UNIVERSITY OF VERONA

DEPARTMENT OF COMPUTER SCIENCE

GRADUATE SCHOOL OF NATURAL SCIENCE AND ENGINEERING

DOCTORAL PROGRAM IN COMPUTER SCIENCE

CYCLE XXXIII

Modeling and Simulation Methodologies for Digital Twin in Industry 4.0

S.S.D. ING-INF/05

Coordinator: Prof. Massimo Merro

Tutor: Prof. Franco Fummi

Doctoral Student: Stefano Centomo

Abstract

The concept of Industry 4.0 [1] represents an innovative vision of what will be the factory of the future. The principles of this new paradigm are based on interoperability and data exchange between different industrial equipment. In this context, Cyber-Physical Systems (CPSs) cover one of the main roles in this revolution. The combination of models and the integration of real data coming from the field allows to obtain the virtual copy of the real plant, also called Digital Twin. The entire factory can be seen as a set of CPSs and the resulting system is also called Cyber-Physical Production System (CPPS). This CPPS represents the Digital Twin of the factory with which it would be possible analyze the real factory. The interoperability between the real industrial equipment and the Digital Twin allows to make predictions concerning the quality of the products. More in details, these analyses are related to the variability of production quality, prediction of the maintenance cycle, the accurate estimation of energy consumption and other extra-functional properties of the system.

Several tools [2] allow to model a production line, considering different aspects of the factory (*i.e.* geometrical properties, the information flows etc.)

However, these simulators do not provide natively any solution for the design integration of CPSs, making impossible to have precise analysis concerning the real factory. Furthermore, for the best of our knowledge, there are no solution regarding a clear integration of data coming from real equipment into CPS models that composes the entire production line.

In this context, the goal of this thesis aims to define an unified methodology to design and simulate the Digital Twin of a plant, integrating data coming from real equipment. In detail, the presented methodologies focus mainly on: integration of heterogeneous models in production line simulators; Integration of heterogeneous models with ad-hoc simulation strategies; Multi-level simulation approach of CPS and integration of real data coming from sensors into models.

All the presented contributions produce an environment that allows to perform simulation of the plant based not only on synthetic data, but also on real data coming from equipments.

Contents

Part I Preliminary

1 Introduction . . . 3
- 1.1 Introduction . . . 3
- 1.2 Objectives & Methodology Flow . . . 5

Part II Homogeneous Models

2 Integrating synthetic and real components of a cyber-physical production system 9
- 2.1 Introduction . . . 9
- 2.2 Background . . . 11
  - 2.2.1 OPC UA Communication Protocol . . . 11
  - 2.2.2 AutomationML . . . 12
  - 2.2.3 Functional Mockup Interface . . . 15
  - 2.2.4 MATLAB/Simulink . . . 15
  - 2.2.5 Tecnomatix Plant Simulation . . . 16
- 2.3 Methodology . . . 16
  - 2.3.1 Mapping Simulink components in AutomationML . . . 16
  - 2.3.2 AML to MATLAB . . . 20
  - 2.3.3 OPC UA server from FMU . . . 22
- 2.4 Experimental Results . . . 23
- 2.5 Simple producer-consumer . . . 24
- 2.6 The ICE lab model . . . 27
- 2.7 Conclusions . . . 28

Part III Heterogeneous Models

3 Automatic Integration of HDL IPs in Simulink using FMI and S-Function Interfaces . . . 33
- 3.1 Introduction . . . 33

3.2 Related Works . . . 34
3.2.1 Running example . . . 35
3.2.2 FMI-Standard . . . 35
3.2.3 Simulink C MEX S-Functions . . . 37
3.2.4 Automatic abstraction of HDL IPs . . . 37
3.3 Methodology . . . 38
3.3.1 Data-type abstraction . . . 41
3.3.2 Automatic generation of Functional Mockup Units . . . 42
3.3.3 Automatic generation of C MEX S-Functions. . . . 46
3.4 Experimental Results . . . 48
3.5 Conclusions and Future Outlook . . . 50

4 Generation of Functional Mockup Units for Transactional Cyber-Physical Virtual Platforms . . . 53
4.1 Introduction . . . 53
4.2 Background and Related Work . . . 55
4.2.1 Functional Mock-up Interface (FMI) Standard 2.0 for co-simulation . . . 55
4.2.2 Simulation coordination in the FMI standard . . . 56
4.2.3 Related Work . . . 58
4.3 FMI Standard Advantages and Limitations . . . 59
4.4 Methodology . . . 60
4.4.1 Functional Mock-up Units (FMUs) generation and timing backward propagation . . . 60
4.4.2 A better coordinator for discrete systems . . . 62
4.5 Methodology Application . . . 63
4.6 Recent Development and Discussion . . . 67
4.7 Concluding remarks . . . 69

5 Cyber-Physical Systems Integration in a Production Line Simulator . . . 71
5.1 Introduction . . . 71
5.2 Plant Simulation and Integration Alternatives . . . 72
5.2.1 Production Line Simulators . . . 72
5.2.2 Siemens Plant Simulation: SimTalk C-Interface . . . 73
5.2.3 Functional Mockup Interface (FMI) . . . 74
5.3 Integration Methodology . . . 75
5.3.1 Cyber System: Modelling and FMU Generation . . . 76
5.3.2 Physical System: Modelling and FMU Generation . . . 77
5.3.3 Cyber-Physical System: Coordination and Integration . . . 77
5.4 Methodology Application . . . 78
5.4.1 Bending machine CPS . . . 79
5.4.2 Alternatives Taxonomy . . . 81
5.4.3 Simulation speed . . . 81

5.4.4 Simulation errors . . . 82
5.5 Conclusions . . . 82

6 Automatic Integration of Cycle-Accurate Models into Cyber-Physical Virtual Platforms . . . 83
6.1 Introduction . . . 83
6.2 Background . . . 84
6.2.1 Related work . . . 84
6.2.2 Code Generation for Virtual Platform Integration . . . 85
6.2.3 Interface technologies for system simulation . . . 86
6.3 Integration methodology . . . 87
6.3.1 Mapping HDL primitives to FMI and S-Functions . . . 87
6.3.2 Monolithic model approach . . . 88
6.3.3 Hub-based approach . . . 90
6.3.4 Alternatives Taxonomy . . . 91
6.4 Methodology Application . . . 92
6.4.1 Simulation performance . . . 93
6.4.2 Design Space Exploration . . . 94
6.5 Conclusions . . . 96

7 Improving FMI-based simulation by Exploiting System–level Information . . . 97
7.1 Introduction . . . 97
7.2 Preliminaries . . . 99
7.2.1 Standardized interfaces and the FMI standard . . . 99
7.2.2 FMI Standard for co-simulation . . . 99
7.2.3 Simulation coordination in the FMI standard . . . 100
7.2.4 Specification Languages for Hardware Platforms . . . 101
7.2.5 Automatic abstraction of digital components . . . 102
7.3 Methodology Overview . . . 104
7.3.1 Running Example . . . 106
7.4 Generation of Cycle-Accurate FMUs . . . 107
7.4.1 Data-type mapping . . . 108
7.4.2 Automatic generation of Functional Mockup Units . . . 109
7.5 Generation of Transactional FMUs . . . 111
7.6 Generating the coordinators . . . 112
7.6.1 Cycle-Accurate Coordinator . . . 114
7.6.2 Smart Coordinator . . . 115
7.7 Experimental Results . . . 116
7.7.1 Experimental setup . . . 116
7.7.2 Experiments overview . . . 117
7.7.3 In-depth analysis of the experimental results . . . 118

Part IV Multi-Level Modeling and Simulation

8 From Multi-Level to Abstract-Based Simulation of a Production Line 123
8.1 Introduction 123
8.2 Background 123
8.3 Multi-Level Modeling and Simulation 125
8.3.1 Application of Multi-Level Simulation to a Production Line 126
8.4 Abstract-Based Modeling and Simulation 126
8.4.1 Application of Abstract-Based Simulation to Production Line 127
8.5 Experimental Results 127
8.5.1 Multi-Level Experiment 128
8.5.2 Abstract-based Experiment 129
8.5.3 Simulation Results Comparison 130
8.6 Concluding Remarks 131

9 A Design Methodology of Multi-level Digital Twins 133
9.1 Introduction 133
9.2 Background 135
9.2.1 AutomationML - IEC 62714 135
9.2.2 CAEX - IEC 62424-2 135
9.2.3 Plant and Kinematics Simulators 136
9.2.4 Running Example 137
9.3 Methodology in Action 138
9.3.1 Plant Topology Generation 138
9.3.2 Plant OPC-UA Infrastructure Generation 141
9.3.3 Process PLC Infrastructure Generation 143
9.4 Experiments 144
9.5 Conclusion 146

Part V From Real Data to Information

10 Industrial-IoT Data Analysis Exploiting Electronic Design Automation Techniques 149
10.1 Introduction 149
10.2 IIoT Data Analysis 151
10.2.1 Severity levels 151
10.2.2 Measurement points 152
10.2.3 P-F intervals 153
10.2.4 Common variables in condition monitoring 154
10.3 Predictive Maintenance 155
10.4 Experimental Validation 158

10.4.1 Experiment Setup . . . 158
10.4.2 Abstract model of a real machine . . . 159
10.4.3 Methodology validation with a mutation analysis technique . . . 160
10.4.4 Experiment results . . . 160
10.5 Conclusions . . . 162

11 The Design of a Digital-Twin for Predictive Maintenance . . . 163
11.1 Introduction . . . 163
11.2 Background & State of the art . . . 165
11.2.1 Technologies . . . 165
11.2.2 Related Work . . . 166
11.3 Modeling The Problem . . . 167
11.4 From Sensor Data to Severity Levels . . . 167
11.5 Monitoring State Machine (MSM) . . . 169
11.6 Developed Technique . . . 169
11.6.1 Predictive Maintenance Supervisor . . . 169
11.7 Alternative Use of MSM and PSM . . . 172
11.7.1 *Structure Validation Mode* . . . 172
11.7.2 *Condition-based-Maintenance Continuous Mode* . . . 172
11.7.3 *Predictive Maintenance Structure Training Mode* . . . 173
11.7.4 *Predictive Maintenance Mode* . . . 173
11.8 Methodology Application . . . 173
11.8.1 Experimental Setup . . . 173
11.8.2 Abstract Model of a Real Machine . . . 175
11.8.3 Structure Validation Mode in Action . . . 176
11.8.4 Experimental results . . . 176
11.8.5 Standards . . . 177
11.9 Conclusions . . . 178

Part VI Unified Example & Conclusions

12 Industrial Computer Engineering Laboratory & Conclusions . . . 181

13 Published Contributions . . . 183

References . . . 185

List of Figures . . . 193

List of Tables . . . 195

Part I

Preliminary

1

Introduction

1.1 Introduction

The concept of Industry 4.0 [1] represents an innovative vision of what will be the factory of the future. A smart factory should be able to optimize efficiency and productivity by extending the capabilities of both manufacturing devices and people. By focusing on creating an agile, iterative production process through data collection, smart factories can aid decision-making processes with stronger evidence. By continuously improving the productivity of manufacturing processes, smart factories can lower costs, reduce downtime and minimize waste. Identifying and reducing misplaced or underused production capacities mean opportunities for growth without investing in additional monetary and/or physical resources. This concept defines a set of design principles needed to enable industrial plant transformation into smart factories, capable to take decisions based on the status of the plant.

- **Interoperability**: This first principle explores the ability of machines, devices, sensors, and people to connect and communicate with each other via the Internet of Things (IoT). Connect your Smart Tools , sensors and operators on the shop floor to collect valuable data and integrate it with your Manufacturing Execution System (MES),Enterprise Resource Planning (ERP) or other smart factory solution for real-time analysis. It creates a network of interconnected data-generating points that can be accessed and manipulated anytime anywhere. This principle dwells on the technology's ability to provide enhanced information for future decision-making.
- **Modularity**: This is the essence of the production by order. It provides the possibility to change certain parts of a product or an equipment during production in accordance with customer's desires. This principle allows to obtain a dynamic and reconfigurable production line.
- **Virtualization**: The ability of information systems to create a virtual copy of the physical world by enriching digital plant models with sensor data. This requires the aggregation of raw sensor data to higher-value context information. In other words, embracing this industrial revolution's design principle helps monitor processes on the shop floor and allows the management to instantly adjust and optimize for higher efficiency.

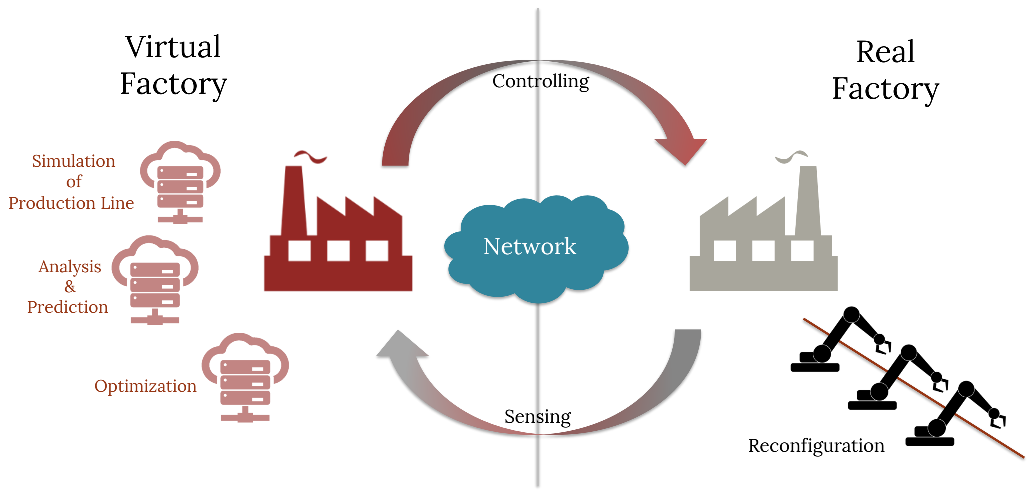


Fig. 1.1: Role of Digital Twin in Industry 4.0.

- **Decentralization**: The decentralization of decisions stems from the ability of Cyber-Physical Systems (CPSs) to make decisions on their own and to perform their tasks as autonomous as possible. Only in case of exceptions, interferences, or conflicting goals, tasks are delegated to a higher level. A decentralized system is also highly adaptable and scalable which determines how efficiently can you respond to industry changes.

The virtualization of the production plant covers a central role in this new revolution. This virtual copy is also called Digital Twin and has the role to mime, predict and optimize what is happening in the real plant (see Figure 1.1).

The entire factory can be represented as a set of CPSs and the resulting system is also called Cyber-Physical Production System (CPPS). This CPPS represents the Digital Twin of the Factory with which it would be possible to make analysis regarding the real factory [3]. The interoperability between the real industrial equipment and the Digital Twin [4] allows to make predictions concerning the quality of the products. Several tools [2, 5] allow to model a production line, considering different aspects of the factory (*i.e.* geometrical properties, the information flows). However, these design principles define only guidelines and actually they are not totally explored in research with methodologies, tools or solutions.

For instance, plant modeling tools [2, 5] do not allow to model the plant with a certain level of details. Some works defines coupling solutions between physical and production simulators, but without defining a unified flow to design and simulate the entire factory. Furthermore, most of the works defined only simple equipment models without considering the integration of raw data coming from real equipment. In particular, the problem of model realignment with the data coming from sensors. Scope of this thesis is the investigation on Digital Twin in Industry 4.0 in order to define a set of methodologies to enable virtualization and the interoperability principle.

1.2 Objectives & Methodology Flow

This thesis proposes a set of methodologies to define solutions for the design of Digital Twin in Industry 4.0 (see Figure 1.2). These novel solutions can be summarized in

- Design methodology for Digital Twin;
- Integration of equipment models;
- Integration of real data to enable Predictive Maintenance.

The entire flow is based on the definition and integration of modeling information of the plant in AutomationML, a neutral standard used to exchange data between different designers and stakeholders. Figure 1.2 shows all the different flows that cover the proposed methodologies. On the top of the Figure, there is the abstracted description of the factory, in AutomationML. We have first to distinguish between the information of the plant (topology and material) and the information of each node that composes the entire factory (equipment models) that needs to be integrated in AutomationML. The proposed metholodogies focus mainly on:

① Homogeneous Models;
② Heterogeneous Models;
③ Multi-Level Models;
④ From Real Data to Information.

Each equipment that composes the entire factory can be seen as a CPS, where the cyber part controls physical processes.

The term *Homogeneous model* is used to represents CPS that are modeled using a unique framework or language. For instance, environments that support model-based design paradigm

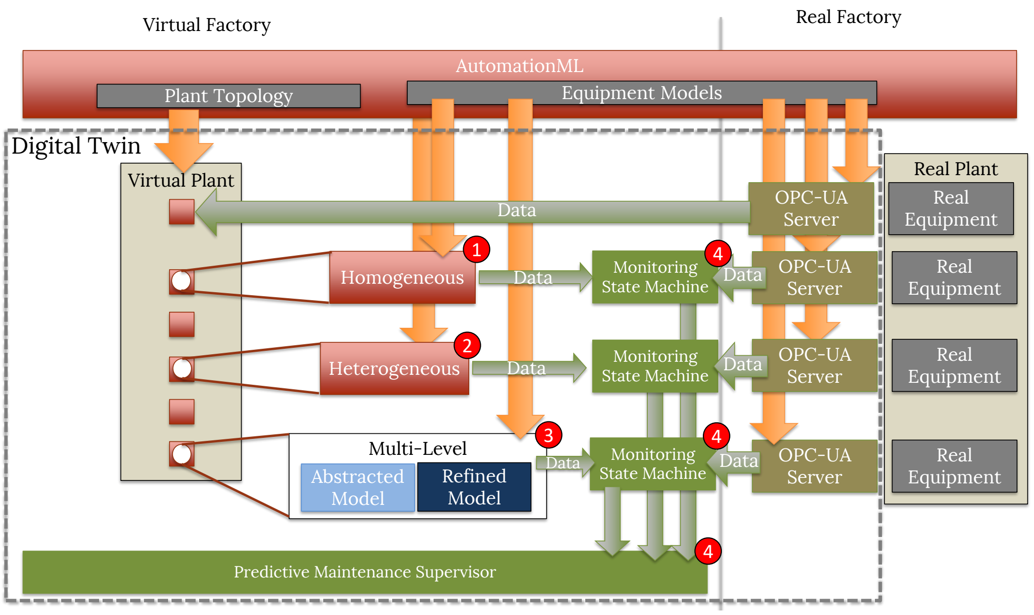


Fig. 1.2: Overview of the proposed methodologies. Circles represents the aspects discussed in this thesis.

like Ptolemy, Modelica, or Simulink. The proposed approach allows to obtain synthetic nodes, through the OPC Unified Architecture (OPC UA) transmission protocol, that can be used to evaluate equipment models or production recipes effects on the production line.

With the term *Heterogeneous Models*, we refer to CPSs that are modeled considering different tools for cyber and physical models. In particular, the use of Hardware Description Languages (HDLs) to model the cyber part and tools to model dynamic systems for the physical process. In this scenario, we focused mostly on the definition of simulation strategies, moving from co-simulation environments to simulation. Moreover, the investigation on the definition and automatic generation of ad-hoc simulation coordinator based on system-level information.

In the *Multi-level* approach, the use multiple descriptions of the same system is discussed. The goal of this approach is to reduce the global complexity of the system by considering two descriptions of the same system with different levels of detail. This approach is based mainly on a simple idea: switching between two models, referring to the same system, to reduce computational effort or increment the level of details when desired.

Finally, with *From Real Data to Information* the investigation on the integration of real data coming from the field. This part seems to be far from what has been explained, but the Digital Twin can not be based only on models. It needs data from the field to be synchronized to analyze the status of the plant. The interconnection within models and real data is necessary for the Digital Twin in order to perform predictions or take decisions regarding equipment maintenance or production plans. From the Digital Twin perspective, raw data needs to be manipulated and transformed in order to dig up information that can be used to plan strategies. This part of the thesis aims to put a first step in this direction, defining a methodology to retrieve equipment status from real data coming from sensors.

The thesis is structured as follows: Part II introduces the proposed approach for the automatic generation of synthetic nodes by integrating *Homogeneous Models* in AutomationML. Part III explores the modeling and simulation of *Heterogeneous Models*, with a particular focus on the discrete part of the system, proposing solutions regarding ad-hoc coordinators based on system-level information. The *Multi-level* approach is shown in Part IV. Part V explains the obtained approach regarding the detection of information from raw data coming from equipment sensors. Finally, Part VI sum-up all the proposed methodologies in a unified example, drawing conclusions, while reporting the publications developed during this thesis in Chapter 13.

Part II

Homogeneous Models

2

Integrating synthetic and real components of a cyber-physical production system

2.1 Introduction

Since their first introduction in 2003 by Michael Grieves, *Digital Twins* have come a long way. They are one of the fundamental pillar of the *Fourth Industrial Revolution* and for a good reason, as they can make things like smart manufacturing a realistic thing.

When we talk about *smart manufacturing*, we refer to a series of technologies that through the usage of interconnected *Cyber-Physical Systems* manage to reach a great production flexibility and things like customized products can be automatically produced by an already operational assembly line, which is impossible with the current mass production as it would require a complete manual readjustment of the assembly line machines. This is only one aspect of *Industry 4.0*. There are a lot of other technologies involved like *Cloud Computing*, *Big Data* analysis, etc., each of them having a specific role for covering a specific aspect.

Digital twins saw many implementations, all of them having usage of models in common. Machine models can be created in many ways at various degrees of abstraction. The important aspect is that these models have to communicate with each other and work together. We call this *Machine-to-Machine* communication and this is what let them to self-configure depending on the condition of the other machines. The most promising protocol for realizing Machine-to-Machine (M2M) communication is OPC Unified Architecture (OPC UA), which offers a platform-independent service-oriented infrastructure. `AutomationML` [6–8] represents a new standard that allows to exchange information of equipment of plant, between different vendors or designers to make models and in general plant descriptions to be freely exchanged by modeling tools.

This research work analyzes these technologies and a then presents a case study about the automatic generation of synthetic OPC UA server. Figure 2.1 shows the proposed methodology. The model is defined in a vendor-neutral standard, `AutomationML`,then synthesized in `MATLAB-Simulink`, and exported as a stand-alone block with the use of Functional Mock-up Interface (FMI) standard. Finally, the obtained block,called Functional Mock-up Unit (FMU), is wrapped into an OPC UA server. The automatic generation of an OPC UA server from a FMU and its integration within a real virtual twin is also presented. Figure 2.1 reports the overview of the proposed methodology.

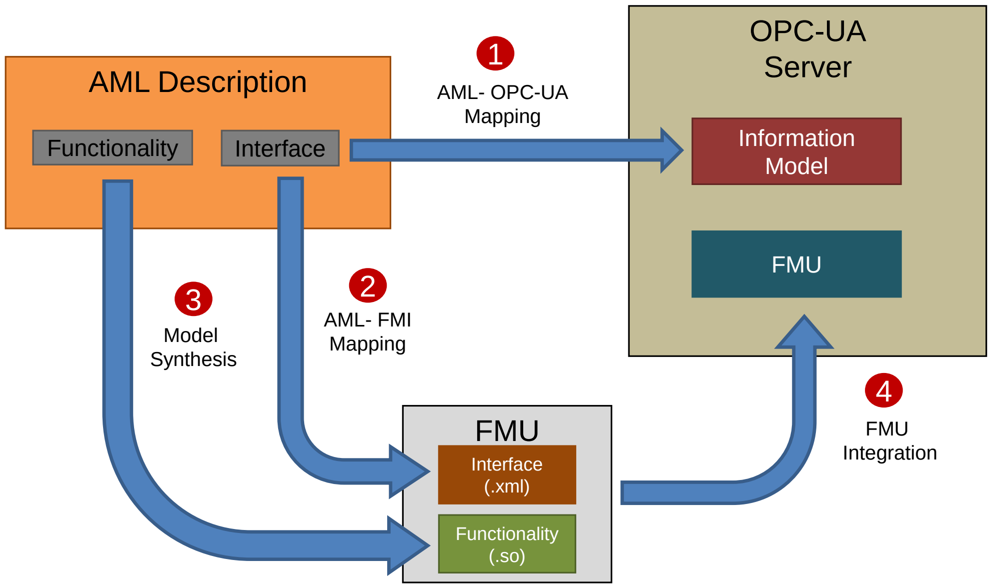


Fig. 2.1: Overview of the proposed methodology.

The benefits of the proposed approach are :

- Integration of Simulink semantic in `AutomationML` vendor-neutral standard;
- Automatic synthesis flow of mechanical models from `AutomationML`;
- Fully automatic generation of synthetic `OPC UA` servers for equipment evaluation.

The presented work focuses on mapping `Simulink` into `AutomationML` as a proof-of-concept, buts supporting other modeling frameworks can be done by following a similar flow.

The sematic gap between `AutomationML` and `Simulink` has been filled with the definition of a special `AutomationML` class called `SimulinkRoleClassLib`.

A `Simulink` model can be the *AutomationML Editor* using this `SimulinkRoleClassLib`.

The use of FMI standard allows to export the synthesized model as a standalone block called FMU.

This can be embedded inside an OPC UA server automatically, and then be added to a production plant simulator to perform production product evaluation. A real digital twin integration example is then presented using Siemens Tecnomatix Plant Simulation, by adding an automatically generated FMU to the plant description of the ICE lab of University of Verona.

Some works tried to generate simulable models from an `AutomationML` description [9]. The problem of this approach is that while it's very powerful and flexible, it requires the definition and usage of complex ontologies. In [8] the authors explored the usage of semantic web technologies to support `AutomationML` model exchange, but the major issue comes from lacks in the definition of the used onthology.

This research work is organized as follows: Section 2.2 introduces the main concepts needed to properly understand the rest of the work, like a general overview of the aforementioned standards.Section 2.3 will discuss about the proposed methodology, while Section 2.4 will show practical appliance of the work to a real world example, the ICE laboratory from University of Verona. Finally, Section 2.7 will present some remarks and shows possible future works.

2.2 Background

2.2.1 OPC UA Communication Protocol

OPC Unified Architecture (OPC UA, [10]) aims to standardize M2M communication [11].

Definition of OPC specifications [12] started to simplify and to standardize data exchange between software applications in industrial environment. The rapid diffusion of the first version of OPC specifications was due to the choice of Microsoft DCOM as the technological basis. However, exactly this point raised the majority of criticism regarding OPC because it was too focused on Microsoft, platform-dependent and not firewall-capable, and thus not suitable for use in cross-domain scenarios and for the Internet. When XML and Web Services technologies have become available, the OPC Foundation adopted them as an opportunity to eliminate the shortcomings of DCOM. Since 2003 the *OPC XML Data Access* (DA) specification has offered a first service-oriented architectural approach besides the "classic" DCOM-based OPC technology. This Web services-based concept enabled applications to communicate independently of the manufacturer and platform.

Few years later, the OPC Foundation has introduced the OPC UA standard which is based on a *service-oriented*, *technology* and *platform-independent* approach, creating new and easy possibilities of communicating with Linux/Unix systems or embedded controls on other platforms and for implementing OPC connections over the Internet. The new possibilities of using OPC components on non-Windows platforms, embedding them in devices or implementing a standardized OPC communication across firewall boundaries allow speaking of a change of paradigms in OPC technology. OPC UA servers can be varied and scaled in their scope of functions, size, performance and the platforms they support. For embedded systems with limited memory capacities, slim OPC UA servers with a small set of UA services can be implemented; at the company level, in contrast, where memory resources are not that important, very powerful OPC UA servers can be used with the full functionality.

OPC UA specifications now offer a security model, which wasn't available in the previous versions of OPC specifications; the OPC UA security governs the authentication of clients and servers and ensures data integrity, trustworthiness and authorization within OPC communication relationships [13].

The OPC UA architecture models Clients and Servers as interacting partners. Each system may contain multiple Clients and Servers. Each Client may interact concurrently with one or more Servers, and each Server may interact concurrently with one or more Clients. An application may combine Server and Client components to allow interaction with other Servers and Clients. *Server to Server* [14] interactions in the Client Server model are interactions in which one Server acts as a Client of another Server. Server to Server interactions allow for the development of servers that:

- exchange information with each other on a peer-to-peer basis, this could include for example redundancy;
- are chained in a layered architecture of Servers to provide aggregation of data from lower-layer Servers, higher-layer data constructs to Clients, concentration interfaces to Clients for single points of access to multiple underlying Servers.

OPC UA can be used at different levels of the automation pyramid for different applications within the same environment. At the plant floor level, an OPC UA server may run in a controller providing data from field devices to OPC UA clients (e.g. HMIs, SCADA). On top of the plant floor at operation level, an OPC UA application may be a client collecting data from the server at the lower level, performing special calculations and generating alarms; an example is represented by an OPC UA client integrated in an ERP system, obtaining information about used devices in the plant floor (e.g. working hours) and creating a maintenance request.

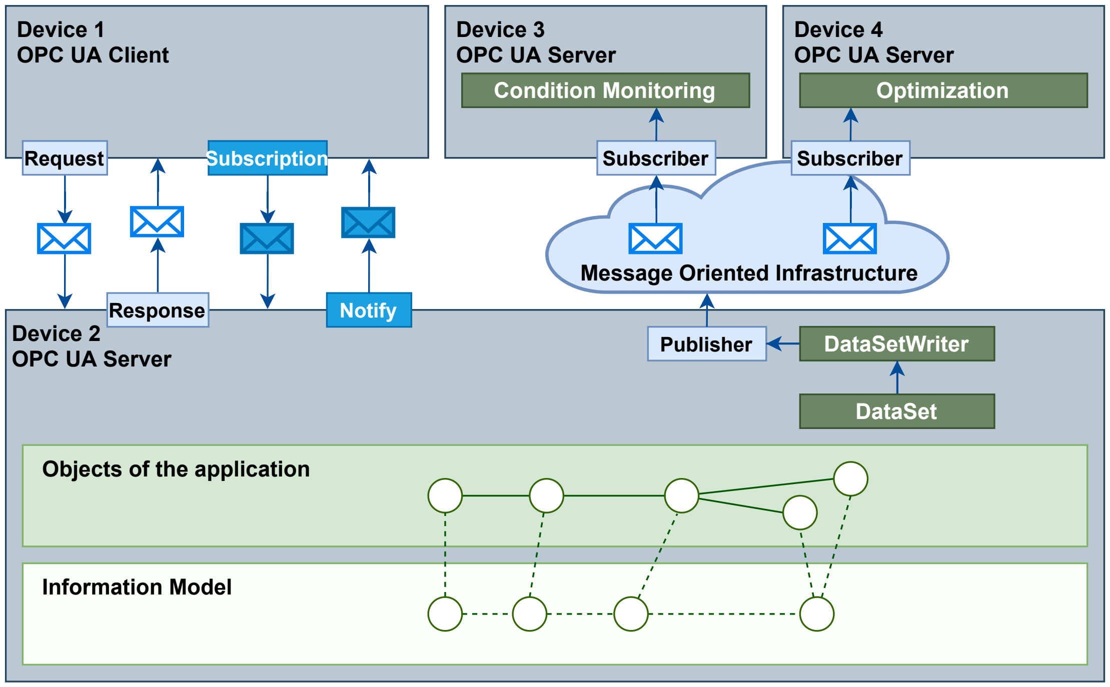


Fig. 2.2: OPC UA stack example

2.2.2 AutomationML

`AutomationML` is a standard based on `XML` which aims to provide reliable data exchange in the engineering process of production systems [15]. An interesting aspect of `AML` is that it doesn't develop any new data format for achieving its purpose, but instead it uses already existing formats, adapted and extended when needed, then merged properly. So far, the representation of plant specific data in general and in special the plant structure, geometry and kinematics, and logic description is possible. Additional representations for networks, mechatronics systems, and others are in progress. Within IEC62714 all parts of AML are going to be standardized internationally. AutomationML has a lean and distributed file architecture. It does not define any new file format but combines existing established XML data formats which have been proven in use for their specific domain. This is why the normative part of the IEC62714-1:2018 document consists of 32 pages only. The data formats for the following modelling domains are:

- *object topologies* including hierarchies, properties and relations of objects: CAEX according to IEC 62424
- *geometries* and *kinematics* of objects: COLLADA 1.4.1 and 1.5.0 (ISO/PAS 17506:2012)
- *discrete behavior* of objects: PLCopen XML 2.0 and 2.0.1; in addition, IEC62714-4 will allow the usage of IEC61131-10

CAEX according to IEC62424 forms the base of AutomationML. It stores object-oriented engineering information, e.g. a plant hierarchy structure (see AutomationML topology). Each CAEX object can contain properties and reference geometry, kinematics or logics information stored in third party XML files. This enables cross-domain modelling and is designed for future extension.

Topologies

Object hierarchies in CAEX form the core of AutomationML (as in figure 2.3). A CAEX object is a data representation of any asset. It can model physical assets, e.g. a motor, a robot, a tank; or abstract assets like a function block, a model or a folder. CAEX allows to link those objects to systems, since every physical or logical system is characterized by internal elements (objects) which may contain further internal elements, and all elements may have interfaces, attributes and connections with each other. Finally, CAEX allows the modeling of any plant topology, communication topology, process topology, resource topology etc.

Geometry and kinematics

As mentioned above AutomationML exploits the international standard COLLADA 1.4.1 and 1.5.0 for the representation of geometry and kinematics information which is standardized as ISO/PAS 17506:2012. Therefore, AutomationML has developed a two-stage process:

1. relevant geometries and kinematics are modelled as COLLADA files.
2. these files and the data objects within them are referenced out of the CAEX file.

COLLADA stands for COLLAborative Design Activity. It was developed by the KHRONOS association under the leadership of Sony as an intermediate format within the scope of digital content creation in the gaming industry.

It is designed to enable the representation of 3D objects within 3D scenes covering all relevant visual, kinematic, and dynamic properties needed for object animation and simulation.

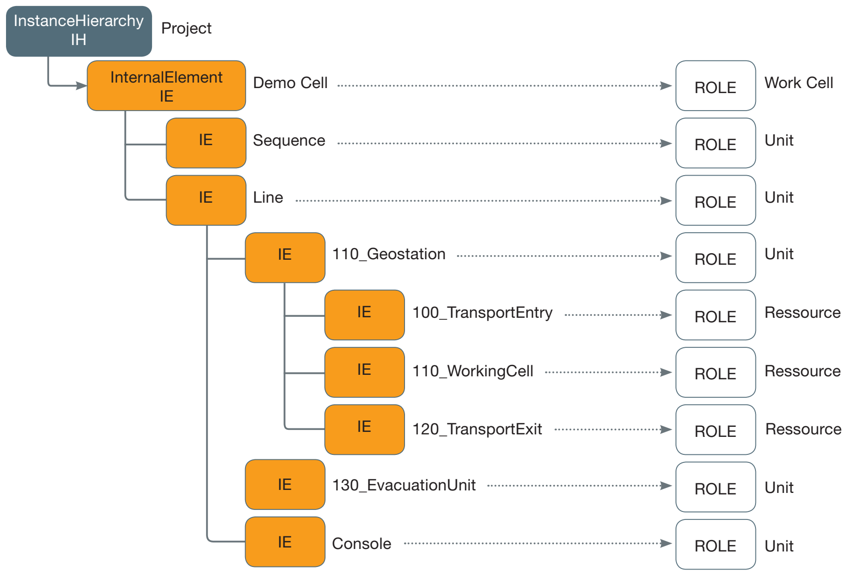


Fig. 2.3: Example of a plant topology in AML.

COLLADA is an XML-based data format with a modular structure enabling the definition of libraries of visual and kinematic elements. It can contain libraries for the representation of geometries, materials, lights, cameras, visual scenes, kinematic models, kinematic scenes, and others.

The most important feature of AutomationML is the clear identifiability of objects in COLLADA files, which allows the integration of these files into AutomationML. Several data objects within a COLLADA file have a unique identification (ID) like geometries, visual scenes, kinematic models and kinematic scenes.

In order to reference these objects, AutomationML has defined a special interface class within the AutomationMLInterfaceClassLib named COLLADAInterface which shall be applied to derive the needed interfaces for geometry integration. This interface class itself is derived from the interface class ExternalDataConnector and therefore has an attribute refURI. This attribute can be used to reference into a COLLADA file, thereby referring to an ID of an object modelled in the COLLADA file.
Thus, the value of the refURI attribute shall contain a string structured like file:///filename.dae#ID. The attribute refType is used to differentiate between various ways of embedding objects in a modeled scene. It can provide information on how static an object in the scene is in relation to other objects, e.g. whether a work piece and the conveyor belt move at the same time.

Modeling behaviour

AutomationML with its object-centric modelling approach enables a dedicated storing of logic information on object level. For this purpose, certain object semantics as well as object interface semantics are developed. In addition to that, logic models are identified, commonly used in the engineering process of a production system, and made exchangeable, but also transformable among each other. This allows and supports an information enrichment process in terms of logic information that is required for the scope of `AutomationML`. All concepts are going to be standardized in `IEC 62714-4`.

Logic information is an important aspect for raw system planning, electrical design, `HMI` development, `PLC` and robot control programming, for simulation purposes, and virtual commissioning. To support the different phases in the iterative production system engineering process covering different levels of detail, `AutomationML` needs to be able to store logic information from different tools and disciplines.

2.2.3 Functional Mockup Interface

The FMI standard defines an interface that allows to encapsulate models from different tools. The primary goal is to support the exchange of simulation models between suppliers and OEMs even if a large variety of different tools are used. Actually the `2.0` version of the standard has been released, and it consists of two main parts: Model Exchange and Co-Simulation.

The *Model Exchange* interface provides a method to generate C-code in the form of an input output block. This method is used by different simulators to export only the descriptions of their models without exposing their internal solver algorithm which, in general, is closed source.

The *Co-Simulation* interface provides a method to export a model in the form of a block including also a mathematical solver needed to execute correctly that model. This allows a simulator to load and execute other models correctly even when the correct solver is not available. The provided model description is very similar to the *Model Exchange* one. The main difference between the two approaches is the location of the solver: in the first case it is provided by the simulator tool while in the second is integrated into the model exported.

A component which implements the interface is called FMU.

2.2.4 MATLAB/Simulink

`MATLAB` is a multi-paradigm numerical computing environment, featuring a proprietary scripting language developed by MathWorks. `MATLAB` allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages. The environment is highly configurable and extensible with plugins, and in fact one of the most popular one is Simulink, which frequently comes directly with the default `MATLAB` installation. `Simulink` adds graphical multi-domain simulation and model-based design for dynamic and Embedded Systems. In Model-Based Design, a system model is at the center of the workflow. Model-Based Design enables fast and cost-effective development of dynamic systems, including control systems, signal processing systems, and communications systems. Model-Based Design allows you to:

- use a common design environment across project teams
- link designs directly to requirements
- refine algorithms through multi-domain simulation

- automatically generate embedded software code and documentation

Till version `2019b Simulink` had a plugin, called "Tool-Coupling Co-Simulation FMU Export", that allowed to export a model as an FMU, supporting the Co-Simulation technique. The problem is that this approach required to have a local `MATLAB` installation to effectively simulate the model, and it wasn't a robust approach. Starting from `MATLAB 2020a`, Simulink allows to export a standalone FMU without requiring an external `MATLAB` instance, but embedding a fixed-step solver.

2.2.5 Tecnomatix Plant Simulation

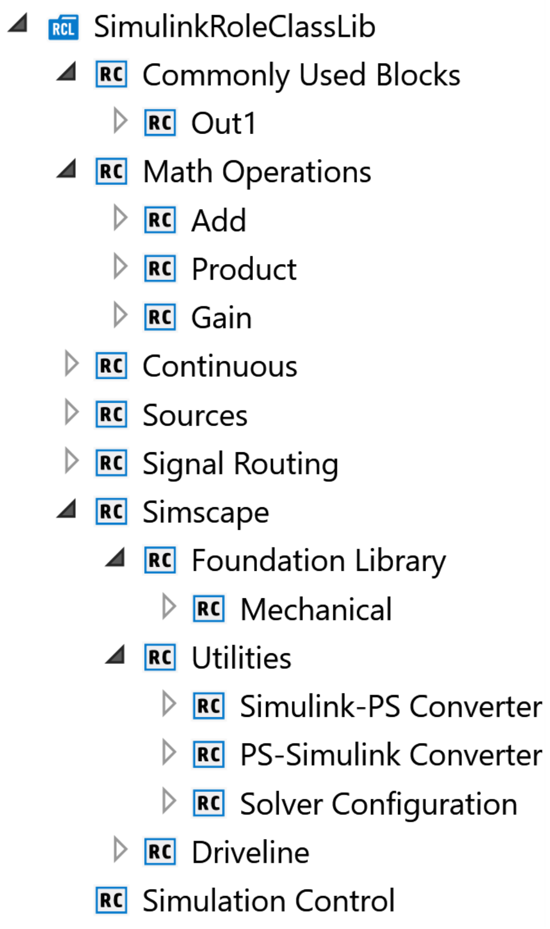
`Tecnomatix Plant Simulation` is a computer application developed by *Siemens PLM Software* for modeling, simulating, analyzing, visualizing and optimizing production systems and processes, the flow of materials and logistic operations. By using `Tecnomatix Plant Simulation`, users can optimize material flow, resource utilization and logistics for all levels of plant planning from global production facilities, through local plants, to specific lines. `Tecnomatix Plant Simulation` belongs to the *Product Lifecycle Management Software* (PLM) portfolio. `Plant Simulation` is a *Material Flow simulation* or *Discrete Event Simulation* (DES) Software. *Material flow* refers to the description of the transportation of raw materials, pre-fabricates, parts, components, integrated objects and final products as a flow of entities. This means that a computer model allows to execute experiments and to run through "what if scenarios" without either having to experiment with the real production environment or, when applied within the planning phase, long before the real system exists.

2.3 Methodology

This section describes the methodology adopted to implement the synthetic OPC UA node, starting from `AutomationML`. First, the semantic map between Simulink and `AutomationML` is explained. Then, the entire generation flow is discussed. Detailed steps are explained for the actual `AutomationML` document parsing and also a look at the final OPC UA server, discussing the adopted solutions.

