

The following XML fragment shows an example of prototype configuration. All the libraries of virtual commissioning entities developed in the validation phase are based on this format.

```

prototype uuid="a7315040-3e5c-42cb-b5ee-b532b7e08e5d" name="Pusher">

  <input name="command" type="DINT">
    <map task="PusherSensor" property="pusherCommand_RegistryIn"/>
  </input>
  <!-- Pusher Extended TRUE:is completely extended -->
  <output name="limitExtended" type="BOOL">
    <map task="PusherSensor" property="pusherLimitExtended_RegistryOut"/>
  </output>
  <!-- Pusher status TRUE:is moving -->
  <output name="isRunning" type="BOOL">
    <map task="PusherSensor" property="pusherIsRunning_RegistryOut"/>
  </output>
  ...
  <properties>
    <property name="pusherAxisValue" defaultValue="500" type="double">
      <map task="PusherSensor" property="pusherAxisValue"/>
    </property>
    <property name="pusherSpeed" defaultValue="2" type="double">
      <map task="PusherSensor" property="pusherSpeed"/>
    </property>
    <property name="sensorColor" defaultValue="255;0;0" type="integer[]">
      <map task="PusherSensor" property="appearanceColorRGB"/>
    </property>
    ...
  </properties>

  <!-- TASKS -->
  <!-- PusherSensor task -->

```

```

<task type="synesis.sensors.PusherBoxSensorModule" name="PusherSensor">
  <property name="boxPosition" defaultValue="0 0 0" type="vector3"/>
  <property name="boxDimentions" defaultValue="120 120 20"
    type="vector3"/>
  <property name="pusherAxisName" defaultValue="$.Spintore.T"
    type="string"/>
</task>

<task type="synesis.sensors.BoxSensorModule" name="PusherLightBarrier">
  <property name="boxPosition" defaultValue="230 0 0" type="vector3"/>
  <property name="boxDimentions" defaultValue="460 10 10"
    type="vector3"/>
</task>
...
</prototype>

```

## 6.4 The integration API

If the data model provide the automation and the simulation IDE with a common ground for the definition of the digital twins of the physical devices, the Integration API layer represents the means by which the two applications collaborate to co-design the virtual commissioning solution. Through this layer, the actions performed by the automation expert induce automatic modifications of the simulation model that in this way is kept in synch with the IEC 61499 logics.

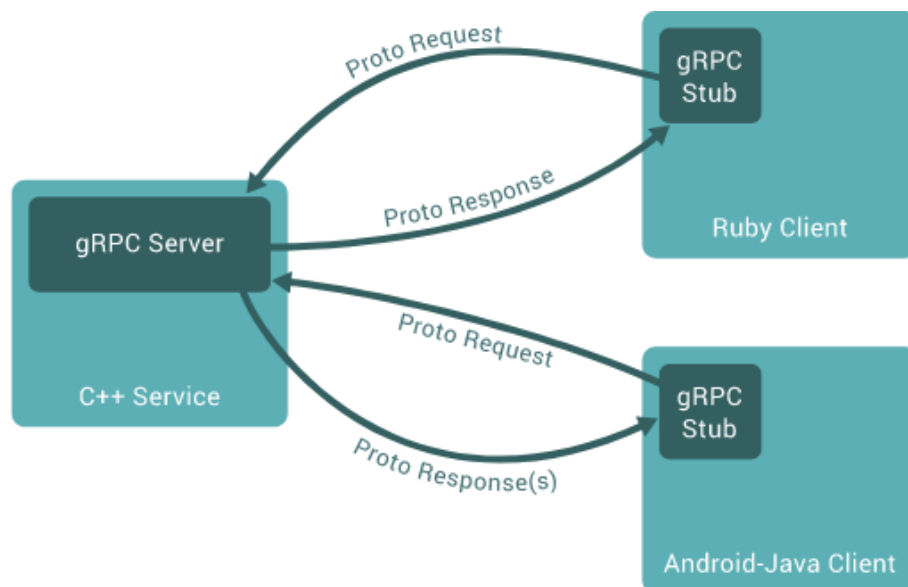
The Design Integration API can be considered the dual of the Runtime Communication Layer and the natural completion of the upstream development process. Nevertheless, the two interfaces are meant to support two so different phases that the requirements for each one are really different.

The design API supports operations happening at slow rate, triggered by the end user actions, therefore it does not demand for the same high speed and low latency required by the runtime communication. The performance in terms of throughput becomes not so important because there is no real time data exchange.

On the contrary, the Design Integration API requires the exposure of high-level functionalities, identifiable with actions or methods of a high-level software interface. For this reason, the choice of the communication protocol is based on criteria that consider the capability to efficiently support complex operations in a cross-platform approach.

A viable approach could be represented by the adoption of the RESTful API design pattern of web services providing client-server access to internal functionalities of the simulation IDE. This solution has the advantage of using very simple and accepted communication protocols based on HTTP and HTTPS but, as drawbacks, its way of exposing functionalities is unstructured, and the communication is error prone because the meaning of each service and in particular of its in/out data structures is not built in with its definition. Moreover, a RESTful API is not suitable to easily manage observer patterns unless switching to the SSE protocol (Server Sent Event) or to a pure WebSocket notification channel.

The best integration technology for this specific use case is the represented by the gRPC [23] which is a high performance, open-source universal remote procedure call framework. It provides support to most of the common development platforms used for software engineering like C++, Java, C#, and others. The gRPC framework uses Protocol Buffers, an industry ready open source mechanism for serializing structured data provided by Google, as both its Interface Definition Language (IDL) and as its underlying message interchange format.



*Figure 30 gRPC framework*

From the architecture point of view, gRPC relies on the implementation of a server artifact and of client stubs that manage the function calls in a way that is transparent both to the server and to the client applications. The framework, starting from the Protocol Buffer defined interface,

generates the server and stubs code in the specific target language of the hosting application, so that both the endpoints of the communication can use local compiled methods. Relying on a binary socket implementation, the procedure calls run not only locally but also remotely through the network. This technology brings a set of benefits that can be summarized in:

- **Transparency of the underlying transport protocol:** the application code doesn't have to comply with specific patterns like the stateless mode of the RESTful API;
- **Performance:** the default binary implementation of the Protocol Buffers is designed for performance so the impact on the normal operation times is negligible;
- **Vendor independence:** the possibility to use the protobuf format as IDL to define the API, allows to open the specification to any vendor interested in integrating with the architecture;
- **Complexity management:** the possibility to define data structures and articulated method signatures brings within the integration layer the same design patterns that can be applied with normal OOP (Object Oriented Programming) code, i.e. the observer pattern.

#### 6.4.1 The gRPC API

The requirements defined at the beginning of the chapter have been translated into use cases and then formalized into functions and data structures of the gRPC IDL using the Protocol Buffer version 3 syntax ("proto3", [24]). The following snippet reports the structure of the developed API:

```
syntax = "proto3";

package simulation.server;

/**
 * Simulation service.
 */
service SimulationService {
    ... [rcp definitions]
}

[message definitions]
```

The following table shows a synoptics overview of the main rpc signatures and the corresponding messages that have been defined, grouped according to the target object.

<b>Group</b>	<b>Procedure name</b>	<b>Input messages</b>	<b>Output message</b>
Project	<b>CreateProject</b>	CreateProjectRequest	ProjectHandle
	<b>OpenProject</b>	OpenProjectRequest	ProjectHandle
	<b>CloseProject</b>	ProjectHandle	Result
	<b>DeleteProject</b>	ProjectHandle	Result
Prototype	<b>CreatePrototype</b>	CreatePrototypeRequest	ResourceHandle
	<b>DeletePrototype</b>	ResourceHandle	Result
	<b>GetPrototypes</b>	ProjectHandle	ResourceList
Instance	<b>CreateInstance</b>	ResourceHandle	CreateResult
	<b>DeleteInstance</b>	ResourceHandle	Result
	<b>GetInstances</b>	ProjectHandle	InstanceList
	<b>SetInstanceProperty</b>	SetInstancePropertyRequest	Result
	<b>SetSignalInstance</b>	SetSignalInstanceRequest	Result
Deploy and run	<b>CompileProject</b>	ProjectHandle	Result
	<b>RunProject</b>	RunProjectRequest	Result
	<b>StopProject</b>	ProjectHandle	Result
	<b>QueryProjectRunning</b>	ProjectHandle	Result



Transaction	<b>BeginTransaction</b>	ProjectHandle	Result
	<b>CommitTransaction</b>	ProjectHandle	Result
	<b>RollbackTransaction</b>	ProjectHandle	Result

Figure 31 Synoptics view of the IDE Integration API

The full IDL in proto3 language syntax is reported in Appendix A, where each procedure has the corresponding embedded documentation and the input/output message structures are visible.