2.3.1 Mapping Simulink components in AutomationML

This section discusses about the semantic mapping between `Simulink` and `AutomationML`. `AutomationML`, or more precisely `CAEX`, uses the concept of *Role* to define semantics. A role describes an abstract functionality without defining the underlying technical implementation. Thus, it has to be seen as an indicator for the semantics of an object which can be described in an abstract way. Roles can include general attributes (`size`, `number of axes`, etc.) and interfaces (`PPRConnector`, ...) to describe the interaction possibilities of the element which assigns this role. Roles are organized hierarchically in libraries and can have interrelations to other roles and further elements to describe their dependencies.



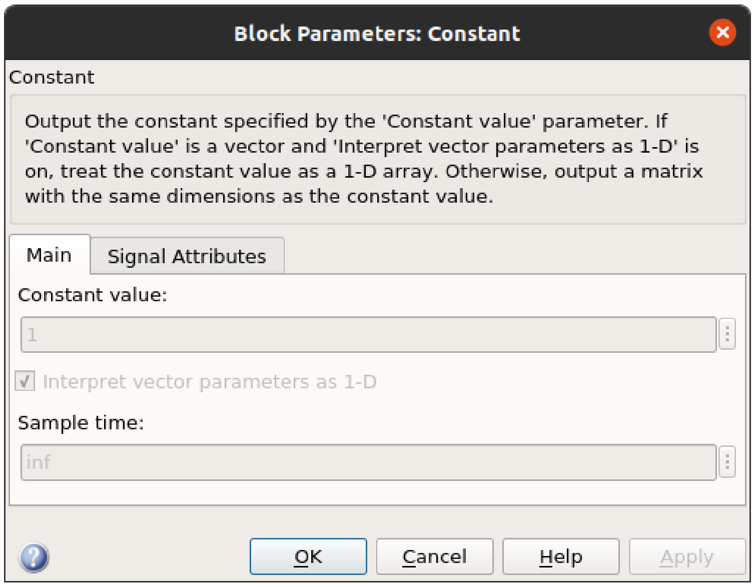
- SimulinkRoleClassLib
  - Commonly Used Blocks
    - Out1
  - Math Operations
    - Add
    - Product
    - Gain
  - Continuous
  - Sources
  - Signal Routing
  - Simscape
    - Foundation Library
      - Mechanical
    - Utilities
      - Simulink-PS Converter
      - PS-Simulink Converter
      - Solver Configuration
    - Driveline
  - Simulation Control

Fig. 2.4: The final result of SimulinkRoleClassLib

AutomationML defines some basic role sets by default. These roles are based on the general role class library (AutomationMLBaseRoleClassLib) defined in the AutomationML standard. The role class AutomationMLBaseRole in the AutomationMLBaseRoleClassLib is a basic abstract role type and the base class for all standard or user-defined role classes. All AML objects shall be associated directly or indirectly to the role class AutomationMLBaseRole to have a common basis, e.g. for simplifying the implementation. Role class libraries can be defined for general use cases but can also relate to specific domains. They don't need to be standardized or defined by AutomationML but can also be created by user groups or single users of AutomationML. The main contribution of this work is the definition of a specific AutomationML RoleClassLib, called SimulinkRoleClassLib, that contains all the Simulink components semantic with their respective properties (see figure 2.4).

Mapping of Constant

Simulink Constant, represents a simple component that gives as output a constant value. Inside the *Library Browser*, the Constant can be found in Sources section, thus the correspondent RoleClass path will be SimulinkRoleClassLib/Sources/Constant. As the



Block Parameters: Constant

Constant

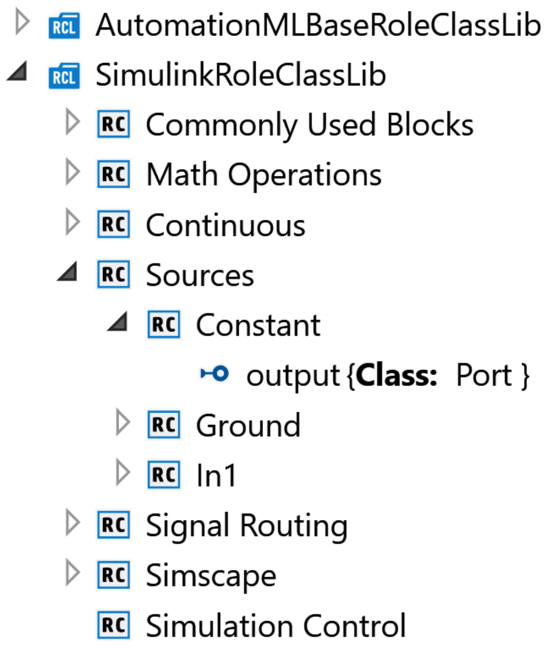
Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

Constant value: 1

☑ Interpret vector parameters as 1-D

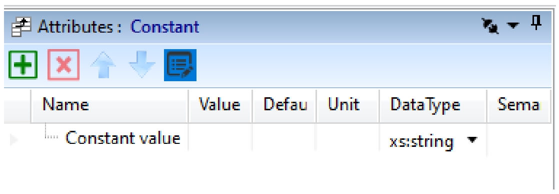
Sample time: inf

Fig. 2.5: Attributes of the Simulink Constant component



- AutomationMLBaseRoleClassLib
- SimulinkRoleClassLib
  - Commonly Used Blocks
  - Math Operations
  - Continuous
  - Sources
    - Constant
      - output {**Class:** Port }
    - Ground
    - In1
  - Signal Routing
  - Simscape
  - Simulation Control

Fig. 2.6: The Constant RoleClass



Attributes : Constant

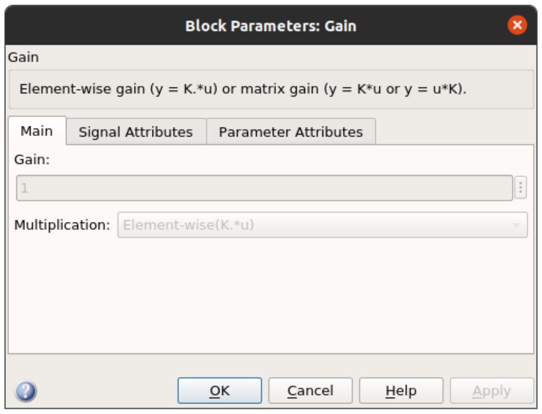
Name	Value	Defau	Unit	DataType	Sema
Constant value				xs:string	

Fig. 2.7: The Constant RoleClass only mapped attribute

`Constant` component doesn't have any input port, in the correspondent `AML RoleClass` only the output port was instantiated (2.6). Attributes of this component are very simple, the only interesting one being `Constant value`, mapped in AML through a `xs:string` data type (2.7). In general, all the attributes will be mapped as strings, because `Simulink` accepts only strings (that will be parsed by `MATLAB` internally).

Mapping of Gain

Another simple yet fundamental component being mapped is the `Simulink Gain` (2.8). In this case, the component has both an input and an output port (2.10) representing the initial value and the value multiplied by the gain, and the gain's value itself is stored in the appropriate attribute (2.9). It is worth mentioning the port attributes, common to all I/O ports in


Block Parameters: Gain

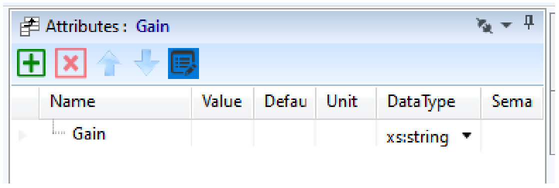
Gain

Element-wise gain (y = K.*u) or matrix gain (y = K*u or y = u*K).

Gain: 1

Multiplication: Element-wise(K.*u)

Fig. 2.8: Attributes of the Simulink Gain component


Attributes : Gain

Name	Value	Defau	Unit	DataType	Sema
Gain				xs:string	

Fig. 2.9: The Gain RoleClass only mapped attribute

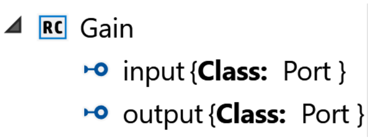
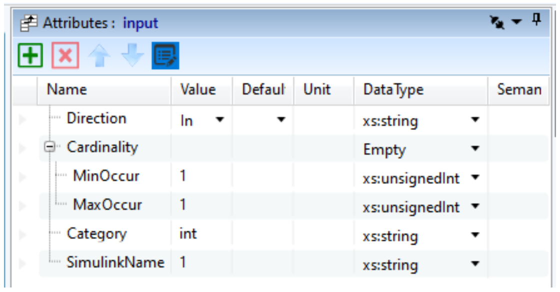


Fig. 2.10: The Gain RoleClass


Attributes : input

Name	Value	Defaul	Unit	DataType	Seman
Direction	In			xs:string	
Cardinality				Empty	
MinOccur	1			xs:unsignedInt	
MaxOccur	1			xs:unsignedInt	
Category	int			xs:string	
SimulinkName	1			xs:string	

Fig. 2.11: The Gain input port attributes

`SimulinkRoleClassLib`. Being instance of the `Port` base `AML` interface, all shares the following attributes:

- `Direction`
  It can be `In`, `Out` or `InOut` and it specifies if that port receives or sends data.
- `Cardinality`
  This combined attribute specifies how many connections the port is supposed to have.
- `Category`
  This specifies the type of the data that's passing through that port.

In addition to these, a fourth attribute was added, `SimulinkName`. This is because `MATLAB` uses special names for manually linking components' ports. When `Simulink` components are used, ports gets numbered starting from 1, both for inputs and outputs, in the way shown in 2.12.

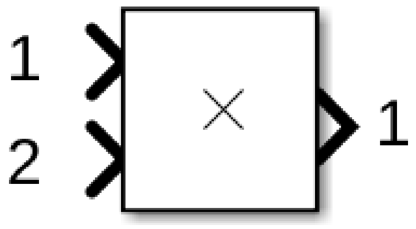
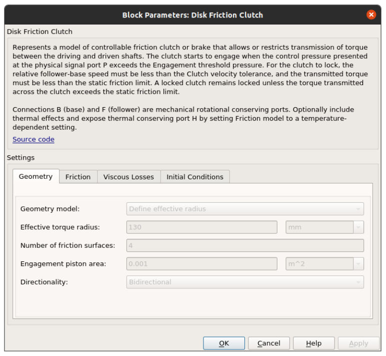


Fig. 2.12: How the ports gets named for a Simulink component

Mapping of Disk Friction Clutch

This component belongs to the `Simscape` library, an extension toolbox of Simulink, which provides a lot of specific and complex blocks for modeling and simulating multidomain physical systems (mechanical, rotational, electrical *etc.*).



Disk Friction Clutch

Represents a model of controllable friction clutch or brake that allows or restricts transmission of torque between the driving and driven shafts. The clutch starts to engage when the control pressure presented at the physical signal port P exceeds the Engagement threshold pressure. For the clutch to lock, the relative follower-base speed must be less than the Clutch velocity tolerance, and the transmitted torque must be less than the static friction limit. A locked clutch remains locked unless the torque transmitted across the clutch exceeds the static friction limit.

Connections B (base) and F (follower) are mechanical rotational conserving ports. Optionally include thermal effects and expose thermal conserving port H by setting Friction model to a temperature-dependent setting.

Source code

Settings

Geometry model:	Define effective radius	
Effective torque radius:	130	mm
Number of friction surfaces:	4	
Engagement piston area:	0.001	m^2
Directionality:	Bidirectional	

Fig. 2.13: Attributes of the Simscape Disk Friction Clutch component

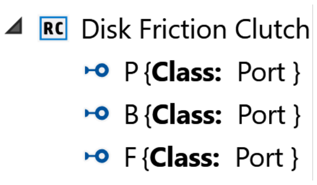
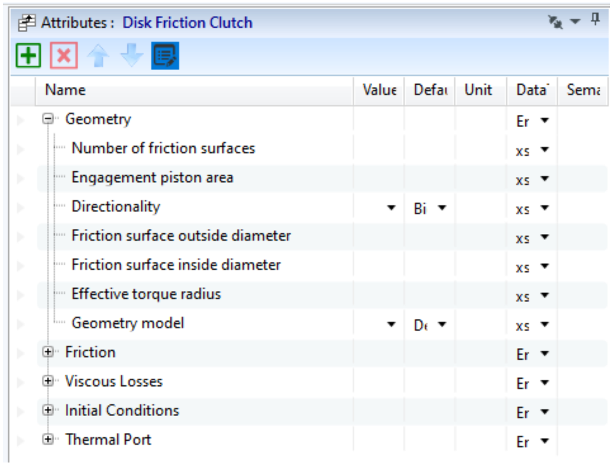


Fig. 2.14: The Disk Friction Clutch RoleClass

This component has a lot of attributes, distributed among tabs. Inside `AutomationML` this logical division can be achieved through "composite" attributes as shown in 2.15. Being a `Simscape` component, the `SimulinkName` attribute is different. In fact here ports get named by position plus number, so the first port at the right will be named `RConn1`, while the second will be `RConn2` and so on. The same goes for the left sided ports, the only difference being the first letter which will be L, so we'll have `LConn1`, `LConn2`, etc. as shown in 2.16.

2.3.2 AML to MATLAB

First thing to do is analyzing the `AutomationML` document and grab the useful information for converting it to a `MATLAB` script. Being `AutomationML` a combination of `CAEX`, `COLLADA` and `PLCOpenXML`, parsing it is equal to parsing `XML`. Before beginning to write a parser, research was done in order to understand if there were already existing parsers, also seeing if they could be


Attributes : Disk Friction Clutch

Name	Value	Defa	Unit	Data	Sema
Geometry				Er	
Number of friction surfaces				xs	
Engagement piston area				xs	
Directionality		Bi		xs	
Friction surface outside diameter				xs	
Friction surface inside diameter				xs	
Effective torque radius				xs	
Geometry model		D		xs	
Friction				Er	
Viscous Losses				Er	
Initial Conditions				Er	
Thermal Port				Er	

Fig. 2.15: The Disk Friction Clutch mapped attributes

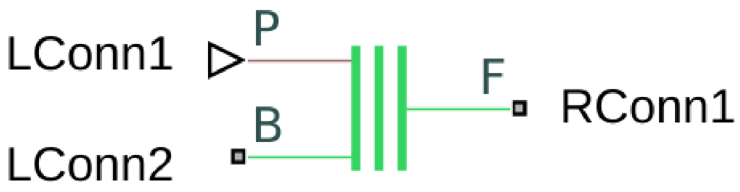


Fig. 2.16: Port names for Simscape components

reused for our purpose. The methodology has been implemented in an automatic parser written in C++.

The conversion from an `AutomationML` model written with `SimulinkRoleClassLib` to a `MATLAB` script takes place in a dedicated class, called `AML2MATLAB`. Before the actual conversion process, the `MATLAB` scripting commands for generating `Simulink` models are explained.

MATLAB script

`MATLAB` provides two basic primitives respectively for adding and connecting `Simulink` blocks:

- `add_block()`
- `add_line()`

In addition to these commands there are a set of other methods needed to set model parameters, auto-arrange the model's layout, and so on.

The `add_block()` primitive takes n parameters as input, with the first one being the `Simulink` absolute path of the object, the second one the relative path of the model along with the block instance name, and from the third onwards the object's parameters, expressed as

```
'NAME_OF_THE_PARAMETER','PARAMETER_VALUE'
```

Listing 2.1: Matlab script that generates simple Simulink model.

```
1  model = 'AdderAndMultiplier';
2
3  new_system(model);
4  open_system(model);
5
6  add_block('simulink/Commonly Used Blocks/In1',[model,'/Input']);
7  add_block('simulink/Commonly Used Blocks/Sum',[model,'/Sum']);
8  add_block('simulink/Commonly Used Blocks/Product',[model,'/Product']);
9  add_block('simulink/Commonly Used Blocks/Out1',[model,'/Output']);
10
11 add_line(model,'Input/1','Sum/1');
12 add_line(model,'Input/1','Product/1');
13 add_line(model,'Input/1','Product/2');
14 add_line(model,'Product/1','Sum/2');
15 add_line(model,'Sum/1','Output/1');
```

The `add_line()` primitive takes three input. The first is the name of the system, the second is the origin of the line (`RefPartnerSideA` in `AML`) and the third is the endpoint (`RefPartnerSideB` in `AML`).

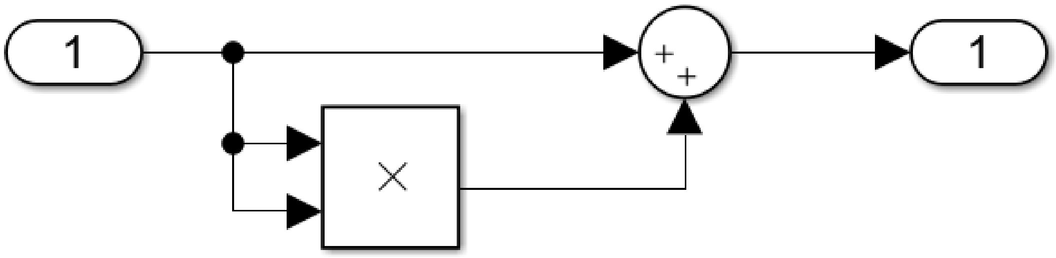


Fig. 2.17: The result of the script reported in listing 2.1

Listing 2.1 shows a simple matlab script that represents the simple model shown in figure 2.17. Lines 6,9 represent the instantiation of all the blocks that composes the model. All the blocks are then linked `add_line` function (lines 11,15).

2.3.3 OPC UA server from FMU

The generated model can be easily exported as an FMU through a set of predefined function, using `MATLAB` GUI or also through a set of function and embbeded in the matlab script. Now that the `Simulink` script has been produced, it can be run inside the `MATLAB` environment to produce a standalone co-simulation `FMU`. The limitation of `MATLAB FMU` exporter is that the encapsulated solver used a fixed-step strategy that is not precise like the variable-step solvers.

For the simulation of the generated FMU inside an OPC UA server, a wrapper is needed. The wrapper has to load the FMU, read its I/O variables, start an OPC UA server and put those I/O variables in it. This routine has to be executed with a certain frequency and for each loop executing a simulation step, using the values from the OPC UA server for I/O, as shown in 2.18. This wrapper has been developed using `C++`, using `FMI4CPP` to manage the FMU and the

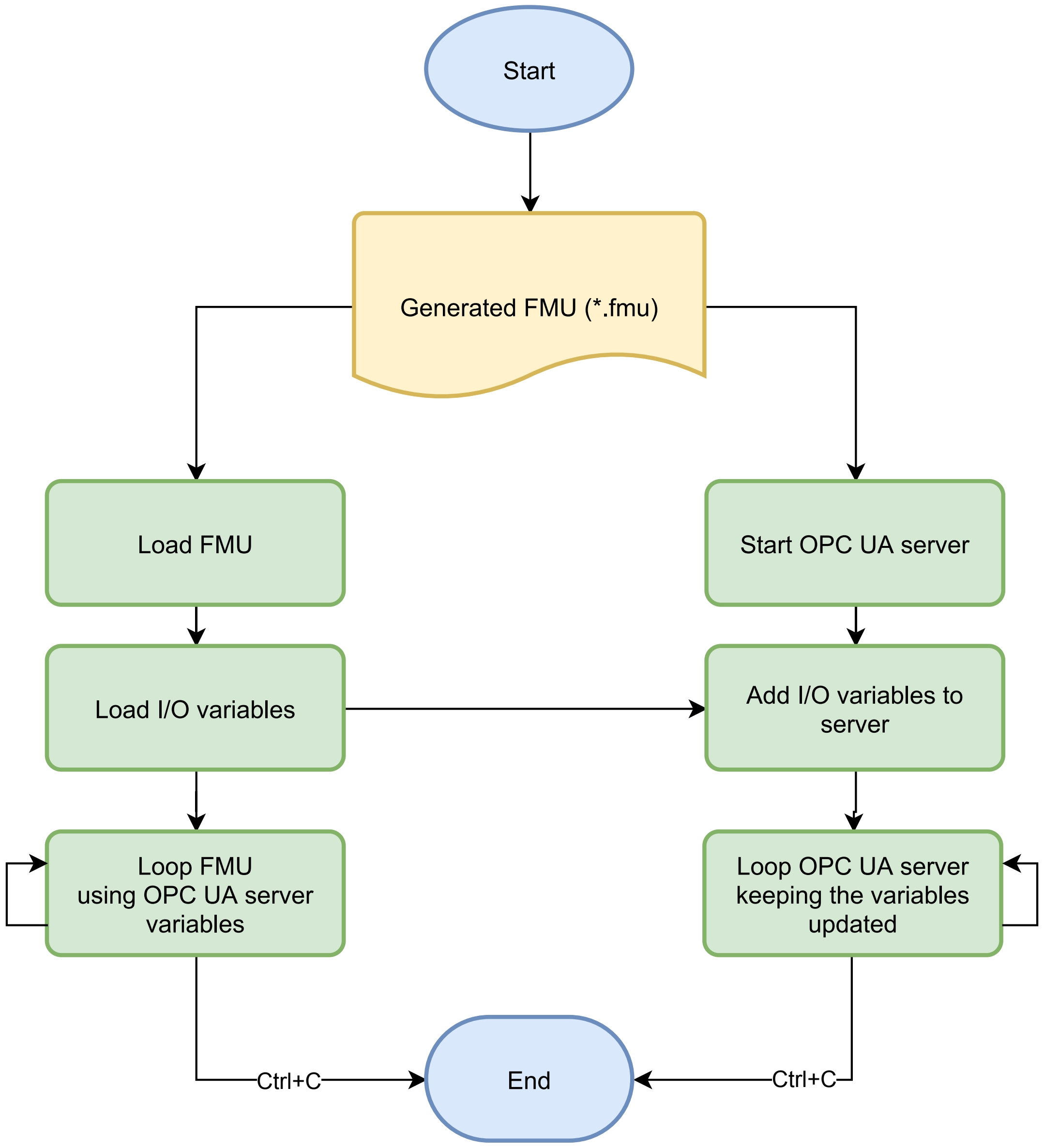

Start
Generated FMU (*.fmu)
Load FMU
Start OPC UA server
Load I/O variables
Add I/O variables to server
Loop FMU using OPC UA server variables
Loop OPC UA server keeping the variables updated
Ctrl+C
End
Ctrl+C

Fig. 2.18: Structure of the wrapper routine.

`open62541` OPC UA stack [16] to create the OPC UA node.

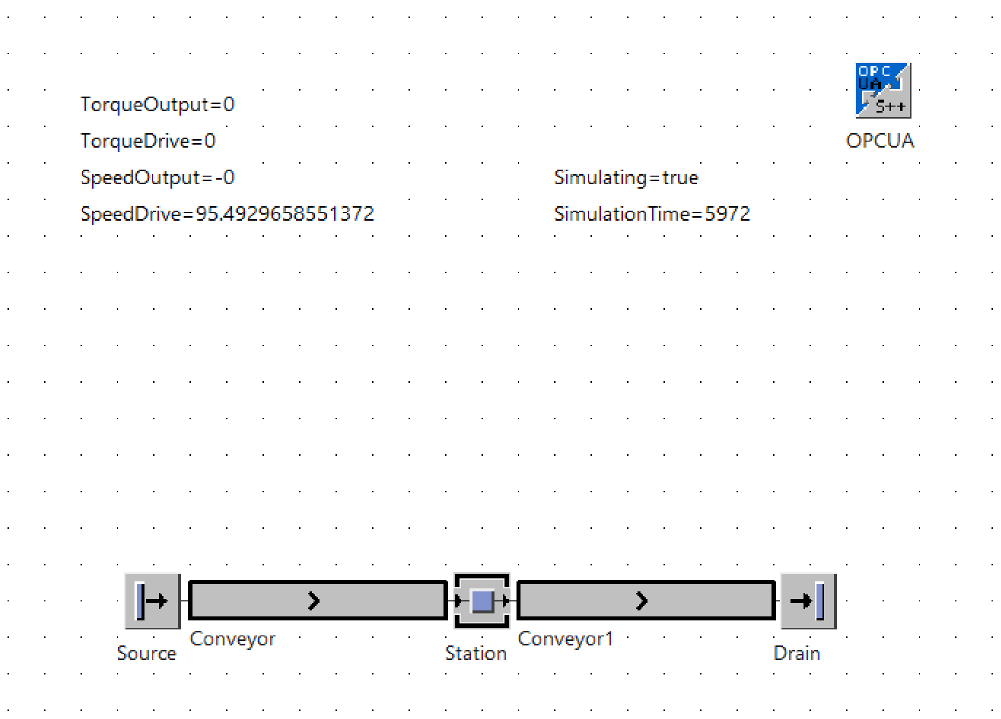
2.4 Experimental Results

In this Section the result of the model generation flow is integrated first into a simple model and then into the *Digital Twin* of the ICE lab, an educational lab of University of Verona for demoing Industry 4.0 concepts and research. Both the examples are made using `Tecnomatix Plant Simulation`.

The model which gets integrated into both project is the *Two Speed Transmission* which represent an electric mechanical engine with two speeds gearbox. It was first mapped to `AutomationML` using the `SimulinkRoleClassLib`, then parsed with `amlparser` and the resulting script was run inside `MATLAB`, thus generating an `FMU`. This `FMU` is then hosted by running `FMU2OPCUA`.

2.5 Simple producer-consumer

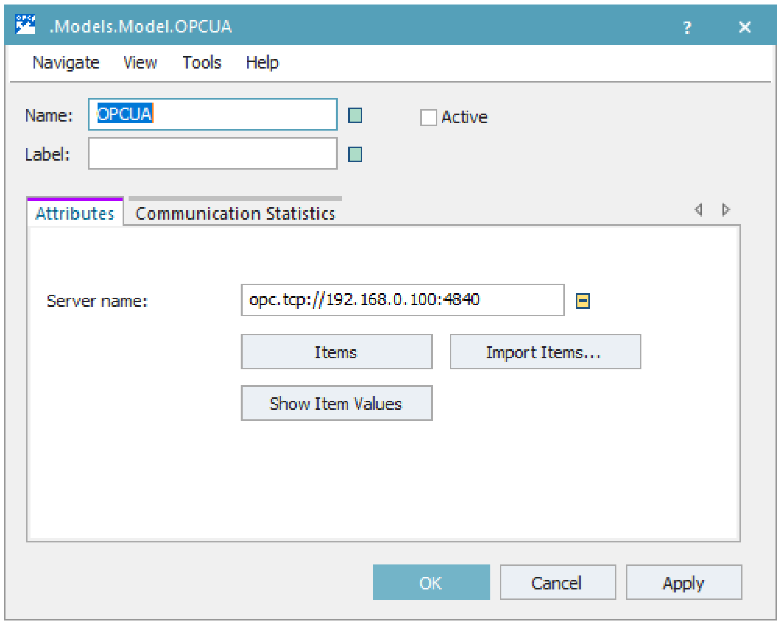
The simple model (2.19) is composed of a *Source* which produces *Mobile Units* (MUs) in a certain amount, a *Station* which represents a generic processing step and a *Drain* which consumes the processed unit.



TorqueOutput=0
TorqueDrive=0
SpeedOutput=-0
SpeedDrive=95.4929658551372
Simulating=true
SimulationTime=5972

Fig. 2.19: The simple model scheme.

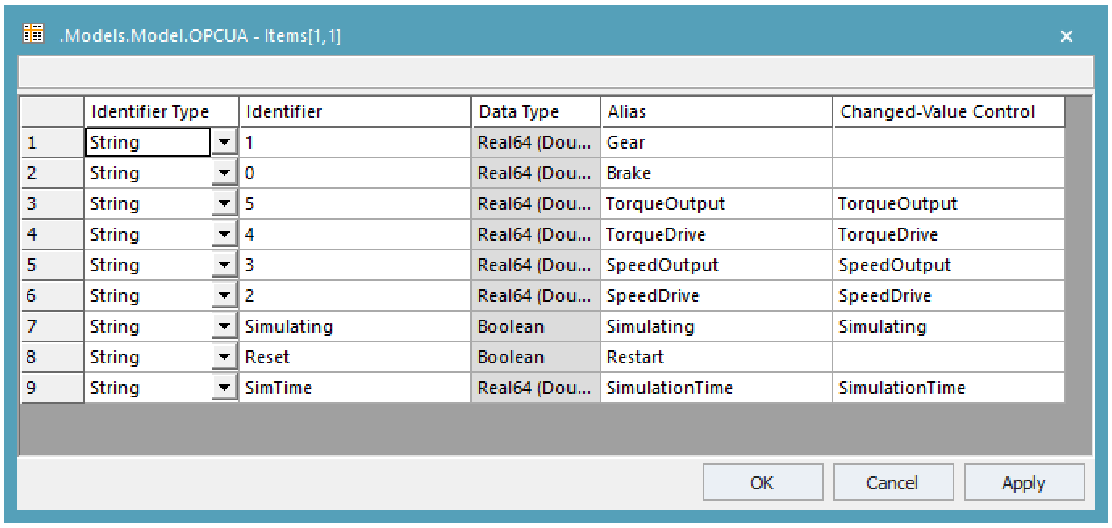
Plant Simulation offers an integrated OPC UA client (the one on the top right of 2.19) which is fairly simple to use. Once added to the model, its dialog window asks for the IP address and the port (2.20). Global model variables can be directly connected to the ones exposed by the server through the `Items` dialog as shown below in 2.21. By doing this way, whenever the server variables' values change, the Plant Simulation variables written in the far-right column gets updated with the same value automatically.



Name: OPCUA
Label:
Active [ ]

Server name: opc.tcp://192.168.0.100:4840

Fig. 2.20: OPC UA client main dialog.



	Identifier Type	Identifier	Data Type	Alias	Changed-Value Control
1	String	1	Real64 (Dou...	Gear	
2	String	0	Real64 (Dou...	Brake	
3	String	5	Real64 (Dou...	TorqueOutput	TorqueOutput
4	String	4	Real64 (Dou...	TorqueDrive	TorqueDrive
5	String	3	Real64 (Dou...	SpeedOutput	SpeedOutput
6	String	2	Real64 (Dou...	SpeedDrive	SpeedDrive
7	String	Simulating	Boolean	Simulating	Simulating
8	String	Reset	Boolean	Restart	
9	String	SimTime	Real64 (Dou...	SimulationTime	SimulationTime

Fig. 2.21: Model variable - Server variable connections.

On the *Station* block in the middle, a method is added on the exit control section, which means that whenever an MU exits the machine the method is going to get called. As the `FMU2OPCUA` program supports different simulation modes, one can either choose to produce a predefined input signals set based on time and use the `SimulationTime` exposed variable to change signal values accordingly, or just rely upon the default stop experiment time and just set values in that restricted time slice. There's also another way which was the chosen one, which is the manual mode. In this mode the simulation gets controlled by the client through the `DoStep` port.

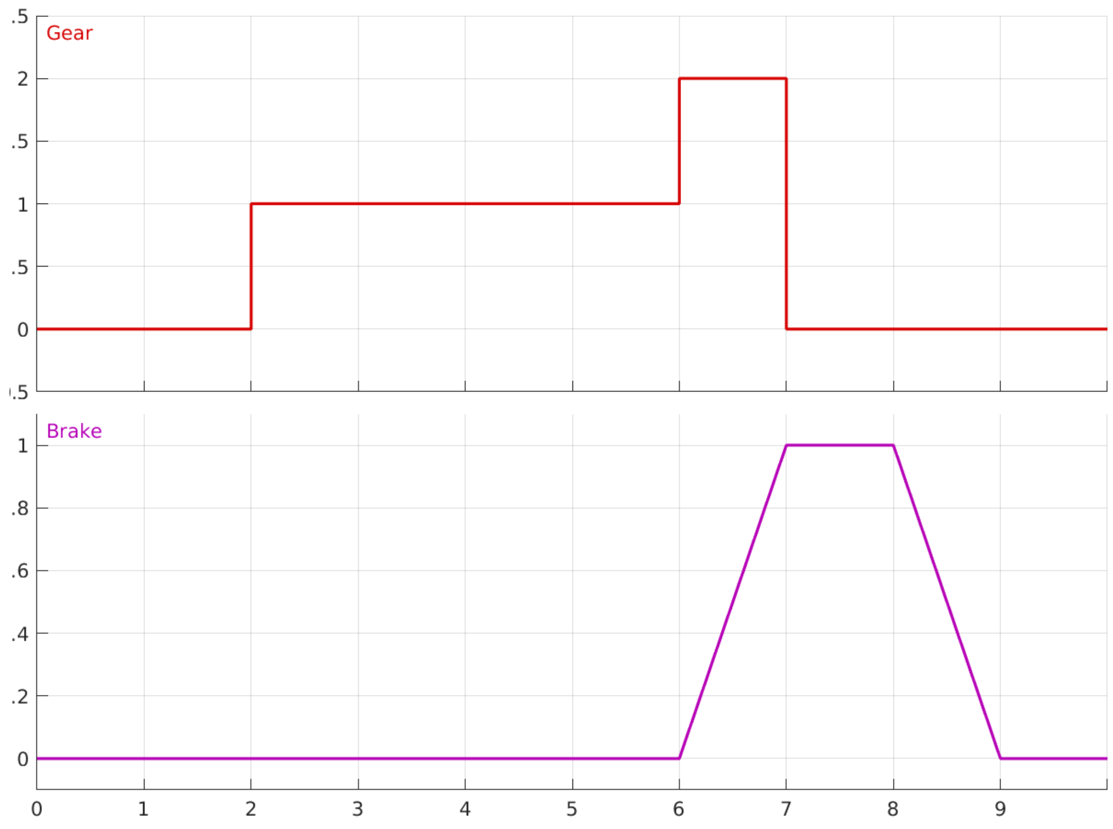


Fig. 2.22: The Brake and the Gear signals for the Station.

In this case, the response is quiet, as the maximum gear value reached is 1 and the brake gets up till 0.5, so when the simulation time is 10 the `SpeedOutput` will already be 0. The result is that for every MU produced by the source, 10 seconds will pass when entering the station, regardless of the simulation speed.

2.6 The ICE lab model

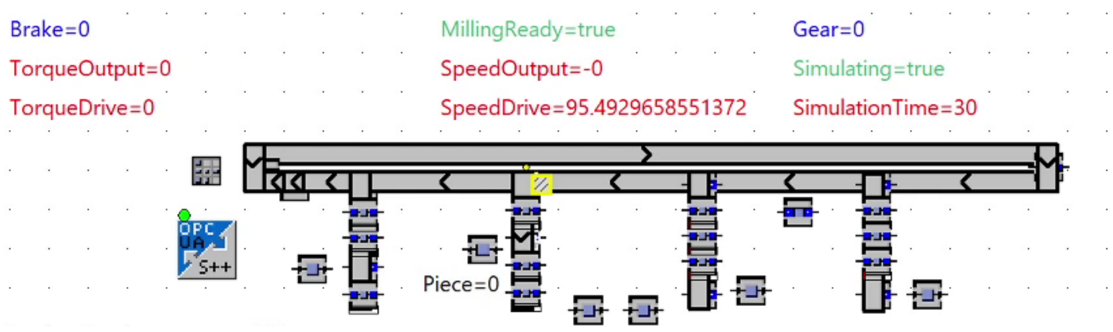


Fig. 2.23: The ICE lab Digital Twin.

The ICE lab model is quite complicated as shown in 2.23. It's composed of many parts:

- A vertical storage
- The conveyor belt system and a mini pallet
- An electronic control panel
- A milling machine
- Two 3D printers
- An assembly station
- A robotic vision system

This digital twin simulates the following process:

1. The MUs stored in the vertical storage are taken on the mini pallet
2. When the mini pallet reaches the stations, it gets processed
3. The milling station is where the `FMU` has been inserted

The *Two Speed Transmission* model can be used to mimic the processing time of the milling machine. Different "recipes" can be used, each one representing a different signal set, selectable from a menu on the top right corner. This is useful for simulating the different effects that different input signal sets can have on the processing times.

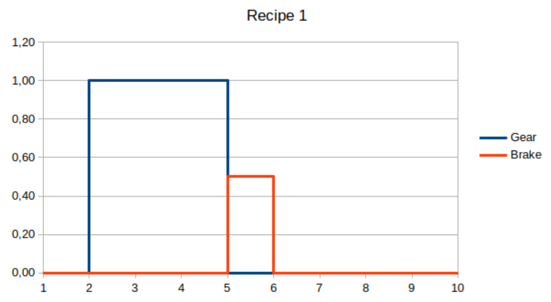


Fig. 2.24: Recipe 1.

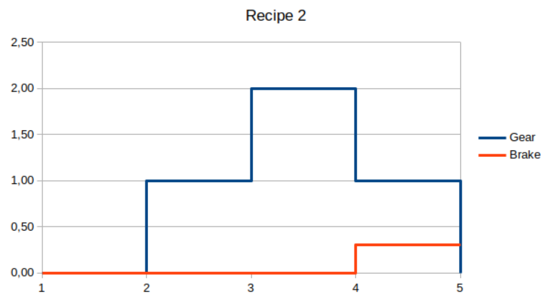


Fig. 2.25: Recipe 2.

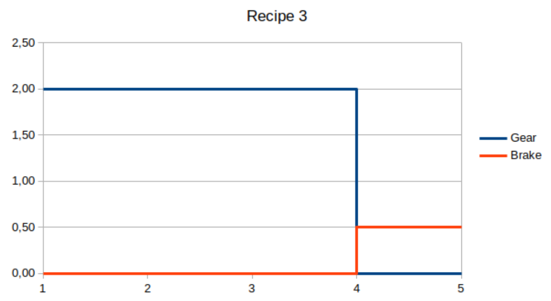


Fig. 2.26: Recipe 3.

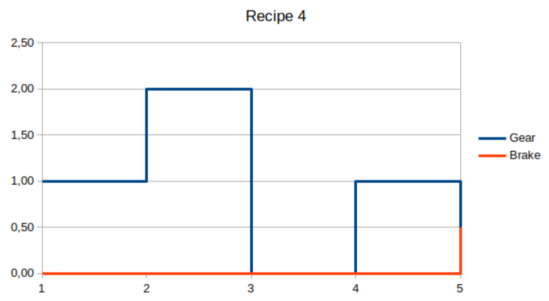


Fig. 2.27: Recipe 4.

Fig. 2.28: The four recipes input signals

Four recipes were made, represented in 2.28. Figure 2.33 reports the different output obtained with the recipes.

It can be observed that when the gear reaches value 2, the speed output goes up to $250-300$, while in the first recipe it doesn't even go beyond 30. In the first recipe the gear has a maximum value of 1 and it lasts 2 seconds, while the brake is also more powerful reaching 0.50. In the other recipes the relation between the speed and the brake intensity and duration can be easily observable. The As the brake gets 0 after the 5th second of simulation, the speed slowly goes down to 0 because of the inertia.

Note that the timing reported in the graphs are the ones of Plant Simulation, which is running at $47\times$ the real speed, while the `FMU` is running at real time, so for example in the second recipe the actual time that the motor took for reaching a speed value of 0 is 30 seconds.

2.7 Conclusions

This work proposed a flow to model generic models in a vendor neutral language as `AutomationML` and showed a semi-automatic flow of model generation and simulation.

This work showed the possibility to map a `Simulink` model into `AutomationML` without the use of ontologies [9].

Then, it showed the integration of `OPC UA` protocol that enable the interconnection with real equipment and synthetic equipment.

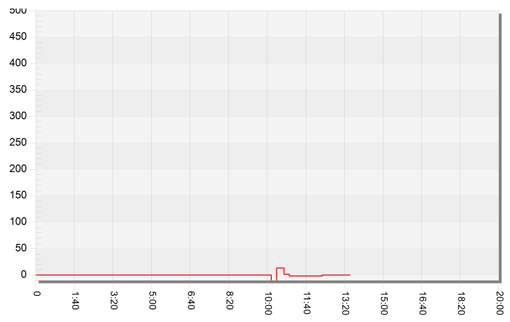


Fig. 2.29: Recipe 1 Output

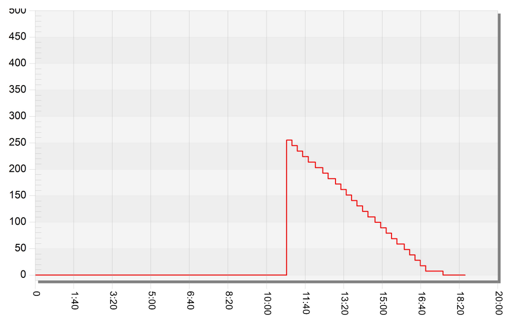


Fig. 2.30: Recipe 2 Output

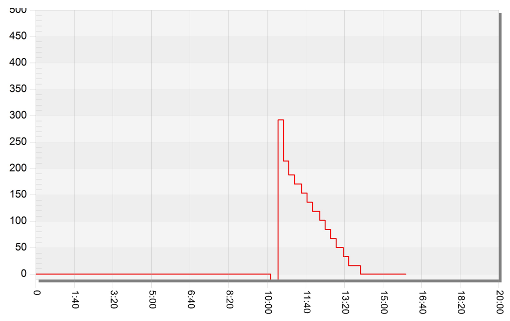


Fig. 2.31: Recipe 3 Output.

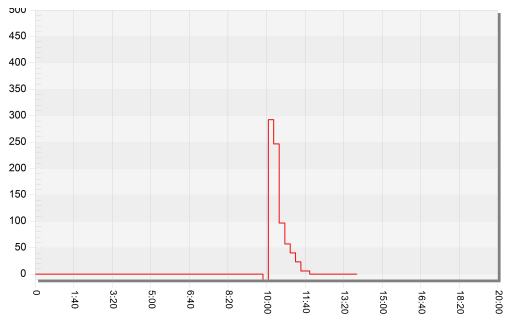


Fig. 2.32: Recipe 4 Output.

Fig. 2.33: Output of the four recipes.

There are many possible areas of improvement in this flow, and also some ideas that could be useful in some scenarios. Improvements that can be applied to the prototypes for making them usable software products are:

- Map in the `SimulinkRoleClassLib` the entire `Simulink` library
- Further expand the `AML` lib with `Simscape` library
- Add correspondent mapping behavior in the `JSON` files
- Implement support for subsystems during the `MATLAB` model generation
- Making `AML` parser better in supporting `SystemUnits` and multiple `RoleClasses`

In addition to these improvements, there are also some ideas based on this work that could be worth exploring, such as:

- Explore the possibility to develop a `MATLAB` plugin that exports a `Simulink` model in `AutomationML`
- After having implemented support for subsystems, implement support to combine different models in `AML`, expressing a full production chain
- Mapping other modeling languages such as `Modelica`

Part III

Heterogeneous Models

3

Automatic Integration of HDL IPs in Simulink using FMI and S-Function Interfaces

3.1 Introduction

Model-based design is nowadays one of the most used approach to tackle heterogeneity and complexity of modern systems [17]. High-level models are step-by-step refined to reach the final system implementation. Over the years, Simulink by Mathworks [18] became the standard “de-facto” in Model-based systems engineering. It provides a nice graphical environment that allows designers to easily model physical systems and their controllers. It provides many different libraries (*i.e.*, toolboxes) full of models and functionalities useful to build and analyze simulations. These features lead it to became the favorite tool of many control and system engineers.

However, Simulink does not provide mechanisms that allows to simulate computational systems. For instance, it does not natively allow to simulate the exact behavior of a SW running on top of an actual HW platform. A task that can be necessary to accurately evaluate timing of HW/SW components controlling cyber-Physical systems [19]. Thus, to perform such kind of analysis in Simulink it will be necessary to exploit HW-in-the-loop and co-simulation techniques. These require specific expertise and are extremely error prone and time consuming: as such, they may negatively impact the time-to-market. This limitation must be overcome as cyber-physical systems and smart devices are everyday more used to control physical processes. In this work we propose a methodology to automatically generate Simulink-compliant blocks from HW Description Language (HDL) models.

The methodology starts from either a Verilog or a VHDL Register Transfer Level (RTL) model. The HDL model is automatically abstracted into an equivalent cycle-accurate C++ model by a state-of-the-art abstraction methodology [20]. We extend this code-generation step to map the abstracted models into two interfacing technologies supported by Simulink: the Functional Mock-up Interface (FMI) [21] and the proprietary C MEX S-Functions.

The models of HW devices generated by the presented methodology can be easily imported within Simulink. Thus, they provide a simpler and more efficient alternative to co-simulation and HW-in-the-loop techniques. To show the advantages of the approach we compared the performance in terms of simulation speed on a set of HDL benchmarks. First, we integrated them within Simulink by co-simulating them using a commercial HDL simulator. Then we integrated the benchmarks by applying the proposed methodology. The experimental results showed up to one order of magnitude speed-up with respect to state-of-the-art co-simulation environments, while preserving accuracy.

Section 3.2 presents some literature about heterogeneous systems simulation, the necessary background and will introduce a running example used throughout the paper. Section 3.3 will present the methodology and its application to the running example. Section 3.4 reports the experimental evaluation of the methodology. After discussing our results, in Section 3.5 we draw some conclusions and give an overview about our ongoing and future research directions.

3.2 Related Works

Model-Based Systems Engineering [22, 23] requires a multitude of tools to be integrated at each design step. This is imposed by the amount of heterogeneous domains involved in modern systems [24]. Many design steps (*e.g.*, validation or performance estimation, *etc.*) require holistic system simulation, usually achieved through co-simulation [25]. Multiple domain-specific simulators are connected to each other; one of the simulator takes care of coordinating and synchronizing all the involved simulators to achieve the complete system emulation. At the state-of-the-practice, Mathworks Simulink [18] is the standard de-facto system simulation tool. For this reason, many attempt to extend its capability to specific domains have been carried out. It has been connected to network simulators [26,27], digital HW simulators [28], instruction-set simulators [29] and many other different kinds of simulators. Furthermore, it has been coupled also with other specific multi-physics simulators. For instance, Haoping et al. used Synopsys Saber [30], Wang et al. proposed an approach using Adams [31] or While in [32] it has been presented a scenario mixing PSpice with Simulink. Another work shows the benefits of coupling Simulink with a Manufacturing Simulator, with the objective of obtaining more accurate estimation about the production quality of a manufacturing system [33]. Other approaches propose solutions to couple complex computational systems with Simulink in order to model and verify cyber-physical systems. In [34] Kawahara et al. connected SysML and Simulink to test and verify the correctness of an embedded system. In [35] Tudoret et al. uses the SIGNAL programming language to model real-time constraints of a software controlling a a physical scenario designed using Simulink. Kung et al. coupled an HDL commercial simulator with Simulink for early validation of HW constraints [36]. All the approaches mentioned above use co-simulation techniques that have been proven to be computational demanding, while its setup may be an error-prone and time consuming processes [24]. As such, some alternative approaches have been defined: they aim at integrating models by translating and importing them into the target simulation environment [37, 38]. However none of the previous works provides neither abstraction nor automation. These features are focal in the approaches presented in [39, 40]: the heterogeneous models of the system components are translated into a homogeneous holistic representation of the cyber-physical system to simulate. However, these approaches requires that the designer can access the original source code of each single part of the system: a rare eventuality in real design flows. The methodology presented in this chapter aims at exploiting tools integration, while providing automation. It automatically integrates cycle-accurate models of digital HW components within Mathworks Simulink through automatic abstraction and translation of the original HW IP cores, and then automatically enriching the generated code to interface it with the target simulator.

3.2.1 Running example

For the sake of clarity, in the following of the chapter we pair the presentation with a running example that represents a IP core HDL description. Its code is depicted in Listing 3.1. It is a Verilog model of a HW module counting the number of positive bits in a 64-bit integer given as input to the module. It carries on such a task by employing a synchronous process performing combinational operations and an asynchronous process controlling the counting algorithm. It has been written to provide a minimal while complete example for the proposed methodology.

3.2.2 FMI-Standard

The basic blocks of a FMI-based simulation environment are called *Functional Mock-up Units (FMUs)*. Multiple FMUs can be imported within simulation environments such as Simulink that takes care of managing FMUs execution. Each FMU may implement the *Model Exchange* or the *Co-simulation* version of the FMI standard. A Model Exchange FMU requires an external solver to simulate. A Co-simulation FMU must provide the solver within its functionalities. This work focuses only on the Co-simulation version of the standard, being more suitable to model discrete behavior.

A co-simulation FMU is composed by an XML file and a dynamic library implementing its functionality. The XML file specifies all the variables that are exposed to the simulation environment by the FMU [21]. For each variable, the XML file must specify its name, causality (*e.g.*, input, output, parameter, *etc.*), its type and a value reference. The supported variable types are 32 bit integer, real, string and boolean. The value reference of a variable is required to be unique for all variables of each type: each variable will be uniquely identified by its type and value reference pair.

The dynamic library must be generated using C-like linking. It must implement the functionality through a set of functions defined by the standard:

- The `fmi2SetupExperiment` function is usually used to initialize the internal variables of the FMU
- The `fmi2Set` function sets the value of an internal variable of the FMU *i.e.*, assigns a value to an input.
- The `fmi2Get` function gets the value of an internal variable of the FMU *i.e.*, returns the value of an output.
- The `fmi2DoStep` advances the simulation time of the component executing the behavior defined by the model.

Listing 3.1: Original Verilog code of the running example.

```
module bit_counter(clk,reset,number,nready,result,rready);

input clk, reset, nready;
input [63:0] number;
output reg rready;
output reg [4:0] result;

integer state, next_state, index;
localparam state_reset=0, state_counting=1, state_output=2;

always @ (posedge clk or negedge reset) begin
    if( reset == 1'b0 )
        state <= state_reset;
    else begin
        state <= next_state;
        case( next_state )
        state_reset: begin
            index <= 0;
            result <= 5'b00000;
            rready <= 1'b0;
        end
        state_counting: begin
            if( number[index] == 1'b1 )
                result = result + 1;
            index = index + 1;
            rready <= 1'b0;
        end
        state_output:
            rready <= 1'b1;
        endcase
    end
end

always @ ( state or nready or index or number ) begin
    case( state )
    state_reset:
        if( nready == 1'b1 ) next_state <= state_counting;
        else next_state <= state_reset;
    state_counting:
        if( index > 63 ) next_state <= state_output;
        else next_state <= state_counting;
    state_output:
        next_state <= state_output;
    endcase
end
endmodule
```

While the standard defines the signature of all the functions to implement, it does not define the sequence in which these functions should be called. It rather defines only some limitations on the possible combinations.

In the last months the FMI Steering Committee announced a new version of the standard (*i.e.*, version 3.0) introducing a new interface called *Hybrid Co-Simulation*, and inspired by the some recent research [41]. It should introduce the possibility to easily handle Discrete-Event Systems. As of today, this new interface is in a pre-alpha status and some of the currently proposed features might end up to be not approved in the final version of the standard update.

3.2.3 Simulink C MEX S-Functions

C MEX S-Functions are the main mechanism provided by Mathworks Simulink to import custom C/C++ code. They provide similar concepts with respect to the FMI standard. However, they do not impose the signature of the functions to implement. S-Functions require to fill a configuration file specifying, among other parameters, the signature of some functions (*i.e.*, callback methods) that will be used by the simulators to execute the functionality. In this work we will use two of these callback methods, that are:

- the *Initialization function* performs initialization actions at the simulation startup. Its name is specified in the configuration file through the `mdlStart` method.
- The *Outputs function* is a C function that takes as parameter a set of input values and a set of references to output variables. It defines the functionality the S-Function must implement at each simulation time. It is specified through the `mdlOutputs` callback method.

The simulator executes the initialization function when the model is instantiated. Then, it executes its output function at each simulation step. Thus, the order in which the input-reading, execution, and output-writing operations are performed is managed internally to the output function by the programmer.

3.2.4 Automatic abstraction of HDL IPs

Efficient HDL simulation has been achieved in the recent years by applying automatic abstraction [42] and code generation [43, 44]. HDL models are translated into functionally equivalent, cycle-accurate C/C++ models.

Since this work aims at generating cycle-accurate models based on the FMI Standard or Mathworks' S-Functions, state-of-the-art abstraction and code generation techniques are suitable to be reused. This work relies on the technique described in [20] to generate the C++ models that will be later wrapped within FMUs or S-Functions. The approach in [20] works as follow:

- a front-end phase parses and analyzes the input HDL model. It extracts the digital processes described in the model and all the dependencies between processes (*e.g.*, sensitivity lists, signal writing and reading, *etc.*). Dependencies analysis is required in order to manage descriptions involving both synchronous and asynchronous processes.
- The analysis produces a dependencies graph: it is used to generate a process scheduling that allows to reproduce the cycle-accurate behavior of the model. The scheduling generation starts from synchronous processes and then proceeds considering the dependencies

of already scheduled processes. Furthermore, the scheduling generation procedure resolves eventual circular dependencies between processes: resolution feasibility is guaranteed by synthesizability of the considered HDL models. As such, the RTL protocol is abstracted into a Transaction-level protocol [42] and the abstracted model is externally synchronous while reproducing asynchronicity internally.

- The model is translated into C++. Each process is translated into an equivalent C++ implementation. A mechanism based on replicated variables, flags and supporting functions is automatically generated to emulate processes concurrency. Each transaction is executed by the execution of a function (*i.e.*, `simulate` function) to which is passed a pointer to a payload data structure (*i.e.*, `model_iostruct`). This structure contains a field for each input or output port of the original model (except for the clock signal, being it abstracted away). As such, each simulation cycle starts by populating the input/output data structure, then the simulation function is called and at its completion the data structure is read.

Listing 3.2 sketches the code generated by applying [20] to the guiding example in Listing 3.1. The `simulate` function (Lines 1-10) performs an input phase to read the values in the `io_exchange` structure. Then, execution of synchronous processes is managed by `synch_elaboration` function (Lines 12-20). Concurrency is reproduced by combining the `flag_elaboration` (Lines 22-29) and `update_event_queue` (Lines 31-53) functions that manages variables replication and the supporting flags.

We extend the last step of the approach (*i.e.*, automatic C++ code generation) to embed generated models within FMU and S-Functions.

3.3 Methodology

We present two different code-generation alternative to integrate cycle-accurate descriptions within Mathworks Simulink models. Both alternatives rely on C++ models generated when applying the automatic abstraction of HDL IP cores described above. The C++ models are automatically customized to be compliant with one of the two interfacing technology supported by Simulink: FMI Standard and C MEX S-Functions. In the former case, the cycle-accurate model can be imported within Simulink as a FMU using the native FMI interface introduced in the latest versions of the simulator. In the latter case, the model becomes a native Simulink block that can be easily integrated within any model. Figure 3.1 gives an overview of the abstraction, code generation and interfacing flow presented in this paper.

When using HDLs for simulation, constructs are meant to describe HW simulation events. These primitives may represent internal events of the device, or interface-level events of the IP. The latters are those events that are visible to whatever interacts with the IP. They may be the model initialization, input reading or output writing operations and, in the case of cycle-accurate models, the execution of a simulation cycle with the consequent temporal progress of the model. We summarizes in Table 3.1 the mapping defined between the HDL constructs necessary to interact with an IP, and the primitives defined by the two target interfaces.

Listing 3.2: Automatically generated RTL processes scheduler.

```
void bit_counter::simulate(bit_counter_iostruct * io_exchange)
{
    input_phase(io_exchange);
    synch_elaboration();
    while (process_in_queue) {
        flag_elaboration();
        update_event_queue();
    }
    output_phase(io_exchange);
}

void bit_counter::synch_elaboration()
{
    process();
    flag_elaboration();
    flag_neg_reset = false;
    flag_number = false;
    flag_nready = false;
    update_event_queue();
}

void bit_counter::flag_elaboration()
{
    if (flag_neg_reset) process();
    if (   flag_state || flag_nready ||
           flag_index || flag_number )
        process_0();
    if (flag_result_out_sig) result_update_process();
}

void bit_counter::update_event_queue()
{
    process_in_queue = false;
    if ( index != index_new)
    {
        index = index_new;
        flag_index = true;
        process_in_queue = true;
    } else flag_index = false;
    next_state = next_state_new;
    if ( result_out_sig != result_out_sig_new)
    {
        result_out_sig = result_out_sig_new;
        flag_result_out_sig = true;
        process_in_queue = true;
    } else flag_result_out_sig = false;
    if ( state != state_new)
    {
        state = state_new;
        flag_state = true;
        process_in_queue = true;
    } else flag_state = false;
}
```

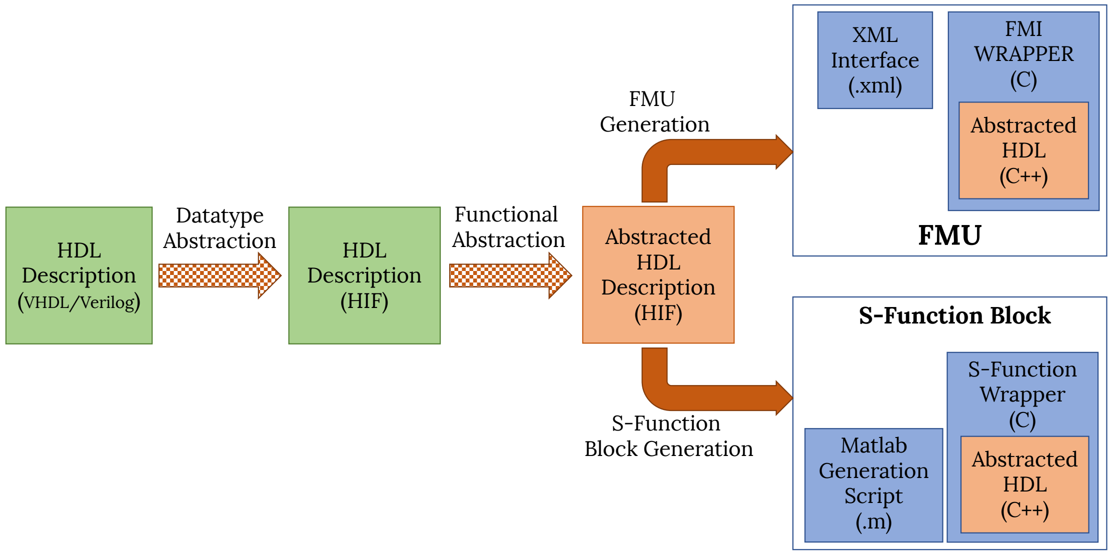


Fig. 3.1: Main steps of the proposed methodology. The boxes represent the different files involved and generated during the different steps of the methodology. Green boxes identifies models at the Register Transfer Level of Abstraction; orange boxes represents abstracted models; blue boxes represents interface-specific files. The different transformation steps of the methodology are represented by the arrows and their corresponding labels. Checked arrows represent steps reused from the methodology in [20], solid arrows represents novel steps. For automation reasons, the flow is built on top of HIFSuite [44], and the different manipulations are performed on the Heterogeneous Intermediate Format (HIF) provided by the suite.

Table 3.1: Mapping of HDL events onto FMI and S-Functions primitives.

HDL simulation Events	**FMI Standard** Primitives	**S-Functions** Primitives
Initialization	Sequence of assignments in initialization mode	Initialization function defined by the `mdlStart` callback method
Simulation Cycle Execution	`fmi2DoStep`	Output function defined by the `mdlOutputs` callback method
Input Signals Reading	`fmi2SetInteger` and `fmi2SetBoolean`	Parameters passed to Compute Output function by value
Output Signals Writing	`fmi2GetInteger` and `fmi2GetBoolean`	Parameters passed to Compute Output function by reference

- During *model initialization*, a HDL simulator instantiates all the necessary data-structures required to carry on the simulation and it assigns initial values to signals defined by the model. Two functions can be used to reproduce it through APIs defined by the FMI Standard: `fmi2Instantiate` allocates any internal data-structure that may be required by the model to simulate. The `fmi2SetupExperiment` function is used to set variables initial values. The same is reproduced by S-Functions using the `mdlStart` callback method. The function specified will perform a set of assignments setting variables to their default values.

Table 3.2: Mapping of HDL data-types to FMI and Simulink.

HDL data-types	C/C++	FMI Standard data-types	Simulink data-types
`Boolean`, `Bit`, `Logic`	`bool`	`Boolean`	`int8`
`Unsigned`, `Bit Vector`, `Logic Vector`	`uint64_t`, `uint32_t`, `uint16_t`, or `uint8_t`	`Integer`	`uint32`, `uint16`, or `uint8`

- The *Simulation of a cycle execution* is reproduced by an FMU through its implementation of the `fmi2DoStep` function. C MEX S-Functions reproduces it using the `mdlOutputs` callback method that implements the IP core *output function*.
- HDLs *input signal reading* is reproduced by an FMU thanks to the `fmi2Set` functions. This work relies only on two functions: `fmi2SetInteger` and `fmi2SetBoolean`. S-Functions use parameters of their output functions to get input. More precisely, all the parameters that are passed by value to the output function are intended to be input values.
- HDLs *output signal writing* is reproduced by an FMU through the `fmi2Get` functions. This work uses the `fmi2GetInteger` and `fmi2GetBoolean` functions. S-Functions use parameters of their output functions to return output values: all the parameters that are passed by reference to the output function are intended to store output values after a cycle execution.

We now detail the features of the code generated by the proposed methodology.

3.3.1 Data-type abstraction

Simulink relies on the *continuous dataflow model* of computation, where blocks are connected through typed connections. However, Simulink, and C MEX S-Functions, supports only a limited set of data-types[1] for such connections. HW-specific data-types (*e.g.*, many-valued logic, bit, logic vectors, *etc.*) are not supported. Also integers are limited in bit span and only signed and unsigned `int8_t`, `int16_t`, `int32_t` are supported. Furthermore, a boolean data-type is not provided. On the other hand, the FMI standard provides the Boolean data-type but integers are supported only as 32 bit signed values by the current FMI standard API. It is thus necessary to map types of the IP cores interface onto the supported types.

Table 3.2 summarizes the mapping chosen between HDL-specific, FMI and Simulink data-types. The HDL to C/C++ automatic abstraction provides also data-types abstraction: HW-specific data-types are represented as described in the Table's second column. *Boolean*, *single bit* and *multi-valued logic* values are abstracted to boolean values (*i.e.*, `bool` in C/C++). This is mapped onto FMI's `Boolean` type and onto the `int` Simulink type, as the boolean type not provided in this latter case.

[1] https://www.mathworks.com/help/simulink/ug/data-types-supported-by-simulink.html

Any *unsigned*, *bit vector* or *logic vector* is abstracted into a corresponding C/C++ standard unsigned integer (*i.e.*, `uint64_t`, `uint32_t`, `uint16_t`, and `uint8_t`). The standard unsigned integer chosen is the one using the minimum amount of bit capable to represent the original value. If the original type span is greater than 64 bits, then the port is splitted into multiple ports with a span not greater than 64 bit before being abstracted. When generating FMUs, this is mapped onto the `Integer` type, while the C/C++ standard integer is preserved when generating C MEX S-Functions. Both the FMI Standard and Simulink supports at most 32 bits integers: if a port is wider than 32 bits it is splitted into multiple ports.

Consider the running example in Listing 3.1, and the resulting FMU XML definition file (*i.e.*, Listing 3.3) and S-Function (*i.e.*, Listing 3.7). Lines 3 to 6 of Listing 3.1 defines the types of the ports. Single-bit multi-valued logic ports, such as `nready` and `rready` are mapped onto boolean variables to build the corresponding FMU (Listing 3.3, lines 34-38 and 40-44), while they are mapped onto `int` when generating the corresponding S-Function (Listing 3.6[2] line 4 and Listing 3.7 lines 3 and 6). Logic vectors, such as `number` and `result` are mapped onto Integer (Listing 3.3, lines 10 to 26). `number` has a width greater than 32-bit: it has been split into two different 32-bit variables: `number_1` and `number_2` (lines 10-14 and 16-20 in Listing 3.3). Something similar has been done to generate the equivalent S-Function (lines 4-5 in Listing 3.7).

3.3.2 Automatic generation of Functional Mockup Units

In an FMU input and output variables are specified by using an XML description. This step automatically generates such a file:

- the input HDL model is parsed and analyzed to identify its top-level design unit.
- All the input and output ports specified in the top-level unit are analyzed and their types are manipulated according to the mapping described in the previous Section and sketched in Table 3.2.
- The header part of the XML file is printed according to the information gathered during the HDL model analysis. The header part contains the name (*i.e.*, `modelName`) and the identifier (*i.e.*, `guid`), some misc information about the FMU and the co-simulation features it provides.
- For each port it is specified its causzality (*i.e.*, input or output), its description, the port name, its value reference and its variability (*i.e.*, continue or discrete). Then, a XML tag specifies the port type. The tag is chosen according to the mapping described above, thus only the `Integer` or `Boolean` tags are generated by the methodology. The parameter `starts` is inserted in the tag to specify the variable initial value.

Listing 3.3 reports the XML automatically generated from the `bit_counter` running example. The header part is composed by lines 1-6, while the variables corresponding to the original

[2] Names of input and output parameters of the output function are fixed by Simulink. Inputs are named `uN` and outputs are named `yM`, where `N` and `M` are Natural numbers. Output variables must be arrays, being them passed by reference to the output function.

Listing 3.3: `modelDescription.xml` file of the `bit_counter` module.

```
<fmiModelDescription copyright="generated by HIFSuite"
 guid="e820d24d-aeaf-47fa-b9b2-a185950d71a9"
 modelName="bit_counter"
 fmiVersion="2.0">

 <CoSimulation/>

 <ModelVariables>

  <ScalarVariable causality="input" description="int"
                  name="number_1" valueReference="0"
                  variability="discrete">
       <Integer max="18446744073709551615" min="0" start="0"/>
  </ScalarVariable>

  <ScalarVariable causality="input" description="int"
                  name="number_2" valueReference="1"
                  variability="discrete">
       <Integer max="18446744073709551615" min="0" start="0"/>
  </ScalarVariable>

  <ScalarVariable causality="output" description="int"
                  name="result" valueReference="2"
                  variability="discrete" initial="approx" >
       <Integer max="255" min="0" start="0"/>
  </ScalarVariable>

  <ScalarVariable causality="input" description="bool"
                  name="reset" valueReference="0"
                  variability="discrete">
       <Boolean start="false"/>
  </ScalarVariable>

  <ScalarVariable causality="input" description="bool"
                  name="nready" valueReference="1"
                  variability="discrete">
       <Boolean start="false"/>
  </ScalarVariable>

  <ScalarVariable causality="output" description="bool"
                  name="rready" valueReference="2"
                  variability="discrete" initial="approx">
       <Boolean start="false"/>
  </ScalarVariable>

 </ModelVariables>
 <ModelStructure/>
</fmiModelDescription>
```

Verilog ports are specified from line 8 to 46. Variables `number_1` and `number_2` are the result of the manipulations applied to the `number` port to make it compliant with the 32-bit limitation imposed by the FMI Standard. It is worth noticing that the clock variable is not present: this is due to the RTL to cycle-accurate abstraction applied to obtain the starting C++ code.

Each variable is uniquely identified by the pair composed by its type (*i.e.*, Boolean or Integer) and its value reference. For this reason, different variables may have the same value reference if they belong to different types. It is the case for the `number_1` and `reset` variables in lines 10-14 and 28-32. While this intuitively may lead to ambiguity, it is properly managed by the input and output C functions defined by the FMI standard and implemented by the C++ model implementation.

Other than the XML file, the FMU must provide the implementation of the C API, defined by the standard, in the form of a dynamic library. The methodology continues by manipulating the C++ code generated during the automatic model abstraction summarized in Section 3.3.1. A sketch of the starting C++ code was reported in Listing 3.2. The functions provided by the scheduler embedded after abstraction must be “wrapped” within the FMI Standard API. Listing 3.4 reports a sketch of the C/C++ code generated for our running example:

- A constant is defined with the value of the FMU’s GUID (*i.e.*, `MODEL_GUID`). Any operation using the GUID of the model uses this constant (line 3).
- A C structure called `ModelInstance` is declared and used as a container for all the information necessary to store during the model execution. It contains a pointer to an instance of the model implementation, a pointer to the input/output data-structure used by the abstracted model to communicate, some other information such as the local time and number of executed cycles (line 5 to 12).
- The `fmi2DoStep` function is implemented (lines 14-17): it takes care of simulating one execution cycle of the model and update the FMU internal time (lines 19-26).
- The `fmi2SetInteger`, `fmi2GetInteger`, `fmi2SetBoolean` and `fmi2GetBoolean` are implemented to manage the input and output phases (lines 29-43). Listing 3.5 exemplifies the implementation of the `fmi2SetInteger` function for the running example. `vr` is the array of size `nvr` containing the value references of the integer variables to set. The `value` array contains `nvr` integers that are the values to be set. The `for` loop (lines 10-21) takes care of setting the correct values to the specified variables. The structure of the other input and output functions recall the one presented.

The generated C++ code must be compiled to produce a shared library. Note that, it is possible to compile different libraries supporting many different architectures. The library Application Binary Interface (ABI) must be compatible with the C API. As such, it can be compiled by using any C++ compiler, however its linking must be compatible to the C linking and it must not perform names mangling.

Finally, the shared libraries and the XML files can be compressed and packet within an `.fmu` file. Such a file can be imported by any simulator supporting the FMI Co-Simulation 2.0 Standard, such as Simulink using the new native interface developed by Mathworks or any other FMI-compliant toolbox.

Listing 3.4: Skeleton of the FMI implementation of the `bit_counter` module.

```
#include <fmi2Functions.h>
#include "inc/bit_counter.hh"
 #define MODEL_GUID "352e3781-f5a3-4914-abd7-687397bff7fe"
...
typedef struct ModelInstance{
    bit_counter * model;
    bit_counter::bit_counter_iostruct * iostruct;
    char * instanceName;
    int32_t cycle_number;
    fmi2Real time;
    ...
} ModelInstance;
...
fmi2Status fmi2DoStep( fmi2Component c,
    fmi2Real currentCommunicationPoint,
    fmi2Real communicationStepSize,
    fmi2Boolean noSetFMUStatePriorToCurrentPoint )
{
    bit_counter::bit_counter_iostruct * iostruct;

    ModelInstance * comp = ( ModelInstance *) c;
    bit_counter * model = comp->model;
    iostruct = comp->iostruct;
    model->simulate( iostruct, comp->cycle_number );
    comp->time = comp->time + communicationStepSize;
    return fmi2OK;
}
...
fmi2Status fmi2GetInteger( fmi2Component c,
    fmi2ValueReference * vr,
    size_t nvr,
    fmi2Integer * value )
{
    ... // Implementation of read port operations.
}
...
fmi2Status fmi2SetInteger( fmi2Component c,
    fmi2ValueReference * vr,
    size_t nvr,
    fmi2Integer * value )
{
    ... // Implementation of write port operations.
}
```

3.3.3 Automatic generation of C MEX S-Functions.

While the just discussed FMI-based generation alternative is portable to multiple tools, the second code generation alternative relies on proprietary C MEX S-Functions. S-Functions allows to specify custom Simulink blocks expressing their functionalities as C/C++ functions. Being a native Simulink technologies S-Functions generation is simpler, but not portable. As for FMUs, S-Functions generation starts from the C/C++ models generated after applying the HDL to

Listing 3.5: Skeleton of the `fmi2SetInteger` function implementation for the `bit_counter` module.

```
fmi2Status fmi2SetInteger(
 fmi2Component c, fmi2ValueReference * vr,
 size_t nvr, fmi2Integer * value ) {
    bit_counter::bit_counter_iostruct * iostruct;

    ModelInstance * comp = ( ModelInstance *) c;
    iostruct = comp->iostruct;
    size_t i = 0L;
    .... // Check for errors...
    for (i = 0L; i < nvr; i = i + 1L) {
        switch (( int32_t) (*(i + vr))) {
            case ((int32_t)0L):
                iostruct->number = *(i + value);
                break;
            case ((int32_t)1L):
                iostruct->result = *(i + value);
                break;
            default:
                break;
        };
    }
    return fmi2OK;
}
```

Listing 3.6: Matlab generation script for the running example.

```
def = legacy_code('initialize');
def.SFunctionName = 'bit_counter_mex_system';
def.StartFcnSpec  = 'createbit_counter()';
def.OutputFcnSpec = 'void bit_counter_Output(
 int8 u1, int8 u2, uint32 u3, uint32 u4,
 int8 y1[1], uint8 y2[1])';
def.TerminateFcnSpec = 'delete_bit_counter()';
def.HeaderFiles   = {'bit_counter.hh'};
def.SourceFiles   = {'bit_counter.cc'};
def.IncPaths      = {'inc'};
def.SrcPaths      = {'src'};
...
def.SampleTime = [10*10^-6 0];
def.Options.language = 'C++';
...
```

C/C++ automatic abstraction. Then, a MATLAB `.m` file must be generated to specify which functions will implement the required callback methods. The MATLAB function described in the file generates the Simulink block. While the entire specification of such a configuration file is available in the Simulink documentation, Listing 3.6 reports the main part of the MATLAB file generated by the proposed methodology:

- the *name of the block* that will implement the model is specified by the `SFunctionName` attribute.

Listing 3.7: Output function for the C MEX S-Function implementing the `bit_counter` model.

```
...
void bit_counter_output(  int8_t reset,
         int8_t nready,
         uint32_t number_1,
         uint32_t number_2,
         int8_t * rready,
         uint8_t * result)

{

    input_phase( reset, nready, number_1, number_2 );
    synch_elaboration();

    while (process_in_queue) {
        flag_elaboration();
        update_event_queue();
    }

    output_phase( rready, result);
}
...
```

- The C/C++ initialization function responsible of allocating resources and initialize the data-structure of the module is specified by the callback method `StartFcnSpec`. This function is also responsible of initializing the Simulink block once instantiated.
- The concrete implementation (*i.e.* output function) of the system is specified by the `OutputFncSpec` callback method. The function must take care also of managing input and output of the block. For this reason, input and output variables are specified in its signature as parameters. The input variables are passed by value, while output variables are passed through reference. The input variable names have the prefix `u`, while output variable names are prefixed `y`. The data-types of the variables are assigned according to the mapping discussed above.
- Some other parameters must be generated to specify the location of the source and header files, the input language and (optionally) the methodology provide a sample time to the block. If it is not specified, it must be specified manually by the user before generating the block from MATLAB.

The functions specified in the MATLAB file are implemented by manipulating the abstracted model. In particular, the initialization function is automatically generated to perform all the assignments necessary to initialize the block. Then, the output function generated "wraps" the main simulation function generated by the abstraction procedure, as well as the input and output phase. Listing 3.7 shows the implementation of the output function generated for the `bit_counter` example. The `reset`, `nready`, `number_1` and `number_2` parameters are input variables corresponding to `u1`, `u2`, `u3` and `u4` of line 5 of Listing 3.6. The `rready` and `result` variables are output of the block and corresponds to the `y1` and `y2` parameters of line 6 of

Table 3.3: Characteristics of the benchmarks and time required for the automatic code generation.

Benchmark	LoC	# Ports		# Interface bits		Generation time (s)	
		Input	Output	Input	Output	FMU	S-Func.
DES56	1186	6	8	132	169	110.61	89.46
AES	1854	6	2	260	129	57.59	40.95
CAMELLIA	284	7	2	260	129	14.15	2.94
XTEA	374	6	2	195	64	15.74	5.26
ECC	180	9	2	26	64	21.72	8.31
MLITE CPU	2122	5	5	36	98	28.42	17.66
SMART DEVICE	3498	4	9	35	52	105.17	72.04

Listing 3.6. Variable types are chosen accordingly to the mapping proposed above. Line 11 of Listing 3.7 assign the input value parameters to the internal variables of the system, while lines 14 to 17 are executing a simulation cycle of the model. Finally, line 19 set the output values.

Once both the C++ implementation and the MATLAB file have been generated, the latter can be executed to generate the Simulink block reproducing the cycle-accurate behavior of the initial HDL description. The block can be easily integrated within any Simulink model.

3.4 Experimental Results

The methodology has been implemented on a prototypical tool. It has been developed extending the methodology proposed by [20] and exploiting an academic license of HIFSuite [44]. It has been tested on a subset of the benchmarks used in [20] that we were able to find as open-source IPs on the OpenCores.org portal (*i.e.*, DES56, AES, CAMELLIA, XTEA, MLITE CPU). Furthermore, we tested the methodology on two custom benchmarks: an IP implementing Error Correction Code algorithm (*i.e.*, ECC) and the SMART DEVICE IP. The latter is a more complex system composed by a MOS Technology 6502 micro-controller, a ROM memory, a RAM memory and a bus connecting the CPU to some peripherals. Table 3.3 reports for each benchmark the number of lines of code (LoC) of the original HDL IP, the number of input and output RTL ports and the number of bits in the IP interface. Furthermore, the Table reports, for each benchmark, the code generation time (in seconds) required by the prototypical tool we implemented to automatize the proposed methodology. The table reports the time required to generate both FMUs and S-Functions.

Each benchmark has been integrated within a Simulink model acting as a testbench for the IP. The model relies on a Stateflow diagram to generate input and react to output signals of the IP. Initially they have been integrated using their original HDL description. They have been simulated by building a co-simulation environment involving Simulink and a commercial HDL simulator. Then, benchmarks underwent the proposed methodology to generate both their equivalent FMUs and S-Functions. Last two columns of Table 3.3 show that the tool implementing our methodology is capable of automatically generate FMUs or C MEX S-Functions

Table 3.4: Results obtained on the set of benchmarks.

Benchmark	Co-simulation (seconds)	FMI (seconds)	Speed-up	S-Function (seconds)	Speed-up
DES56	61.40	7.87	7.80x	3.79	16.20x
AES	55.47	10.83	5.12x	6.07	9.13x
CAMELLIA	36.74	4.99	7.36x	3.07	11.97x
XTEA	36.43	4.63	7.87x	2.69	13.54x
ECC	30.22	4.84	6.24x	2.79	10.83x
MLITE CPU	40.79	4.60	8.87x	3.27	12.47x
SMART DEVICE	64.09	14.30	4.48x	7.53	8.51x

from some quite complex HDL IPs in a very small amount of time. In fact, all the benchmarks required less than two minutes for the automatic code-generation step.

We replaced and compared the HDL simulation with both the equivalent FMUs and S-Functions. All the simulations have been performed using Matlab R2018a on a 64-bit machine running Ubuntu 16.04, equipped with 16 GB of memory and an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz.

Table 3.4 reports the obtained simulation results. It shows the time required to emulate the behavior of one second of the real system execution when using co-simulation, the FMI standard and S-Functions to import the benchmarks in the Simulink model. Furthermore, it reports the speed-up achieved using the models automatically generated by applying the methodology proposed in this paper.

The simulation environments obtained by applying our automatic generation technique of Simulink blocks always outperform co-simulation. This is mainly due to two reasons. First, the methodology relies on an automatic abstraction technique that performs many different optimizations. The generated C/C++ code is managed by a highly optimized scheduler that is obtained through a deep static analysis performed on the process dependency graph of the HDL description [20]. Furthermore, slow and inefficient HDL data-types are replaced through abstraction by faster and more efficient C-native data-types. Second, interprocess communication is computationally demanding. During simulation the operating system must perform many different context switch operations each time two different simulators of the co-simulation environment need to synchronize with each other. When co-simulating an HDL description with Simulink it happens at each clock cycle. On the other hand, both FMI standard and C MEX S-Functions relies on internal Simulink data-structure, and as such no interprocess communication is required.

Table 3.4 compares also the two different alternatives proposed by this paper. S-Function implementations are always faster than FMI. This seems reasonable since C MEX S-Functions are custom (and proprietary) interfaces thought to be used specifically within Simunlink, while the FMI Standard is meant to be portable on different tools. As such, it is reasonable to assume that S-Functions are better integrated and optimized for the target simulation environment. Still, FMUs provided good performance and their automatic generation is justified by the frequent need of tool-independent models. Furthermore, as Simulink recently started to support FMUs natively, we can imagine a future improvement on their performance.

Finally, it may be interesting to compare the speed-up achieved by applying our methodology *w.r.t.* the results in [20]. Speed-up values in [20], ranges from 7.7x to an impressive 441.3x while the speed-up values presented in Table 3.4 are lower even though both works rely on the same abstraction methodology. The main reason for this loss of performance improvement is due to the target simulation environment. In [20] system simulation relies on highly optimized and customized virtual platforms, exploiting the advantages of the discrete-event model of computation. As such, it is possible to perform strong optimization by allowing variable length of transactions. This feature allows to perform automatic protocol abstraction, as presented by Bombieri et. al. [42]. Protocol abstraction allows to execute a single transaction for each phase of the communication protocol of the original IP. In fact, [20] reaches its best performance when it can perform also protocol abstraction.

However, protocol abstraction is not yet available to us since this work targets dynamic-system simulation environments, such as Simulink. Such a simulator relies on the synchronous data-flow model of computation, rather than discrete-events. As such, the blocks generated by the presented methodology are constrained to execute periodically with a fixed time-step. Aiming at preserving cycle accuracy *w.r.t.* the model, it is thus necessary to choose a simulation time granularity that allows to reproduce any possible transaction of the communication protocol of the IP. Since HDL IPs are usually reactive at each clock cycle, for instance to manage a reset signal, the fixed simulation time step for the generated Simulink blocks must be equal to the clock period. As a conclusion, it is necessary to sacrifice part of the simulation speed to gain the possibility to perform still efficient cycle-accurate simulation within Mathworks Simulink. However, in the next session we will introduce some ongoing works that aims at improving such limitations.

3.5 Conclusions and Future Outlook

In this chapter we presented the basic steps towards the integration of HDL in Simulink. The main contribution is a fully automatic methodology for the generation of Simulink blocks starting from HDL descriptions. The generated blocks are functionally equivalent and cycle-accurate with respect to the original models. This enables the possibility to “import” timing accurate models of HW/SW devices within Simulink system models: a feature that may contribute in decreasing design time of complex heterogeneous systems by providing an efficient alternative to co-simulation and HW-in-the-loop methodologies.

The methodology has been implemented into an automatic tool and then applied to a set of benchmarks. It shows both effectiveness and efficiency by providing up to 16x speed-up with respect to state-of-the-art co-simulation environments. Thus, it is a good starting point to develop a set of techniques to integrate efficient cyber-physical virtual platform to use in variety of design steps. Ongoing activities with particular interest on the FMI standard, try to cover different aspects of cyber-physical systems simulation starting from this work.

While this work focused only on the generation of a single components, in [45] we focused on the integration of multiple Cyber FMUs within Simulink. During the modeling phase of a Control Platform of a Physical System, a designer models the Hardware Platform reusing existing components and composing them together. In this work we consider a set of HDL IPs exported as FMUs that represent components of a Virtual Platform (*i.e.* CPU, Memory, Bus, *etc.*). This allows the designer to switch between these components directly within Simulink environment, and to evaluate the performances of the components. As such, it enables early design space exploration. In [45] we deal with the problems caused by the co-existence of different Models of Computation: the data-flow used by Simulink and the discrete-event model used by FMUs generated from HW IP cores.

In the same direction, we are trying to extend the work presented in this chapter to allow coarser synchronization mechanisms when using FMI. In [46] we identify some of the limitations of the FMI standard that prevent a more efficient synchronization mechanism. The main issue we identified is the fact that the simulation of an FMU is imposed unidirectionally from the Master Algorithm to the FMU. More in details, the Master Algorithm decides the size of the simulation step of the FMU. On the other hand, a coarser synchronization may be provided by Transactional models that may not allow to know a priori the exact time of the next transaction of the model. Thus, they cannot be managed by a Master Algorithm compliant with the current version of the FMI standard. In practice, an FMU cannot communicate its internal time to the Master Algorithm if it is different by the one imposed by the `fmi2DoStep` invocation. In [46] we enable backward time propagation between the FMU and Master Algorithm in order to capture this information. Thus, we allow to manage Transactional models. The work also presents a novel simulation strategy for the Master Algorithm based on the backward timing propagation. Interestingly, we achieved this goal by acting on the modeling within the rules imposed by the current standard.

Future works will focus on the lacks of the standard 2.0 trying to better optimize the simulation strategies of the Master Algorithm. In particular, we are exploring some specification languages to pair with the FMI standard to express more information about HW platforms. We are exploring the possibility of using UML and/or SysML to define the protocol, and the IP-XACT standard to model the interconnections between the components of a platform that will be simulated as FMUs. With this increased level of information we aim at improving the simulation strategy, thus obtaining highly specialized master algorithm that could better fit each particular scenario.