The documentation of the API is available also in auto-generated HTML format as shown in the following picture:

## SimulationService.proto

[Top](#)

### CreateProjectRequest

Request to create a new project.

Field	Type	Label	Description
name	string		project display name
path	string		full path of the new project (the directory should not exist)

### CreatePrototypeRequest

Request message to create a new prototype in a project from a prototype definition in the specified directory.

Field	Type	Label	Description
handle	uint32		project handle
path	string		prototype resources path

### InstanceDescription

Field	Type	Label	Description
name	string		
type	string		
input	SignalDescription	repeated	
output	SignalDescription	repeated	
task	TaskDescription	repeated	

### InstanceList

Field	Type	Label	Description
instance	InstanceDescription	repeated	

Figure 32 Portion of HTML auto-generated documentation of the SimulationService.proto

### 6.4.2 User workflows

The API is meant to allow the integration of the automation IDE and the simulation IDE during the engineering phase. It is possible to formalize a set of relevant sequences of operations that are performed by the two application, highlighting the role of each environment in the interaction. In this section is described a complete Workflow for an End-User (Engineer) divided in three phases: start up and creation, stop and close and destroy. The use cases and corresponding sequence diagrams are presented from the Automation Engineer perspective using a generic IEC 61499 IDE, to describe which is the envisioned user experience and how it reflects on the data exchange between environments. This workflows are the same that have been implemented on nxtStudio and DDD Model Editor platforms during the development of the proof of concepts and that have been applied during the test and validation phase.

The interactions are described using UML sequence diagrams, where the messages flowing between lifelines, denoted with << ... >> notation correspond to gRPC calls.

#### 6.4.2.1 *Start up and creation*

In this workflow the user starts the IDE applications and then, following the natural engineering process, creates a new project, adding as many CPS instances as needed and setting their parameters, and finally compiles, deploys, and runs the project. What is enhanced is the fact that during the compiling and run, the corresponding simulation model is transparently built, deployed, run, and connected to the control application giving the end user the possibility to immediately execute virtual commissioning operations.

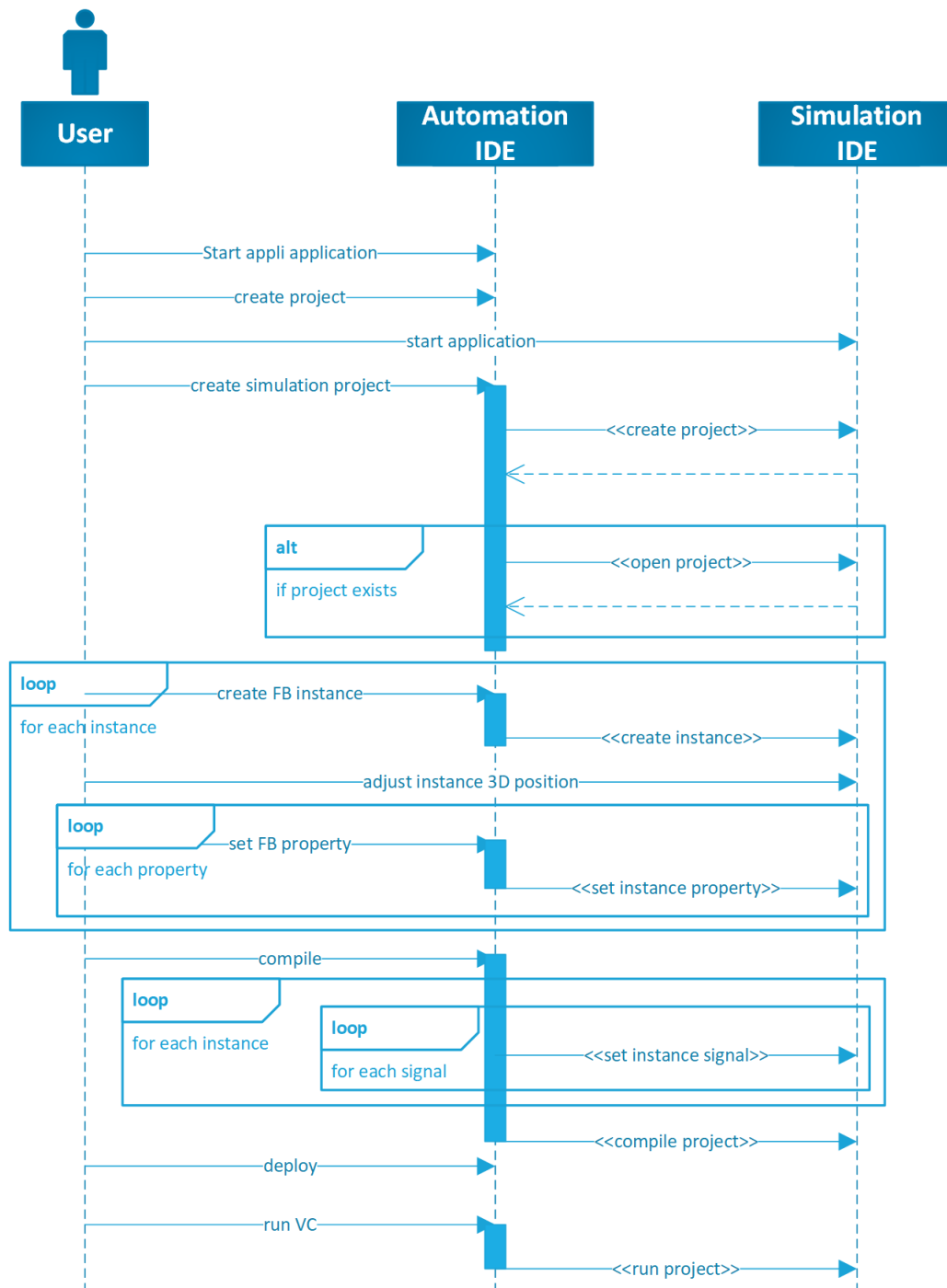


Figure 33 Startup and creation workflow with gRPC API

#### 6.4.2.2 Stop and close

In this workflow the user stops a running project, closes it, and exits the application. This workflow is very simple and it is meant only to highlight the fact that the interaction between IDEs involves the management of cleanup operations to ensure the correct release of the system resources.

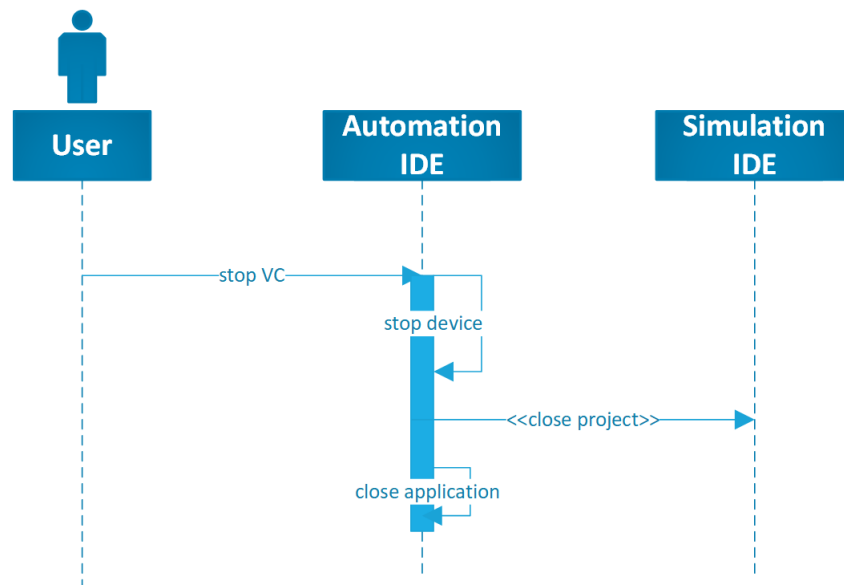


Figure 34 Stop and close workflow

#### 6.4.2.3 Destroy

In this workflow the end user deletes an instance and, if the originating prototype is not used anymore, the automation IDE is in charge of removing the prototype from the simulation project, to allow its garbage collection.

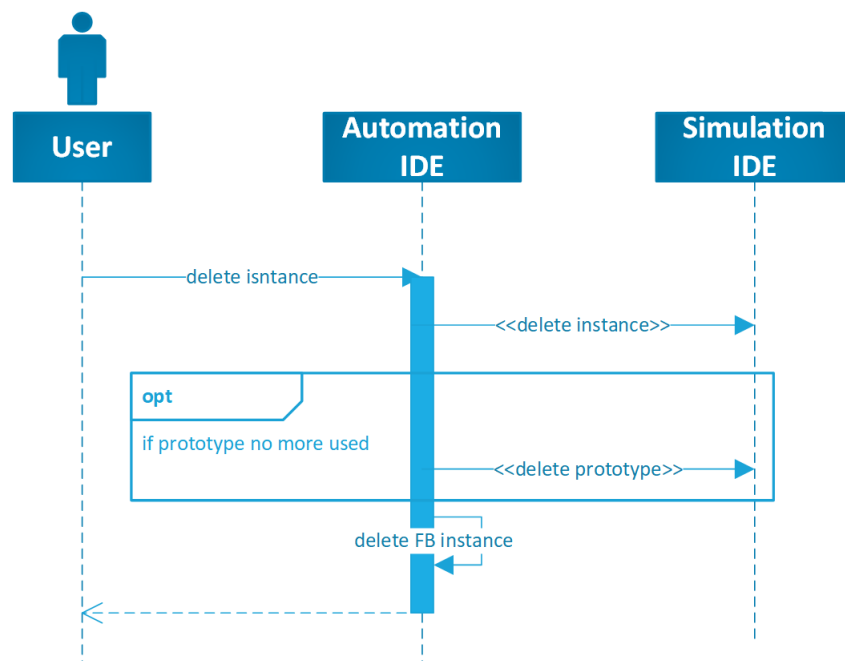


Figure 35 Destroy workflow

### 6.4.3 Implementation within target environments

The implementation of the Integration API has been carried out on the two target environments provided by the partners of the EU research projects: nxtStudio and DDD Model Editor. Figure 36 shows the placement of the gRPC interface component within the internal architecture of each platform. The following paragraphs describe the role of the depicted software modules of the DDD Model Editor application.

The implementation on the automation side, in nxtStudio platform, has been completed by the R&D department of NxtControl GmbH and its documentation is part of a complementary work that will be published separately.

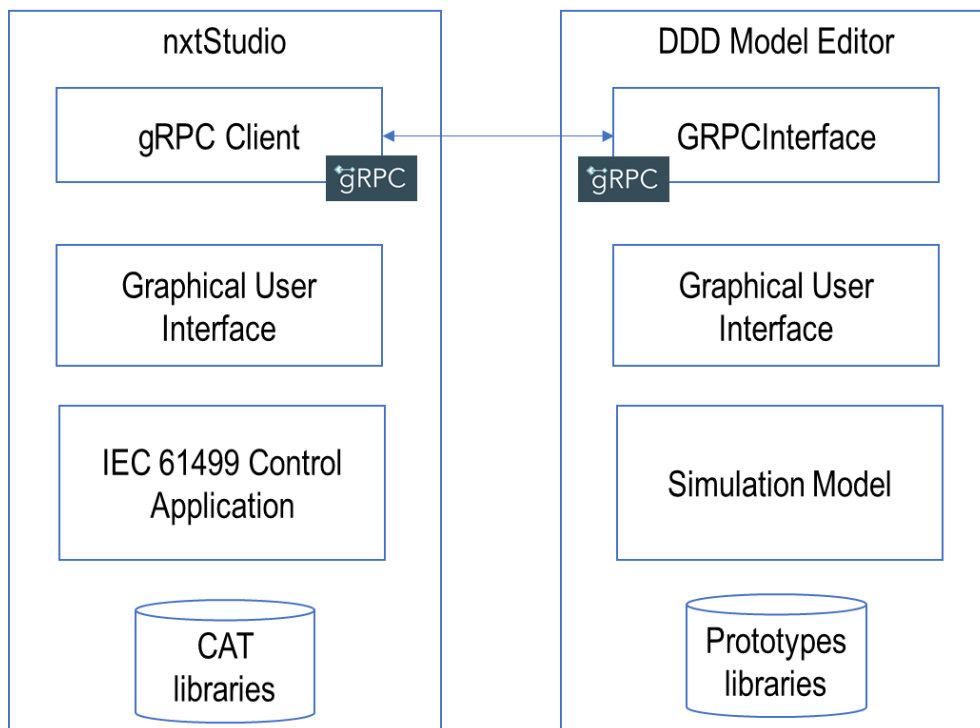


Figure 36 Component view of the implementation

#### 6.4.3.1 GRPCInterface and Graphical User Interface

From the Integration API perspective, the DDD Model Editor application represents the serving software, providing the functionalities to operate on the simulation project, its prototypes, and its instances. The gRPC Server component is a Java package that has been generated using the

tools made available by the gRPC framework. The generated classes has been connected with the internal project management API to control the CRUD operations of the simulation model, of its instances and the execution of the virtual commissioning sessions, as required by the Integration API specification. The gRPC server is part of the modular architecture of the DDD Model Editor application, that allows the installation of external plugins interacting with the core editing functionalities. Figure 37 highlights the internal organization of the plugin.

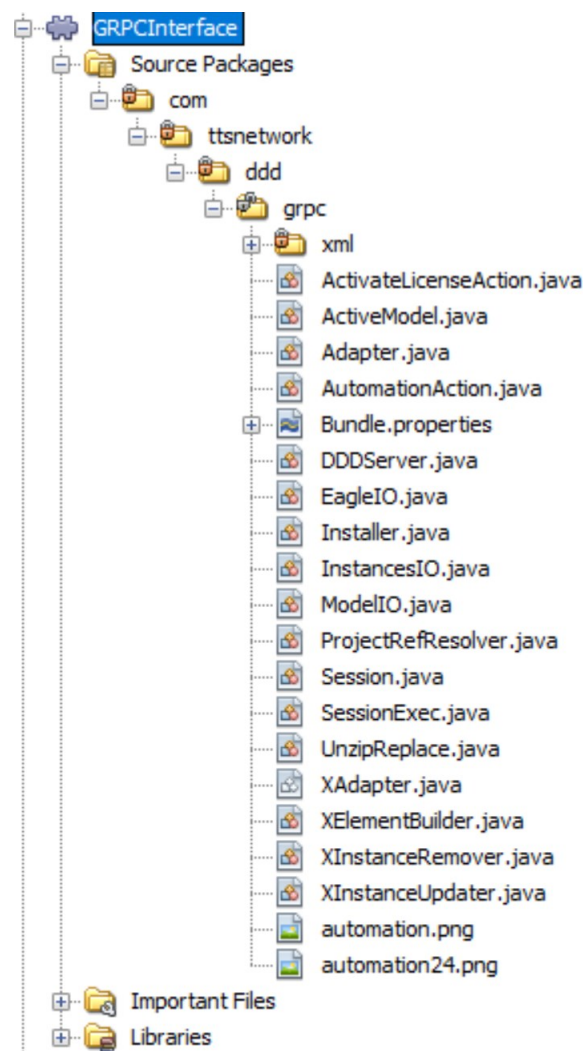


Figure 37 Internal structure of the GRPCInterface plugin

It is important noting that this component is mainly operating in the back end logics of the application and its presence is almost transparent to the graphical user interface, expect for the

startup message that informs the user that the plugin gRPC interface is active, as shown in Figure 38.

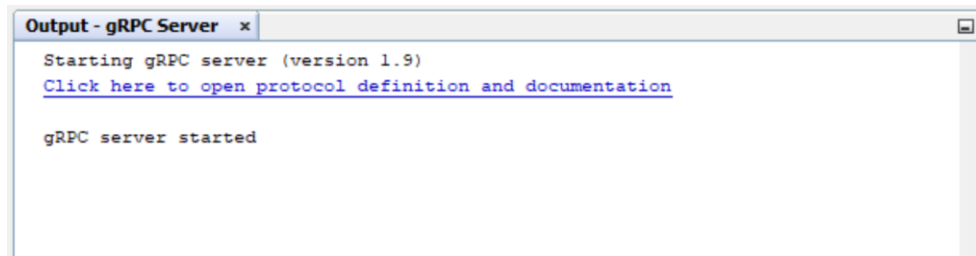


Figure 38 Startup of the gRPC server module

#### 6.4.3.2 Prototype libraries

The possibility to instantiate simulation entities inside the project 3D model, depends on the availability of the prototypes representing the digital twins of the automation Function Blocks. As documented in §3.4.4, the prototypes are grouped and installed into the editing environment within libraries of components. Therefore for each CAT (Composite Automation Type) that represents a physical element on the nxtStudio side, there exist a simulation prototype in the corresponding library.