The integration of HW components within cyber-physical system simulators are applicable to many fields. One of our ongoing activities aims at exploiting the FMI standard to integrate cyber models into commercial production line simulators. This to better estimate the quality deviation of the manufacturing processes [47]. In future, these activities could collapse to model different levels of abstraction of models, with ad-hoc simulation algorithms and integrate them in production line models.

4

Generation of Functional Mockup Units for Transactional Cyber-Physical Virtual Platforms

4.1 Introduction

Cyber-Physical Systems (CPSs) are shaping todays world. They are an enabling technology for many different ongoing technological disruptions, such as smart manufacturing, autonomous driving, *etc.* As such, improving design methodologies for CPSs is crucial to advance a wide set of system engineering sub-disciplines [48].

System design requires models to be simulated providing designers with the feedback necessary to evaluate the quality of their ideas [49]. The heterogeneity of CPSs makes modeling and simulation pretty intricate tasks [50]. To achieve holistic simulation of such heterogeneous systems, designers must either rely on complex co-simulation environment aggregating specialized simulators for the many design domains involved in the system, or to produce a single holistic model of the system [51]. However, the latter solution requires to access, often unavailable, open specifications for every component of the system. On the other hand, co-simulation requires interfacing different simulation tools. Such tools often provides incompatible interfaces, thus requiring time-consuming adapters [52].

In this scenario, the Functional Mock-up Interface (FMI) standard for co-simulation emerged as one of the most promising technologies to interface heterogeneous simulators [21]. It defines an Application Programming Interface (API) that must be implemented by the simulator. As such, FMI is well suited to build *Cyber-Physical Virtual Platforms* emulating both the "cyber" and "physical" parts of a CPS [53].

Even though the FMI standard proved to be a powerful tool to build such Cyber-Physical Virtual Platform, its focus is still strongly oriented to the simulation of continuous dynamic systems [54]. Thus, simulation of digital components still requires adapting the use of the standard to replicate the semantics of HW simulators [53]. Improvements to support Hardware Description Language (HDL) models in FMI have been addressed [53,55]. However, the advantages in terms of simulation speed of higher-level models, such as *Transaction-level models* [56], have not been exploited so far due to some limitations of the standard. This chapter aims at analying and discussing such limitations. Then, it proposes a set of adjustments in the use of FMI constructs defined in the current standard for co-simulation (*i.e.*, version 2.0). Furthermore, it presents a simulation coordination scheme that exploits such adjustments. These contributions

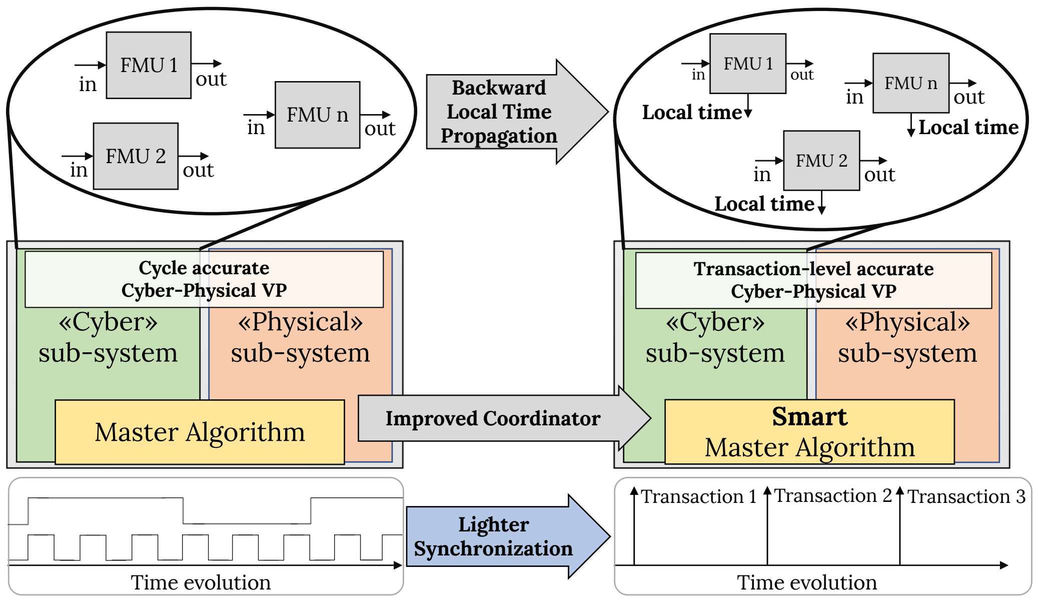


Fig. 4.1: Overview of the contribution.

together allows generating *Transaction-level Functional Functional Mock-up Units (FMUs)* for Cyber-Physical Virtual Platforms.

In the last few months, the FMI Steering Committee announced a new interface (version 3.0) that aims to introduce the hybrid co-simulation concept [41]. However, it is still in alpha release and, as highlighter by the analysis presented in this chapter (Section 4.6), it still require many improvements to effectively enhance the support of digital components into FMI-based simulation environments. On top of the time that will be necessary to develop the new standard, any new version of the standard will also require time to be accepted from all the tools supporting the previous standard. Meanwhile, using the current version 2.0, as we do in the approach presented by this chapter, guarantees compatibility with the current version of the tools.

Figure 4.1 summarizes the contributions of this work. On the left, the CPS to be designed is simulated by using a *Cycle-accurate Cyber-Physical Virtual Platform*. The virtual platform is composed by exploiting the FMI standard. It is composed of both the models of the "cyber" and the "physical" sub-systems of the model. In this work we focus on the "cyber" part of the system modeled by aggregating different FMUs, each of them representing a digital components of the system. The simulation is managed by a *Master Algorithm* coordinating the FMUs. The time evolution of the virtual platform on the left-sideis accurate with respect to the clock cycle of the system: each simulation step simulates a single clock cycle, synchronizing at each step. This work improves the left side configuration by proposing two modifications to the platform and its components:

- The functionality within the **FMUs** composing the digital part of the system are abstracted to **transaction-level**. Their interfaces are modified to make them communicate their internal local time backward to the master algorithm.

- The **master algorithm** is improved to exploit the information about the local time of the FMUs in the model.

These modifications allow to produce the ***Transaction-level accurate Cyber-Physical Virtual Platform*** on the right-side of the figure. The platform synchronizes at each transaction defined by the communication protocol. Thus, it benefits the lighter synchronization for improving the simulation speed.

The Chapter is organized as follows: Section 4.2 gives the necessary background about FMI, and summarize the state of the art. Section 4.3 discuss the advantages and the limitations in the current version of the FMI standard and discuss a set of possible improvements. Section 4.4 presents the methodology proposed by this paper. The presented approach is implemented by building an automatic tool-chain and then experimentally evaluated in Section 4.5. Then, Section 4.6 updates the discussion we previously presented [57] about the current support for digital models within FMI-based simulation environment. It presents the current efforts being made by the FMI Steering Committee to develop a novel version of the standard, and it discuss the improvements necessary to improve the support of discrete models into hybrid systems. Finally, in Section 4.7 we draw some conclusions.

4.2 Background and Related Work

FMI is a tool-independent standard aiming to enhance the interoperability between tools of different vendors in the field of systems design [21, 58]. It supports both model exchange and co-simulation of dynamic models produced by using different tools and languages. The standard has been originally developed by Daimler AG, and maintained initially by the MODELISAR Consortium, and by the Modelica association after the MODELISAR European Project ended. The latest version of the standard is the 2.0 of 2014. Currently, the version 3.0 is under development. The basic blocks of any FMI-based simulation environment are called FMUs. Multiple FMUs can be imported within a simulation tool to be executed. Each FMU may implement only one of the two variations of the current standard: *Model Exchange* or *Co-Simulation*. Model exchange FMUs describe functionalities by using differential, algebraic and discrete equations with time-, state- and step-events [58]. The equations must be solved by an external solver, that is thus required to simulate model exchange FMUs. Meanwhile, Co-simulation FMUs must model the functionality and implement the solver as well. As such, the model described within a co-simulation FMU does not require any external solver.

At its current state, the standard for model exchange does not suit well for describing discrete-event models [53]. Thus, chapter focuses on co-simulation which main features and structure are described hereby.

4.2.1 FMI Standard 2.0 for co-simulation

Practically, an FMU is an archive containing an XML file describing the component interface and a dynamic library providing its implementation. Furthermore, the dynamic library contained in any FMU for co-simulation must implement also the solver necessary to execute the

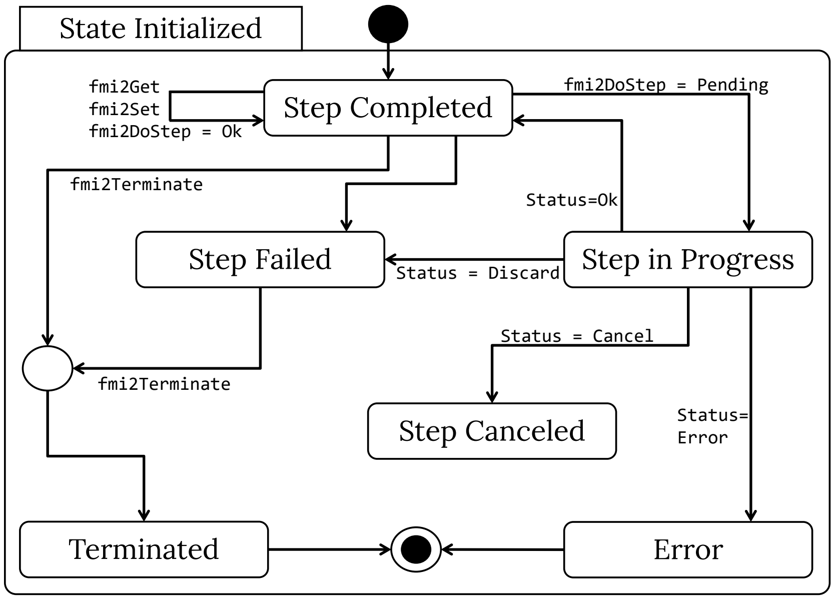


Fig. 4.2: Statechart representation of the coordinator algorithm for a FMU.

functionality. The XML file must specify all the variables of the FMU visible to the simulation environment [21]. Each variable is characterized by a *name*, *causality* (*e.g.*, input, output, parameter, *etc.*), a *type* and a *value reference*. The value reference of a variable must be unique among the variables of each type. Each variable is uniquely identified by the pair made of its type and value reference. The dynamic library must implement the functionality by implementing a set of functions defined by the standard. The most important, among the many defined in the current version of the standard, are:

- `fmi2SetupExperiment`: initializes the internal variables of the FMU.
- `fmi2Set`: sets the value of an internal variable of the FMU *i.e.*, it assigns a value to an input.
- `fmi2Get`: gets the value of an internal variable of the FMU *i.e.*, it returns the value of an output.
- `fmi2DoStep`: advances the simulation time of the component executing the behavior defined by the model.

The dynamic library must be generated using C-like linking [55], as such the functionality is usually expressed by using either C or C++. The standard defines the signature for all the C functions to be implemented by the dynamic library. However, it does not impose how they should be used, as it rather defines only some limitations on the possible combinations.

4.2.2 Simulation coordination in the FMI standard

Any model having one or more FMUs requires a coordination mechanism compliant with the FMI standard. Version 2.0 of the standard [21,58] defines the concept of *master algorithm* as the module of managing communication and synchronization for sets of FMUs. Communication is managed by the master algorithm by invoking the `fmi2Get` and `fmi2Set` functions of the co-simulation API. Meanwhile, synchronization and simulation advancement is implemented by

Listing 4.1: Sketch of the C implementation of a basic master algorithm compliant with the FMI standard. The algorithm executes a thousands iterations, each of those advances the local and global time of 10 time units.

```c
int main(int ac, char * av[]){
  fmi2Component component_1 = load_fmu("./component_1.fmu");
  fmi2Component component_2 = load_fmu("./component_2.fmu");
  ...
  fmi2Status st;
  ...
  st = fmi2SetupExperiment(component_1);
  st = fmi2SetupExperiment(component_2);
  ...
  time = 0; step = 10;
  ...
  fmi2Integer in_1, in_2, out_1, out_2;
  // Simulation starts here.
  for(int i = 0; i < 1000; ++i) {
    st = fmi2GetInteger(component_1, 0, &out_1);
    st = fmi2GetInteger(component_2, 0, &out_2);
    in_1 = out_2; in_2 = out_1;
    st = fmi2SetInteger(component_1, 1, in_1);
    st = fmi2SetInteger(component_2, 1, in_2);
    st = fmi2DoStep(component_1, time, step);
    st = fmi2DoStep(component_2, time, step);
    time = time + step;
  }
}
```

carrying on the components execution by invoking the `fmi2DoStep` functions of the FMUs composing the model being simulated. The standard defines some rules about how the master algorithm should be. However, the exact definition of the algorithm is not part of the standard. In fact, the rules explicitly defined are mostly imposing some limitations on the structure.

Figure 4.2 reports a statechart simplified version of the master algorithm. It shows the functions that the algorithm must invoke for each FMU in the model. The figure reports only the execution of a initialized FMU that already successfully went through the FMU setup state. Once a FMU has been initialized, its execution reaches the *Step Completed* atomic state within the *State Initialized* sub-machine. The master algorithm may invoke the `fmi2Get` or the `fmi2Set` functions, respectively reading or writing values of the FMU external variables. Otherwise, the algorithm may invoke the `fmi2DoStep` function by passing as a parameter the amount of time that must be simulated. Then, the machine moves to the *Step in Progress* state. The FMU simulates by executing its functionality: if the step is not canceled or discarded, and no errors are catched during the FMU execution, the `fmi2DoStep` returns and the machine goes back to the *Step Completed* state, and the FMU advances its own local time according to the one previously passed as a parameter. These steps iterate until no `fmi2Terminate` function is invoked. A simulation tool may implement the simplest coordinator for FMI by iterating this process for each FMU, or it may implement some more complex mechanism, still adhering to the state-chart in Figure 4.2. Finally, the standard explicitly states that is not legal to call a `fmi2Get` function after `fmi2Set` functions without calling the `fmi2DoStep` in between.

Listing 4.1 shows a C implementation of a trivial master algorithm using the functions defined by the FMI standard for co-simulation. The procedure loads the FMUs instantiating two variable of type `fmi2Component` that will points to the FMU implementations (Lines 3–6). The status variable is declared (Line 8): every function defined in the standard returns a status. The master algorithm initializes the FMUs, the timing variables and defines four integer variables (Lines 10–15). Then, a thousand simulation cycles are executed: the algorithm reads the output from the FMUs and assigns it to the input variables (Lines 18–20). Then, it sets the input variables of the FMUs (Lines 21–22). Finally, the algorithm executes the functionalities, advancing the global time of the FMUs and update the global time (Lines 23–25).

4.2.3 Related Work

Some of the FMI standard weaknesses have been first identified in [59].The main issues concern the managing of hybrid and discrete-event systems. The analysis highlights how FMI is more suited for physical, continuous-time (or discretized) systems, rather than discrete-event systems. Thus, it is tricky to use FMI when models require discrete events. The semantic gap between continuous-time models, and discrete-event models in FMI has been addressed [54] by proposing to use of tokens synchronizing the FMUs in the model when discrete-events happen. However, such a mechanism introduce many synchronization points in the execution thus slowing down the simulation. This may be particularly inconvenient when simulating models coming from HDL descriptions, as we showed in [53]. In the same work we proposed an ad-hoc synchronization methodology to reproduce the cycle-accurate behavior of HDL descriptions. It manages the synchronization locally to each FMU, while the data are exchanged by an additional FMU acting as a communication hub for the data in the system. The approach relies on automatic code generation to generate the FMUs implementing such mechanism. Automatic code generation of FMUs for co-simulation from HDL descriptions has been previously presented [55,60]: it relies on a state-of-the-art abstraction technique [20] to translate HDL models into C descriptions. The generated descriptions are finally wrapped by an interface using the FMI co-simulation API. While none of the approaches described above is proposing modifications to the standard, a number of papers do it. [41] proposes an additional mechanism to add to the FMI standard, aside to the model exchange and the co-simulation mechanisms. The novel mechanism is called *Hybrid Co-simulation*, and it is thought to manage hybrid models. The authors of [61] proposed some modifications to the API specified by the FMI standard for co-simulation. In particular, they proposed adding a *interrupt and preempt* mechanism to the `fmi2DoStep`. It allows the execution of an FMU to be interrupted when events must be managed.

To the best of our knowledge, **none of the previous work proposed to raise the abstraction level of FMUs to the *transactional level***. This is due to the fact that the master algorithm must always know in advance the next step size for each FMU [59, 62]. This chapter shows how we overcame this limitation, enabling transactional level Cyber-Physical Virtual Platforms assembled relying on more abstract FMUs.

4.3 FMI Standard Advantages and Limitations

As a first contribution of this paper, we discuss the standard's features useful to create cyber-physical virtual platform. Then we will discuss some limitations that make integration of virtual platforms difficult. Our discussion will be from a "cyber" point of view, as we aim at highlighting the weaknesses of the standard when dealing with discrete-event and cycle-accurate components.

Indeed the standard allows to ease the integration of different tools. It simplifies the interfacing of heterogeneous description. It allows the designer to care only marginally about communication and synchronization between simulators. Furthermore, it is reasonable to assume that complex CPSs are designed by multiple teams of designers. For instance, a team might be in charge of the physical part while the other designs the computational infrastructure. The FMI standard allows to easily integrate the models produced by different teams, to build a holistic simulation of the system.

However, as hinted in Section 4.2.3, the standard has been strongly oriented to continuous systems and dynamics. We can identify different drawbacks when modeling discrete components, and in particular when simulating digital components.

The set of *data types* provided by the standard is limited. When modeling digital HW it happens to use multi-valued logic values, or signals that uses an arbitrary number of bits. Meanwhile, FMI allows only integer, real, string and boolean. Thus, HDL data-types must be mapped on the provided types. Different mappings have been already proposed in the past. Multi-valued logics as well as arbitrary long bit vectors have been mapped onto strings [63], and (more efficiently) abstracted to unsigned integer [55]. Still, none of the previous mapping is ideal even though they partially solve the problem.

The data-types provided by FMI are even more insufficient when modeling digital HW models at higher levels of abstraction, or when modeling SW. In such case, models may require aggregate data types, *e.g.*, to represent sockets' payloads in transaction-level description, or classes of SW models. In this case, FMI does not provide any other solution than breaking down any aggregated type into its basic components.

The standard does not provide any mechanism to *specify the Model of Computation* employed by the FMU to implement the functionality. In the case of a digital HW description assignments are concurrent. However, simulators usually rely on sequential models of computation (*e.g.*, data-flows). When aggregating digital HW components using FMI, complex synchronization structures must be built [53, 54] to guarantee the functional equivalence of the aggregated model of the system.

It is not possible to retrieve the *internal time of an FMU*. The master algorithm "imposes" to each FMU its internal timing. The main issue is related to the `fmi2DoStep` function behavior: it is called by the master algorithm and it carries on the simulation time while executing an FMU functionality. The execution of an FMU cannot be preempted by external events. Neither the FMU is allowed to simulate an amount of time different with respect to the one imposed by the master algorithm, since the FMU cannot communicate back to the master algorithm its effective internal timing. For this reason, the master algorithm must alway be able to know

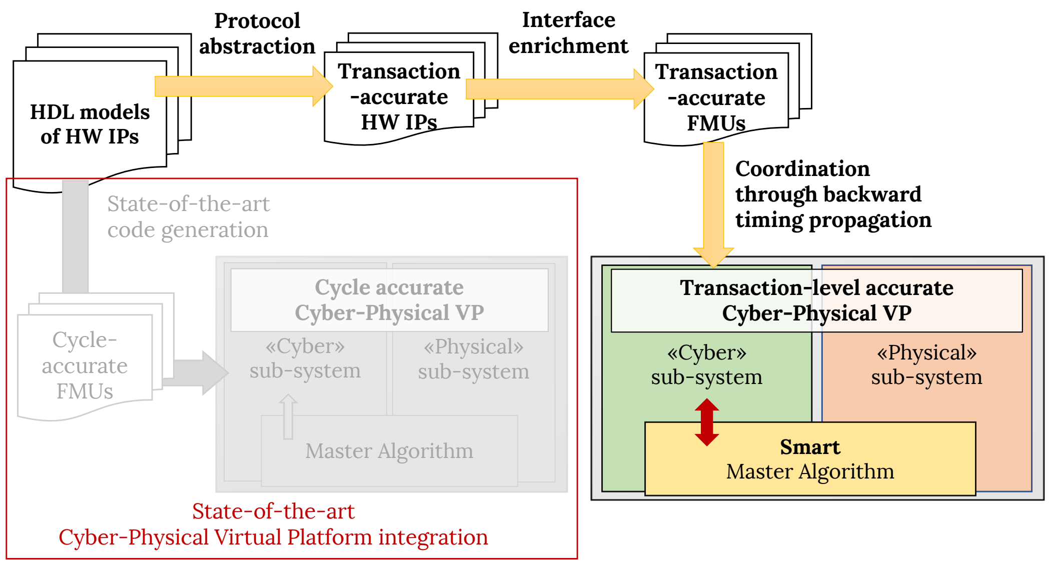

Protocol abstraction
Interface enrichment
HDL models of HW IPs
Transaction -accurate HW IPs
Transaction -accurate FMUs
Coordination through backward timing propagation
State-of-the-art code generation
Cycle-accurate FMUs
Cycle accurate Cyber-Physical VP
«Cyber» sub-system
«Physical» sub-system
Master Algorithm
State-of-the-art Cyber-Physical Virtual Platform integration
Transaction-level accurate Cyber-Physical VP
«Cyber» sub-system
«Physical» sub-system
Smart Master Algorithm

Fig. 4.3: Overview of the proposed approach, and comparison with the state-of-the-art methodology presented in [55].

exactly the length of the next time step of each FMU. This forces the master algorithm to call the `fmi2DoStep` function of an FMI using the shortest time step available, or to perform multiple step revisions. Thus, this limitation leads to an higher number of synchronization points in the simulation and makes impossible to use advanced synchronization techniques, such as *temporal decoupling*. Thus, it is not well suited to manage discrete events that might be generated by system's components. In the case of HW description, this usually forces to simulate each FMU with a time granularity equal to the clock cycle [53].

4.4 Methodology

Figure 4.3 summarizes the proposed methodology. It starts from a set of HDL Intellectual Propertys (IPs) models. An approach we precedently presented [55] (*i.e.*, red box in Figure 4.3) was simply translating the IPs into C models then wrapped into FMUs. Here we present a more advanced approach where HDL IPs undergoes an abstraction and manipulation process (colored arrows in Figure 4.3). The produced models rely on a transaction-level synchronization mechanism. Finally, these FMUs are inserted within the Cyber-Physical Virtual Platform, where they will be coordinated by a master algorithm that is aware of the shifting in synchronization and communication granularity achieved by applying the transformations.

4.4.1 FMUs generation and timing backward propagation

As identified in Section 4.3 FMUs cannot propagate their local time back to the coordinator. This is a major issue that must be tackled to achieve an efficient discrete-event simulation. In fact, solving such issue will allow the master algorithm to decide the next simulation step length

Listing 4.2: `modelDescription.xml` file of the `component_1` with time port.

```
...
<ModelVariables>

<!-- Input Ports -->
  <ScalarVariable name="in_1"
                  causality="input" \
                  valueReference="0">
                  <Boolean start="false"/>
  </ScalarVariable>

  <ScalarVariable name="in_2"
                  causality="input"
                  valueReference="1">
                  <Boolean start="false"/>
  </ScalarVariable>

<!-- Output Ports -->
  <ScalarVariable name="fmi2TLifaceTime"
                  causality="output"
                  valueReference="-1">
                  <Integer start="0"/>
  </ScalarVariable>

  <ScalarVariable name="out_1"
                  causality="output"
                  valueReference="0">
                  <Integer start="0"/>
  </ScalarVariable>

  <ScalarVariable name="out_2"
                  causality="output"
                  vr="1">
                  <Integer start="0"/>
  </ScalarVariable>

</ModelVariables>
...
```

more efficiently. Furthermore, it will allow each FMU to simulate in a decoupled way, without defining the simulation step size. As such, when the Master Algorithm calls the `fmi2DoStep`, a FMU can simulate until it does not need to synchronize or communicate with other system components.

The proposed methodology starts by generating Transaction-Level models starting from HW descriptions. This is achieved by using the methodology defined in [64]. It takes as input a HDL model described at the Register Transfer Level (RTL) together with its communication protocol, and it generates a functionally equivalent Transactional Level Modeling (TLM) description. The HW descriptions can be provided by using the most common HDLs (*i.e.*, VHDL or Verilog). The protocol of a component can be specified in different ways. The state-of-the-art implementations of the RTL-to-TLM abstraction methodology relies on ad-hoc protocol specification languages [20]. The resulting description is a C++ class representing a Transaction-Level model of the original component. Each transaction of the system is executed by invoking its `simulate` function, and it emulates one transaction of the specified protocol. The internal time of the model is annotated as `Integer` datatype, which represents the number of clock-cycles executed in the last transaction. The abstraction procedure computes the number of clock cycles for each transaction, and annotate it within the generated model.

The interface of the model is isolated in a structure embedded inside the C++ class. The structure contains a set of fields representing the original ports of the HW models. The data-types of these fields are abstracted into C native data-types. For instance, a 32-bit `logic_vector` datatype is abstracted into `uint32_t` C datatype. The methodology relies on automatic abstraction of HDL data-type [20] to perform this transformation. Furthermore, the interface structure also contains the time annotation of the model.

Our methodology goes on by wrapping the generated C++ class within the FMI functions. Thus, it generates the set of `fmi2Set` and `fmi2Get` necessary to write and read, respectively, input and output variables from and to the components. It also generates the `fmi2DoStep` function that calls the generated `simulate` function emulating a component transaction. The `fmi2DoStep` function still accepts the step length to stay compliant with the standard. However, it ignores it as the actual internal time of the FMU is computed by the `simulate` function.

The methodology generates also the XML file for the FMU. The original ports of the HW model are mapped in the FMI data-types: the `Boolean` and `Integer` FMI types are used to represent respectively single bit (or logic) and bit (or logic) vectors. The *value reference*, is assigned to each port starting from 0 for each data-type. Listing 4.2 depicts the definitions of the ports in the XML file for a component originally having two input and two output ports.

The methodology enriches the interface of the FMU with the internal time annotation of the transaction-level model, that is exposed as a new `Integer` port of the FMUs (see Listing 4.2, line 18-22). The value reference `-1` is reserved for the timing port. This assures that it can be uniquely identified once the FMU is loaded by a simulator. Furthermore, the timing port is called `fmi2TLifaceTime` in order to decrease the chances of name clashing with the other ports of the FMU. This last solution is helpful to increase also the readability of the produced FMUs.

4.4.2 A better coordinator for discrete systems

Listing 4.1 depicts a trivial Master Algorithm able to executes cycle-accurate FMUs. It must synchronize the components of the system at each clock cycle. Thus, the time step of each `fmi2doStep` is set to be equal to the clock period of the system being modeled. Such a solution is indeed precise; however, it uses an unnecessarily high number of synchronization points. The backward propagation of the FMUs internal time can be exploited to reduce the number of synchronization points.

Figure 4.4 shows the execution scheme of the proposed *Smart Master Algorithm*. Its core is the *FMU Coordinator*: it is in charge of storing the internal time values of the FMUs in the system, and it decides at each simulation step which components must be executed. Initially, the *Smart Master Algorithm* simulates all the FMUs, without defining a step size. All the FMUs return to the coordinator their internal time after their first execution. Then, the *Smart Master Algorithm* iterates the following steps (as in Figure 4.4).

- ① *Time-Data Storing*: the internal time and the new data of each FMUs are retrieved from the master algorithm and passed to the *FMU Coordinator* that stores them.

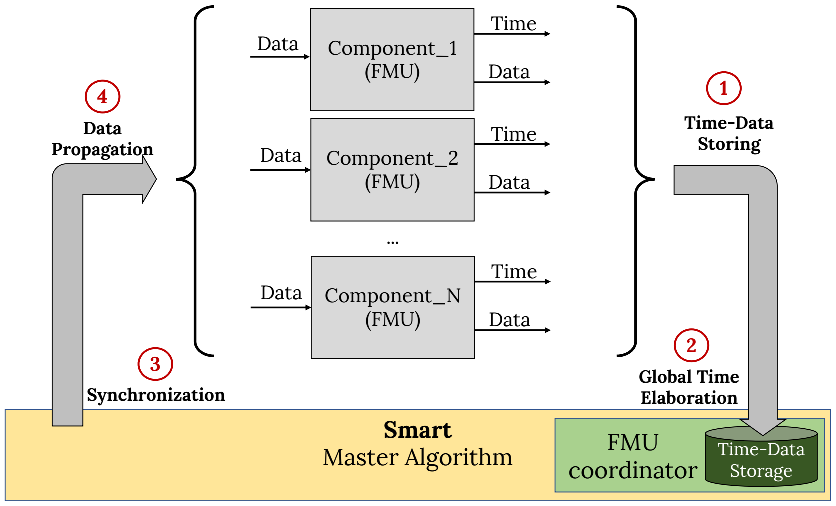


Fig. 4.4: Scheme of the *Smart Master Algorithm* with the *FMU Coordinator* of Transaction-Level FMUs.

- ② *Global Time Elaboration*: the *FMU Coordinator* elaborates the new Global Time of the simulation as the minimum value among all the internal times of the FMUs.
- ③ *Synchronization*: any FMU having the internal time equal to the Global Time is inserted into the list of *runnable FMUs*. Data read after the last execution of each runnable FMU, and previously stored by the coordinator, are shared with the system (*i.e.*, the values become valid for the entire system).
- ④ *Data Propagation*: the *Smart Master Algorithm* propagates the data and simulates the FMUs present in the list of runnable FMUs.

Listing 4.3 shows a sketch of the proposed Smart Master Algorithm. It reports only the most important parts of a possible C++ implementation of the coordination mechanism. Initially (Lines 2–9) it declares a status variable, an integer variable tracking the global time, and an array of components. The FMUs composing the system are stored in the array after being loaded. Furthermore, an array is declared to store the local times of each FMU. The same position in the two arrays refers always to the same FMU. Then, the coordinator initializes the simulation (Lines 11–16) by executing all the components once without advancing the global time. This step allows to generate the first set of events of the system, thus firing the event-based simulation mechanism, thus populating the set of runnable FMUs. For each execution, the local time is retrieved (Lines 13) and stored (Line 14). Then, all the output values written by the FMU are retrieved and stored (Line 15). Then, the system is simulated (Lines 18–39). At each simulation cycle a set containing the runnable FMUs is created empty, and populated after the global time has been update (Lines 19–25). Then, data previously produced by the runnable FMUs are propagated (Line 27). Finally, each runnable FMU is executed (Lines 28–37).

4.5 Methodology Application

We implemented the methodology by assembling a tool-chain performing the abstraction, manipulation and translation steps. We relied on the API provided by the HIFSuite framework [44]

Listing 4.3: Sketch of the C++ implementation of the *Smart Master Algorithm* exploiting backward timing propagation.

```
...
fmi2Status st;
unsigned int global_time=0;

fmi2Component components[num];
components[0]=load_fmu("./component_1.fmu");
components[1]=load_fmu("./component_2.fmu");

unsigned int local_time_vector[num];

for(int i=0; i < num; i++) {
 st=fmi2DoStep(components[i], global_time, 0);
 st=fmi2GetInteger(component[i], -1, &local_time);
 local_time_vector[i]=local_time;
 retrieve_and_store_output(component[i]);
}

while(global_time < 1000) {
 set< fmi2Component > runnable_FMUs;
 global_time=find_minimum(local_time_vector[0]);

 for(int i=0; i < num; i++) {
  if(local_time_vector[i] == global_time)
   runnable_FMUs.insert(components[i]);
 }

 propagate_data(runnable_FMUs);
 set< fmi2Component >::iterator it;
 for(it=runnable_FMUs.begin;
   it != runnable_FMUs.end; it++)
 {
  fmi2Component * component=*it;
  st=writeInputs(component);
  st=fmi2DoStep(component, global_time, 0);
  st=fmi2GetInteger(component[i], -1, &local_time);
  local_time_vector[i]=local_time;
  retrieve_and_store_output(component[i]);
 }
}
...
```

to extended the automatic code generation presented in [55]. The automatic abstraction of HDL descriptions is performed by specifying the components' protocols to generate the corresponding transaction-level C++ descriptions as defined in [64]. The models produced by the abstraction are enriched with the timing backward propagation mechanism. Finally, a tool wraps the model within the FMI APIs for co-simulation. We applied the tool-chain to a set of benchmarks varying with respect to two dimensions: the protocol latency and the number of FMUs composing the system. We aim at estimating the scalability of the proposed approach with respect to these two dimensions. We implemented the same functionality within each HW component

Table 4.1: Execution time of FMUs simulation using trivial Master Algorithm, with different number of iterations.

# iterations (clock cycles)	Execution of FMUs (seconds)				
	2	5	10	20	40
100 K	4.76	10.75	21.89	43.34	82.46
1 M	41.87	104.13	198.74	405.25	834.34
10 M	421.93	1021.64	2015.55	4129.17	8322.22
20 M	886.78	2062.32	4267.29	8219.65	16466.54

Table 4.2: Execution Time Comparison of Normal Master Algorithm and Smart Master Algorithm with different protocol latencies. In all the scenarios, 10 million clock cycles of the system have been simulated.

Base Latency (clock cycles)	Execution of FMUs (seconds)									
	2		5		10		20		40	
	Trivial	Smart	Trivial	Smart	Trivial	Smart	Trivial	Smart	Trivial	Smart
20	421.93	115.54	1021.64	250.23	2015.55	465.48	4129.17	889.39	8322.22	1752.71
speed-up:	3.65x		4.08x		4.33x		4.64x		4.75x	
50	421.93	60.28	1021.64	135.65	2015.55	253.43	4129.17	481.36	8322.22	964.92
speed-up:	7.00x		7.53x		7.95x		8.58x		8.62x	
100	421.93	44.71	1021.64	95.57	2015.55	179.34	4129.17	344.25	8322.22	702.17
speed-up:	9.44x		10.69x		11.24x		11.99x		11.85x	

of the system, since the paper focuses on the interfaces of the components, rather than on their internal functionalities. The internal functionality is kept extremely simple in order to let the communication and synchronization overhead to be predominant in the simulation. Each component is simply counting the number of clock cycles until its pre-defined latency is reached. For each experiment, we have considered components with different latencies. In the experiments we refer to the *base latency* of an experiment as the minimum latency of the component in that experiment.

We generate two FMUs of different types for the same HW model: the cycle-accurate FMU and the transaction-level FMU with backward timing propagation. All the experiments have been performed on a 64-bit machine running Ubuntu Linux 16.04, equipped with 16 GB of memory and an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz.

Table 4.1 reports the execution time by using the Trivial Master Algorithm, with different cycle-accurate FMUs and different numbers of iterations. The protocol latency dimension is not considered in this Table because the Trivial Master Algorithm simulates only cycle-accurate FMUs. Using the Trivial Master Algorithm the protocol latency does not affect the coordination overhead in the simulation. The results show that moving in both the dimensions (number of FMUs or iterations) the execution time increases almost linearly.

Table 4.2 compares the simulation speed achievable by using the Trivial and the Smart Master Algorithm. The performance obtained by using the Smart Master Algorithm depends on the protocol latency. On the contrary, the Trivial Master Algorithm performance is not influenced by such dimension. The Smart Master algorithm with the transaction-level FMUs achieves up

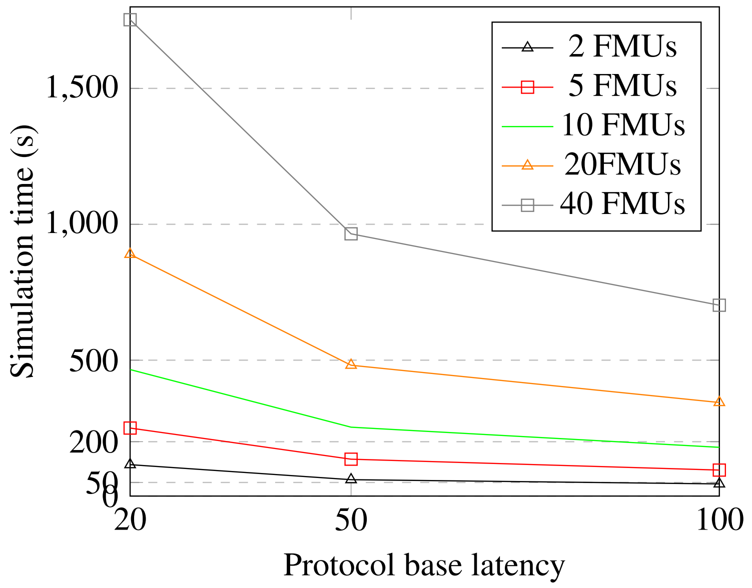


Fig. 4.5: Trend of the simulation overhead using the Smart Master Algorithm with respect to the protocol latency.

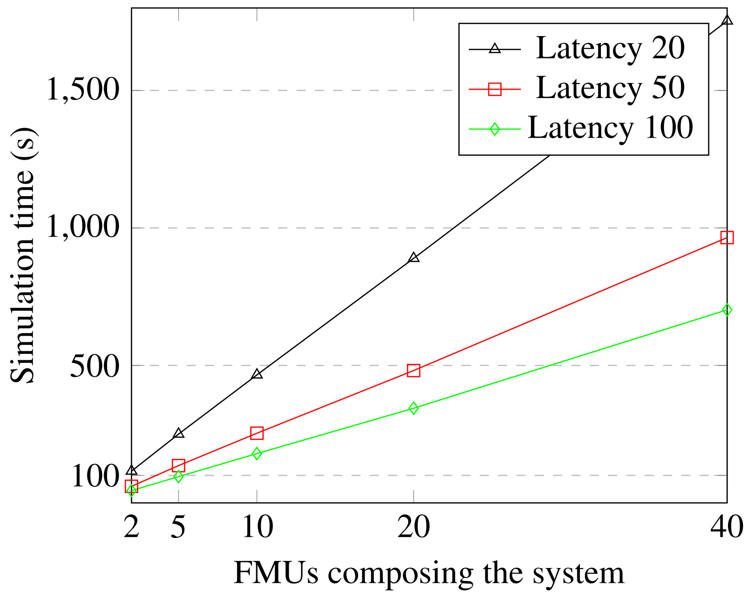


Fig. 4.6: Scalability of the Smart Master Algorithm with respect to the number of FMUs.

to 11x speed-up when using the largest protocol latencies considered. Reducing the protocol latencies of the transaction-level FMUs, the Smart Master Algorithm is less beneficial because of the increasing number of synchronization points. Of course, when the protocol latency is equal to one clock cycle (*e.g.*, when modeling combinatorial circuits) we have a degenerate case: the transaction-level and the cycle-accurate implementations will have the same amount of synchronization points. As such, only in that case, the Smarter Master Algorithm is slightly outperformed by the trivial one, due to the higher amount of computation required by the coordinator.

Listing 4.4: Datatypes comparison between FMI Standard 2.0 and the new 3.0 versions.

```
//FMI 2.0 DataTypes
typedef double          fmi2Real;
typedef int             fmi2Integer;
typedef int             fmi2Boolean;
typedef char            fmi2Char;
typedef const fmi2Char* fmi2String;
typedef char            fmi2Byte;

// FMI 3.0 Datatypes
typedef float           fmi3Float32;
typedef double          fmi3Float64;
typedef   int8_t        fmi3Int8;
typedef  uint8_t        fmi3UInt8;
typedef  int16_t        fmi3Int16;
typedef uint16_t        fmi3UInt16;
typedef  int32_t        fmi3Int32;
typedef uint32_t        fmi3UInt32;
typedef  int64_t        fmi3Int64;
typedef uint64_t        fmi3UInt64;
typedef int             fmi3Boolean;
typedef char            fmi3Char;
typedef const fmi3Char* fmi3String;
typedef char            fmi3Byte;
typedef const fmi3Byte* fmi3Binary;
```

Figures 4.5 and 4.6 give a graphical representation of how the simulation overhead changes when changing the protocol base latencies and the number of FMUs respectively. The vertical axes of both table reports the simulation time, while the horizontal axes reports the two considered dimensions. The trends in Figure 4.5 show how performance improve by increasing the latency. This is because a longer latency allows for more temporal decoupling, thus less synchronization and communication overhead. Figure 4.6 shows that the simulation time increases linearly with the number of involved FMUs. Thus, it shows the minimal impact of the more sophisticated master algorithm proposed in this paper.

4.6 Recent Development and Discussion

In 2018 the FMI Steering Committee has announced a new version of the standard, called FMI 3.0. The committee also published the list of new standard intended additional features[1]. In our opinion, among the proposed features, the most interesting aiming at providing a better support for digital components are:

- new datatypes.
- Structured ports and multi-dimensional variable.
- Intermediate output values and support for hybrid co-simulation.

[1] https://fmi-standard.org/news/2018/05/30/fmi-3-0-alpha-feature-list.html

The Steering Committee also underlines that not all the mentioned features might be introduced in the final version of the new standard. Still, the entire project is now stored in a public repository[2], it is thus possible to monitor the status of the development for the new standard. Currently, only new datatypes have been added to the features list. In details, it is now possible to specify integer values spanning from a single byte Integer (`fmi3Uint8,fmi3Int8`) to 64 bits Integer `fmi3Uint64, fmi3Int64`).