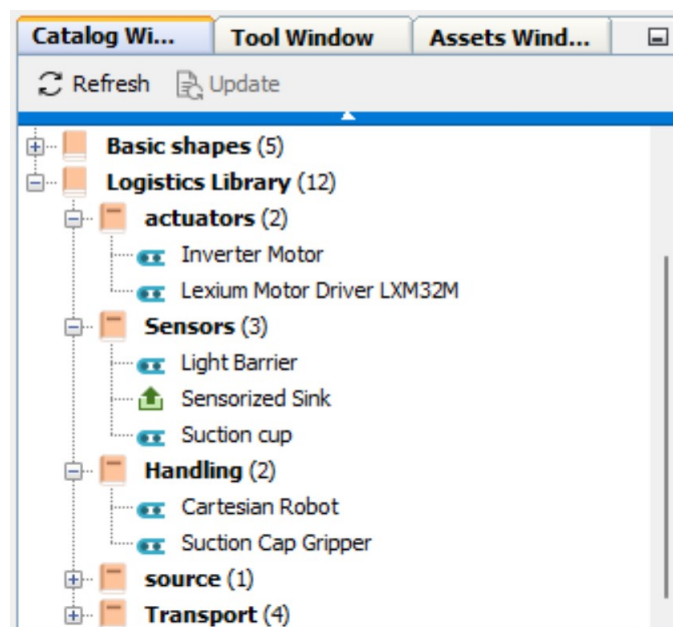


Figure 39 Library of simulation prototypes corresponding to IEC61499 device FBs



#### 6.4.3.3 Simulation Model

It is the instance of simulation project organized as explained in §3.4.4 and target of the gRPC integration API. As reported in the user workflows, all the times the user modifies a device Function Block, the corresponding instance are created/deleted or parametrized in the simulation model.

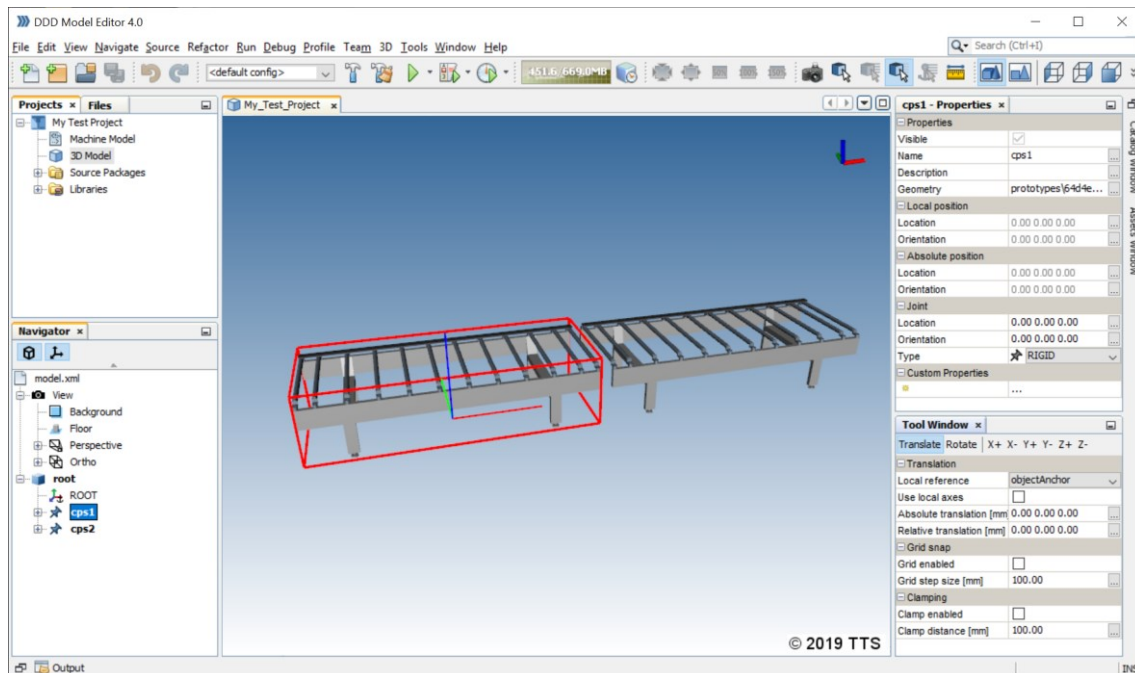


Figure 40 Simulation model composed of 2 instances of conveyor prototype

#### 6.4.3.4 Workflow

The main workflow presented as a platform independent sequence diagram for the startup and creation, the is instantiated with the target software application. Figure 41 shows the main order of the actions triggered by the application engineer. In particular it is important highlighting that, except for the point number 1 – Write Automation Code, that corresponds to his everyday work, all the other steps happen automatically and are managed by the internal logics of the two interacting platforms whenever the automation expert request to start a virtual commissioning session. This is fundamental from the workload perspective because it shows how the complexity of the whole system can be kept almost transparent to the final end user, without

creating additional burden but providing a high value added service for testing the control application under development.

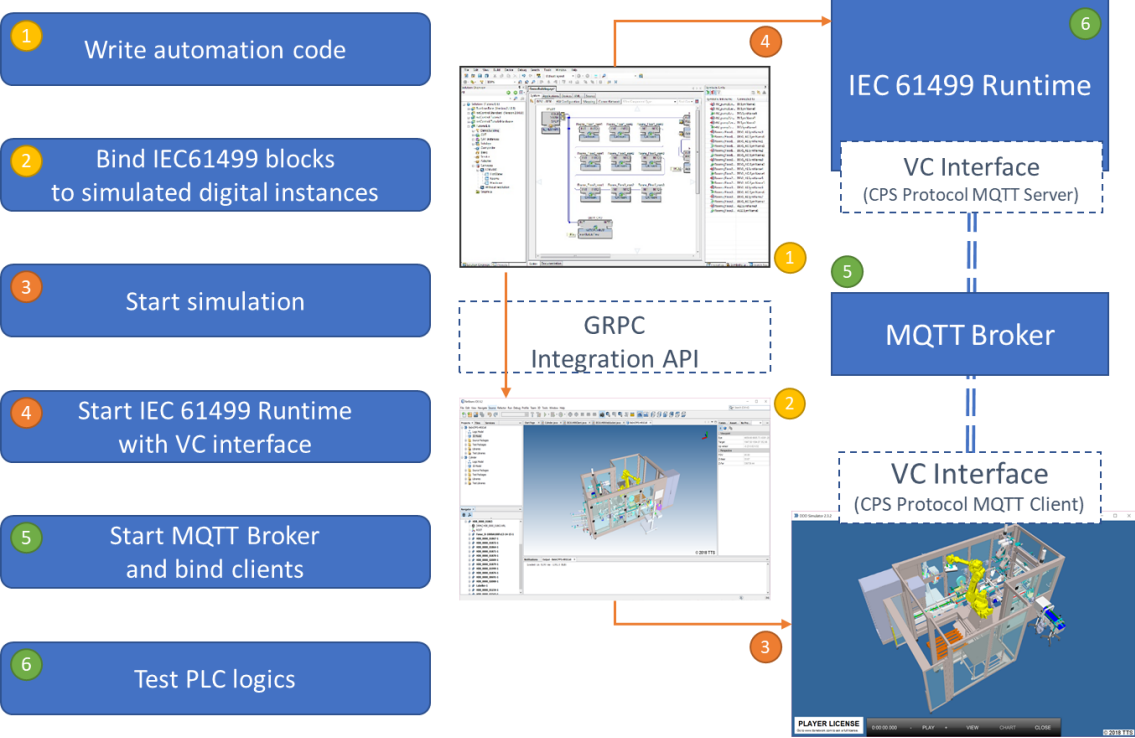


Figure 41 Instantiation of the workflow in the target applications

## 7 Global validation scenarios

The validation of the results achieved has been carried on within the contexts of the Daedalus and 1-SWARM European research projects, and the availability of different industrial show cases has provided an ideal set of high-value added real scenarios. The development of the runtime connection and of the Integration API followed an iterative incremental path made of early prototypes implementations, small focused tests and re-design of the solutions. Thus, the low level functionalities of each components of the overall picture has been guided by the evaluation results during the implementation, correcting pitfalls as they appeared. This approach allowed the consortium of partners collaborating to the whole topic to progressively adjust the objectives and the steps to achieve them. In this way the single components deployed in the respective automation and simulation platforms, both for runtime and for design time, could reach a good level of stability and it has been possible to start the demonstration phase leveraging on a reliable architecture.