Listing 4.4 shows a comparison between the datatypes provided by the standard 2.0 and the new standard 3.0. Only lines 2-7 were present already in the previous standard. The new 3.0 standard extends the previously existing datatypes by adding the definitions reported in lines 10-24. It is important noticing that some of the new definitions replace those of the former standard. For instance, `fmi2Integer` (line 3) has been replaced with all the primitive C types that allows addressing specific amount of bytes (lines 12-19). These new datatypes allow choosing between Signed or Unsigned and from a single byte to 64 bits Integer. *fmi2Real* (line 2) has been splitted into `fmi3Float32` and `fmi3Float64` (lines 10-11), where 32-64 represent the size of the datatype (Floating-Point single precision or Double precision). Moreover, the introduction of these new datatypes implies consequentially the introduction of new methods for data-exchange (*i.e.* `fmi3GetUint8`, `fmi3SetUint8`, *etc.*). On the other hand, it is not clear if the new standard will provide compatibility with FMUs written by using the previous standards. One of the main purpose of the FMI standard is to support exchange of models among different teams, organizations and tools. As such, it is our opinion that will be necessary to provide interoperability between models produced by different organization, or by the same in different times, and that may thus rely on different versions of the standard. Furthermore, we also think that keeping the possibility of using more generic types, such as generic integer or generic real, may help designers in the initial phases of the modeling process when some details may still be unknown. Moreover, it is our opinion that providing the possibility of using generic integer and real types may makes the standard more attractive to users whose background is not in computer science. Another novel addition is the `fmi3Binary` data type. It is a opaque binary data type that may be useful to carry the information from complex sensors data to computational components, to model complex binary streams, or to model communication of closed-source components.

Structured ports and multi-dimensional variables are listed in the intended features list. Of course, supporting multi-dimensional variables will drastically move forward the standard toward the possibility of representing communication mechanisms typical of computing systems. In fact, it should provide the possibility of representing arrays, records and other software typical data structures. The same will be true for hardware bus-based communication, that can be represented by structured ports, similarly to what happens with the *payload* structures used in transactional models. For instance, structured ports will allow to simplify the approach presented in this chapter by using a single port representing the entire payload of the component modeled in a transactional FMU. However, at the current state of the work, the development

[2] https://github.com/modelica/fmi-standard/

of structured ports and multi-dimensional variables has been only announced, and any further detail has not been presented yet.

The possibility of accessing internal data in the middle of the `doStep` function is an extremely promising feature to enable hybrid co-simulation. In fact, accessing internal events of module being simulated enable the possibility of modeling mechanisms similar to interrupt, that are crucial to model reactive systems. Furthermore, such a feature will enable a better managing of events and time. Since important events may be visible to the master algorithm even before the termination of a FMU execution, the master algorithm can speculate by increasing the length of the FMU execution. This feature will also make obsolete the solution proposed in this chapter: here, an FMU simulates until the first interesting event, and then the master algorithm must retrieve the internal FMU time by the additional port proposed in Section 4.4. With the new feature, the master algorithm may impose a longer FMU execution, while monitoring eventual internal events of interest thus decreasing the number of required synchronization and communication points. Then, better mechanisms of handling the co-existence of FMUs governed by different model of computation, as well as mixed continuous- and discrete-time dynamics must be incorporated. However, even though some extensions have been proposed in the literature [41, 61], at its current state, the implementation does not clarify if such extensions will be integrated into the new standard. Integrating such features will be crucial to support digital models within FMI-based simulation frameworks. Otherwise, users will continue to be forced performing sophisticated manipulations to models, such as those proposed in this chapter.

4.7 Concluding remarks

This chapter showed discussed the current version of the FMI standard, and proposed a methodology to extend it to simulate FMUs representing digital components at transaction-level. The approach adds some information to the FMUs interface. Then, it adopts an ad-hoc master algorithm that is still conformed to the standard.

The experimental results showed the positive impact of the methodology. However, the approach requires design effort to explicitly force the standard to accept the transaction-level FMUs we defined. The analysis of the current effort to extend the standard shows both the importance of the proposed extensions, as well as the necessity to better target such extensions to support discrete-event models. Meanwhile, the proposed methodology will allow to cover where the current standard is still lacking.

5

Cyber-Physical Systems Integration in a Production Line Simulator

5.1 Introduction

Industry 4.0 represents the fourth industrial revolution and its goal is to evolve the actual factories into smart factory systems [1]. A smart factory is a factory that can make analysis and rational decisions to optimize and maximize the entire production. To be able to do that the smart factory requires a simulable model of the factory, usually called *Digital Twin*. A *Digital Twin* is a combination of Cyber-Physical System(CPS), where each CPS represents a process of the real factory. Nowadays, there are several tools developed by different stakeholders, that allow to model a production line [2]. All these modeling tools use Model-based design approaches, thus giving to modelers intuitive and easy-to-use environments. Unfortunately, these tools do not provide mechanisms to simulate Cyber-Physical Systems making impossible to model the production line processes with more details. Because of this limitation, the resulting simulation is not accurate enough in order to make precise analysis and planning optimal strategies. For instance, [4], couples real equipments with a production line simulator. The approach is promising but it does not consider the models of the physical processes, thus making the solution not usable to make accurate analysis. Mosterman et al. [65] proposed an example of a simple logistic production line, modeling the entire process by using Simulink. This is a complex solution, but not easily reusable in different scenario.
Some others [66, 67] try to fill this weakness by using co-simulation techniques. Co-simulation approches require complex environments and they are extremely error prone in the connection of components and time consuming. For this reason, in this work we present a methodology to integrate Cyber-Physical Systems in a Production Line Simulator but avoiding Co-simulation issues (see Figure 5.1). The proposed methodology starts from the modeling phase of the CPS by using specific domain languages and tools. For the digital system this methodology starts from VHDL or Verilog models at Register Transfer Level (RTL). On the contrary, the physical system is modeled by using OpenModelica tool [68], a state-of-the-art tool for this field. Both of the systems are then exported by using the FMI technology [21], as FMUs. Finally, the methodology relies on a coordinator written in C language, that manages the FMUs representing the two systems on the production line side. This work adopts Siemens Plant Simulation [69], that provides proprietary C-Interface. With this interface, it is possible to import C dynamic libraries

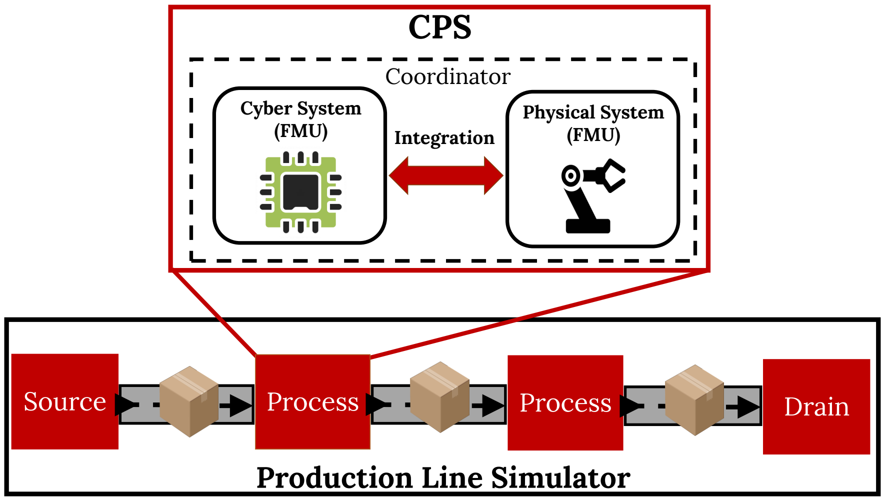


Fig. 5.1: Overview of the CPS integration in a production line simulator

(.dll) inside Plant Simulation. The proposed coordinator, and the FMUs, are then compiled as a C dynamic library and integrated in the simulator. The resulting dynamic Library contains a CPS, which represents a process of the production line. To enable the CPS integration, there are some issues to face with:

- FMU Generation for homogeneous and standardize communication;
- FMU Coordination;
- Integration of the CPS in the simulation environment;
- Timing scale differences among plant, cyber and physical models.

The paper examines and solves all such issues, allowing the designer to easily integrate CPS in a plant simulator, thus enabling precise analyses about the production line in order to optimize all the manufacturing processes.

The experimental results show the benefits of the integration of CPS in a plant simulator, in terms of simulation accuracy. The paper is organized as follow: Section 5.2 presents the necessary background about Production Line Simulators and about the FMI standard. Section 5.3 explains the integration methodology of the CPSs in Siemens Plant Simulation. Section 5.4 shows experimental results related to the proposed methodology, while Section 5.5 reports conclusions and possible future works.

5.2 Plant Simulation and Integration Alternatives

This Section presents two main background concepts: an overview of production line simulators, particularly Siemens Plant Simulation, and a brief description of the FMI standard. The two concepts are necessary to understand the methodology explained in section 5.3.

5.2.1 Production Line Simulators

In these years several providers proposed different tools to model Manufacturing processes [2, 5]. Report [5] summaries periodically all tools by proposing comparisons on their main char-

Listing 5.1: Sketch of Simtalk method that use an external library

```
openConsole

var lib_path := to_str("../library.dll")
var lib_ref : loadLibrary(lib_path)

var op1 : integer := 2
var op2 : integer := 3

var result := callLibrary(lib_ref, "add", op1, op2)

end
```

acteristics. The last report showed the main differences between tools in terms of provided functionalities, usability, multiprocessing execution, costs and other characteristics. These data help in the selection of the most suited plant simulator with respect to the parameters to be evaluated. A production line is mainly the composition of a chain of production processes. These processes require handling information like geometric properties of the products, the processing time, the energy consumption, the failure rate, *etc.*. The simulation of production systems, with this information, allows making decisions in order to optimize the entire production line in terms of cost, quality and productivity. These simulators have some common principles such as:

- *Layout Planning*: Represents the geometrical structure of the production line. All of these simulators have a library of components (*i.e.*, generic processes, assembly stations, buffers, *etc.*) which is possible to model the factory, by considering physical constraints.
- *Material Flow/Fluid Simulation*: Represents the transportation of the products from a process to the others. This is made possible with components like line transporters or pipe, depending if the product of the production line is solid or fluid.
- *Process Simulation*: Represents the physical transformation made by the processes to products.

All of these simulators use the Discrete-event Model of Computation. When a product enters in one of the production process chain, an event is triggered and that specific process can execute the relative action.

5.2.2 Siemens Plant Simulation: SimTalk C-Interface

Plant Simulation [69] is the widely used production line simulator developed by Siemens. It offers an internal programming language called *SimTalk*. This language gives the entire control over the simulation in order to customize the production line models. *SimTalk* allows to define methods that can be used inside every available object of the Plant Simulation library. In particular, it is possible to couple *SimTalk* methods with occurring events (*i.e.* Entrance or Exit of a product from a process) in every object of the production line model. Furthermore,

Listing 5.2: Simple C function implemented using SimTalk C-Interface

```c
1  #include "cwinfunc.h" // C-Interface library
2
3  extern "C" __declspec(dllexport)
4  void add(UF_Value *ret, UF_Value *arg){
5
6    int op1 = arg[0].value.integer;
7    int op2 = arg[1].value.integer;
8
9    ret->type = UF_INTEGER;
10   ret->value.integer = op1 + op2;
11 }
```

it is also possible to import dynamic libraries written in C, with a proprietary interface, called *C-Interface*.

SimTalk provides a `callLibrary` method that permits to call a function of an external dynamic library that is implemented using the *C-Interface* (Listing 5.1, line 9). Listing 5.2 shows a sketch of a function written in C implemented by using *SimTalk C-Interface*. The function has two parameters that represent the input and the output values, in order to respect the *C-interface* structure (Listing 5.2, line 4). The `UF_VALUE` is a structure defined by *SimTalk C-Interface* and it has to be used as the datatype of the function parameters. The `UF_VALUE` structure is composed of two fields: `type` and `value`. The `type` field represents the datatype of the parameters (Listing 5.2, line 9), while the `value` field represents the value of the parameters (Listing 5.2, line 10). *SimTalk C-Interface* allows the functions to have multiple inputs but only one output value. [2, 5] show comparisons between different simulators. From the survey [5] and the simulator comparison [2], we decided to adopt Siemens Plant Simulation for the purpose of this work because of *SimTalk* utilities. In particular the *C-Interface* is a very usable modeling environment that allows to extend the tool by creating models not natively supported.

5.2.3 Functional Mockup Interface (FMI)

In the state-of-art there are some solutions that try to uniform the interfaces and functionalities of the models. For instance, IP-XACT [70] is a standard for the definition of of Hardware model interface, but it does not consider their functionalities. Other solutions, like S-functions provided by Mathworks, define both interface and functionality of the models, but they are proprietary and are not supported by other tools.

The Functional Mockup Interface [21] is a relatively new standard that defines an interface that allows to export and imports models from different tools. The main goal of the standard is the simulation of heterogeneous systems. A component which implements the interface is called FMU. The standard defines two different interfaces: *Model Exchange* and *Co-Simulation*. An FMU may implement one or the other. In a Model Exchange FMU the simulation environment has to provides an external numeric solver. With *Co-Simulation* FMU the numeric solver is provided inside the model. This work considers only *Co-Simulation* Interface being more portable and easier to use. An FMU is composed of two parts: an XML file and a dynamic library. The XML file contains the information about the interface of the FMU, that will be visible to

Listing 5.3: Sketch of SimTalk method that uses CPS library functions

```
openConsole

var libPath := to_str("../CPS.dll")
var CPSLib : loadLibrary(libPath)

//get properties from the product
var prodType := product.prodType


//simulate the CPS with the product properties
callLibrary(CPSLib, "simulateCPS",prodType)


//Get the properties
var time:=callLibrary(CPSLib,"getTime")
var energy:=callLibrary(CPSLib,"getEnery")
var prodProperty:=callLibrary(CPSLib,"getProdProp")


//Set the properties in Plant Simulation Objects
SingleProcess.time:=time
SingleProcess.energy:=energy
//Set the new properties of the product
product.property:=prodProperty
end
```

the simulation environment [21]. Each port of the interface has different properties like *name*, *causality* (*e.g.*, input, output, parameter, *etc.*), a *type* and a *value reference*. The value reference of a port represents a numeric identificator that must be unique. The dynamic library must implement the functionality through a set of functions defined by the standard. The most relevant functions are:

- `fmi2SetupExperiment`: initializes the initial condition of the FMU.
- `fmi2Set`: sets the value of an input port of the FMU.
- `fmi2Get`: gets the value of an output port of the FMU.
- `fmi2DoStep`: executes the model contained in the FMU.

5.3 Integration Methodology

The integration relies on the FMI standard [21], and *SimTalk C-Interface*. The Cyber and the Physical systems are modeled and exported as FMUs. Then, a coordination algorithm with the *SimTalk C-Interface* is implemented, in the view of enabling the integration with the production line simulator.

Listing 5.4: Sketch of an FMU coordinator using FMISupport library with *SimTalk C-Interface*

```
#include "cwinfunc.h" //C-Interface library
#include "FMISupport.h" //FMUs Support library

extern "C" __declspec(dllexport)
void simulateCPS(UF_Value *ret, UF_Value *arg)
{

 Product prod=getProductProperties(arg);
 bool eventFlag=false;

 time = 0;
 while(!eventFlag)
 {

  //FMU Simulation
  fmuDoStep(cyberFMU,time,step);
  fmuDoStep(physicalFMU,time,step);

  //Data Exchange between FMUs and product
  eventFlag=dataExchange(cyberFMU,physicalFMU,prod);

  time=time+step;

 }
}
```

5.3.1 Cyber System: Modelling and FMU Generation

Cyber Systems are designed by using HDL languages (VHDL & Verilog) usually at Register Transfer Level (RTL). The entire Cyber System needs to be exported as a portable FMU, to allow the integration with the Physical System.

Thus, a first step is required to translate the HDL model into a C/C++ equivalent model, in order to simply wrap it with the FMI interface. In the years, the translation issue has been addressed by different works. For instance, [71] considered only the translation from VHDL to C. Verilog is not supported. In [72], VHDL and Verilog are both considered, but no other manipulation tools are provided.

This methodology relies on HIFSuite framework [44]. HIFSuite provides a set of manipulation tools for VHDL and Verilog models and also offers a set of APIs that allows to extend its functionalities. Some of these manipulations allow generating semantic equivalent models written in C++ [20]. After the manipulation and generation of C++ code, HIFSuite can also generate the XML file and the C wrapper, necessary for the composition of the FMU [53, 55]. As explain in Section 5.2 an FMU is a zip file that contains an XML file and a dynamic library. Then, the resulting C++ and C code is compiled in order to obtain the dynamic library. Finally, to generate the FMU, the XML and the dynamic library are zipped together. The used abstraction [20] changes the model of computation of the Cyber System from event-driven to cycle-accurate or even transaction-level, but preserving the initial behavior. This step is fundamental for the correct FMU integration with the other FMUs.

5.3.2 Physical System: Modelling and FMU Generation

Physical Systems can be modeled by using two different solutions, depending on the complexity of the system to be represented.

$$\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) \\ y(t) = Cx(t) + Du(t) \end{cases} \tag{5.1}$$

The first solution is used for Linear Time-Invariant (LTI) systems represented by a set of equations (see Eq. 5.1) and four matrices (*A,B,C,D*). The first equation represents the evolution of the internal states of the system. The second represents the output function, that depends on the input and the actual state of the system. To easily model LTI systems, we provided a C template, which allows to define these four matrices. The template uses a fixed step integration as numeric solver for the LTI system. The template is then wrapped with the FMI interface and compiled as dynamic library, to obtain an FMU of the system.

The second solution is used when dealing with more complex dynamic systems. In this cases a modeling framework like OpenModelica [68] can be used. OpenModelica is an open-source Model-based design environment. It offers a library of components that allows to model the physical system as the composition of these blocks, by using the Model-based design paradigm. Figure 5.2 shows a simple OpenModelica model that represents a second-order dynamical system, with a PID, secondOrder and feedback block. The model has one input port, "*u*", and one output port, "*y*". Furthermore, OpenModelica exports the system as an FMU. OpenModelica exposes the input and the output ports in the XML file of the FMU. The obtained Physical System uses Synchronous Data Flow Model of Computation (MoC) and it needs to be synchronized with Cycle-accurate MoC of the Cyber System [53].

5.3.3 Cyber-Physical System: Coordination and Integration

The two FMUs, representing respectively the Cyber and the Physical system, required to be connected and simulated together to obtain a CPS behavior. In order to do that an FMU coordinator is needed. For this methodology, we have developed a C library called *FMISupport* that allows managing Co-Simulation FMUs. The coordinator has to be integrated into Plant Simulation using the *SimTalk C-Interface*, to allow precise analysis concerning the production process. Listing 5.4 sketches a basic coordinator algorithm that uses the *FMISupport* library and *SimTalk C-Interface*. The function `simulate_CPS` is defined with the C-Interface signature and it represents the simulation function of the CPS (Listing 5.4, line 5). This function is called by a *SimTalk* method (Listing 5.3, line 11) associated to the Entrance event of a product in a process called *SingleProcess*. The properties of this product (*i.e.* type of the product, geometric properties, *etc.*) are passed to the `simulateCPS` function as parameter, and saved in a local structure (Listing 5.4, line 5-8). Then, the two FMUs are simulated for the same amount of time, using the `fmuDoStep` function provided by the *FMISupport* library (Listing 5.4, lines 16-17). After the simulation step of the line, data are exchanged between the FMUs and the product (Listing 5.4, line 20). The simulation stops when a certain event is reached. This event

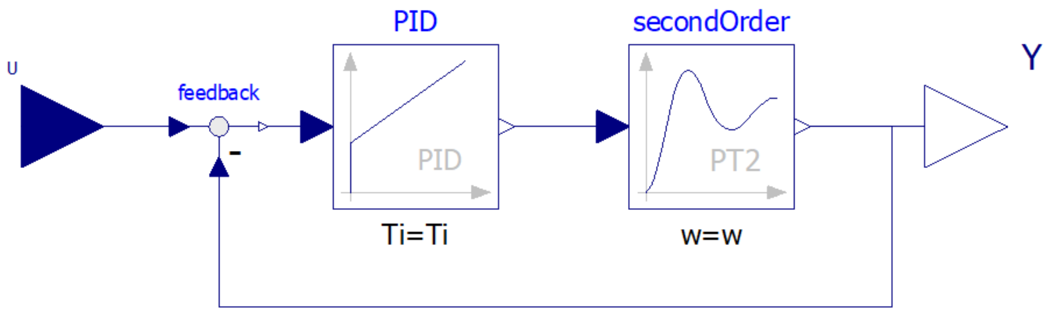


Fig. 5.2: Example of an OpenModelica Model

represents, for example, the end of the physical process. The time of the entire process could depend on the product properties of the line. For instance, if the process has to deal with different types of product, it could affect the processing time. Furthermore, the processing time could also affect the energy consumption of the process. *SimTalk C-interface* does not define a structure to return more than one value for each function. Because of this limitation related to *SimTalk C-interface*, it is needed to define a get function for each desired property. For that reason, some functions are defined in the Coordinator to retrieve the information related to the product and the process. Regarding the process, this methodology focuses on two aspects: the processing time and the energy consumption. In Listing 5.3 it is possible to see the *SimTalk* method that uses the CPS Library implemented in Listing 5.4. The *SimTalk* method retrieves the product properties (Listing 5.3, line 7) and then calls the `simulateCPS` with the product properties as parameter (Listing 5.3, line 11). After the execution of the `simulateCPS`, the *SimTalk* method retrieves the processing time, the energy consumption and the new product properties (Listing 5.3, lines 15-17). Finally, the *SimTalk* method propagates the properties to the product and to the *SingleProcess* (Listing 5.3, lines 20-24).

5.4 Methodology Application

The proposed methodology has been applied to a three-process production line. Metal sheets cross the line that must bend sheets to a desired angle. Involved processes are the following:

- *Sensing*: it analyzes sheets to discover their bend angle which is written over an applied barcode;
- *Bending*: it reads angles from barcodes, then it computes the bending actions and it controls the bending machine;
- *Checking*: it ensures that sheets have been correctly bent; thresholds are defined to separate correct, acceptable and damaged sheets.

This example has been chosen because it contains the three-steps sequence common to many production lines: generation, production and validation of items.Plant Simulation models this production line with different nodes, as shown in Figure 5.3:

- *Source* node provides metal sheets to the line;

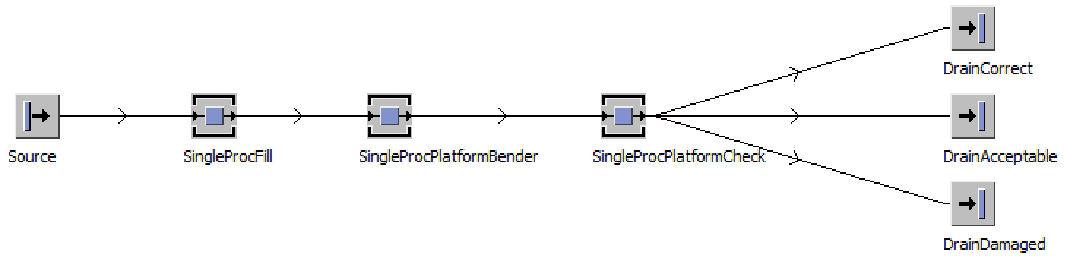


Fig. 5.3: Overview of proposed production line in Plant Simulation

- *Drain* node receives manufactured metal sheets;
- *SingleProcess* nodes represent the three processes above.

Plant Simulation provides *SimTalk* to describe production line functionalities: this leads to a high-level simulation which is very fast but not much detailed. The proposed methodology integrates CPSs into *SingleProcess* nodes to get simulations enriched with time, energy and mechanical wear. This is done by using the *SimTalk C-interface* which permits to call customized C-functions directly from the simulator.

5.4.1 Bending machine CPS

This CPS consists of a digital virtual platform and an analog model of a bending machine. The digital platform is composed of M6502 CPU, RAM and ROM memory, bus, sensor and bender I/O interfaces as shown in Figure 5.4. CPU can access bus peripherals by MMIO. For The bending machine there are two models obtained by using two solutions proposed in Section 5.3.2. The first model is a partial model of the bending machine and it is described using the C template, defining the values of the four matrices A,B,C,D (Section 5.3.2, Eq. 5.1). This model allows to have faster simulation but does not have accurate details about the energy consumption of the equipment. The second model is more detailed than the first it is described using OpenModelica tool [68], integrating information concerning the energy consumption. These models are characterized by different bending speeds according to bending direction, different power consumption, different bending times, increasing bending error due to bender wear. Two versions of this CPS are provided:

- *Partial platform*: it is composed of the digital platform and a partial bending machine model, described as LTI system.
- *Complete platform*: it is composed of the digital platform and the complete bending machine model, described using OpenModelica;

The CPU reads the angle from the previously applied barcode, then computes the bending action as delta angle and bend direction. The bending action is provided to the bending machine through the bus. When the bending process completes, metal sheet can advance to the next node. This behavior is shown in figure 5.4. The models of the digital platform are written in VHDL and Verilog. The entire digital platform is automatically abstracted into their C++ cycle-accurate level representation by state-of-the-art [53] and then wrapped in order to ob-

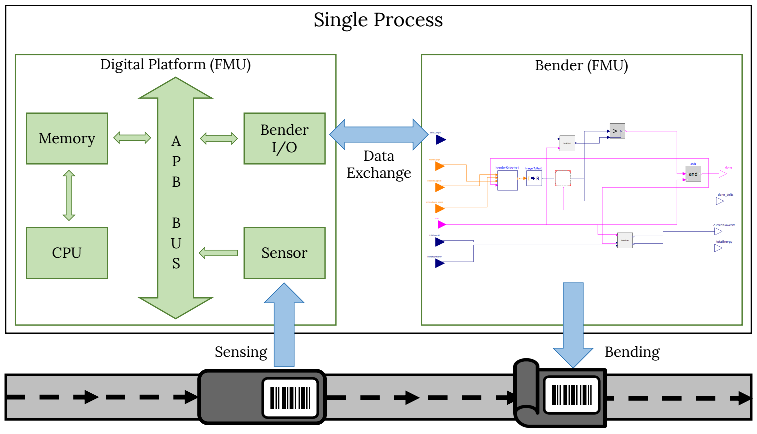


Fig. 5.4: Overview of bending machine CPS

Table 5.1: Properties evaluated by the different simulation line versions

Version	Functionality	Time	Energy	Mechanical Wear
SimTalk	✓	≈		
Partial Platform	✓	✓	≈	≈
Complete Platform	✓	✓	✓	✓

Table 5.2: Execution Time comparison of different approaches.

Metal Sheets	SimTalk		Partial Platform		Complete Platform			
					Basic Coordinator		Optimized Coordinator	
	Total Time	Time per Sheet	Total Time	Time per Sheet	Total Time	Time per Sheet	Total Time	Time per Sheet
10	0.01	0.0001	18.17	1.8165	48.33	4.8334	1.40	0.1404
100	0.01	0.0001	202.35	2.0235	499.89	4.9989	2.85	0.0285
500	0.03	0.0001	1021.89	2.0438	2562.85	5.1257	9.06	0.0181
1000	0.06	0.0001	2290.06	2.2901	6364.07	6.3641	17.04	0.0170

tain the Digital Platform FMU. The CPU Software is cross-compiled stored inside the Digital Platform FMU. The two models of the Bending machine, partial and complete, are exported as FMUs using the C template and the OpenModelica framework. The Digital platform FMU and one of the Bender FMUs, are manually interconnected with a coordinator(see Listing 5.4). The FMUs and the coordinator are wrapped using the *SimTalk C-interface* exposing the functions needed to simulate the CPS (`simulateCPS`) and retrieve the properties (*i.e.*,`getTime` and `getEnergy`). Finally, the entire code is compiled into a dynamic Library ready to be loaded from the simulator (see Section 5.3.3). A model composed of only the bending machine analog model is not provided. **This is due to the digital platform simulation time that is negligible when compared with the one required to simulate the bending machine model.**

Table 5.3: Simulation times and percentage of errors of the proposed approaches, respect to the real bending machine.

Metal Sheets	Partial Platform		Compl. Plat. Basic		Compl. Plat. Opt.	
	Sim Err.	Time/Sheet	Sim Err.	Time/Sheet	Sim Err.	Time/Sheet
10	0.00%	1.8165	0.00%	4.8334	0.00%	0.1404
100	33.33%	2.0235	0.00%	4.9989	25.00%	0.0285
500	42.86%	2.0438	0.00%	5.1257	33.93%	0.0181
1000	40.30%	2.2901	0.00%	6.3641	19.40%	0.0170

5.4.2 Alternatives Taxonomy

Table 5.1 summarizes all aspects supported by the different modeling alternatives proposed in this paper. It compares the standard Plant Simulation production line model with CPS-enriched ones. The *SimTalk* approach can only simulate bending functionalities and it provides an estimation of the simulated time. This is the highest level of abstraction because a bending machine simulable model is not used. With a CPS model, more information can be retrieved. *Partial Platform* grants detailed information about CPU processing time. Partial bending machine model used in Partial Platform provides only an estimation of energy and mechanical wear but it can give the precise bending time. With *Complete Platform* instead, it is possible to get detailed information about all properties. This is possible thanks to the refined model.

5.4.3 Simulation speed

Simulation data are analyzed from different points of view to demonstrate the usability of this work. First adopted method is to measure the simulation times and the required times to simulate the bending of a single metal sheet: it is important that simulated line can advance its time at least as the real factory does.

Table 5.2 reports the simulation time respect to the total number of processed metal sheet. This paper proposes two different coordinators for the *Complete Platform*: the *Basic* and the *Optimized Coordinator*. The difference between *Basic* and *Optimized Coordinator* concerns the step size used to simulate an entire operation of the bending machine FMU. The *Basic Coordinator* always uses a step size equal to the clock period of the digital FMU. The *Optimized Coordinator* adopts longer steps during bendings, that decouple the simulation of the FMUs and reduce the number of synchronization point. The *Complete Platform* with the *Optimized Coordinator* achieves up to 374x speed-up, respect the *Basic Coordinator*, with 1000 bending operations. The real bending machine requires on average $\sim$ 1 second at every new sheet ($\sim$ 1.8 seconds in the worst case). This leads to about 3 seconds per metal sheet considering also times required to transport sheets between nodes. In table 5.2, *Complete Platform* with *Basic Coordinator* requires more than 3 seconds to simulates a single bending operation. This result does not met real-time requirements in order to plan strategies on the real factory. In conclusion, *SimTalk*, *Partial platform* and *Complete platform* with *Optimized Coordinator* are solutions that can be adopted to advance in parallel to real production line.

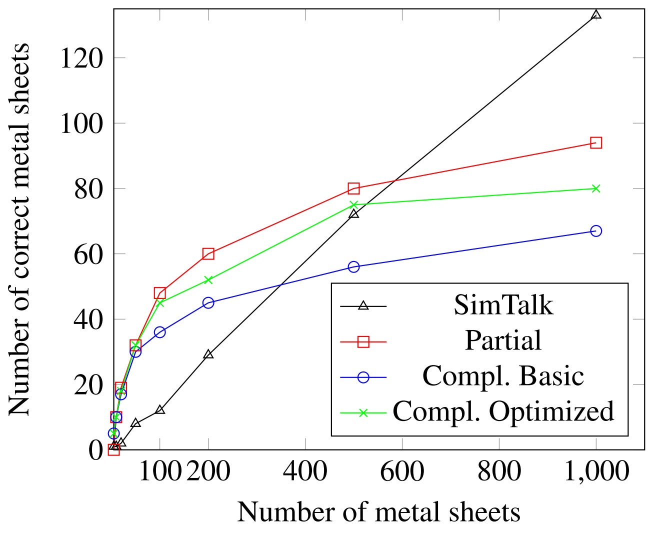


Fig. 5.5: Number of correct metal sheets

5.4.4 Simulation errors

The second analysis focuses on the measurements of simulation results: an abstracted model leads to less detailed simulations. Figure 5.5 shows the number of correct metal sheets produced by the proposed approaches. *Complete Platform* with *Basic Coordinator* represents the correct trend: it is the most detailed simulation, but requires a lot of time and can not be used in a real-time context. *SimTalk* version grows almost linearly because it does not take care of mechanical wear. *Partial Platform* leads to smaller error thanks to its approximated wear model. *Complete Platform* with *Optimized Coordinator* leads to an error smaller than *Partial Platform* one. It is due to different approximations done by analog numeric solvers when changing step size. Thus, *Complete Platform* with *Optimized Coordinator* represents an optimal trade-off solution between simulation time and the percentage of errors.

5.5 Conclusions

We presented a methodology for the integration of Cyber-Physical Systems in a production line simulator. This work represents an efficient alternative to co-simulation methodologies for the modeling of Digital Twin concerning Industry 4.0.

The methodology has been implemented and applied to a common use case scenario in manufacturing process. The results obtained from the simulation clearly show the benefits from the integration of Cyber-physical systems in terms of accuracy. **CPS integration allows to estimate properties of the production line that could not be estimated elsewhere**. The simulation time required to compute the CPS is 100 times faster then the real processing time. These promising results enable the possibility to make precise analysis in order to plan a strategy to optimize the entire production line. Future work will focus on improving the simulation speed proposing a new coordination of the FMUs and mixing models with different levels of abstraction.

6 Automatic Integration of Cycle-Accurate Models into Cyber-Physical Virtual Platforms

6.1 Introduction

Virtual platforms are powerful tools to co-design HW/SW devices [73] as they allow to execute SW while considering constraints imposed by the architecture [74]. In the case of CPSs, digital HW/SW is used to control physical processes. Thus, virtual platforms for CPSs must be able to capture evolution of both continuous (*i.e.*, physical processes) and discrete (*i.e.*, HW/SW devices) parts of the system. Unfortunately, virtual platforms for CPSs are not yet available and control SW designers still prefer to rely on multi-domain dynamic system simulators, such as Simulink, that allow them to design control strategies without considering architectural details. Of course, a given architecture may introduce timing artifacts that make control strategy ineffective, thus forcing re-implementation [75]. Cyber-physical virtual platforms would be thus beneficial to reduce this risk if they allow simulating together the timing-accurate behavior of HW/SW devices and the continuous dynamics of physical components of the CPS being simulated. Such a simulation environment may be obtained by integrating models of HW components within dynamic system simulators. It is thus necessary to overcome the semantic gap between cycle-accurate HW models and data-flow continuous-time models [76].

This work assumes that cycle-accurate HW models are originally described as RTL IPs using HDL. Therefore it faces the problem of mapping the HDL constructs onto the target simulation environment primitives. Then, it defines two different synchronization techniques to reconcile the concurrent semantics of HDLs with the sequential execution of data-flows.

As depicted in Figure 6.1, the approach starts from a set of discrete-event HDL models of HW IPs, and a data-flow continuous-time model of the physical processes (*i.e.*, plant and environment). Each discrete-event HW model undergoes a state-of-the-art abstraction process [20,43,44] to produce its cycle-accurate C++ representation. Then, the novel synchronization and code generation techniques are used to integrate cycle-accurate models to data-flow models. We adopt Simulink as target simulation environment, being the most popular simulator among system engineers. However alternative code-generation steps are presented to provide either Mathworks' C MEX S-function implementations, or portable FMUs compliant with the FMI standard [21]. The approach is completely automatic: this and the other advantages will be shown in the experimental section of this paper.

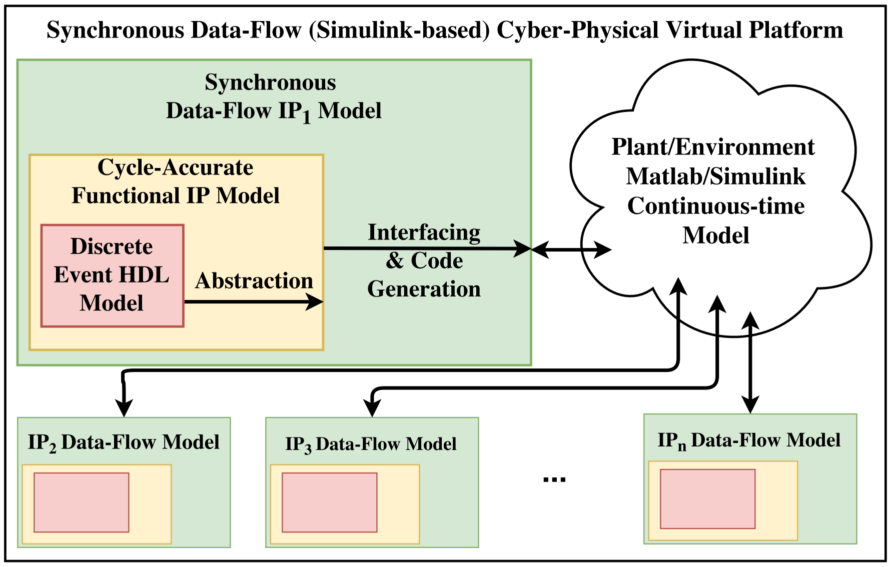

Synchronous Data-Flow (Simulink-based) Cyber-Physical Virtual Platform
Synchronous Data-Flow IP$_1$ Model
Cycle-Accurate Functional IP Model
Discrete Event HDL Model
Abstraction
Interfacing & Code Generation
Plant/Environment Matlab/Simulink Continuous-time Model
IP$_2$ Data-Flow Model
IP$_3$ Data-Flow Model
...
IP$_n$ Data-Flow Model

Fig. 6.1: Structure of the target simulation environment.

6.2 Background

6.2.1 Related work

As of today Virtual platforms focus on digital HW/SW systems [77] and support of continuous-time models is limited to analog devices [24, 78], and Micro-Electro Mechanical Systems [79]. CPSs SW development is still strongly based on top-down methodologies [19], and Model-based design tools such as Mathworks' Simulink [18], rely on abstract models thus imposing HW-in-the-loop techniques to perform precise timing estimation of the computational unit. Naderlinger [80] proposes *xTask blocks*, *i.e.* Simulink S-Functions representing time-annotated SW tasks: the approach is promising but it is still an approximation based on manual timing estimations. Alternatively, virtual platforms development can rely on Analog-Mixed Signals HDLs: [81] proposes Verilog-AMS and VHDL-AMS as modeling languages for heterogeneous systems. However, system and control engineers are reluctant to rewrite their models.

Co-simulation is widely used to emulate heterogeneous systems [25]. Standardized interfaces, *i.e.* FMI, allow to build complex simulation environments. Integration of SystemC models using FMI has been explored [82]: however, SystemC is a modeling rather than a design language. [76] formalizes the FMI primitives and proposes a synchronization mechanism to enable co-simulation of discrete-event models and data-flow descriptions. However, it is thought to be used in top-down design flows, without reusing IPs. The automatic generation of cycle-accurate FMI and Simulink blocks from HW models has been proposed by us in [83]. However, we did not address the problem of composing multiple blocks preserving functional equivalence, that is a major target of this paper.

Fig. 6.2: Flow diagram implemented by HDL simulators.

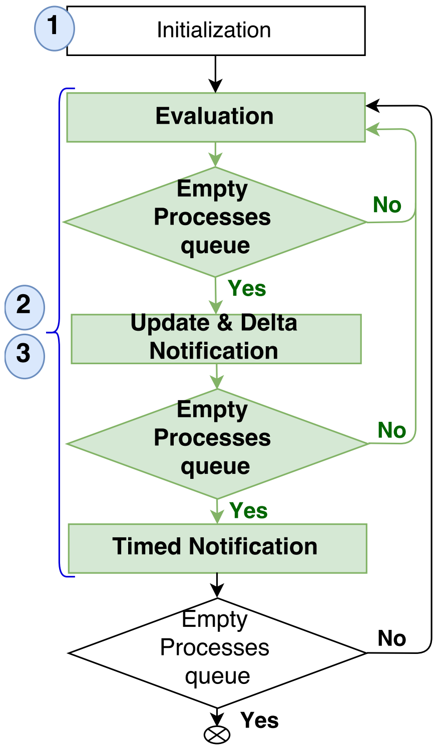


Fig. 6.3: Statechart representation of HDL simulation.

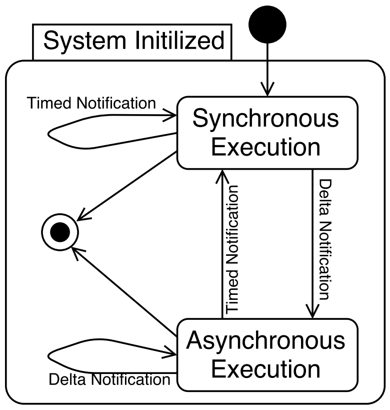


Fig. 6.4: HDL simulation statechart after abstraction.

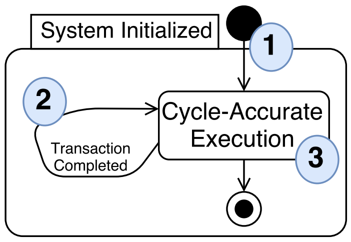


Fig. 6.5: Execution schemes involved in the RTL to cycle-accurate abstraction and translation methodology.

6.2.2 Code Generation for Virtual Platform Integration

Tools as Carbon Model Studio [43], Verilator [72], HIFSuite [44] and methodologies have been proposed to automatically translate HDL IPs into cycle-accurate C++ models.

The abstraction methodology presented in [20] is used as a starting point for this work: it produces a C++ model that is cycle-accurate with respect to a starting HDL description. Its interface is a function executing a simulation cycle at each call. Inputs and Outputs are managed by a payload structure passed to the function as a parameter. Figure 6.5 depicts the different computation schemes involved in this abstraction. Figure 6.2 is the flow chart representing the HDL scheduler. Green boxes highlight the steps executed at each clock cycle. Figure 6.3 is the state chart implementing the scheduler in Figure 6.2. Figure 6.4 shows the state chart obtained by applying the abstraction methodology to the one in Figure 6.3: it has a unique state and a self-transition executing the set of green boxes in Figure 6.2 at each simulation function call. Circled numbers of Figures 6.5 and 6.6 will be used in the following of the paper to highlight the relations between execution schemes.