This chapter reports on the validation tests performed on two reference cases, each one characterized by peculiarities that made them an optimal playground:

- A pilot plant scenario constituted by an automated de-palletization line
- A industrial scenario constituted by lines for food packaging

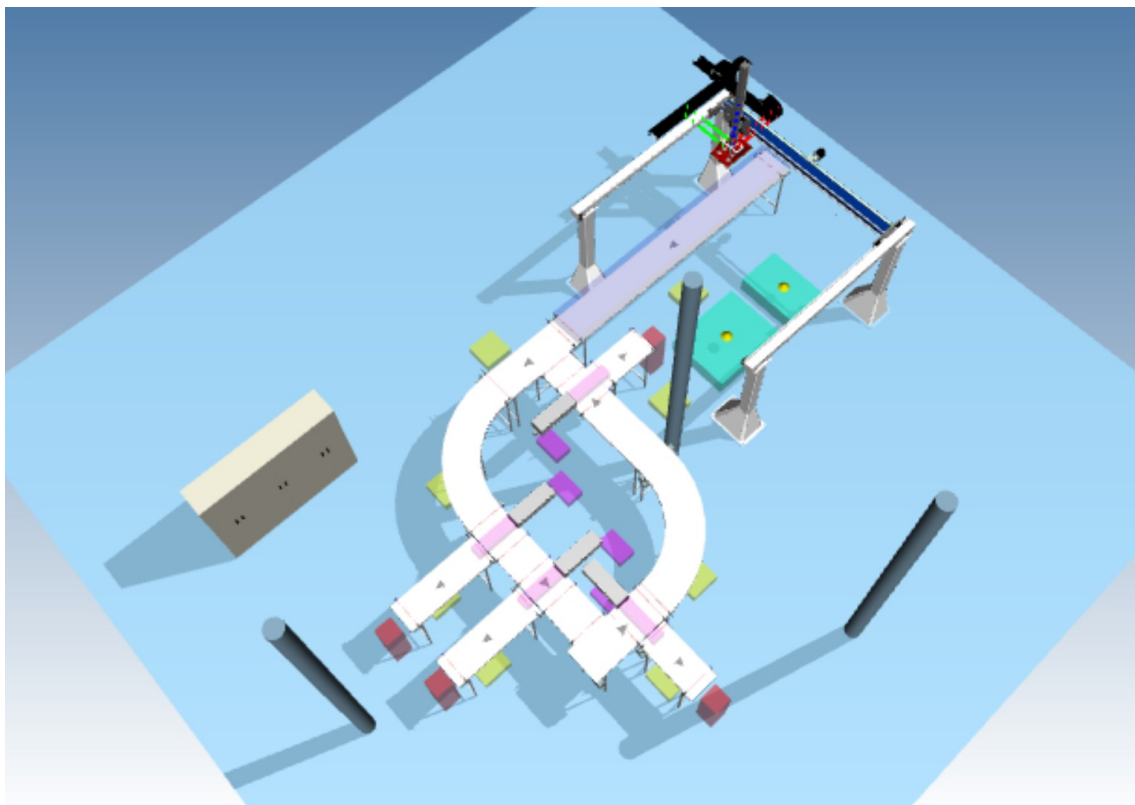
On both cases, the full set of libraries containing the simulation prototypes for the DDD platform and the CAT models for the nxtStudio IEC 61499 environment has been developed from scratch in compliance with the proposed specifications data models. The nxtStudio and DDD platforms, equipped with the extension documented in this thesis have been deployed and extensively applied to create the IEC 61499 control applications and their 3D simulation counterparts. In this way, the global workflow presented in Chapter 6 has been applied step by step to confirm the effectiveness of the framework and to reach an initial evaluation of the improvements that it could bring to the virtual commissioning on IEC 61499 systems.

The evaluation procedures have been based on the comparison of the observed behaviors and performances with the initial expectations formalized in the requirements lists produced for the

runtime engines and for the authoring environments. Therefore the final section of this chapter contains a synoptic table with an assessment of the level of fulfillment of the acceptance criteria for each original requirement. This report, beyond providing a global rating of the whole work, constitutes the starting point for the future improvements.

## 7.1 Logistics

The logistic scenario refers to the De-Palletization plant present in Como Next, Lomazzo (IT). It is a pilot plant, therefore not dedicated to real production, but mainly to showcase the benefits of applying distributed IEC 61499 automation to control systems composed of heterogeneous devices supplied by different vendors. Figure 42 De-palletization pilot plant model, shows the structure of the line.



*Figure 42 De-palletization pilot plant model*

The importance of this scenario is related to the following aspects:

1. The possibility to develop and test in a complex and complete playground without impacting on the production of a real line; this made the logistic scenario ideal for the initial validation, the most critical one from the point of view of the stability of the applied solutions.
2. The richness of devices composing the line: the presence of conveyor lines, pneumatic actuators like the pushers and the suction grippers, and complex kinematics structures requiring motion control, like the cartesian robot, gave the possibility to develop a wide set of prototypes, evaluating the behavior of the framework with different type and frequencies of signals.

### 7.1.1 Prototypes

This section documents the device prototypes based on the Digital Avatar Data Model developed to perform the virtual commissioning of the plant.

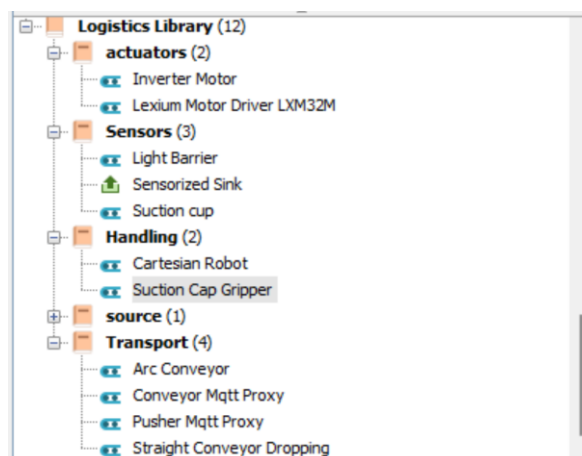
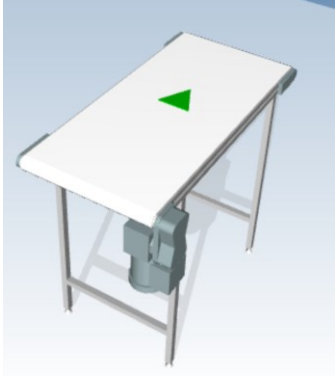


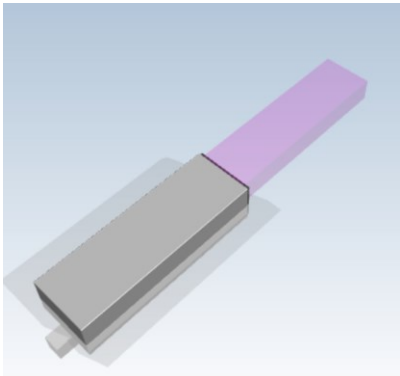
Figure 43 Library of prototypes developed for testing the logistics scenario

Other modules have been defined to emulate the behavior of the loading and unloading of the bays with pallets and boxes but they are not controlled by the automation.