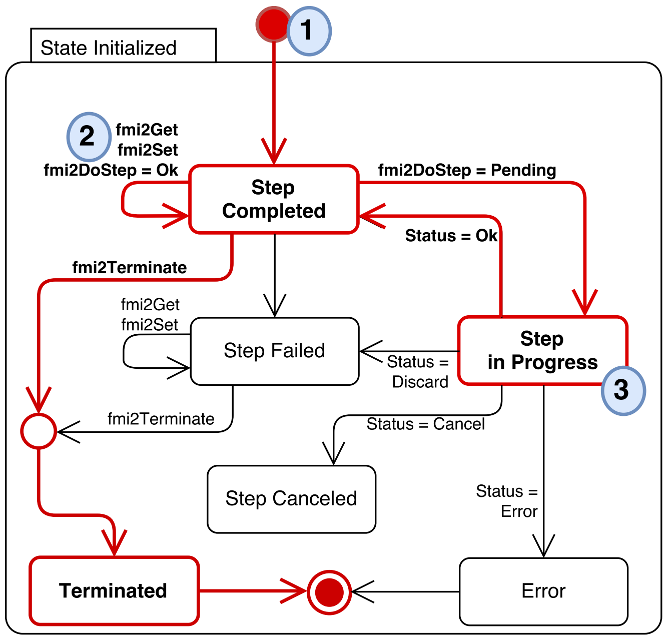


Fig. 6.6: Statechart representation of the calling sequence of Co-simulation C functions to simulate an initialized FMU.

6.2.3 Interface technologies for system simulation

This work exploits two standardized interfaces to import and connect heterogeneous models: the FMI standard 2.0 [21], and Mathworks' proprietary C MEX S-Functions [18]. Both provide C-based interfaces to model components functionality.

FMUs [21] are the basic blocks of a simulation environment based on the FMI standard. Each FMU represents a component and it contains all the information necessary to be simulated. It may implement either the *Model Exchange* or the *Co-Simulation* version of the FMI standard: this work focuses only on Co-Simulation version, as it is more suitable for discrete-time models. An FMU consists of an XML file and a C-based implementation. The XML file describes the variables exposed by the FMU. The functional implementation of an FMU is based on the FMI standard C interface defining a set of functions implementing system behavior in terms of input-reading, output-writing and execution operations. The standard does not define how functions are scheduled at each simulation step: it rather specifies some constraints. As such, the same set of FMUs may act differently [84] on different simulators. The simulation evolves according to the state chart in Figure 6.6. Circled numbers, red thicker lines and bold text in Figure 6.6 will be used in the following Sections of this paper.

S-Functions rely on similar concepts, although they provides a simpler interface. A configuration file specifies the functions in charge of managing the simulation. The simulator executes the *initialization function* when the model is instantiated. Then, at each simulation step, it executes its *output function* in which the designer specifies the order of the input-reading, execution, and output-writing operations.

6.3 Integration methodology

This work provides different alternative and engineers can tail the virtual platform integration process to best suit their needs. We evaluate the proposed approaches and the state-of-the-art and we identified the following list of desired features:

- *Automatic reuse of components* by automatically importing previously designed HW IPs into a virtual platform.
- *Automatic generation* of model: to allow HW engineers to model each device or IP in their favorite HDL.
- *Seamless integration* to allow system engineers to import HW IPs into a virtual platform with no prior knowledge about internal details of devices, thus enabling *Compositional design*.
- *Portability* of generated models to support different systems simulators that may be used by different engineer teams, thus not being bonded to a specific vendor.

This paper addresses the generation of “cyber models” targeting:

- *Portable FMUs*: the code generated after performing automatic abstraction is mapped onto FMI standard primitives.
- *Mathworks’ S-Functions*: the code generated by automatic RTL to cycle-accurate abstraction is mapped onto vendor-specific primitives provided with Mathworks’ Simulink.

Finally, synchronization between different cyber modules into a platform can be managed by two alternative approaches:

- in the *Monolithic model* approach IPs composing the computational infrastructure of the system are integrated before applying the RTL to cycle-accurate abstraction. It will produce a single module, where synchronization and communication between IPs is managed internally [20, 24].
- The *Hub-based* approach allows to keep IPs separated. An additional module (called Hub) takes care of connecting the cyber modules with each other and to the physical sub-model of the system. It aims at overcoming the gap between the sequential semantics of data-flows and the concurrent semantics of HDLs.

The hub-based approach has two variations since an interface may grant complete (*e.g.*, S-Functions) or partial (*e.g.*, FMI standard) freedom about the sequence of primitive calls.

To integrate HDL IPs into dynamic system simulators we first introduce a mapping between the starting HDLs and the target interfacing technologies. This is necessary to create the modules to integrate. Then, we define two synchronization approaches providing alternative features.

6.3.1 Mapping HDL primitives to FMI and S-Functions

Figure 6.5 summarizes the two different execution schemes involved in the RTL to cycle-accurate abstraction for digital IPs. Figure 6.6 shows the execution scheme imposed by the

Table 6.1: Mapping of HDLs simulation events onto the primitives offered by the FMI standard and S-Functions.

HDL simulation Events	**FMI Standard** Primitives	**S-Functions** Primitives
Initialization	Sequence of assignments in initialization mode	Initialization function defined by the `mdlStart` callback method
Input Signals Reading	`fmi2SetInteger`	Parameters passed to Compute Output function by value
Output Signals Writing	`fmi2GetInteger`	parameters passed to Compute Output function by reference
Simulation Cycle Execution	`fmi2DoStep`	Output function defined by the `mdlOutputs` callback method

FMI standard. S-functions rely on a similar schema and differs only on primitives naming. Circled numbers in figures label relations among schemes: ① labels the initialization phases, ② input and output phases and ③ the execution phase. Labels in Figure 6.6 show that the *Step Completed* and *Step In Progress* states are necessary to reproduce HDL semantic.

Table 6.1 reports the FMI and S-Function primitives used to map HDL events. That is how correspondences highlighted by circled numbers in Figures 6.5 and 6.6 are implemented.

HDL model initialization (*i.e.*, ①) is reproduced by two FMI primitives. The `fmi2Instantiate` function allocates the memory necessary to load the model, and creates the indexes identifying the external variables of the FMU. The `fmi2SetupExperiment` function leads internal values to their initial configuration. To reproduce the same behavior in an S-Function it implements a *setup function* specified by the `mdlStart` method of the S-Function configuration file. The setup function leads internal values to their initial configuration.

HDL input and output events (*i.e.*, ②) are reproduced in FMI by the `fmi2SetInteger` and `fmi2GetInteger` functions, respectively. They are called by the simulator at each simulation cycle for all the variables exported by the FMU as input or output variable. After the data-type abstraction is performed to generate the C cycle-accurate models, all the variables are integer. S-Functions input and output variables are parameters of the output function. Inputs are passed to the function by value, while outputs by reference.

Simulation of cycle-accurate models is a sequence of periodic executions (*i.e.*, ③). Once a cycle-accurate model is transformed into an FMU, the `fmi2DoStep` primitive is called at each clock cycle. The same behavior is reproduced using S-Functions by defining an *output function*, specified by the `mdlOutputs` callback method. The output function takes the input values and the references of the output variables as parameters.

6.3.2 Monolithic model approach

Blocks (*i.e.*, FMU or S-Function) generated applying the mapping discussed above can be inserted in a data-flow model, such as the ones used in Simulink. Designers may be led to model a system using different HW IPs trivially connecting the generated blocks according to the structure in Figure 6.7, without any particular precaution. This may lead to errors: synchronization problems arise while trying to represent HW models using data-flow-based tools because of

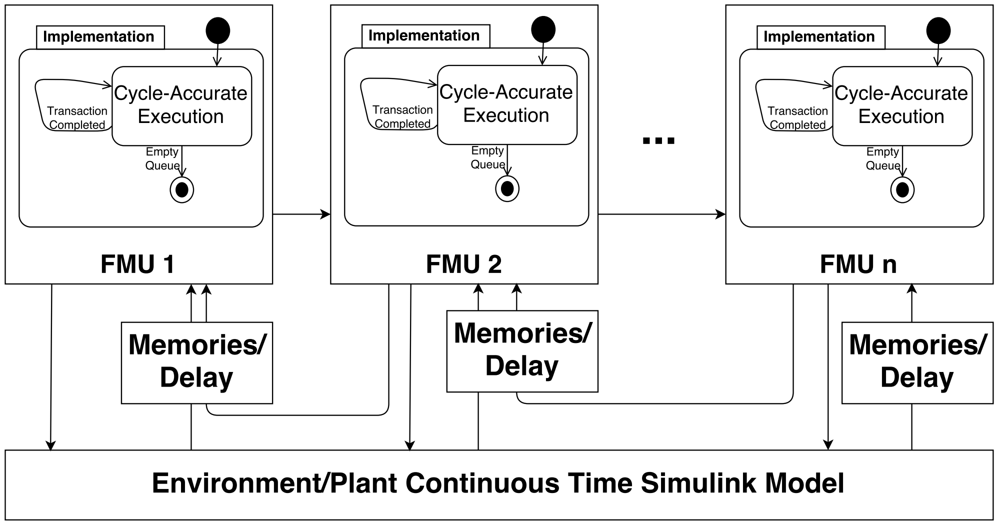


Fig. 6.7: Naïve structure of Simulink model with multiple computational components.

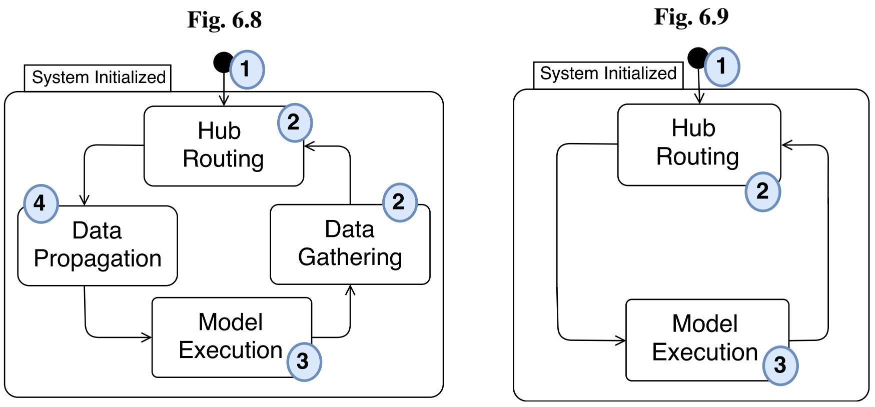


Fig. 6.10: State charts representing the FSMs used to manage synchronization locally to (a) FMUs and (b) S-functions.

the different execution semantics involved. Data-flow models are intrinsically sequential, while HDL semantics is concurrent.

A first approach delegates integration to the abstraction procedure applied to create cycle-accurate models. It schedules, and thus integrates, any set of synchronous and asynchronous processes of an HDL hierarchical model. Thus, different HW IPs can be hierarchically composed into a unique HDL description and abstracted into its equivalent cycle-accurate model. Finally, it undergoes the mapping proposed above.

This solution manages the synchronization and communication between digital HW components internally to the monolithic model of the HW platform. Unfortunately, it requires defining the entire HW architecture of the system before its integration in the cyber-physical virtual platform. Thus, the designer must sacrifice the possibility to replace components freely.

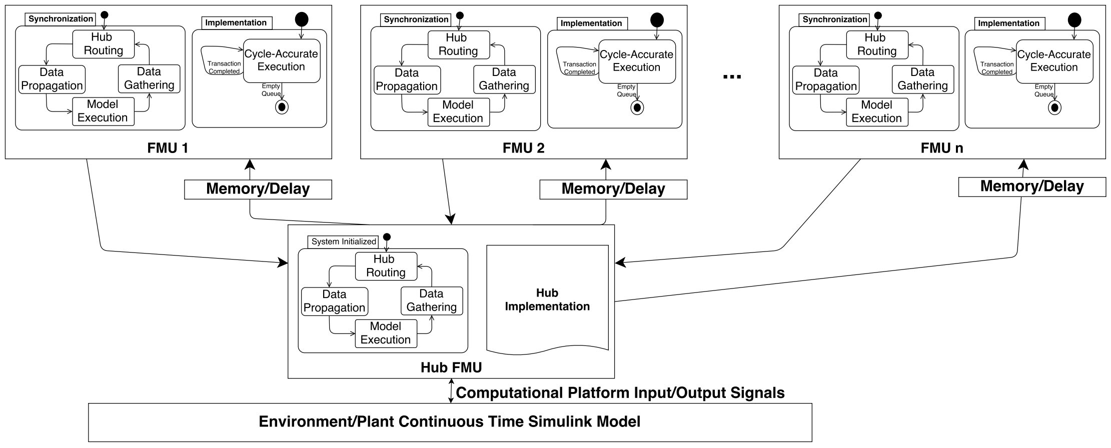


Fig. 6.11: Structure of the simulation environment combining the continuous-time model of the physical plant and a computational infrastructure composed by multiple FMUs.

6.3.3 Hub-based approach

The second solution reconciles the models of computation involved, by enabling concurrent execution of components within data-flows. The main idea is to decouple IPs execution from their synchronization and communication. IPs must read their inputs and perform one execution step to produce their outputs before to exchange data with each other.

All the clusters of communicating IPs are identified. **Communication, synchronization, and data exchange are managed globally** for each cluster by using a **dedicated module called Hub**. Then, the **synchronization is managed locally to each module** by using a **Finite State Machine (FSM)**.

Two situations may occur while integrating modules into a CPS simulator using integration technologies when considering the algorithm regulating the input-reading, execution and output-writing phases (*e.g.*, the FMI master algorithm for co-simulation): the algorithm can be fixed or it can be configurable. If the algorithm is fixed, all the possible sequences of primitive calls must be managed. Otherwise, some optimization may be possible. Figure 6.10 shows the FSMs managing the two cases. The machine in Figure 6.8 is used when the scheduler is fixed, the machine in Figure 6.9 is used otherwise. The reference FMI standard toolbox available for Simulink [18] fixes the sequence of actions for each simulation cycle to *execution, output, input*: thus requiring the first solution. S-Functions grant more freedom: the solely “output function” is executed at each simulation cycle. It takes care internally of reading input, executing and writing output: it thus allows to use the second solution.

The states in Figure 6.10 are hereby described referring to the mapping defined above for the FMI standard:

- *Hub Routing*: the Hub updates its output variables using the values of its input variables read at the previous execution: it mimics the signals propagation defined when binding signals in RTL models. FMUs implementing components of the platform do not perform any computation.

- *Data Propagation*: the values written by the hub are stored into Simulink memory units, allowing to break algebraic loops. FMUs implementing components and the hub are not performing any computation.
- *Model Execution*: the FMUs modeling components execute one simulation cycle, write their output variables and update their input values reading memories output values.
- *Data Gathering*: the hub updates its inputs reading values from the components FMUs and the Simulink model.

Circled numbers in Figure 6.10 highlight relations between FSM states and the execution schemes to reproduce (Figure 6.5). *Data Propagation* (④) has no correspondence in previous schemes being an artifact stalling the execution to assure correct values propagation. The 4-state FSM can manage the most critical family of scheduling sequences allowed by the FMI standard, *i.e.*, simulation cycles with input reading operations scheduled after the model execution.

On the other hand, if the scheduler is not using a critical sequence it is possible to use the 2-state FSM. The latter is the case when executing S-Functions where we are free to impose input-reading as the initial action of each simulation step.

Each module (*i.e.*, FMU or S-Function) contains both its cycle-accurate implementation and the chosen FSM. It is now possible to define a schematic structure overcoming problems emerged using the structure in Figure 6.7. Multiple modules representing different IPs are connected to each other as depicted in Figure 6.11. Such a structure is required by the Hub-based approach to faithfully reproduce the communication scheme between multiple HDL models. All the input or output ports of modules produced from IPs are connected to the module implementing the hub. The latter takes care of propagating values from each module to the others, and from the modules implementing the HW infrastructure of the system to the remainder of the model. Finally, the simulation step of each module is set to be equal to the device's *clock period divided by the number of states* in the FSM. In this way, the hub module "parallelizes" components' executions, while interfacing them to the sequential semantics of the model.

6.3.4 Alternatives Taxonomy

Table 6.2 summarizes the desired features supported by the different alternatives proposed by this paper. It compares them to the state-of-the-art, (*i.e.* token-based synchronization methodology [76]). The approach found in the literature is thought to be used in a top-down scenario, where components reuse is not considered. Despite its portability, granted by the FMI Standard, it does not support automatic generation and seamless integration since it requires to enrich interfaces of discrete-event models adding event ports.

The other entries in the table point out the features of solutions combining the interfacing technologies and synchronization approaches proposed above. Contrary to S-Functions, FMI-based solutions are always portable. Both interfacing techniques, as well as both the synchronization approaches, allow component reuse and automatic model generation. The Hub-based approach provides seamless integration: it allows to easily modify the structure of the digital devices minimizing the number of model regenerations. The monolithic models require to regenerate the module at each change of the HW architecture. As such, the former is more suitable

Table 6.2: Comparison between feature of the state-of-the-art [76] methodology and the proposed approaches.

Interface Technology	Synch. Approach	Components Reuse	Automatic generation	Seamless integration	Portability
FMI	Token-based [76]	×	×	×	✓
	Monolithic Model	✓	✓	×	✓
	Hub-based	✓	✓	✓	✓
S-Functions	Monolithic Model	✓	✓	×	×
	Hub-based	✓	✓	✓	×

for design-space exploration. However, Section 6.4 will show that monolithic models provide faster simulation.

6.4 Methodology Application

The methodology has been implemented in a tool on top of HIFSuite [44] to exploit its already available abstraction features [20]. The tool has been applied to a case study composed of a physical plant controlled by a SW running on a CPU. The case study has been composed to be representative of many CPS families, and Figure 6.12 shows its structure.

Our approaches have been used to integrate accurate models of the digital components into Simulink, building a cycle-accurate cyber-physical virtual platform. Then, we use this case study to evaluate the efficiency of the proposed methodology regarding simulation speed, while the effectiveness of the methodology is shown by exploiting a cycle-accurate virtual platform to perform some design-space exploration steps.

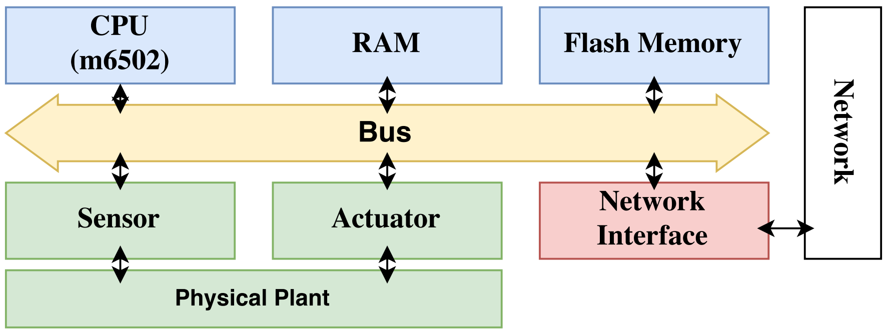


Fig. 6.12: Architecture of the CPS used as case study. Colors represents the original modeling languages of the components: blue for Verilog, yellow for VHDL, green for Simulink, while red closed source HDL IPs.

Table 6.3: Simulation overhead necessary to simulate one minute of the system by using different modeling alternatives.

#	Interface Technology	Synchronization Methodology	Configuration	Execution Time (s)
0	–	–	Stateflow controller in Simulink model	0.32
1	S-Functions	Hub-based	4 C MEX Compiled S-Functions	4.29
2			3 C MEX Compiled S-Functions	3.78
3			2 C MEX Compiled S-Functions	3.63
4		Monolithic Model	1 C MEX Compiled S-Function	1.98
5	FMI	Token-based [76]	4 FMUs	10.74
6			3 FMUs	8.88
7			2 FMUs	5.16
8		Hub-based	4 FMUs	15.94
9			3 FMUs	13.14
10			2 FMUs	11.90
11		Monolithic Model	1 FMU	2.61

6.4.1 Simulation performance

Last column of Table 6.3 reports the time (seconds) required to simulate 1 second of the real system by using different alternatives of the virtual platform. The first column enumerates scenarios. All the simulations have been executed on an Intel i7-3770 CPU at 3.40GHz and 16 GB of DDR3 Ram Machine running Linux Ubuntu 16.04 and Mathworks' Simulink 8.8.

Simulation time is obviously minimized when modeling the entire system within Simulink (Scenario 0). However, Section 6.4.2 will point out how this model lacks in accuracy.

Scenarios 5 to 7 relies on the FMI standard and the state-of-the-art synchronization methodology in [76]. They can be compared to scenarios implementing the hub-based synchronization (Scenarios 1 to 3 using S-Functions and 8 to 10 using FMI). S-Functions always outperform both alternatives. The token-based approach is slightly faster than the FMI-implemented hub-based technique. However, [76] is purely top-down, and it does not easily support reuse of components. Moreover, generation of FMUs cannot be automatized, depending on FMUs instantiation. All the approaches we proposed are instead thought to maximize component reuse, automatic generation, and integration.

Scalability is analyzed by comparing Scenarios 1, 5 and 8, respectively to Scenarios 3, 7 and 10. The analysis is shown in Figure 6.13 reporting the simulation time of each approach (y-axis), given the number of instantiated units (x-axis). The constant overhead required by each

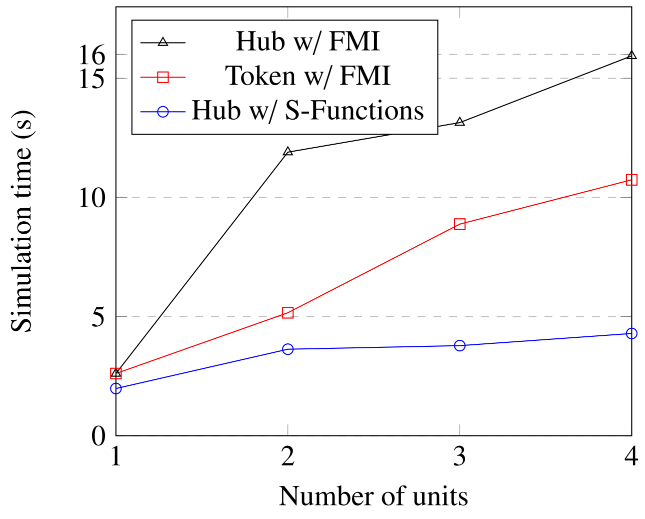


Fig. 6.13: Scalability of the proposed approaches.

approach must be evaluated when two units are instantiated. The *Hub-based* synchronization has a heavier constant overhead *w.r.t.* to [76] when using the same interface technology, *i.e.* FMUs. Nonetheless, the *Token-based* approach overhead grows faster adding units: from 5.16 to 10.74 seconds (2.08x) for [76], from 11.90 to 15.94 seconds (1.33x) for the Hub-based approach using FMI: the Hub-based approach seems to be more suitable when virtual platforms are composed of many digital modules. The Hub-based synchronization implemented using S-Functions outperforms both the others. S-Functions introduce lighter overhead than FMUs. Furthermore, doubling the number of units the simulation time grows slower: from 3.63 (Scenario 3) to 4.29 seconds (Scenario 1) (1.18x).

Finally, Scenarios 4 and 11 synchronize IPs by exploiting automatic abstraction and integration. They provide the fastest virtual platforms and they are the best choice to effectively execute SW and not to explore alternative platforms.

6.4.2 Design Space Exploration

Figure 6.19 summarizes the development and refinement steps carried on to obtain the controller HW/SW implementation. It aims at showing the benefits of embedding cycle-accurate models within cyber-physical virtual platforms:

- The system is originally modeled by using Simulink (Scenario 0 of Table 6.3). The HW/SW controller is modeled using Stateflow. Figure 6.14 shows the model's time evolution. The Controller easily manages the system leading to stability after around 450 seconds.
- The model is refined by introducing the FMUs implementing the cycle-accurate models of HW components. A SW running on the CPU implements the controller. HW resources are very limited: fixed-point arithmetic is not available, and the bus causes a communication bottleneck. Figure 6.15 shows the new model evolution: the SW cannot produce the actuation signals in time to control the system due to architectural constraints.

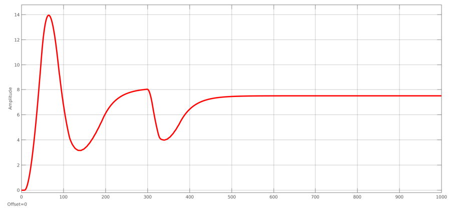


Fig. 6.14: Time evolution of the ideal Simulink model.

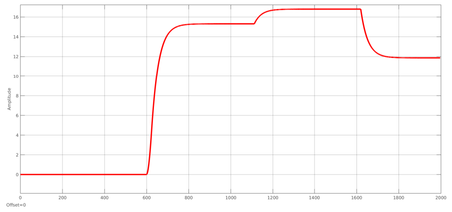


Fig. 6.15: HW-aware model evolution.

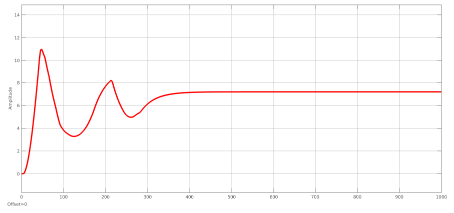


Fig. 6.16: Time evolution of the model after the architectural optimization.

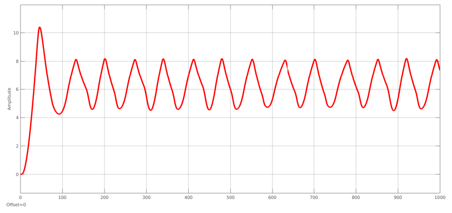


Fig. 6.17: Time evolution of the model using the Network Interface.

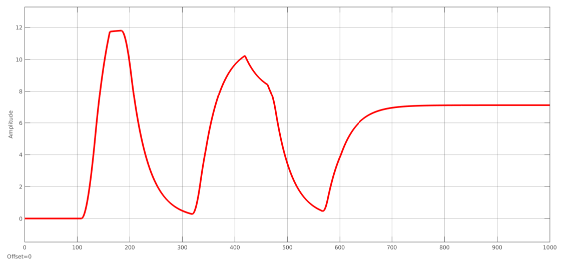


Fig. 6.18: Time evolution of the model after the SW optimization.

Fig. 6.19: Time evolution of the CPS model in the different phases of SW design exploration and optimization.

- Analyzing the previous model execution may help designers: SW-implemented multiplications was burdening the SW. As such, the designer inserts a further IP providing HW fixed-point arithmetic. Figure 6.16 shows the system reaching stability after 400 seconds.
- Previous models were not considering the Network Interface peripheral that requires a SW-implemented polling mechanism. The peripheral is imported as an FMU using the *Hub-based* mechanism. The SW is modified to send data through the network. Figure 6.17 shows that the polling mechanism introduces overhead causing the controller inability to control the system.
- The SW has been modified to run while the Network Interface is sending data. The previous model allows identifying the part of computation that can be executed while waiting for the peripheral to complete. In the evolution depicted in Figure 6.18 the system is now stable after 700 seconds.

The steps just presented show the importance of relying on accurate models when developing control SW for CPS. They highlight the positive impact of automatic integration of cycle-accurate models into cyber-physical virtual platforms.

6.5 Conclusions

This paper enables integration of cycle-accurate models within dynamic system simulators, to create cyber-physical virtual platforms. It proposes two integration solution to efficiently synchronize the discrete-event and data-flow models of computation. We evaluated the different techniques by using the same case study, highlighting for each alternative its advantages and drawbacks. This analysis may guide designers to choose the solution better suited to any design phase.

Experiments showed the benefits of using cyber-physical virtual platforms. In particular, a case study clearly shows the importance of using cycle-accurate models when developing timed-constrained control SW for CPSs.

7

Improving FMI-based simulation by Exploiting System–level Information

7.1 Introduction

CPSs are complex, heterogeneous systems that interact with the physical environment, under the control of complex software. Thus, their correctness depends on either their physical part as well as the software controlling them [85]. In this context, *virtual platforms* play a crucial role during the system design flow. A virtual platform [73] is an executable model able to emulate the behavior of the system for which a piece of software is being written. Traditionally, they are used for classical embedded systems to emulate the behavior of a hardware platform running the software being developed. In the case of digital devices, virtual platforms are already a widely mature technology, and a well-known practice [86]. Unfortunately, the heterogeneity characterizing CPSs makes way more difficult building virtual platforms providing efficient system-level simulation [87].

In fact, emulating heterogeneous systems requires the integration of different simulation paradigms. This as been so far achieved by developing, either academic [88] and commercial [18], heterogeneous modeling and simulation tools. However, these tools are strongly oriented to strictly top-down design flow, where abstract models are produced for the entire system, and then refined into implementations [89]. Thus, designers are forced to rely on abstract models as starting point of the design, hindering the possibility to re-use previously designed and validated implementations [87]. This is particularly limiting when designing the digital HW/SW platforms embedded in the CPS being developed. In fact, a very powerful and efficient practice in the design of embedded devices proved to be the re-use and integration [90] of previously designed, validated and tested components, *i.e.*, IP modules or IPs.

Integration of reused IPs and high-level physical models for simulation, is crucial to improve design of CPSs and it is the target of this work as summarized in Figure 7.1. This paper proposes a set of methodologies to build cyber-physical virtual platforms integrating abstract models of the physical parts of the system, and digital subsystems modeled by reusing hardware components' models. The IPs components composing the digital subsystem are abstracted and integrated to build a virtual platform through automatic code generation. The generated code is enriched to provide a standardized interface that can be used to import the model into system-

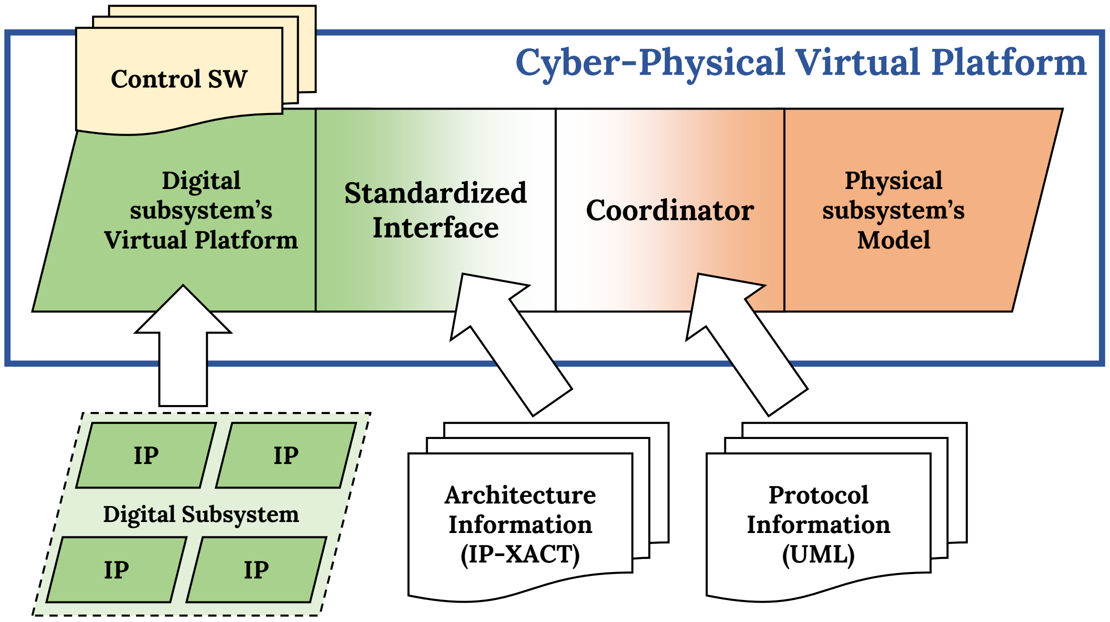

Cyber-Physical Virtual Platform
Control SW
Digital subsystem's Virtual Platform
Standardized Interface
Coordinator
Physical subsystem's Model
IP
IP
Digital Subsystem
IP
IP
Architecture Information (IP-XACT)
Protocol Information (UML)

Fig. 7.1: Target simulation environment produced by this work. A cyber-physical virtual platforms allowing the simulation of complex CPS, while allowing integration of previously designed IP components. The cyber-physical virtual platforms must allow running control software for validation purposes. Architectural and protocol informations are used to automatically generate standardized interfaces, and the system-level coordinator enabling heterogenous co-simulation.

level simulators. This work targets both the generation of portable interfaces compliant with the *FMI standard*.

In our previous work [46,55], we introduced the automatic code generation of cyber-physical virtual platforms. In this paper, we build a structured methodology allowing to apply optimizations to the virtual platforms by exploiting system–level information. In particular, the optimizations are enabled by providing information about the architecture and the communication protocols of the system. For each available piece of information, this paper shows the enabled optimizations. Information stored into *IP-XACT models* of the architecture will allow to optimize the automatically generated standardized interfaces of the components. Meanwhile, protocol information modeled as *UML timing diagrams* is used to optimize the coordination between the digital and the physical subsystems. Therefore, the virtual platforms produced by the presented approach are both able to emulate accurately the entire system, and allow reuse of components. Thus, providing a crucial tool for the early validation of control software.

The following of the paper is organized as follow. Section 7.2 introduces the background and the related work. Section 7.3 provides the overview of the different steps of the contribution, while describing their connections. Furthermore, it introduces a case study used through the paper as a running example to describe the different steps. The steps are then deeply described in Sections 7.4 to 7.6. Finally,Section 7.7 presents an experimental evaluation of the contribution.

7.2 Preliminaries

This section first introduce the interfaces available to build system-level executable models for CPSs, and it will specifically focus on the details of the FMI standard. Then, it recalls the main concepts about the specification languages typically involved in the design of hardware platforms, and about the automatic abstraction techniques for hardware descriptions. Finally, this section summarizes the major results related to this work available in the literature, highlighting the novelty of the presented work.

7.2.1 Standardized interfaces and the FMI standard

The ability to interface different models is crucial for effective system–level design. For this reason, many co-simulation technologies have been proposed in the last decades [91]. However, most of the time, co-simulation environments relied on interfaces tailored on the specific simulation tools being interfaced. Custom co-simulation interfaces come with two major drawbacks. On one hand, designers are forced to either use tools for which they already developed interfaces, or to put effort in developing new interfaces. On the other hand, custom interfaces were developed with weak formal support and thus, they do not provide strong correctness assurance.

In this context, the FMI standard has been proposed to enhance the interoperability between tools of different vendors for systems design [21]. It aims at providing support for model exchange and co-simulation of dynamic models that may be produced by using different tools and languages.

FMI-based simulation environment is a composition of FMUs. Multiple FMUs can be imported within a simulation tool to be executed. Each FMU must implement one of the two interfaces defined by the current standard: *Model Exchange* or *Co-Simulation*. Both types of FMU must contain an XML file describing the interface of the component modeled within the FMU, and the model of its functionalities. The main differences between the two types of FMU lay in the latter. Model exchange FMUs describe functionalities by using differential, algebraic and discrete equations with time-, state- and step-events. The behavior specified in a model exchange FMUs is reproduced by using external solver. On the contrary, co-simulation FMU must contain within the model both the functionality and implementation of the solver. As such, co-simulation FMU must contain an executable specification of the component being modeled.

The FMI standard has been originally developed by a community mostly focused on the physical side of CPS. For this reason, at its latest release, the standard is better tailored for expressing continuous-time behavior rather than discrete systems. This is particularly evident in the standard for model exchange. However, as the co-simulation standard requires to embed solvers within FMUs, it is better suited to support discrete-time models. For this reason, in this work we focus on the co-simulation FMI standard, which main features are described hereby.

7.2.2 FMI Standard for co-simulation

An FMU for co-simulation usually models a system component. It must contain an XML file describing the component interface, and the implementation of its functionality as a dynamic library written in C, implementing both the model of the functionalities and the solver necessary to execute the model. The XML file must specify all the variables and parameters of the model that the FMU makes visible to the simulation environment [21]. For each variable, it specify its *name*, *causality* (*e.g.*, input, output, parameter, *etc.*), its *type* and a *value reference* (*i.e.*, a variable identifier that must be unique among the variable of the same time). The simulator identifies each variable by its type and value reference pair.

The dynamic library must implement the set of functions defined by the standard. In other words, the standard for co-simulation defines a set of function signatures that must be implemented within the dynamic library contained in the FMU. The functions most relevant for this work are:

- `fmi2SetupExperiment`: initializes the internal variables of the FMU.
- `fmi2Set`: sets the value of an internal variable of the FMU *i.e.*, it assigns a value to an input.
- `fmi2Get`: gets the value of an internal variable of the FMU *i.e.*, it returns the value of an output.
- `fmi2DoStep`: advances the simulation time of the component executing the behavior defined by the model.

While the standard defines the signature of all the functions to implement, it does not imposes their usage. However, the standard defines some limitations on the order in which the functions can be invoked by the simulator.

7.2.3 Simulation coordination in the FMI standard

A FMU, regardless whether it is a model exchange or co-simulation FMU, to be simulated must be loaded by a coordinator managing the communication and the synchronization between FMUs. Coordinators must implement communication and synchronization mechanisms compliant with the FMI standard, which defines [21] the concept of *master algorithm* as the actor in charge of exchanging data between FMUs and synchronizing the simulation of the involved solvers. The `fmi2Get` and `fmi2Set` functions are invoked by the master algorithm to read and write data to the FMU. Meanwhile, the `fmi2DoStep` functions of the FMUs in the system are invoked to advance the simulation. The exact definition of the master algorithm is not part of the standard which only imposes some limitations to the order of functions to be called.

The master algorithm initialize the FMUs in the system by calling their `fmi2SetupExperiment` functions. Then for each initialized FMU, the algorithm follows the execution defined by the statechart in Figure 7.2. The execution starts from the *Step Completed* atomic state within the *State Initialized* sub-machine. The master algorithm may invoke the `fmi2Get` or the `fmi2Set` functions to read or write values of the FMU external variables, or the `fmi2DoStep` function to execute a simulation step of the FMU. The standard forbids to call a `fmi2Get` function immediately after calling the `fmi2Set` function. When calling the `fmi2DoStep` function, the algorithm passes the amount of time to be simulated as a parameter. The machine moves to the *Step in Progress* state and the FMU executes its functionality. Whenever the step is not canceled nor discarded, and

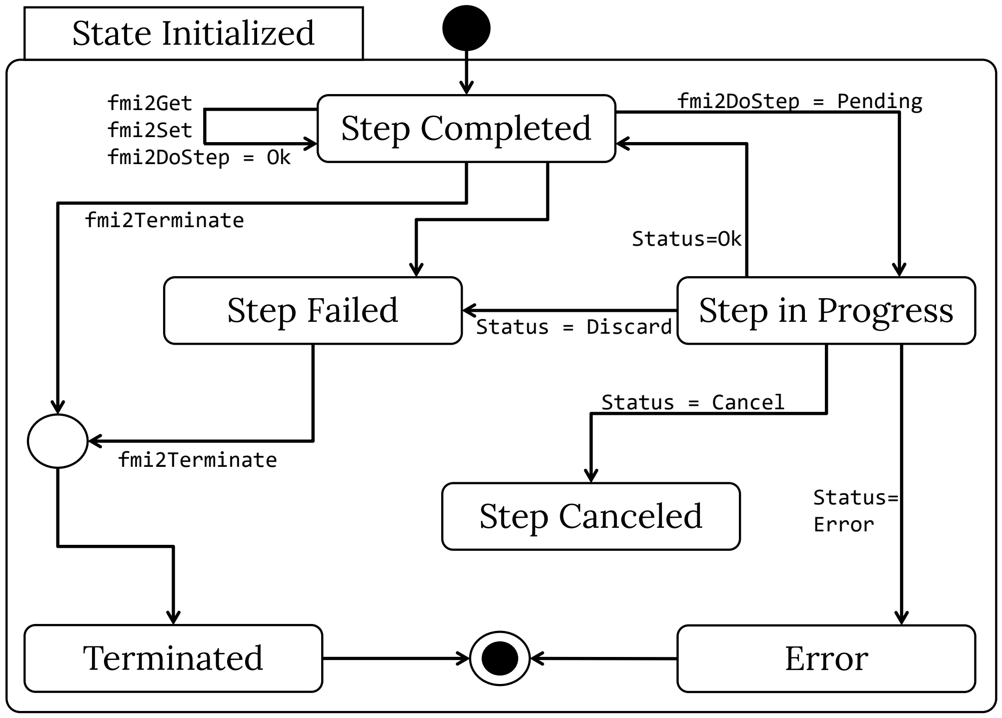


Fig. 7.2: Statechart of the FMI standard's master algorithm.

whenever no errors happens during the execution, execution goes back to the *Step Completed* state and the FMU advances its own local time. While the `fmi2Terminate` function is not invoke, the execution continues indefinitely.

A simulation tool may implement a coordinator for FMI executing in any order, as long as it complies with the rules summarized above. Thus, the standard allows many different master algorithms, each of them providing different executions.

7.2.4 Specification Languages for Hardware Platforms

A large variety of languages have been produced to model HW/SW platforms [87]. Each of them, specialized in capturing different views and aspects of the design [92]. In this work, we rely on IP-XACT [93] and SysML [94] to specify, respectively, the architecture and the communication protocol of a HW platform.

Platform specification using IP-XACT

IP-XACT [93] is a standard (IEEE P1685) defining a XML-based format for describing HW IPs to support the design of digital platforms and System on a Chips (SoCs).

An IP-XACT *component* describes one or more implementations of an IP. Each IP is identified by four elements: *vendor*, *library*, *name*, and *version* which compose the IP's *VLNV identifier* used by IP-XACT editors and tools to uniquely identify IPs. A component model the IP by describing its interface. Thus, it lists the component ports. Each *port* is characterized by a name and it may be either a *wire* in the case the IP is express at RTL, or *transactional* is used by a transactional model. A special type of IP-XACT component is a *Bus definition*. It defines a set of interfaces and communication protocols.

IP-XACT supports compositionality and hierarchy, as a *design* is specified as a (potentially) hierarchical assembly of components. Components are connected by connecting their ports, or their transactional interfaces. As such, a IP-XACT design models the architecture of a system built by assembling, and usually reusing, hw IPs. In order to enhance reuse, it provides many constructs to support multiple IP configurations and abstractions. Furthermore, the standard provides a native extension mechanism that allowed to extend the language to support reconfigurable computing [95], extra-functional characterization of IPs [96], and modeling of software features [97].

These features gave the IP-XACT standard vast popularity among HW/SW systems designers, to the point that Xilinx's Vivado design suite relies entirely on IP-XACT to store and track projects of HW/SW platforms and IPs [98]. In many industrial context, IP-XACT have been used to produce new systems integrating IPs [99], or to track data throughout the production process [100].