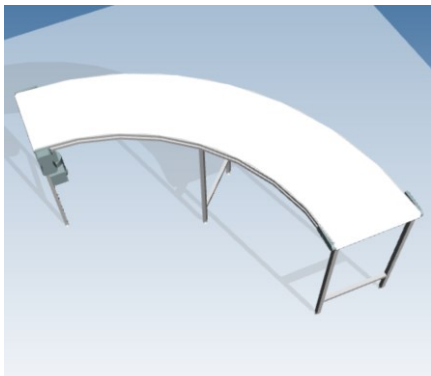
### *Straight Conveyor*

	Prototype signals:	IN: velocityTarget IN: motorDirection OUT: velocityActual OUT: driveEnabled OUT: driveRunning OUT: directionalActual OUT: lightBarrierA OUT: lightBarrierB
---	--------------------	---


### *Pusher*

	Prototype signals:	IN: command OUT: limitExtended OUT: limitRetracted OUT: isRunning OUT: lightbarrierA OUT: lightbarrierB
--	--------------------	--

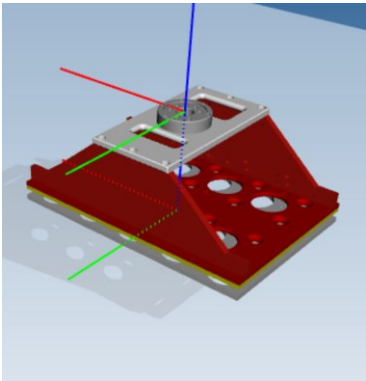
### *ConveyorCurve*

	Prototype signals:	IN: velocityTarget IN: motorDirection OUT: velocityActual OUT: driveEnabled OUT: driveRunning OUT: directionalActual OUT: lightBarrierA OUT: lightBarrierB
---	--------------------	---

*CartesianRobot*

	Prototype signals:	IN: XTarget IN: YTarget IN: ZTarget IN: RTarget IN: grip OUT: cartesianArrived OUT: actualPosX OUT: actualPosY OUT: actualPosZ OUT: actualPosR
---	--------------------	---

*SuctionCapGripper*

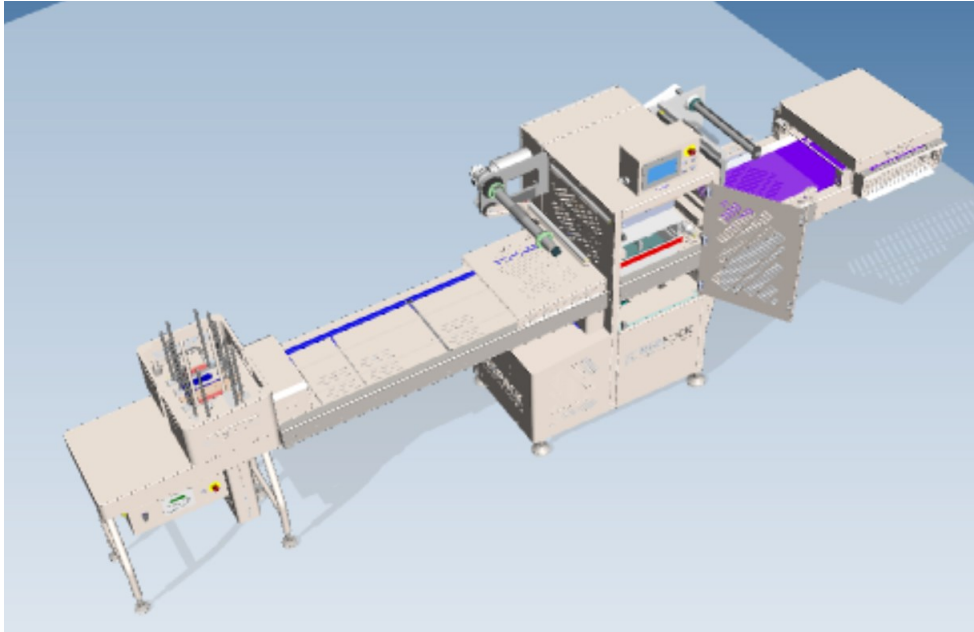
	Prototype signals:	IN: suctionOn IN: blowOn OUT: vacuumOn
--	--------------------	--

7.1.2 Model Statistics

Number of prototypes	7
Number of instances	38
Number of signals	130
Runtime max update frequency	50 Hz (20 ms interval)
Simulation model complexity	about 250K polygons

## 7.2 Packaging

The packaging scenario refers to the production of lines for food packaging of Reepack Srl (IT) [25]. This type of lines is characterized to be quite short with few but complex modules homogeneous because all produced by the same company.



*Figure 44 Reepack food packaging line*

Nevertheless the importance of this test case refers to:

1. The soundness of the test: Reepack lines are industrial products working in real environments and with really demanding requirement in terms of performances and machines customizations
2. The machines perform fast operations, requiring event management on simulation side which is not larger than 10 ms to allow automation to control the main inverter motor governing the motion actuators of the line.

In this scenario preliminary tests have been executed to asses the capability of the MQTT runtime communication channel to stand the required update frequency needed to control the inverter motor with a speed profile. The result of the test has been positive, the updating of



speed input values and the feedback emulating the encoder succeeded in ensuring an update frequency of about 100 Hz.

7.2.1 Prototypes

The main device prototypes developed for this test case are listed in the following tables.

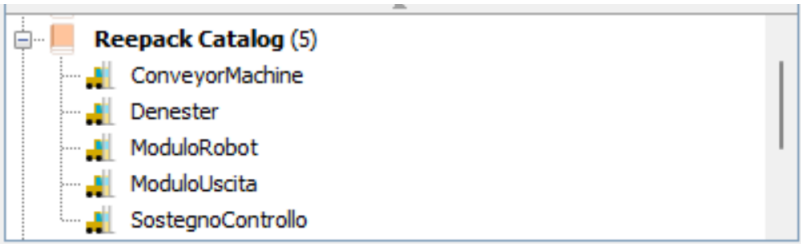
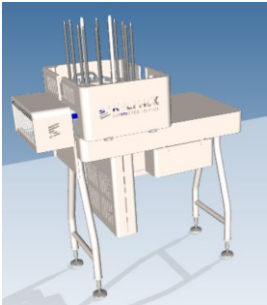
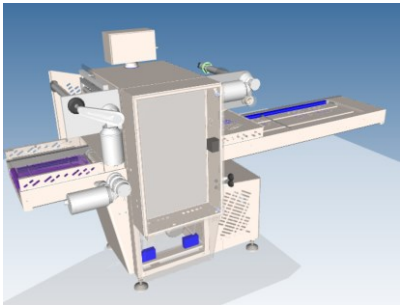
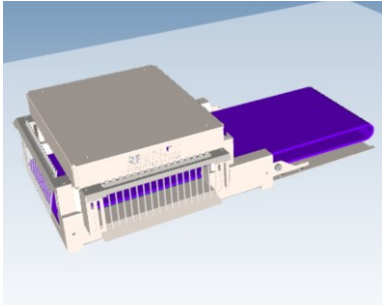
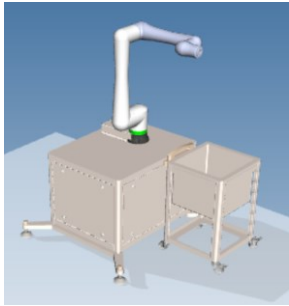


Figure 45 Library of prototypes developed for testing the packaging scenario

All the prototypes have been modelled and endowed of the corresponding signals. The following table report the visual appearance but, for data protection reason, the names of the signals and their meaning is not reported in the document.

Denester module	
Conveyor module	

Exit module	
Collaborative robot module	

### 7.2.2 Model Statistics

Number of prototypes	5
Number of instances	5
Number of signals	80
Runtime max update frequency	100 Hz (10 ms interval)
Simulation model complexity	about 200K polygons

### 7.3 Requirements fulfillment assessment

The following table reports the numbered requirements and the fulfillment level as a number between 0 (not achieved) and 10 (completely achieved). For the sake of summary, for the documentation about each requirement and its acceptance criteria, refer to the respective definition Chapters 5 and 6.