Timing diagrams for protocol specification

Timing diagrams are a type of *interaction specification diagram*, showing the timing of the different interactions between components, and events in a system [101]. They have been widely used in hardware design, and they are the primary feedback of HDL simulation. Timing diagrams express the timing correspondence between events (*i.e.*, changes of values) in the system. As such, since when digital design moved from to chips to SoC, timing diagrams are largely used to specify protocols.

Many different formats of timing diagrams have been proposed by different tools. In this work, we rely on UML timing diagrams: they supports all the major features of timing diagrams, while at the same time they have an open specification, as well as usable editors and a well-defined XML schema for importing and exporting diagrams into different tools.

7.2.5 Automatic abstraction of digital components

Fast simulation of HDL models is crucial to build effective system design methodologies. The effort put by researchers to speed up HDL simulation resulted into highly effective abstraction and code generation. Usually, HDL models are translated into functionally equivalent, highly efficient C/C++ models [43, 102]. Meanwhile, the most effective approaches abstract models from the cycle-accurate to the transaction-level [42], before generating C/C++ executable models. This work relies [20] to translate and abstract HDL descriptions into the C++ models required in the proposed methodology. Listing 7.2 shows the skeleton of the C++ code generated to emulate the RTL behavior in Listing 7.1, which is the skeleton of a generic Verilog module. To produce Listing 7.2 from Listing 7.1, the approach in [20] works as follow:

- the input HDL description is parsed to extract its digital processes, the dependencies among processes (*e.g.*, sensitivity lists, signal writing and reading, *etc.*), identifying both synchronous and asynchronous behaviors.

Listing 7.1: Generic verilog component with Amba interface.

```
module component(    // AMBA Interface
    input pclk,
    input presetn,
    input [31:0] paddr,
    input psel,
    input penable,
    input pwrite,
    input [31:0] pwdata,
    output reg pready,
    output reg [31:0] prdata);

integer state, next_state;

always @ (posedge pclk or negedge presetn) begin
    if( presetn == 1'b0 ) state <= state_reset;
    else begin
        // Combinational implementation
    end
end

always @ ( state ) begin
    case( state )
        // FSM implementation.
    endcase
end
endmodule
```

- the parsing of the HDL model produces a dependencies graph. The graph is used to define a process scheduling able to reproduce the cycle-accurate behavior of the original HDL description. The scheduling starts from synchronous processes and proceeds considering the dependencies of already scheduled processes. The methodology assumes synthesizable RTL models as input descriptions, thus guaranteeing the absence of cycles in the dependency graph. The scheduling and the processes build an externally synchronous model reproducing asynchronicity internally.
- the user may electively choose to abstract the communication protocol of the model. If so, the RTL protocol is abstracted into a Transaction-level protocol [42]. However, this step requires the specification of the component protocol. Otherwise, the model is intepreted as cycle-accurate.
- The model is translated into C++. Each process is translated into an C++ function; replicated variables, control flags and supporting functions implements the scheduling mimicking concurrent behavior of HW. Each transaction starts executing `simulate` function to which is passed a pointer to a payload data structure (*i.e.*, `component_iostruct`). The structure is composed by a field for each input or output port of the original model. Each simulation cycle starts by populating the input/output data structure, then the simulation function is called and at its completion the data structure is read.

Listing 7.2: C++ processes scheduler automatically generated to emulate the RTL behavior of the verilog model in Listing 7.1.

```
void component::simulate(component_iostruct * io_exchange){
    input_phase(io_exchange);
    synch_elaboration();
    while (process_in_queue) {
        flag_elaboration();
        update_event_queue();
    }
    output_phase(io_exchange);
}

void component::synch_elaboration() {
    process(); // Combinational implementation
    flag_elaboration();
    flag_neg_reset = false;
    flag_in = false;
}

void component::flag_elaboration() {
    if (flag_neg_reset) process();
    if (flag_state)
        process_0(); // FSM implementation
    if (flag_result_out_sig) result_update_process();
}

void component::update_event_queue() {
    process_in_queue = false;
    next_state = next_state_new;
    if ( result_out_sig != result_out_sig_new) {
        out_sig = out_sig_new;
        flag_out_sig = true;
        process_in_queue = true;
    } else flag_result_out_sig = false;
    if ( state != state_new) {
        state = state_new;
        flag_state = true;
        process_in_queue = true;
    } else flag_state = false;
}
```

This work enriches the code generation step to embed the generated models within an FMU. Thus, making it FMI-compliant.

7.3 Methodology Overview

The main target of this work is to exploits as much as possible the available system-level information, to automatically generate highly optimized virtual platforms for CPSs. Thus, for each piece of information available we aim at systematically improving the virtual platform by maximizing the simulation speed.

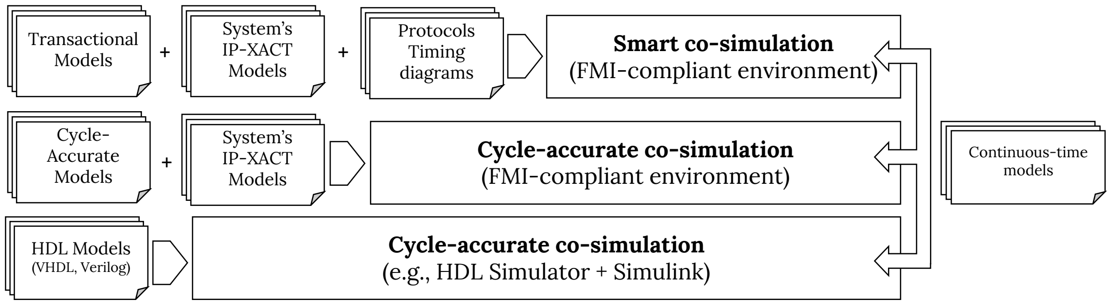

Transactional Models
+
System's IP-XACT Models
+
Protocols Timing diagrams
Smart co-simulation
(FMI-compliant environment)
Cycle-Accurate Models
+
System's IP-XACT Models
Cycle-accurate co-simulation
(FMI-compliant environment)
HDL Models
(VHDL, Verilog)
Cycle-accurate co-simulation
(e.g., HDL Simulator + Simulink)
Continuous-time models

Fig. 7.3: Overview of the Contribution. The paper proposes to exploit system-level information to build FMI-based alternatives to build cyber-physical virtual platforms overcoming the limitation of typical cycle-accurate co-simulation elying on the co-execution of HDL and dynamical systems simulators. From the bottom, up to the top, co-simulation environment exploits an increasing amount of system-level information and, consiquently, improving the simulation performance.

Figure 7.3 summarizes the increasing amount of system-level information exploited by this work to build more and more efficient virtual prototypes of a CPS. Notice, this work focuses on the *cyber* part of the system (*i.e.*, the embedded device). As such, we hypothesize the continuous-time models of the *physical* part of the system are given. From the bottom to the top, the device is specified by its components modeled by using any HDL. In such case, a virtual-platform is built by establishing a co-simulation scenario using a HDL simulator able to communicate with a system simulator able to execute the models of the physical part of the system, such as Mathwork's Simulink. In such a case, as the simulation of the HDL models is cycle-accurate, so will be the co-simulation. Indeed, the integration of the components, as well as the integration between the *cyber* and *physical* parts is going to be performed manually by the designer. Such a scenario, represents the current state of the practice when building virtual-platforms for CPSs.

Exploiting state-of-the-art translation methodologies, HDL models can be translated into cycle-accurate C++ models. Section 7.4 presents the methodology to automatically wrap cycle-accurate C++ models to build cycle-accurate FMUs. Such FMUs can be integrated manually by the designer in any FMI compliant simulator. However, further system-level information allows making the integration step automatic. In particular, whenever the IP-XACT model of the device is available, Section 7.6 will show how to automatically generate the interconnections and the coordinator necessary to integrate the set of cycle-accurate FMU emulating the different components of the system. Thus, having both the cycle-accurate models and the IP-XACT models of the device, the proposed methodologies allow assembling a complete cycle-accurate virtual-platform entirely based on FMI.

Finally, the designer may be provided with also the timing diagrams describing the communication protocols necessary to implement every permitted operation by the device. This will allows to produce transactional models of the components exploiting automatic abstraction. Section 7.5 presents our approach to generate transactional FMU, also able to provide temporal decoupling for faster simulation. In this case, we propose to automatically generates a virtual platform composed of transactional FMUs, which simulation is governed by a *smart coordinator*, *i.e.*, a coordinator ad-hoc for the specific system being designed. Indeed, such a

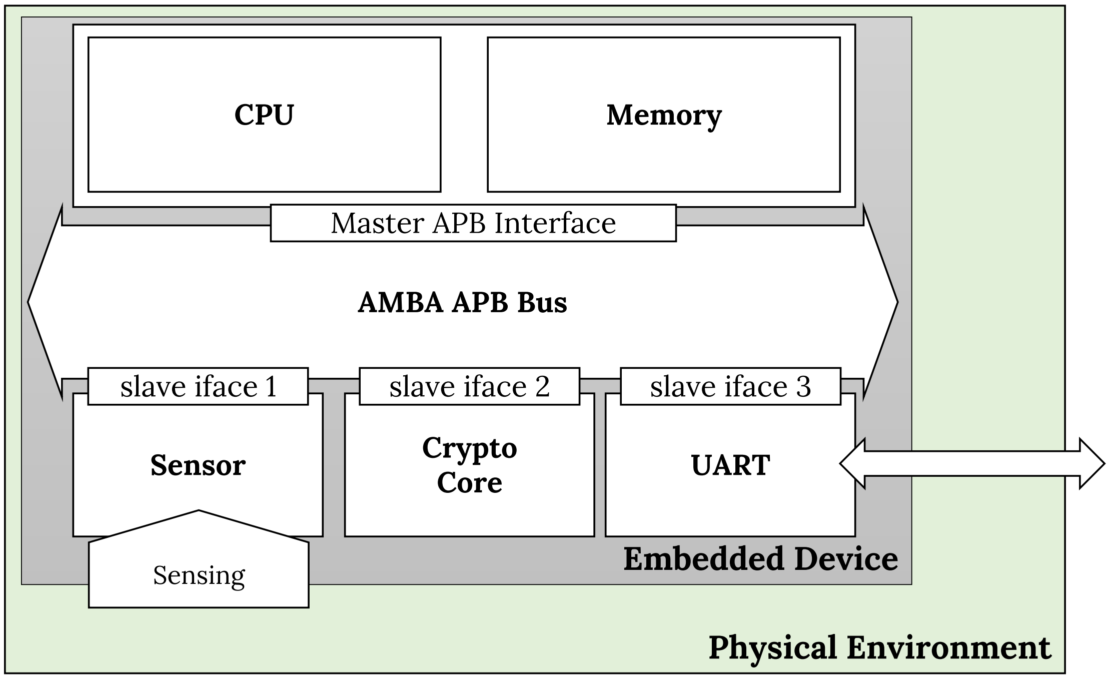


Fig. 7.4: Case study used throughout the paper.

coordinator will provide maximal efficiency during the simulation. The automatic generation of such a smart coordinator is presented in Section 7.6. As we explicitly target the standard FMI, the smart coordinator will be the master algorithm governing the simulation of the FMUs modeling the system.

The following of the paper provides the details of the different steps of the methodology introduced in this Section. For the sake of clarity, the steps descriptions are accompanied by their application to the following case study.

7.3.1 Running Example

The explanation of the methodology is paired with its application to a case study acting as running example throughout the paper. Figure 7.4 summarizes the structure of the system used as case study. It is a digital device, embedded within a physical environment to be monitored. The system is composed by a subsystem composed of a general purpose CPU dialoging with a memory to execute the software loaded in the memory. The CPU and memory subsystem is connected to a bus implementing the ARM Peripherals Bus (APB) protocol. The CPU and memory subsystem acts as master for the bus communication. The Bus is connected to three peripherals, *i.e.*, a sensor sensing signals from the environment; a cryptographical core encrypting the data sensed by the sensor; a Unified Asynchronous Receiver-Transmitter (UART), sending the encrypted data to the external world through a serial port.

The components have different features, as well as different communication and synchronization patterns:

Table 7.1: Mapping of HDL events onto FMI primitives.

Hardware simulation Events	**FMI Standard** Primitives
Initialization	Sequence of assignments in initialization mode
Simulation cycle execution and time progress	`fmi2DoStep`
Input signals reading	`fmi2SetInteger` and `fmi2SetBoolean`
Output signals writing	`fmi2GetInteger` and `fmi2GetBoolean`

- the *CPU and memory subsystem* fetches, decondes and executes the operational codes stored in the memory, and representing the software being executed. For this reason, CPU and memory must synchronize at every clock cycle. As such, at every level of abstraction, they must be executed at each simulated clock cycle in order to preserve functional equivalence.
- The *sensor* samples the environment periodically, and it also periodically quantize and send the data to the CPU and memory subsystem. As such, its latency is fixed, but larger than one clock cycle and the communication and synchronization happens periodically.
- The *cryptographical core* latency depends on the input data it receives. As such, its latency and communication patterns may vary unpredictably. When it completes an operation, it sends an interrupt to the CPU and memory subsystem.
- The *UART* component may operate in different modes. Such modes are set by the CPU that sets the UART internal registers through the bus. As such, the latency is variable but predictable for each operation.

In the case study co-exist different execution patterns, *i.e.*, periodic with single clock latency, periodic with multiple clock latency, variable and unpredictable latency, and finally variable but predictable latency. The case study has been built to be representative for many systems. For instance, we avoided using a fixed-latency cryptographical core, as it would have made the application of the proposed methodology trivial. The HDL implementations of the hardware IPs follows the same structure with the verilog generic component in Listing 7.1. They are characterized by both synchronous and asynchronous processes, as well as asynchronous signals with different widths. Thus, for the sake of compactness, while presenting the methodology we will use Listing 7.1 as a generalization of any HDL model in the case study.

7.4 Generation of Cycle-Accurate FMUs

HDLs constructs are meant to describe hardware simulation events. These primitives may represent internal events of the device, or interface-level events of the IP. The internal events are managed by the state-of-the-art automatic abstraction and translation procedure in Section 7.2.5. On

Table 7.2: Mapping of HDL data-types to the FMI Standard.

Hardware-specific data-types	**C/C++** data-types	**FMI Standard** data-types
`Boolean`, `Bit`, `Logic`	`bool`	`Boolean`
`Unsigned`, `Bit Vector`, `Logic Vector`	`uint64_t`, `uint32_t`, `uint16_t`, or `uint8_t`	`Integer`

the other hand, we must handle the external events described in the model to make them manageable by any FMI-compliant simulator. These events are the model initialization, the input reading or output writing operations and the execution of a simulation cycle and time progress.

Table 7.1 reports the mapping of HDL simulation phases, limited to its external events, onto the corresponding primitives defined by the FMI standard. The IP initialization is realized as a set of assignments while the FMI is in initialization mode. The simulation, as well as the progress of time is implemented within the generated fmi2DoStep function. Finally, any operation related to input and output of values to and from the IP is mapped onto the fmi2Set and fmi2Get functions. The following of this section details how FMUs compliant with this mapping are built starting from the original HDL model.

7.4.1 Data-type mapping

The FMI standard provides the boolean and integer data-types, while HW-specific data-types (*e.g.*, many-valued logic, bit, logic vectors, *etc.*), and custom range integers are not supported. Integers are supported only as 32 bit signed values by the FMI standard API. These differences make necessary defining a precise mapping of data-types between the languages used to express the IPs and the FMI standard data-types. This mapping is summarized in Table 7.2. It is important considering that the HDL to C/C++ automatic translation (Section 7.2.5) already define the first step of the mapping, from HDL specific types to C/C++ native data-types, as reported in the second column of Table 7.2.

Boolean, single bit and multi-valued logic values are represented by boolean values (*i.e.*, `bool` in C/C++). Any signed is represented by a signed integer, while any unsigned, bit or logic vector is represented by an unsigned integer (*i.e.*, `uint64_t`, `uint32_t`, `uint16_t`, and `uint8_t`). The mapping relies on fixed width integers due to the possibility of explicitly choose the amount of bits (*i.e.*, type width) used for the representation. The specific type of integer chosen will be the one with the minimal width necessary to represent the original value. In the case the original IP has a port or signal whose type width is greater that 64, the IP undergoes an intermediate manipulation. The port or signal in the original model is splitted into multiple ports with a span not greater than 64 bit and then each port or signal introduced by this manipulation is mapped according to the corresponding type.

Listing 7.3: `modelDescription.xml` file generated from the generic component in Listing 7.1.

```xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <fmiModelDescription description="component design" fmiVersion="2.0" guid="f123-ab02" modelName="component">
3   <CoSimulation canBeInstantiatedOnlyOncePerProcess="false" canGetAndSetFMUstate="false">
4   <LogCategories>
5    <Category name="logAll"/>
6   </LogCategories>
7   <ModelVariables>
8    <ScalarVariable causality="input" description="int" name="pwdata" valueReference="0" variability="discrete">
9     <Integer max="2^32-1" min="0" start="0"/>
10   </ScalarVariable>
11   <ScalarVariable causality="input" description="int" name="paddr" valueReference="1" variability="discrete">
12    <Integer max="2^32-1" min="0" start="0"/>
13   </ScalarVariable>
14   <ScalarVariable causality="input" description="bool" name="pclk" valueReference="0" variability="discrete">
15    <Boolean start="false"/>
16   </ScalarVariable>
17   <ScalarVariable causality="output" description="int" name="prdata" valueReference="2" variability="discrete">
18    <Integer max="2^32-1" min="0" start="0"/>
19   </ScalarVariable>
20   ...
21  </ModelVariables>
22  <ModelStructure/>
23 </fmiModelDescription>
```

Afterward, the C/C++ data-types can be mapped into the data-types defined by the FMI standard. C/C++'s `bool` values are translated into the FMI's *Boolean* data-type. The C/C++ fixed with integers are mapped into the onto the `Integer` type. However, FMI supports only 32 bits integers in the interfaces definition. Thus, it is necessary to split into multiple 32 bit ports any port having a width higher than 32 bits.

Consider Listing 7.1, and its corresponding XML model description (Listing 7.6): lines 3–5 of Listing 7.1 defines the ports types. Single-bit multi-valued logic ports, such as the `reset` are mapped onto boolean variables to build the corresponding FMU (Listing 7.6, lines 14–15). Logic vectors, such as `in` and `out` are mapped onto Integer (Listing 7.6, lines 8–13 and 17–18). The `in` port's width is greater than 32-bit: it must be split into two different 32-bit variables:`in_1` and `in_2` (lines 8–13 in Listing 7.6).

7.4.2 Automatic generation of Functional Mockup Units

Any FMU specifies the input and output variables through an XML description, that can be automatically generated as follows:

- parse the input HDL model and identify its top-level design unit.
- Analyze the input and output ports specified in the top-level unit. Manipulate them and their types according to the mapping described above and in Table 7.2.
- Print the header part containing the name (*i.e.*, `modelName`), the identifier (*i.e.*, `guid`), and the co-simulation features the FMU provides.
- For each port, specify its causality (*i.e.*, input or output), its description, the port name, its value reference and its variability (*i.e.*, continue or discrete). Then, specify the port type through the correct XML tag, chosen according to the mapping described above. Finally, for each port, the parameter `starts` specifies its variable initial value.

Listing 7.6 reports the XML automatically generated from the generic component in Listing 7.1. Lines 1–6 are the header, while lines 7–20 list the interface variables of the model.

Listing 7.4: Skeleton of the FMU obtained from the generic verilog model, by wrapping the code in Listing 7.2.

```
 1 #include <fmi2Functions.h>
 2 #include "inc/component.hh"
 3  #define MODEL_GUID "352e3781-f5a3-4914-abd7-687397bff7fe"
 4 ...
 5 typedef struct ModelInstance{
 6     component * model;
 7     component::component_iostruct * iostruct;
 8     char * instanceName;
 9     int32_t cycle_number;
10     fmi2Real time;
11     ...
12 } ModelInstance;
13 ...
14 fmi2Status fmi2DoStep( fmi2Component c, fmi2Real currentCommunicationPoint, fmi2Real communicationStepSize)
15 {
16     ModelInstance * comp = ( ModelInstance *) c;
17     component * model = comp->model;
18     component::component_iostruct * iostruct = comp->iostruct;
19     model->simulate( iostruct, comp->cycle_number );
20     comp->time = comp->time + communicationStepSize;
21     return fmi2OK;
22 }
23 ...
24 fmi2Status fmi2GetInteger( fmi2Component c, fmi2ValueReference * vr, size_t nvr, fmi2Integer * value )
25 {
26     ... // Implementation of read port operations.
27 }
28 ...
29 fmi2Status fmi2SetInteger( fmi2Component c, fmi2ValueReference * vr, size_t nvr, fmi2Integer * value )
30 {
31     ... // Implementation of write port operations.
32 }
```

Variables `in_1` and `in_2` result from the manipulations of port `in`, due the 32-bit limitation imposed by the FMI standard. The clock variable is not present, as the clock has been abstracted to produce the starting C++ code in Listing 7.2. Each variable is uniquely identified by its type and its value reference pair. For this reason, different variables may have the same value reference if they belong to different types, as for the `in_1` and `reset` variables.

The methodology continues by manipulating the C++ code generated through automatic model abstraction and translation (*e.g.*, Listing 7.2). The generated C++ code is "wrapped" within the FMI standard functions. Listing 7.4 sketches the resulting code:

- A constant is defined with the value of the FMU's GUID (*i.e.*, `MODEL_GUID`). Any operation using the GUID of the model uses this constant (line 3).
- A C structure, called `ModelInstance`, acts as a container for the information supporting the model execution. It contains the pointer to the instance of the model implementation, the pointer to the input/output data-structure used to communicate, the local time and number of executed cycles (line 5 to 12).
- The `fmi2DoStep` function is implemented (line 14) taking care of simulating one execution cycle of the model and updating the FMU internal time (lines 14–22).
- The `fmi2Set` and `fmi2GetInteger` functions manage the input and output phases (lines 24–32). Listing 7.5 shows the implementation of the `fmi2SetInteger` function of Listing 7.4. `vr` is the array of size `nvr` containing the value references of the integer variables to set. The `value` array contains `nvr` integers that are the values to be set. The `for` loop (lines 6–17) sets the correct values to the specified variables. All the input and output functions in Listing 7.4 shares the same skeleton.

Listing 7.5: Skeleton of the `fmi2SetInteger` function implementation in the FMU in Listing 7.4.

```
fmi2Status fmi2SetInteger(fmi2Component c, fmi2ValueReference * vr, size_t nvr, fmi2Integer * value ) {
    ModelInstance * comp = ( ModelInstance *) c;
    component::component_iostruct * iostruct = comp->iostruct;
    size_t i = 0L;
    .... // Check for errors...
    for (i = 0L; i < nvr; i = i + 1L) {
        switch (( int32_t) (*(i + vr))) {
            case ((int32_t)0L):
                iostruct->pwdata = *(i + value);
                break;
            case ((int32_t)1L):
                iostruct->paddr = *(i + value);
                break;
            default:
                break;
        };
    }
    return fmi2OK;
}
```

The code must is used to produce a shared library whose Application Binary Interface (ABI) must be compatible with the C API. Thus, its linking must be compatible to the C linking and it must not perform names mangling. Finally, the shared library and the XML file are archived in a `.fmu` file that may be imported by any simulator supporting the FMI standard 2.0 for co-simulation.

7.5 Generation of Transactional FMUs

A major problem of FMI is the impossibility for the FMUs to propagate their local time back to the coordinator. Such an issue prevents the master algorithm from providing more efficient strategies for selecting the simulation steps lengths.

Hereby, we describe how to generate FMUs able to simulate in a decoupled way, without defining the simulation step size. Using such FMUs, whenever the Master Algorithm calls the `fmi2DoStep` to a FMU, this simulates while the component does not require synchronization or communication with the other components in the system. This simulation behavior is well-known in HW/SW co-design as it is typical of the transaction-level models. Thus, we call transactional FMU an FMU allowing to simulate a component functionalities without knowing “a priori” the step size, and consequently allowing decoupled simulation.

The first step to generate a transactional FMU from a HDL description of a component relies on a state-of-the-art RTL to transactional-level automatic abstraction technique [20].

The methodology generates Transaction-Level models from HW descriptions. To do so, we apply the methodology defined in [42]. It starts from HW models described at RTL and abstract them to TLM. The methodology supports the most common HDLs (*i.e.*, VHDL or Verilog). The protocol of a component can be specified in different ways. The state-of-the-art implementations of the RTL-to-TLM abstraction methodology relies on ad-hoc protocol specification languages [20]. The abstraction result is a C++ class representing a Transaction-Level model. Each transaction of the system is executed by invoking its `simulate` function which

Listing 7.6: `modelDescription.xml` file of the `component_1` with time port.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <fmiModelDescription description="component design" fmiVersion="2.0" guid="f123-ab02" modelName="component">
3    <CoSimulation canBeInstantiatedOnlyOncePerProcess="false" canGetAndSetFMUstate="false">
4    <LogCategories>
5      <Category name="logAll"/>
6    </LogCategories>
7    <ModelVariables>
8      <ScalarVariable causality="input" description="int" name="pwdata" valueReference="0" variability="discrete">
9        <Integer max="2^32-1" min="0" start="0"/>
10     </ScalarVariable>
11     <ScalarVariable causality="input" description="int" name="paddr" valueReference="1" variability="discrete">
12       <Integer max="2^32-1" min="0" start="0"/>
13     </ScalarVariable>
14     <ScalarVariable causality="input" description="bool" name="pclk" valueReference="0" variability="discrete">
15       <Boolean start="false"/>
16     </ScalarVariable>
17     <ScalarVariable causality="output" description="int" name="prdata" valueReference="2" variability="discrete">
18       <Integer max="2^32-1" min="0" start="0"/>
19     </ScalarVariable>
20   ...
21   </ModelVariables>
22   <ModelStructure/>
23 </fmiModelDescription>
```

emulates one transaction. The internal time of the model is annotated by storing the number of clock-cycles executed in the last transaction.

The model's interface is isolated in a structure embedded inside the C++ class containing a set of fields representing the original ports of the HW models. The data-types of the fields are abstracted into C native data-types, as described in [20]. For instance, a 32-bit `logic_vector` datatype is abstracted into `uint32_t` C datatype. Furthermore, the interface structure also contains the time annotation of the model.

Then, the methodology wraps the C++ class within the FMI functions and it generates the set of `fmi2Set` and `fmi2Get` necessary to write and read, respectively, input and output variables from and to the components. The methodology generates the `fmi2DoStep` function wrapping the generated `simulate`. While the generated `fmi2DoStep` function still accepts the step length, in order to stay compliant with the standard, it is able to ignore it as the actual internal time of the FMU at the end of the execution is computed by the `simulate` function. Then, the methodology continues by generating the XML file of the FMU. The original ports of the HW model are mapped in the FMI data-types as described in Section 7.4.1.

Finally, the methodology enriches the interface of the FMU with the internal time annotation of the Transaction-Level Model. The internal time of the model is exposed as a new `Integer` port of the FMUs (see Listing 7.6, line 18-22).The value reference `-1` is reserved for the timing port, since ports are uniquely identified by their type and value reference pairs. This assures that it can be uniquely identified once the FMU is loaded by a simulator.

7.6 Generating the coordinators

In this section, the generation of different coordinators is treated. The proposed methodology aims to automatically generate coordinators capable to manage discrete and continuous FMUs. Usually, a coordinator reflects the Model of Computation (MOC) of the blocks that compose the entire system. A CPS is the combination of discrete and continuous elements where all

Listing 7.7: IP-XACT Topology description of the platform.

```
...
// Master connection to bus
<spirit:interconnection>
  <spirit:activeInterface spirit:componentRef="bus"
    spirit:busRef="masterAPB_if" />
  <spirit:activeInterface spirit:componentRef="cpu_mem"
    spirit:busRef="ambaAPB_if" />
</spirit:interconnection>

//Slave 1 connection to bus
 <spirit:interconnection>
  <spirit:activeInterface
    spirit:componentRef="bus"
    spirit:busRef="slaveAPB_if_1" />
 <spirit:activeInterface
  spirit:componentRef="sensor"
  spirit:busRef="ambaAPB_if" />

//Slave 2 connection to bus
 <spirit:interconnection>
  <spirit:activeInterface
    spirit:componentRef="bus"
    spirit:busRef="slaveAPB_if_2" />
 <spirit:activeInterface
  spirit:componentRef="crypto"
  spirit:busRef="ambaAPB_if" />

//Slave 3 connection to bus
 <spirit:interconnection>
  <spirit:activeInterface
    spirit:componentRef="bus"
    spirit:busRef="slaveAPB_if_3" />
 <spirit:activeInterface
  spirit:componentRef="uart"
  spirit:busRef="ambaAPB_if" />