#### 7.3.1 Runtime requirements

<b>Name</b>	<b>Description</b>	<b>Priority</b>	<b>LoF</b>
R001	The communication channel must ensure a large bandwidth to handle high sampling frequencies of a large set of I/O signals.	SHALL	10
R002	The communication channel, once activated on the deployed system, must have a minimal footprint on the infrastructure.	SHOULD	8
R003	The information (signal values) must be delivered in both directions (from automation to simulation and vice-versa) assuring the packet ordering	SHALL	10
R004	The information ordering must be ensured without any loss of packet data in both directions.	SHALL	10
R005	The communication channel must accept, at automation level, multiple incoming connection from several simulation clients, thus supporting the unidirectional multicasting of packets generated by the automation runtime towards several simulation clients, even when deployed on a distributed environment	SHALL	10
R006	The number of sockets opened by each connected client simulation model must be minimized.	SHOULD	5

R007	The communication channel allows an initial synchronous (request-response) setup phase to select the signals of interest that should be transferred during the virtual commissioning session and the corresponding maximum update frequencies.	SHOULD	0 in MQTT 8 in WS
R008	The same communication channel (physical socket) must support the multiple flow of asynchronous signal messages, corresponding to the events governing the IEC 61499 FBs.	SHALL	10
R009	The communication channel must be based on widely accepted open standards both at transport layer and at payload level.	SHALL	10
R010	The communication channel must be natively cross-platform in order to easily deployed on multiple different hardware and operating system platforms	SHALL	10
R011	The chosen transport layer must be compliant with industrial and shopfloor network setups.	SHALL	10
R012	The communication channel can be secured, preventing possible exploitations for cyber-attacks to the control hardware.	MAY	8

### 7.3.2 IDE Integration requirements

<b>ID</b>	<b>Description</b>	<b>Priority</b>	
D001	The integration layer must allow the user to automatically create digital counterparts of the automation code written in IEC 61499.	SHALL	10
D002	The integration layer must define automatically the signal mapping between function blocks and simulation entities.	SHALL	10
D003	The simulation model must be a synchronized representation of the automation function blocks.	SHALL	10
D004	The integration layer must support a modular object oriented approach.	SHOULD	8
D005	The integration layer must be based on an open Digital Avatar Data model compatible with the runtime IO Data Model and its implementations.	SHALL	10
D006	The integration architecture must be platform independent, it must be possible to apply the same approach to automation and simulation platforms different from the ones used during the validation.	SHOULD	9
D007	The integration must be based on a cross platform technology. The IDE applications are typically developed on different software platforms (Java, C++, C#, etc.), the chosen technology must ease the implementation of the communication stubs.	SHOULD	10
D008	The integration layer must support the compilation and deployment phase of the artifacts at simulation phase.	SHALL	10

D009	The integration layer must support the control (start/stop) of the execution of the virtual commissioning sessions, running the automation and simulation engine and connecting them without the need for the automation developer to interact with the simulation IDE.	SHOULD	10
D010	The integration layer must be bidirectional, supporting not only the control flow from automation to simulation IDE but also the notifications in the opposite direction.	SHOULD	8
D011	The integration layer must be resilient to modifications of the simulation model that affect the positioning and parametrization of the digital twins of the Function Blocks.	SHALL	8
D012	The integration layer should have a small footprint. The concurrent execution of two complex IDEs like the automation and simulation can require a significant amount of system resources, therefore it is important to avoid that a further increase of this consumption due to the communication between the applications causes instabilities or even system crashes.	MAY	6

## 8 Conclusions and future developments

This PhD document presented a new approach to improve the process of implementing digital twins for complex automated discrete manufacturing systems, basing on an advancement of the IEC 61499 virtual commissioning framework. The design and development of the integration layers of runtimes and engineering platforms for automation and simulation have been documented, motivating the proposed solutions with a detailed analysis of the system requirements, and demonstrating their potentialities with industry derived validation scenarios. The obtained results are encouraging and the possibility to further extend them is already reality at the time of this report. In fact, the European Research initiative Horizon 2020 – 1-SWARM is currently entering its third year of activity and the consortium is preparing advanced industrial test cases where the framework will be tested and improved. The envisioned developments go in the direction of enhancing the capability of the integrated platform, to manage Cyber Physical Systems of Systems, increasing the reliability and usability of the reference implementations presented in this work. This represents an enormous opportunity of promoting the adoption of the proposed solutions as de facto standards for the validation of IEC 61499 control applications, paving the way for the commercial exploitation of the results.

## 9 Bibliography

- [1] M. Mazzolini, "ENGINEERING SUPPORT SYSTEM FOR SUSTAINABLE OPTIMIZATION OF AUTOMATION TASKS SUPERVISION," 2018.
- [2] S.-F. Q. a. K. Cheng, "Future Digital Design and Manufacturing: Embracing Industry 4.0 and Beyond," *Chinese J. Mech. Eng.*, vol. 30, no. 5, p. 1047–1049, 2017.
- [3] L. B. a. T. F. P. Osterrieder, "The smart factory as a key construct of industry 4.0: A systematic literature review," *International Journal of Production Economics*, vol. 221, 2020.
- [4] *H2020 1-SWARM European Research Project*, 2020.
- [5] A. U. a. E. Cevikcan, "Industry 4.0: Managing The Digital," *Cham: Springer International Publishing*, 2018.
- [6] S. B. a. R. Rosen, "Digital Twin—The Simulation Aspect," *Mechatronic Futures*, *Cham: Springer International Publishing*, p. 59–74, 2016.
- [7] S. W. a. F. Q. J. Um, "Plug-and-Simulate within Modular Assembly Line enabled by Digital Twins and the use of AutomationML," *FAC-PapersOnLine*, vol. 50, no. 1, p. 15904–15909, 2017.
- [8] F. A. G. M. A. F. Mauro Mazzolini, "Structured Approach to the Design of Automation Systems through IEC 61499 Standard," *Procedia Manufacturing*, vol. 11, pp. 905-913, 2017.
- [9] SIMIO, "SIMIO SIMULATION SOFTWARE," [Online]. Available: <https://www.simio.com/>.



- [10] M. W. A. Z. J Cabral, "Enable co-simulation for industrial automation by an FMU exporter for IEC 61499 models," *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2018.
- [11] Modelica Association c/o PELAB, IDA, "Functional Mockup Interface," [Online]. Available: <https://fmi-standard.org/>.
- [12] S. P. V. V. Midhun Xavier, "Cyber-physical automation systems modelling with IEC 61499 for their formal verification," *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, 2021.
- [13] U. D. A. V. V. Tuojian Lyu, "Towards cloud-based virtual commissioning of distributed automation applications with IEC 61499 and containerization technology," *IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society*, 2021.
- [14] A. S. Adriano A.Santos, "Simulation and Control of a Cyber-Physical System under IEC 61499 Standard," *Procedia Manufacturing*, vol. 55, pp. 72-79, 2021.
- [15] C. G. L. a. S. C. Park, "Survey on the virtual commissioning of manufacturing systems," *Journal of Computational Design and Engineering*, vol. 1, no. 3, pp. 213-222, 2014.
- [16] Siemens, "Virtual Commissioning," Siemens, [Online]. Available: <https://www.plm.automation.siemens.com/global/it/products/tecnomatix/virtual-commissioning.html>. [Accessed 2021].
- [17] Fanuc, "Fanuc Robot Guide," [Online]. Available: <https://www.fanuc.eu/it/it/robot/accessori/simulation-software-roboguide>. [Accessed 2021].

- [18] V. Vyatkin, "The IEC 61499 standard and its semantics," *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 40-48, 2009.
- [19] NXT, "nxtStudio," [Online]. Available: <https://www.nxtcontrol.com/en/engineering/>. [Accessed 2021].
- [20] International Organization for Standardization, ISO/IEC 20922:2016 Information technology -- Message Queuing Telemetry Transport (MQTT) v3.1.1, 2016.
- [21] Eclipse Foundation, "Eclipse Mosquitto," Eclipse Foundation, [Online]. Available: <https://mosquitto.org/>.
- [22] Eclipse Foundation, "Eclipse Paho," Eclipse Foundation, [Online]. Available: <https://www.eclipse.org/paho/>.
- [23] g. Authors, "gRPC," [Online]. Available: <https://grpc.io/>. [Accessed 2021].
- [24] Google, "Language Guide (proto3)," [Online]. Available: <https://developers.google.com/protocol-buffers/docs/proto3>. [Accessed 2021].
- [25] Reepack, "Reepack home page," [Online]. Available: <https://www.reepack.com/>.
- [26] V. D. a. K. V. J. Müller, "Industry 4.0 and its Impact on eshoring Decisions of German Manufacturing Enterprises," *Supply Management Research, Wiesbaden: Springer Fachmedien Wiesbaden*, p. 165–179, 2017.
- [27] V. Vyatkin, "IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review," *IEEE Trans. Ind. Informatics*, vol. 7, no. 4, p. 768–781, 2011.
- [28] O. M. Group, "Model Driven Architecture," [Online]. Available: [http://www.omg.org/mda/faq\\_mda.htm](http://www.omg.org/mda/faq_mda.htm).

- [29] J. Y. a. V. V. Vyatkin, "Distributed execution and cyber-physical design of Baggage Handling automation with IEC 61499," *2011 9th IEEE International Conference on Industrial Informatics*, p. 573–578, 2011.
- [30] M. Grieves, "Digital Twin: manufacturing excellence through virtual factory replication," 2015.
- [31] T. M. M. O. D. G. a. D. Z. S. Weyer, "Future Modeling and Simulation of CPS-based Factories: an Example from the Automotive Industry," *IFAC-PapersOnLine*, vol. 49, no. 31, pp. 97-102, 2016.
- [32] M. Grieves and J. Vickers, "Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems," *Transdisciplinary Perspectives on Complex Systems, Cham: Springer International Publishing*, pp. 85-113, 2017.
- [33] L. B. O. R. D. S. M. M. a. M. L. J. Vachalek, "The Digital Twin of an industrial production line within the industry 4.0 concept," *2017 21st International Conference on Process Control (PC)*, p. 258–262, 2017.
- [34] M. A. S. M. D. R. a. P. P. M. Ciavotta, "A Microservice-based Middleware for the Digital Factory," *Procedia Manuf.*, vol. 11, p. 931–938, 2017.
- [35] M. R. M. T. a. C. D. T. Bangemann, "Integration of Classical Components Into Industrial Cyber–Physical Systems," *IEEE Proceedings*, vol. 104, no. 5, p. 947–959, 2016.
- [36] D. R. G. D. M. S. M. R. T. Michele Ciavotta, "Towards the Digital Factory: A Microservices-Based Middleware for Real-to-Digital Synchronization," in *Microservices, Science and Engineering*, 2019.

- [37] A. B. S. M. F. C. D. R. G. D. M. G Landolfi, "Design of a multi-sided platform supporting CPS deployment in the automation market," *IEEE Industrial Cyber-Physical Systems (ICPS)*, pp. 684-689, 2018.
- [38] D. R. P. P. M. C. Giovanni Dal Maso, "A Centralized Support Infrastructure (CSI) to Manage CPS Digital Twin, towards the Synchronization between CPSs Deployed on the Shopfloor and Their Digital Representation," in *The Digital Shopfloor: Industrial Automation in the Industry 4.0 Era Performance Analysis and Applications*, River Publishers, 2019, pp. 317-335.

## Appendix A – Simulation Server Protocol Buffer IDL

The section reports the full definition of the interface that could be used for IDE integration.

```
syntax = "proto3";

option java_package = "simulation.server";
option java_multiple_files = true;

package simulation.server;

enum StatusCode {
    ERROR = 0;
    CREATED = 1;
    EXISTING = 2;
}

/**
 * Simulation service.
 */
service SimulationService {

    /**
     * Creates and opens a Virtual commissioning project
     */
    rpc CreateProject(CreateProjectRequest) returns (ProjectHandle);

    /**
     * Opens a Virtual commissioning project
     */
    rpc OpenProject(OpenProjectRequest) returns (ProjectHandle);

    /**
     * Closes the project.
     */
    rpc CloseProject(ProjectHandle) returns (Result);

    /**
     * Deletes the project
     */
}
```

```

    */
    rpc DeleteProject(ProjectHandle) returns (Result);

    /**
     * Creates a new prototype in the project importing a prototype
    definition from the specified directory.
     */
    rpc CreatePrototype(CreatePrototypeRequest) returns (ResourceHandle);

    /**
     * Deletes a prototype (it should not be used, i.e. no instances of the
    prototype).
     */
    rpc DeletePrototype(ResourceHandle) returns (Result);

    /**
     * Gets a list of prototypes.
     */
    rpc GetPrototypes(ProjectHandle) returns (ResourceList);

    /**
     * Creates a new instance of a prototype.
     */
    rpc CreateInstance(ResourceHandle) returns (CreateResult);

    /**
     * Deletes an instance.
     */
    rpc DeleteInstance(ResourceHandle) returns (Result);

    /**
     * * Gets a list of instances.
     */
    rpc GetInstances(ProjectHandle) returns (InstanceList);

    /**
     * Sets the value of a property.
     */
    rpc SetInstanceProperty(SetInstancePropertyRequest) returns (Result);

    /**
     * Sets the source of a signal for an instance.
     */

```

```

rpc SetSignalInstance(SetSignalInstanceRequest) returns (Result);

/**
 * Compiles the project and creates the distributable runtime.
 */
rpc CompileProject(ProjectHandle) returns (Result);

/**
 * Runs the project, launching the runtime application.
 * Note: the project must be already compiled.
 */
rpc RunProject(RunProjectRequest) returns (Result);

/**
 * Stop the project: disconnect and exit runtime.
 */
rpc StopProject(ProjectHandle) returns (Result);

/**
 * Returns true if the runtime process is running, false otherwise.
 */
rpc QueryProjectRunning(ProjectHandle) returns (Result);

rpc BeginTransaction(ProjectHandle) returns (Result);
rpc CommitTransaction(ProjectHandle) returns (Result);
rpc RollbackTransaction(ProjectHandle) returns (Result);
}

/**
 * Request to create a new project.
 */
message CreateProjectRequest {
    // project display name
    string name = 1;

    // full path of the new project (the directory should not exist)
    string path = 2;
}

/**
 * Generic result message of success.
 */

```

```

message Result {
    bool success = 1;
}

/**
 * Create message status.
 */
message CreateResult {
    StatusCode = 1;
}

/**
 * Message that holds a project handle.
 * It is used both as a result for CreateProject and OpenProject, and as a
 * parameter for methods that requires a valid opened project (i.e.
 * CompileProject).
 */
message ProjectHandle {
    // project handle
    uint32 handle = 1;
}

/**
 * Request message to open a project from a directory.
 */
message OpenProjectRequest {
    // full path of the project
    string path = 1;
}

/**
 * Request message to create a new prototype in a project from a prototype
 * definition in the specified directory.
 */
message CreatePrototypeRequest {
    // project handle
    uint32 handle = 1;

    // prototype resources path
    string path = 2;
}

/**

```



```

    * Generic message that holds information about a resource (prototype or
instance).
    * It is used both as a request and a response.
    */
message ResourceHandle {
    // project handle
    uint32 project_handle = 1;

    // name of the resource (name of a prototype or name of an instance)
    string name = 2;

    // resource type (uuid of a prototype or prototype of an instance)
    string type = 3;
}

/**
 * Message that holds a list of resources.
 */
message ResourceList {
    repeated ResourceHandle resources = 1;
}

/**
 * Sets the value of a property.
 */
message SetInstancePropertyRequest {
    // project handle
    uint32 handle = 1;

    // instance name
    string instance = 2;

    // task name
    string task = 3;

    // property name
    string name = 4;

    // property value
    string value = 5;
}

/**

```

```

* Sets the source of a signal for an instance.
*/
message SetSignalInstanceRequest {
    // project handle
    uint32 handle = 1;

    // instance name
    string instance = 2;

    //signal name
    string signal = 3;

    // source name
    string source = 4;
}

/**
* A run argument is a pair of name and value.
*/
message RunArgument {
    // argument name
    string name = 1;

    // argument value
    string value = 2;
}

/**
* Run project message request holds the handle of the project and a list of
run arguments.
*/
message RunProjectRequest {
    // project handle
    uint32 handle = 1;

    // optional run arguments
    repeated RunArgument run_arguments = 2;
}

/**
*
*/
message SignalDescription {

```

```

    string name = 1;
    string type = 2;

    // can be unset (but in proto3 cannot be null)
    string source = 3;
}

message PropertyDescription {
    string name = 1;
    string type = 2;
    string defaultValue = 3;

    // can be unset (but in proto3 cannot be null)
    string value = 4;
}

message TaskDescription {
    string name = 1;
    string type = 2;
    repeated PropertyDescription property = 3;
}

message InstanceDescription {
    string name = 1;
    string type = 2;
    repeated SignalDescription input = 3;
    repeated SignalDescription output = 4;
    repeated TaskDescription task = 5;
}

message InstanceList {
    repeated InstanceDescription instance = 1;
}

```