</spirit:interconnection>
```

the actors affect the opposite side. In this particular situation, the general simulation strategy requires the identification of the simulation step of the discrete system and propagate it in the continuous part. This is due to the fact that data exchange between discrete-continuous elements can happen only in specific moments in time. This strategy is flexible and scalable, but not the optimal solution. In fact, in some systems, not all the blocks are always used. In particular, this is clear when we are dealing with a digital platform representing the discrete part of the CPS, where the peripherals are controlled by a master component and enabled only on-demand. From a general coordinator perspective, where all the blocks are simulated, this requires useless computational effort in terms of simulations and data exchange. In this paper, we discuss the generation of specific coordinators adding external information coming from designers. The methodology relies on the topology system information and communication protocols of the involved components of the digital system. The entire digital platform is based on the definition of roles, one master and multiple slave peripherals, with AMBA peripheral bus standardized interface and communication. Figure 7.3 shows the different coordinators that are possible to obtain mixing topology and protocol timing diagrams.

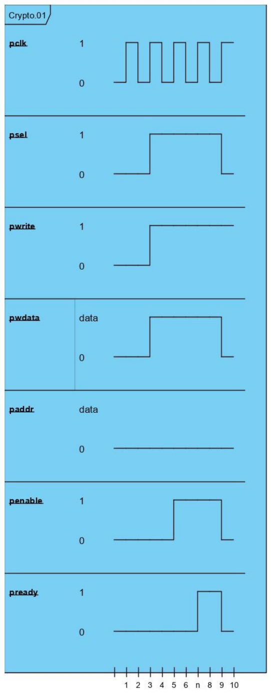


Fig. 7.5: Crypto APB Timing Diagram.

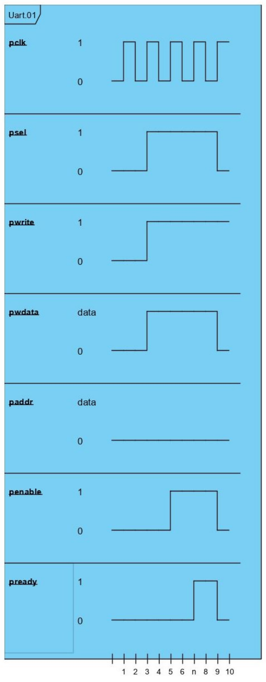


Fig. 7.6: Uart APB Timing Diagram.

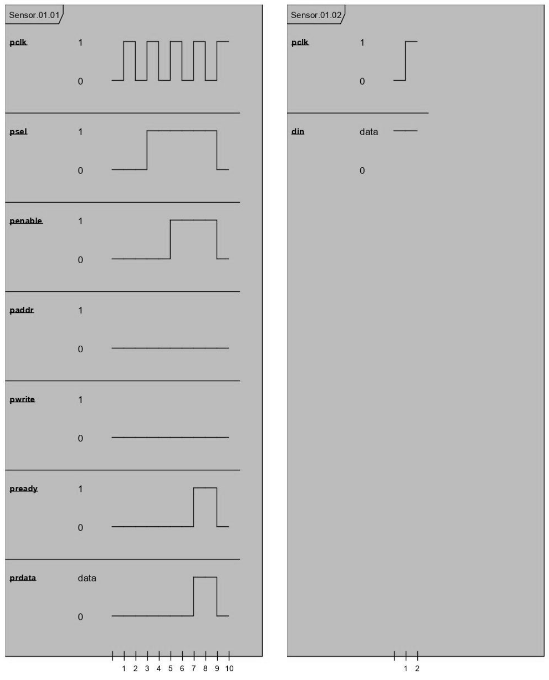


Fig. 7.7: Sensor's Timing Diagrams

Fig. 7.8: Timing Diagram of platform components.

7.6.1 Cycle-Accurate Coordinator

The generation of a cycle-accurate coordinator can be obtained just by considering cycle-accurate models and topology information of the discrete models. In [55] we have presented an automatic methodology to obtain discrete FMUs starting from discrete models described with HDLs. In this work, we have extended the mentioned methodology adding information regarding the topology of the cyber part of the system. Listing 7.7 reports the IP-XACT interconnection between the different components that composes the case study platform (see Fig. 7.4). In particular, lines 2-8 represent the interconnection of the CPU-Memory component to the master interface of the APB bus. Lines 10-17 show the sensor connected as slave peripheral of the Apb bus (lines 10-17). More in detail, the sensor is connected to `slave_APB_if_1` which represents the interface 1 of the slave peripherals of the bus. The rest of the Listing reports the connection of the crypto and uart peripheral respectively on the second and third slave interface of the bus. All the involved digital components adopt the AMBA standard interface. The use of a standardized interface reduces the effort to build the entire platform in terms of interconnections and communications.

The IP-XACT description allows to easily define which is the master component and the peripherals connected to the bus. The resulting coordinator performs the simulation of all the components, miming the concurrency, and then performing data exchange based on the IP-XACT description.

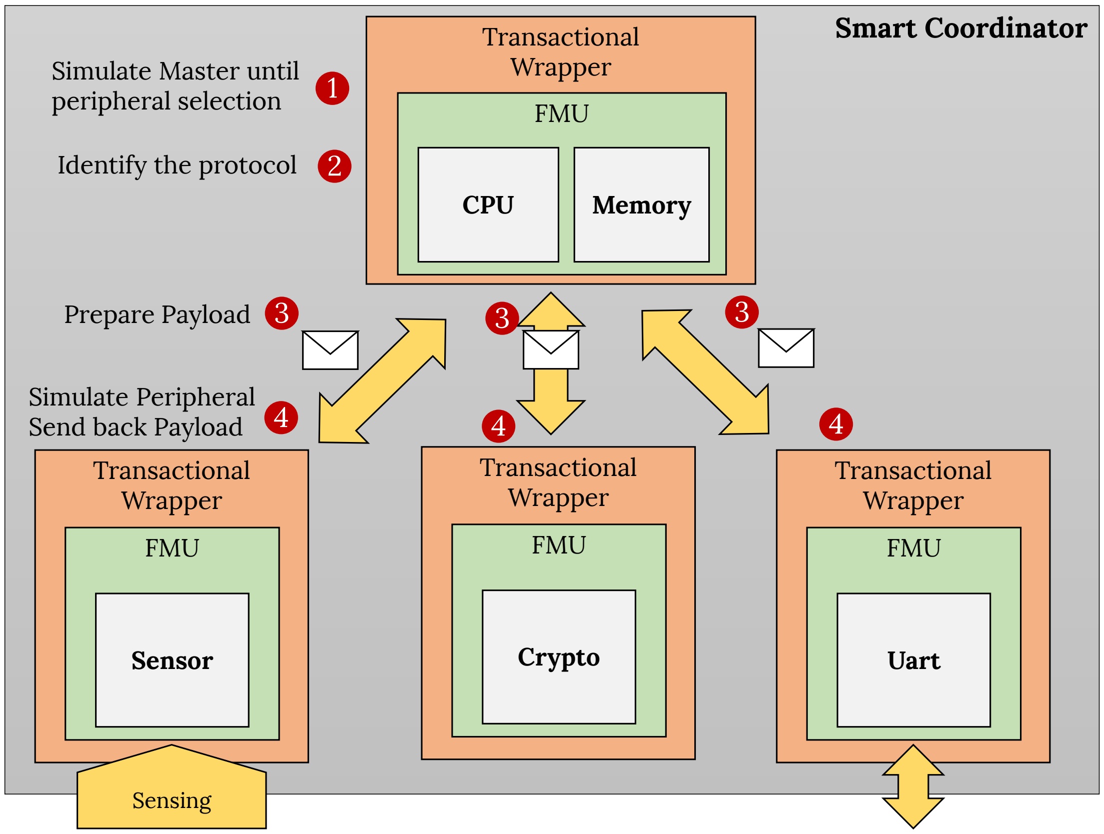


Fig. 7.9: Overview of the smart coordinator obtained mixing System-level information, with all the execution phases.

However, with just the topology information it is not possible to infer how the different models interact, requiring to perform a cycle-accurate simulation of all the involved models.

7.6.2 Smart Coordinator

The smart coordinator is an extension of the cycle-accurate coordinator. It relies on topology system information and the communication protocol between the different involved digital components. The use of system-level information (topology and communication protocols) does not violate intellectual properties by maintaining secret the functionality of the model. With the combination of topology and communication protocols information, the simulation strategy can be optimized by obtaining a smart coordinator that reduces the data exchange between the different components.

The topology is described by using IP-XACT standard (see Section 7.2.4) while the protocols are modeled with UML Timing Diagrams (see Section 7.2.4). The entire platform is based on the definition of roles, one master and multiple slave peripherals, with Amba peripheral bus standardized interface and communication.

Figure 7.5 shows the protocol timing diagram for the crypto (`crypto.01`) and uart peripheral (`uart.01`). As mentioned in this section, the communication protocol of all the digital components relies on the AMBA specifications. The AMBA interface contains a peripheral selector port, called `psel`, used by the master to select the peripheral. The `pclk` port represents the clock of the peripheral and each clock transition is tagged with the time unit reported in the lower axis. The `pwrite` port is used to code the write operation from the master to the peripheral. `pwdata` and `paddr` ports represents the data and the address that are exchanged by master to the slave peripheral. When the peripheral selector is triggered, the peripheral starts receiving data through the `pwdata` port.

The `penable` port is used from the master to enable the peripheral to compute the elaboration. `Pready` port is an output port from the peripheral used to notify the master that the peripheral has concluded the elaboration. When the latency is not known, the time unit referred to `pready` event is set to *n* (see time unit labels). Therefore, the smart coordinator can simulate the peripheral until `pready` is triggered. If a peripheral has more than one operation, the methodology allows to describe another timing diagram referring to the new operation. Let considers the crypto peripheral that executes only crypt operation. If the peripheral needed to execute also a decrypt operation it can be done describing the timing diagram, with the name `crypto.02`. With the timing diagram, the coordinator creates a wrapper around each component, containing an finite state machine that emulates the protocol with static information.

Figure 7.9 shows the overview of the smart coordinator with the use case, with all the execution phases. The first step simulates the CPU-Memory master component until the peripheral selector, `psel`, is triggered. This is implemented inside the transactional Wrapper of the master component. From the smart coordinator perspective, this is done with a single calling function. Then, the transaction wrapper identifies the protocol analyzing the AMBA output ports. The identification phase is performed by analyzing `pwdata` and `paddr` ports, according to the timing diagram of each peripheral. Then, the smart coordinator prepares the payload with the information of the protocol, data to send to the selected peripheral (`psel` and `pwdata[]`,`paddr[]` arrays), and also the global time. In the fourth phase, the smart coordinator simulates only the selected peripheral. The peripheral transactional wrapper receives the payload containing the protocol and the data to execute. The smart coordinator forward the resulting payload to the CPU-Memory block, which represents the master block. Then, the entire execution flow of the smart coordinator can start again simulating the CPU-Memory block.

7.7 Experimental Results

The proposed approach has been validated by automatically generating virtual platforms to emulate the behavior of the case-study described in Section 7.3.1.

7.7.1 Experimental setup

All the simulations have been executed on a i7-7700HQ with 16GB of RAM machine, running Fedora 33 Linux. The virtual prototypes generated by the presented approach have been compared with a state-of-the-art co-simulation environment based on Mentor Graphics' QuestaSim

Table 7.3: Summary of the obtained experimental results. It compares the results considering both the two proposed techniques exploiting system-level information (*i.e.*, cycle-accurate and smart coordinator), and the two realized implementations (*i.e.*, based on PyFMI and based on FMI4CPP). The reference co-simulation environment is based on QuestaSim and Simulink.

Simulated Time	Co-simulation (QuestaSim + Simulink)		FMI-based Cosimulation Cycle Accurate				FMI-based Cosimulation with Smart Coordinator			
	Sample time		PyFMI implementation		FMI4CPP implementation		PyFMI implemenation		FMI4CPP implemenation	
	5 ns	**10 ns**	**Time (s)**	**Speed-up**	**Time (s)**	**Speed-up**	**Time (s)**	**Speed-up**	**Time (s)**	**Speed-up**
1 us	2.33	2.34	0.44	**5.30x**	0.16	**14.56x**	0.55	**4.24x**	0.10	**23.30x**
10 us	2.69	2.60	0.79	**3.41x**	0.16	**16.81x**	0.55	**4.89x**	0.10	**26.90x**
100 us	5.99	4.61	4.38	**1.37x**	0.25	**23.96x**	5.53	**1.08x**	0.40	**14.98x**
1 ms	35.70	21.63	39.57	**0.90x**	1.80	**19.83x**	10.47	**3.41x**	0.70	**51.00x**
10 ms	375.85	185.86	409.66	**0.92x**	18.41	**20.42x**	55.24	**6.80x**	3.59	**104.69x**
100 ms	3389.00	1807.00	3897.83	**0.87x**	183.81	**18.44x**	524.71	**6.46x**	35.02	**96.77x**
1 s	34194.00	17132.00	37287.32	**0.92x**	1864.56	**18.34x**	5020.88	**6.81x**	352.01	**97.14x**

2019.4 as HDL simulation for the embedded device, connected to Mathworks' Simulink 2020b simulating the dynamic system modeling the environment of the device.

This work focuses on the integration of the digital components into the virtual prototypes. Still, in order to stimulate the hardware platform it is necessary to connect it to a model able to generate continuous-time values emulating the physical environment of the system. The model of the physical part of the system generates a set of sinusoidal waves on the input variables. We decided to keep the continuous time model in order to keep the overhead of the dynamical system simulation to the minimum, thus allowing to obtain experimental values dominated by the execution of the digital part of the modeled system. The continuous time signals have been implemented as Mathwork's Simulink models for the reference simulation environment, while they have been implemented within a FMU in the FMI-based simulation environments generated by applying the proposed methodologies.

7.7.2 Experiments overview

Table 7.3 reports the results obtained in our experimental analysis. The first column reports the *Simulated Time* of each simulation, we carried on simulation emulating different amount of time of the actual system. The values ranges between 1 microsecond and 1 second. Then, the table reports the time needed to simulate the system. The *Co-simulation (QuestaSim + Simulink)* section of the table reports the time required to simulate the system using a state-of-the-practice simulation environment based on a HDL simulator connected with the Mathworks' Simulink dynamic simulator. The *FMI-based Cosimulation Cycle Accurate* section of the table reports the results obtained by emulating the system using the virtual platform built by assembling cycle-accurate FMUs, and exploiting only the information carried by the IP-XACT descriptions. Finally, the last section of the table, *i.e.*, *FMI-based Cosimulation with Smart Coordinator*, reports the results obtained by emulating the system using the virtual platform built by assembling transactional-accurate FMUs, and by generating a smart coordinator exploiting the information contained in the IP-XACT and the timing diagrams describing the system's protocols.

For all the reported simulation, both FMI-based co-simulation environments are set to sample the environment every 10 nanoseconds. Meanwhile, in the case of the state-of-the-practice co-simulation environment we report the results obtained by setting the sample time at 5 and 10 nanoseconds. Intuitively, the results obtained by applying the proposed approach should be compared against the results obtained using the same sample time (*i.e.*, 10 nanoseconds). However, when emulating the system using the state-of-the-practice co-simulation environment we have a very limited control over the used numerical integration being used, and over the communication between the discrete and continuous models in the system. Meanwhile, in the FMI-based co-simulation environment, the generated coordinators are fully aware of the interactions between discrete and continuous parts of the system. Thus, keeping the same sampling time in the state-of-the-practice environment, continuous-timed values may be lost with respect to the emulation performed by using the proposed FMI co-simulation environments. As such, in order to be sure to sample the same values in all the executions, and considering the NyquistâĂŞShannon sampling theorem, we choose to use as reference the simulation which sample frequency is doubled (*i.e.*, 5 nanoseconds). For this reason, while the third column of the table reports the simulation time required using a sampling time of 10 nanoseconds, the speed-up reported in the Table refers to the executions in the second column of the Table.

The coordinators generated by the proposed approach have been generated to be compliant with two different FMI simulation engines. A coordinator written in Python and based on the PyFMI framework [103], and a C++ coordinator based on the FMI4cpp[1] framework. We report the time required to simulate and system, and the speed-up *w.r.t.* the reference simulation for both coordinators, for each simulation.

We compared the behavior of each experiment with respect to the behavior of the reference simulation, carried on with state-of-the-art tools. In all of our experiments, the simulations were functionally equivalent, *i.e.*, at each time step, and for each variable, the values of the variables in the reference and the generated models are equivalent. In general, the C++ coordinator proved itself more efficient than the coordinator written in Python. Still, in both cases, the simulation efficiency increases by increasing the amount of information used to generate the virtual platform emulating the system. In particular, considering the more efficient C++ coordinator, the speed-up when exploiting only the information carried by the IP-XACT description is in the 20x ballpark. Meanwhile, it raises to around two order of magnitude when exploiting also the information about the protocols. While these are the main takeaways of our experimental results, we hereby present a more in-depth analysis.

7.7.3 In-depth analysis of the experimental results

In general, the FMI-based cosimulation environments provide better performance than classic co-simulation. However, the cycle-accurate implementation using the coordinator based on PyFMI suffers a performance drop. This is mostly due to low efficiency of the communication between Python and the C-based DLLs containing the FMUs behaviors. The fine-grained

[1] Online: `https://github.com/NTNU-IHB/FMI4cpp`

synthronization required for cycle-accurate simulation, and the poor efficiency of the communication leads the communication and synchronization to dominate the simulation. Meanwhile, the C++-based implementation of the coordinator guarantees to preserve the efficiency even when increasing the number of synchronization and communication points of the simulation. Indeed, for the shortest simulations, the execution time is dominated by the initialization. However, when increasing the simulated time, the speed-up tends to stabilize in a range between the 18 to the 20x.

When improving the simulation environment, also the PyFMI-based implementation is able to preserve the speed-up. This is due to the fact that the smart coordinator allows to drastically reduce the number of synchronization points. Thus, mitigating the inefficiencies due to the communication between the DLLs and the python coordinator. Thus, lenghtening the simulated time the speed-up stabilizes between 6 to 7x.

Finally, the smart coordinator based on FMI4cpp reaches up to two order of magnitude speed-up. However, in some of the intermediate entries of the table (*i.e.*, simulated time set to 100 us and 1 ms) the simulation speed-up drops. We investigated the reasons of such anomalies, and they are due to the characteristics of the case study: the UART component protocol is characterized by a variable latency that, depending on the task being executed, may be either extremely long or short. This leads to a variation in the frequency of the synchronization. The two anomalies are due to executions in which the frequency of synchronization of the UART is higher.

Part IV

Multi-Level Modeling and Simulation

8

From Multi-Level to Abstract-Based Simulation of a Production Line

8.1 Introduction

The concept of Industry 4.0 [1] represents an innovative vision of what will be the factory of the future. The principles of this new paradigm are based on interoperability and data exchange between different industrial equipment. In this context, Cyber-Physical Systems (CPSs) cover one of the main roles in this revolution. The entire factory can be seen as a set of CPSs and the resulting system is also called Cyber-Physical Production System (CPPS). This CPPS represents the Digital Twin of the Factory with which it would be possible to make analysis regarding the Real Factory [3]. The interoperability between the real industrial equipment and the Digital Twin [4] allows to make predictions concerning the quality of the products. Several tools [2] allow to model a production line, considering different aspects of the factory (*i.e.* geometrical properties, the information flows). However, these simulators do not provide natively any solution for the design integration of CPSs, making impossible to have precise analysis concerning the real factory.

This paper proposes two different approaches for the integration of CPS in a production line simulator (see Fig. 8.1). The approach on the left side of the figure relies on the *Multi-Level* simulation where multiple descriptions of the same CPS are managed. These descriptions have different levels of detail and are switched at runtime in order to optimize the overall simulation time. The second approach is based on *Abstraction* techniques where a set of manipulations are made on the whole system in order to obtain semantically equivalent descriptions but reducing the complexity of the starting models. Different coordination strategies are presented in order to integrate and simulate correctly the abstracted models. The two approaches are then integrated with *Siemens Plant Simulation* with a real use case scenario. The obtained results show the benefits of the CPS integration in both of the approaches.

8.2 Background

In these years several providers proposed different tools to model manufacturing processes [2, 5]. Report [5] summarizes periodically all tools by proposing comparisons on their main characteristics (usability, costs, features, *etc.*). These reports are a useful guideline for selecting

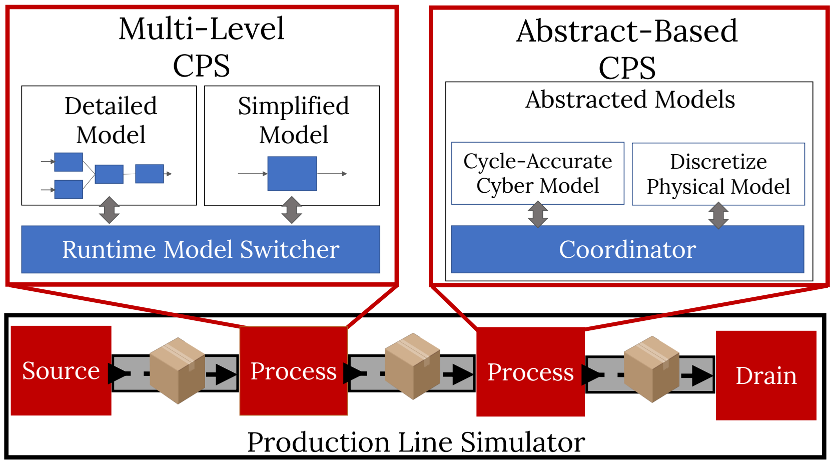


Fig. 8.1: Overview of the contribution of the work. The figure shows the *Multi-level* and *Abstract-Based* approaches for the CPS integration in a production line simulator.

most suited plant simulators with respect to the parameters to evaluate. A production line is the composition of processes, organized into a chain, which has the main purpose of handling information. Despite some differences, all the simulations share the following principles:

- *Layout Planning*: Represents the geometrical structure of the production line. A library of components allows to model the factory, by considering physical constraints.
- *Material Flow/Fluid Simulation*: Represents the movements of products from a process to the others. This is made possible with components like line transporters or pipe, depending on the material state of matter, *i.e.*, solid or fluid.
- *Process Simulation*: Represents the physical transformation made by the equipment of the factory to the products.

From the simulation perspective, most of the available simulators rely on the *discrete-event* model of computation. For instance, when a product enters or exists a process, an event is triggered and the specific equipment can execute its relative action.

This paper makes use of Siemens *Plant Simulation*, a simulator which over the years has become a standard de facto in the designing of production lines, with an intuitive and easy to use environment. It is a Model-Based tool that provides a library of customizable components that represent the basic building blocks of a factory. Combination of these blocks allow to model different aspects of a production chain. The products are called Mobile Unit (MU) and they represent the entities moving among the blocks of the production line.

SingleProcess is the fundamental block provided by the tool used to represents the physical process of an equipment on a MU. Plant Simulation offers the possibility to customize the behaviour of every block with an internal programming language called *SimTalk*. It also provides a functionality called *C-interface* to import dynamic libraries written in C/C++, enabling to customize the behaviour of the simulation and also connect other tools.

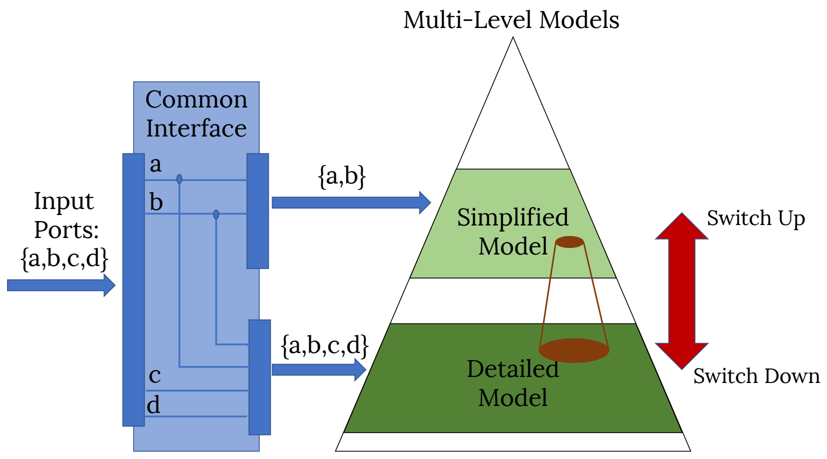


Fig. 8.2: Multi-level Common Interface and switching actions

8.3 Multi-Level Modeling and Simulation

Multi-level approach [104] allows managing multiple descriptions of the same system with different level of detail at runtime. Mixing the execution of these descriptions at runtime allows to speedup the simulation instead of using only the high resolution model, but maintaining a certain level of precision compared to the low accuracy of the low resolution model. Multi-level approach requires to face some issues that can be summaries as follows:

- interfaces of the multi-level models;
- models switching actions;
- state mapping of the models.

Let consider two different behavioural descriptions of a CPS: a *Detailed Model* and a *Simplified Model*. These two models could have different interfaces (see Fig. 8.2). The *Simplified Model* has fewer ports than the *Detailed Model*, but all the ports of this model are a subset of the *Detailed Model* interface. The standardization of a `Common Interface` is needed to easily switch between the two models (see Fig. 8.2).

The exposed ports of this block are represented by the interface of the *Detailed Model*, in order to capture all the properties without losing any information. When the *Detailed* model is in use, the `Common Interface` block receives inputs `a,b,c,d` and redirects all the four inputs to the model. When *Simplified Model* is running, the `Common Interface` block redirects only `a,b`. The second issue to face with Multi-Level approach regards the switching actions, called *switching-up* and *switching-down*, used to switch from a model to another. *Switching-up* action allows switching from a high-resolution model to another with a lower resolution. *switching-down* action switch from a low-resolution model to a high-resolution model.

Every switching action requires to exchange the internal state of the current model to the new switch one, in order to be consistent between the models. This is called *state mapping* [105]. For instance, let consider the *Detailed Model* simulated for a certain amount from *t=0* to *t=h*. At time *t=h* a *switching-up* action is triggered. Without *state mapping*, the simulated time of the *Simplified Model* is zero but the global time of the simulation is *h*. The internal status of the *Simplified Model* refers to its initial conditions and not to the time *h*, making an inaccurate simulation. In this paper all the considered models expose internal storing variables as ports to the `Common Interface` blocks, in order to be able to perform a state mapping.

8.3.1 Application of Multi-Level Simulation to a Production Line

The application of Multi-level approach with *Plant Simulation* requires to define synchronization rules in order to handle correctly the data from the models to the production Line simulator and vice-versa.

The `Runtime Model Switcher` is the actor in charge to exchange the data and manage the simulation of the two models (see Fig. 8.3). The *Common Interface* block is part of the `Runtime Model Switcher`. The `Runtime Model Switcher` receives input data from *Plant Simulator* and then redirect it to the `Common Interface`. After the data exchange, the `Runtime Model Switcher` select the model to simulate performing a switching action. The `Runtime Model Switcher` also stores the shared properties of the models, needed to perform the state mapping. In *Plant Simulation* we assume that the switching actions, can be performed when a `MU` enters in a `Single Process` block that represents an equipment of the factory. With this assumption, we say that the resolution of the switching points is Mobile Unit Accurate (MU-Accurate).

8.4 Abstract-Based Modeling and Simulation

A CPS can be modelled with a low level of details in a unique language but the resulting system will be not accurate. Refine both Cyber and Physical systems requires specific languages of different tools tailored to specific domains that use different Models of Computation (MoC). The simulation of the Cyber and the Physical systems together is time-consuming, because of the synchronization of the different MoC and because of Co-Simulation mechanisms between different tools. The goal of the *Abstract-Based* approach is to abstract the descriptions of the two subsystems in order to reduce the global complexity of the whole Systems. The abstract-Based approach starts from specific-domain languages like Hardware Description Languages (HDLs) for the Cyber part and Verilog-AMS to model the Physical System, that uses a Continuous Time MoC. This paper relies on abstraction technologies provided by a HIFSuite framework [44]. Such a framework allows to manipulate models described using HDLs (Verilog and VHDL) or Analog Mixed Signal (Verilog-AMS). HIF provides a set of methods to import model descriptions, manipulate them and generate semantic equivalent C++ code. The front-end methods translate the model descriptions in a proprietary language (HIF format). The manipulation tools make transformation on the obtained HIF description. Finally, the back-end methods allow generating the C++ code. There are three different abstractions available: *Analog Abstraction*, *Datatype Abstraction* and *Protocol Abstraction*.

The *Analog Abstraction* is a process which aims at simplifying continuous time components for an easier integration and efficient simulation [106]. The challenge is related to their behaviour, which is usually described through a system of differential equations. State of the art

Table 8.1: Simulation times of bending operations for Mobile Units in the two different approaches.

Simulated Bending Operations	Multi-Level			Abstract-based				Simulated Time	
	Simplified model	Detailed model	Hybrid	Abstracted Simplified Model		Abstracted Detailed Model		Simplified Model	Detailed Model
				Cycle-Accurate Coordinator	Transaction-Level Coordinator	Cycle-Accurate Coordinator	Transaction-Level Coordinator		
1	0.22	0.63	0.22	0.208	0.016	0.075	0.008	1.22	0.66
20	5.28	15.12	9.74	4.992	0.384	1.801	0.192	16.70	11.46
50	12.76	36.54	27.69	12.064	0.928	4.351	0.464	39.28	27.67

simulators represent the behaviour as sparse matrices and solve the system at each simulation step, with a consequent loss of performances. With the abstraction, the complexity of solving such systems is anticipated during models generation. It enriches the initial set of equations using Kirchhoff's laws. Then, it solves the resulting enriched system with a symbolic solver and produces a signal-flow representation of the model. The final description contains the simplified equations that describe the relation between inputs and the outputs of the model.

The *Datatype Abstraction* is a process which transforms HDL datatypes, like logics and bit vectors with a many-valued logic, into efficient C++ native ones [20].

The *Protocol Abstraction* is performed on the simulation protocol of the model [107]. This manipulation reduces the internal processes of the module comparing them sensitivity lists and merging them together. Furthermore, it encapsulates an optimized Discrete-Event scheduler inside the C++ final code. The resulting C++ code contains a special structure that represents the interface of the starting model and a set of methods to simulate it.

8.4.1 Application of Abstract-Based Simulation to Production Line

The Cyber and Physical Abstracted models need to be integrated into *Plant Simulation*. First, a simulation coordinator is needed to synchronize the two abstracted systems of the CPS. This paper presents two different coordination strategies. The first strategy is the *Cycle-Accurate* where both of the Systems are simulated with a timestep that is equal to the clock period of the Cyber System. This coordination requires a lot of synchronization points depending on the clock period.

The second is the *Transaction-Level* strategy, and it is event-oriented. For instance, if the Cyber model is blocked, waiting for a certain event from the Physical system, the coordinator will not execute it. This solution is still correct, but requiring fewer synchronization points than the *Cycle-Accurate*. Moreover, it requires to know in advance the synchronization mechanisms between the two parts of the CPS in order to identify which synchronization points can be avoided. The resulting CPS is then integrated in *Plant Simulation* using *SimTalk C-Interface* APIs.

8.5 Experimental Results

The two presented approaches are tested with a real use case scenario of a simple production line. The production line is composed of three processes and represents a bending operation of metal sheets. The first process applies to the metal sheet a barcode containing the information of the desired bend angle. The second represents the bending machine that reads the angle to bend from the barcode and executes the bending operation. The real bending process of the equipment

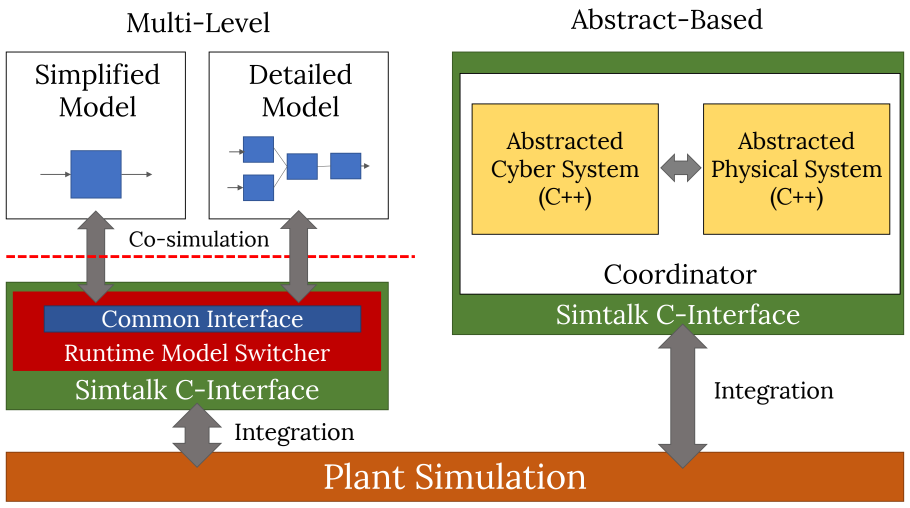


Fig. 8.3: Overview of the Experiment Setup.

Table 8.2: Times needed to simulate one second in the two different approaches.

Average Time to Simulate 1 Second (s)	Multi-Level			Abstract-based			
	Simplified model	Detailed model	Hybrid	Abstracted Simplified Model		Abstracted Detailed Model	
				Cycle-Accurate Coordinator	Transaction-Level Coordinator	Cycle-Accurate Coordinator	Transaction-Level Coordinator
	0.180	0.955	0.437	0.170	0.013	0.114	0.012

requires at most 3 seconds to bend a metal sheet. The last process checks the quality of the bending operation comparing the desired angle and the real bent metal sheet. The two proposed approaches presented in this paper integrate a CPS in the bending process in order to make a more accurate estimation of the production in terms of productivity and quality(see Fig. 8.3). The *Physical System* represents the behaviour of a bending machine, described using Verilog-AMS, that is controlled by the *Cyber System*. The *Cyber System* of the CPS is a Hardware platform composed of a CPU, a Memory, a Bus, a Barcode sensor and an Actuator to the bender physical system. All the components of the HW platform are described by using HDLs.

The bending software reads the angle to bend from the metal sheet using the Barcode Sensor, and then redirects it to the *Physical Systems* with a set of commands.

8.5.1 Multi-Level Experiment

The Multi-Level experiment relies on two different versions of the CPS called *Simplified Model* and *Detailed Model* (see Fig. 8.4). In both of the CPS models, the *Cyber System* is represented with the digital platform explained in the previous section. In the *Simplified Model* the *Physical System* consists of a behavioural description of the bender equipment, with a low level of detail. In particular, the *Physical System* is modelled with an integrator that describes the bending operation. The integrator receives as input a value that represents the constant bending speed to apply. This value is calculated by a *Speed Selector* node, which adopts two different values according to the bending rotation versus. The value provided by the integrator is then compared

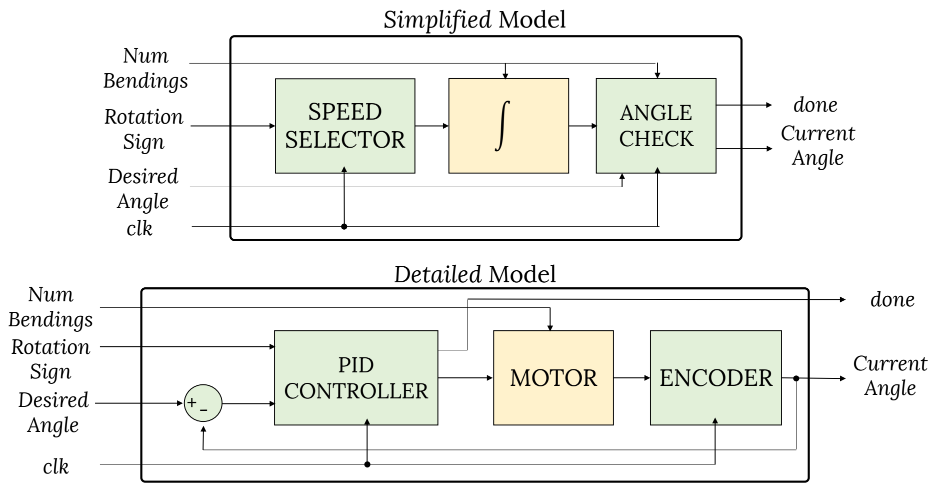


Fig. 8.4: Simplifed and Detailed Physical Models.

with the desired angle by the *Angle Controller* node in order to drive the *done* signal. The *numBendings* is used to model the machinery wearing.

In the *Detailed Model* the *Physical System* allows to make a precise estimation of the execution time and the quality of operation. A *PID Controller* is used to control a DC Motor in order to bring every sheet to the desired angle, which is read by an encoder that translates the motor rotational position in a digital value. The DC Motor model uses the *numBendings* value to modify its internal parameters: this allows to simulate its wearing since those values represent the mechanical and electrical characteristics of the Motor.

The `Runtime Model Switcher` is connected to the CPS simulator using Co-simulation techniques. In the other side, the `Runtime Model Switcher` is integrated in *Plant Simulation* using *C-Interface* proprietary interface. The `Runtime Model Switcher` is then linked to the Bending `SingleProcess` and executed only when a new MU enters in that node of the production line. The simulation starts using the *Simplified Model* of the CPS. Every 20 bending operations, the `Runtime Model Switcher` enables both the models and provides to them the current metal sheet to bend. After the bending process, the quality deviation between the two models is evaluated: if it crosses a certain threshold, the `Runtime Model Switcher` deactivates the *Simplified Model* and switches to the *Detailed Model*. The *Detailed Model* is kept enabled until the quality deviation remains over the threshold. The simulation resolution is MU-Accurate, meaning that the switching actions can be performed only at the entrance of a MU in the `SingleProcess`. When the execution of the bending operation is completed, the `Runtime Model Switcher` retrieves the executed time to bend the metal sheet and returns it to the `SingleProcess` of Plant Simulation.

8.5.2 Abstract-based Experiment

To better evaluate the *Abstract-based* approach performances, the two CPS versions (*Simplified Model* and *Detailed Model*) are considered. As explained in 8.4 the entire abstraction toolchain is based on the HIFSuite framework [44]. The models of the Cyber and Physical systems, are first translated in HIF format, using front-end methods. After the first translation, the *Physical System* is abstracted using the *Analog Abstraction*, defining the time step to use for the discretization. Then, *Datatype* and *Protocol Abstraction* are performed on both the obtained HIF descriptions. Finally, the HIF back-end tools are invoked to generate the C++ code. For the *Cyber System* the *Protocol Abstraction* is performed with Cycle-Accurate (CA) resolution. More in details, when the simulation method of the *Cyber System* is called, it performs a simulation of an entire clock cycle. A *Coordinator* is needed to simulate together the obtained abstracted Systems that compose the CPS. This paper proposes two different coordinators, *Cycle Accurate Coordinator* and *Transaction-Level Coordinator*. The *Cycle Accurate Coordinator* simulates and exchanges the data between the Cyber and the Physical System at every clock cycle period. The *Transaction-Level Coordinator* reduces the number of communication points based on the communication protocol between the two systems. When the *Physical System* is performing a bending operation, the *Cyber System* has to wait until the finishing operation event is triggered. Thus, the *Transaction-Level Coordinator* can pause the *Cyber System* and simulate only the *Physical System* until this event. Finally, the resulting C/C++ code of the CPS is wrapped with the *C-interface* avoiding completely Co-Simulation mechanisms.

8.5.3 Simulation Results Comparison

In Table 8.1 and 8.2 are reported the results of the experiments between the two approaches.

Table 8.1 reports the required time to simulate the metal sheet bending using the different models. The last two columns report the simulated times. The simulated time of the *Detailed Model* is more accurate than the *Simplified Model*. In *Multi-Level* approach, the *Hybrid* column reports the simulation results mixing the *Simplified* and *Detailed Models*. At the beginning of the simulation, *Simplified Model* is activated, in fact, it requires similar simulation times. When the number of bending operations increases, the required time moves towards the *Detailed Model* one, which is enabled to keep the quality deviation estimation under a certain threshold. This approach requires an effort by the CPS simulator that has to simulate every single clock cycle of the HW platform, in order to obtain the most precise simulation. It is also slowed down by Co-Simulation mechanisms. The *Abstracted Simplified Model* with *Cycle-Accurate* coordinator, requires almost the same time as the *Simplified Model* with *Multi-Level* approach. This is due to the very small complexity of the *Simplified Model* that cannot be reduced from the abstraction techniques. The *Abstracted Detailed Model* with the same coordinator achieves a speed-up of 10x compared to the *Detailed Model* of the *Multi-Level* approach. However, the *Cycle-Accurate* coordinator simulates the busy waiting of the *Cyber System* during the physical bending introducing unnecessary communication points. The *Transaction-Level Coordinator* still maintains the same timestep precision but avoiding all the unnecessary communication points. The *Abstracted Simplified Model* with *Transaction-Level Coordinator* achieves a speed-up of $\sim 10x$ than the *Cycle-Accurate* coordinator simulation. The *Abstracted Detailed Model* obtains a $\sim 100x$ speed-up than the *Multi-Level Detailed Model* and $\sim 10x$ as against its *Cycle-Accurate* version. In general, using the *Transaction-Level* coordination algorithm allows reducing simulation times of one degree of magnitude. From Table 8.1 it could seem that the *Abstracted Simplified Model* is slower than the *Abstracted Detailed Model*.

In Table 8.2 is reported the average time needed to simulate one second for the different approaches. In The *Multi-Level* approach, the *Detailed Model* is 5x slower than the *Simplified Model*. The *Abstracted Simplified* and *Abstracted Detailed Model* with the *Transaction-Level coordinator* requires almost the same average time to simulate one second.

8.6 Concluding Remarks

In this work, we proposed two different approaches to integrate CPSs in a production line simulator. The *Multi-Level* approach has shown the benefits of switching between different models at runtime has been proved with the obtained results, but with an approximate simulation. The *Abstract-based* represents a promising alternative solution that abstract models through a set of abstraction steps, but maintaining a certain level of detail. The *Abstract-Based* approach with *Transaction-Level* coordination has proved to be the best choice in terms of simulation time.

The results clearly show that the *Abstracted Simplified* and *Abstracted Detailed* models require almost the same time to simulate, but with different simulation accuracy. This important result allows to consider *Abstracted Detailed Model* with the *Transaction-Level* coordinator as the best solution avoiding approximate simulation of the *Multi-Level* approach or inaccuracy of the *Abstracted Simplified Model*.

9

A Design Methodology of Multi-level Digital Twins

9.1 Introduction

The concept of Industry 4.0 [1] represents an innovative vision of what will be the factory of the future. The principles of this new paradigm are based on interoperability and data exchange between different industrial equipment. In this context, CPSs cover one of the main roles in this revolution. The entire factory can be seen as a set of CPSs and the resulting system is also called CPPS. The CPPS represents the Digital Twin [4] of the factory and it allows to compare the expected behavior with the actual one of the factory. The interoperability between the real industrial equipment and the Digital Twin allows to make predictions concerning the quality of the products. Several tools [2] allow to model a production line, considering different aspects of the factory (*i.e.* geometrical properties, the information flows, *etc.*). However, these simulators do not provide natively any solution for the design integration of CPSs, making impossible to have precise analysis concerning the real factory. Multi-level approaches [104] try to define principles in order to use multiple descriptions of the same system with a different level of detail. This allows to switch at runtime between an abstracted model and a more refined one to obtain precise data regarding equipment operations while reducing the computational effort by using the abstracted model. Several works provide ad-hoc solutions for the integration of different simulators [105, 108, 109]. In [110], authors try to use multi-level approach to integrate CPSs with the use of Functional Mock-up Interface (FMI) standard [21]. In [111] authors proposed a methodology to synthesize models from AutomationML. However, all the mentioned works did not propose a unified design methodology that starts from a neutral standard leading to a multi-level approach. In particular, [110] relies on FMI that does not allow to refine the equipment model after the integration.

This paper proposes a multi-level design methodology that starts from AutomationML [15], synthesizing the entire infrastructure between plant and process simulator. It is not part of this work the synthesis of a physical model from AutomationML. The methodology allows to synthesized the communication infrastructure and the communication protocol of the process. Then, the designer can focuses on the kinematic model of the process in order to obtain the desired behavior. The proposed design methodology allows to evaluate the performance of a new equipment integrated in the real plant through the use the equipment model in a plant sim-

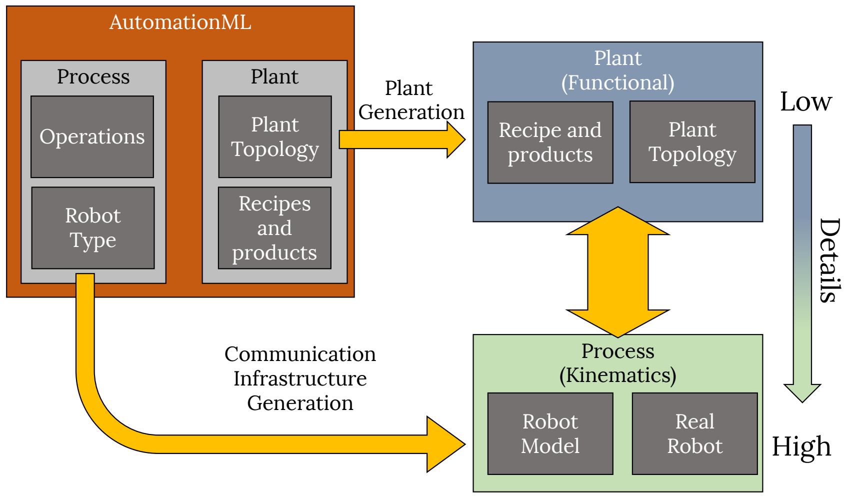


Fig. 9.1: Overview of the proposed multi-level design methodology, starting from AutomationML neutral description.The methodology automatically generates the model of the plant, the process skeleton and the communication infrastructure.

ulator. Moreover, the use of real equipment interface for the equipment model allows to easily switch between the model to the real equipment, reducing time to integrate a real equipment in the digital twin.

Figure 9.1 shows the overview of the entire flow of the proposed design methodology. The design methodology starts from AutomationML, enriched with information regarding the multi-level node to consider. Then, the plant topology is generated in a manufacturing simulator and the communication infrastructure is synthesized. This allows to obtain automatically an integrated environment to perform accurate simulation of physical processes in a functional manufacturing simulator. From the multi-level perspective, the functional simulation of the plant represents the model with a high level of abstraction while the physical process with kinematics, represents the model with a high level of details.

The main contributions of the proposed paper are:

- Unified design methodology;
- Multi-Level approach in manufacturing simulators;
- Standard interface that allows to easily switch from a model to a real equipment;
- Tool independent design methodology.

The paper is organized as follow: Section 9.2 presents the necessary background of the used technologies and the running example that will be use to explain the entire methodology. Section 9.3 explains the entire methodology starting from AutomationML to the generation of the Plant and the communication infrastructure, while Section 9.4 reports the experiments with different time granularity. Finally, Section 9.5 reports conclusion and possible future works.

9.2 Background

9.2.1 AutomationML - IEC 62714

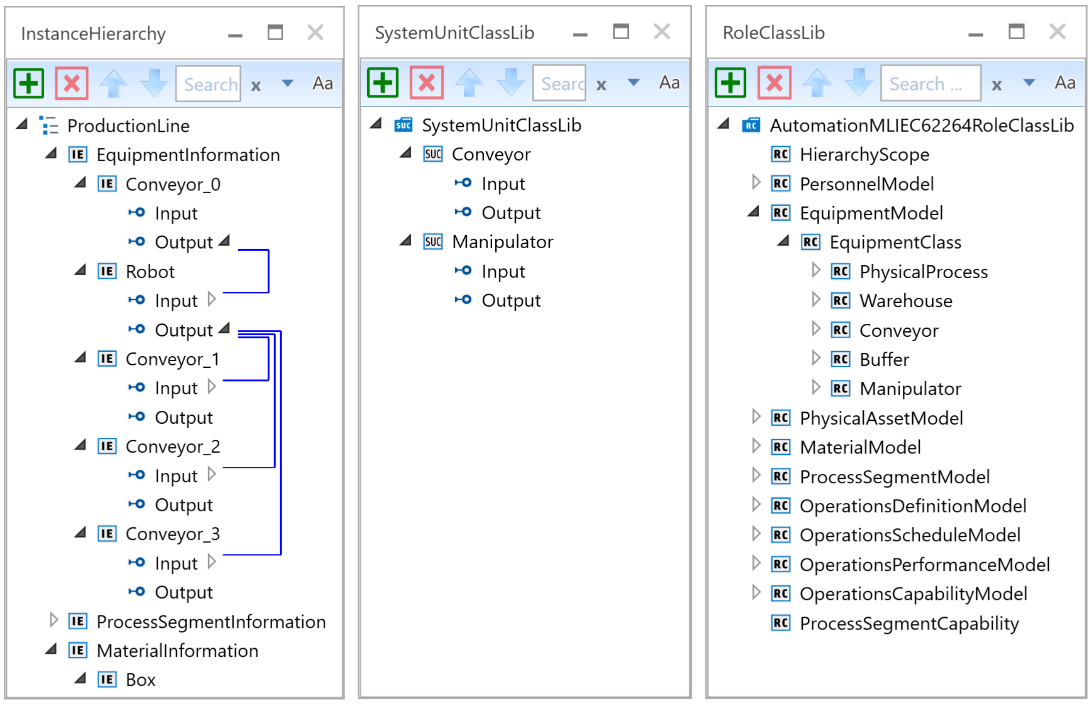
AutomationML [15] standardizes data exchange in the engineering process of production systems. The IEC 62714 does not define a new standard or a new XML schema, but it integrates all the existing standard XML schema to allow a unified semantic of a production plant. All the standards that can be integrated into an AutomationML description came from a different domain. Computer Aided Engineering Exchange (CAEX) - IEC 62424 standard is the core of the AutomationML XML structure. Into a CAEX description, it is possible to store object-oriented engineering information, for example, it is easy to define a machine topology. The topology is a structure description that defines uniquely the components of a machine. It is similar to the topology of a production line, in which the entire line is structured to be integrated into the Manufacturing Enterprise Resource (MES) core. Furthermore, the CAEX description can refer to geometry information, PLC data, logic, kinematics and external files. It defines mainly four classes:

- `InstanceHierarchy`: It represents the class where objects are instantiated. These objects could represent equipments of the plant or systems of an equipment. AutomationML provides also `Internallink` object allowing to link instantiated objects together;
- `RoleClassLib`: This class is reserved to defined the roles of objects giving them semantic;
- `SystemUnitClassLib`: This class is used to define object templates that can be instatiated multiple times;
- `InterfaceClassLib`: In this class it is possible to define interface to external objects or files. In particular, the standard provides interface for external files like COLLADA geometric information, PLC-Open description. The standard allows to define user external interface allowing to integrate other types of files.

9.2.2 CAEX - IEC 62424-2

CAEX is a language based on XML which allows to structure hierarchical information [112]. A CAEX object is a data representation of an object that could be a plant asset. Into the XML schema it is possible to model physical assets (e.g., a motor, a robot, a tank) or abstract assets, like a function block, a model or a folder. CAEX allows to link those objects because every physical or logical system is characterized by internal elements (objects) which may contain further internal elements, and all elements may have interfaces, attributes and connections with each other. This standard allows to model a single machine topology or a production lines topology. Building these topologies into CAEX is possible because this language is not specific, but it is vendor neutral. A machine topology must hierarchically define the internal structure of the machine and also define the communication with all the other devices connected to them.

IEC 62424-2,also known as ISA-95, is the international standard for the integration of enterprise and control systems. It structures a manufacturing enterprise, in four different levels from the lowest that represents the production to the highest where business-related activities needed



```
InstanceHierarchy
ProductionLine
  EquipmentInformation
    Conveyor_0
      Input
      Output
    Robot
      Input
      Output
    Conveyor_1
      Input
      Output
    Conveyor_2
      Input
      Output
    Conveyor_3
      Input
      Output
  ProcessSegmentInformation
  MaterialInformation
    Box
```

```
SystemUnitClassLib
SystemUnitClassLib
  Conveyor
    Input
    Output
  Manipulator
    Input
    Output
```

```
RoleClassLib
AutomationMLIEC62264RoleClassLib
  HierarchyScope
  PersonnelModel
  EquipmentModel
    EquipmentClass
      PhysicalProcess
      Warehouse
      Conveyor
      Buffer
      Manipulator
  PhysicalAssetModel
  MaterialModel
  ProcessSegmentModel
  OperationsDefinitionModel
  OperationsScheduleModel
  OperationsPerformanceModel
  OperationsCapabilityModel
  ProcessSegmentCapability
```

Fig. 9.2: AutomationML description of Plant running example using IEC62264-2 [113].

to manage a manufacturing operation are considered. In [113] authors proposed a solution to map IEC 62424-2 concepts in AutomationML defining a set of namespace and rules over the existing CAEX xml-schema. In particular, IEC 62424-2 consider not only equipment but also personnel and the material of the plant that could be part of the manufacturing production.

The major contribution of [113], can be summarize in the reorganization of the AutomationML `RoleClassLib`. In particular, the RoleClassLib with IEC 62424-2 is structure as follow:

- `PersonelModel`
- `EquipmentModel`
- `MaterialModel`

Each of these objects are used as super classes to model the roles of different type of operators, equipment or materials that are part of the production.

9.2.3 Plant and Kinematics Simulators

Report [5] summaries the most used manufacturing simulators performing a comparison considering different functional and non-functional properties (modeling tool, license cost, interfaces,*etc.*). All of these simulators use discrete-event model of computation. These simulators have some common principles such as:

- *Layout Planning*: Represents the geometrical structure of the production line. Manufacturing simulators provides a set of components (*i.e.*, Source, Station, Conveyor,*etc.*) that can be parametrized with functional and geometric properties (processing time, dimensions, energy consumption,*etc.*).

- *Material Flow/Fluid Simulation*: Represents the transportation of the products from a process to the others. In particular, there are two main classes of products that are material, usually called MU or liquid products.
- *Process Simulation*: Represents the physical transformation made by the processes to products.

The main limitation of this class of simulators comes from the lacks of physical process modeling. In particular, it is possible to perform simulations only with static information of the physical process (time, energy consumption, *etc.*) with some statistical parameters.

However, this approach allows to perform an immediate evaluation of the production but driving to non accurate simulation of the plant due to the fact that the simulator is based on statical analysis and not on a model with kinematics.

In order to validate the entire methodology we adopted Siemens Plant Simulation and Siemens Process Simulate, part of the PLM Siemens Tecnomatix suite[1], to model the plant and a process.

Plant Simulator provides also *SimTalk*, an internal programming language that allows to define routines to customize the model and create or destroy objects at runtime. These routines can be triggered on events when, for instance, a MU enter or exit from a Station. Moreover, a set of interface objects allow to retrieve data from external sources through a set of protocols (OPC Classic, OPC-UA, Socket, COM, *etc.*).

Process Simulators usually allows to model and performing simulations of dynamic systems considering also geometric simulations in a 3D CAD environment. The main objective of these simulator is modeling stations in order to verify and optimize operations that can be performed by the equipment. For instance, defining strategies to avoid collisions between different robots or operators. For this part we adopted Siemens Process Simulate that is another tool part of the Siemens Tecnomatix suite addressed to model manufacturing processes.

As explained in section 9.1, the entire methodology is tool independent at it focuses mainly on the integration of information in AutomationML, needed to build the entire infrastructure. The methodology can be re-oriented to other simulators without changing the AutomationML source description.

9.2.4 Running Example

The following section gives a full explanation of the design methodology. For the sake of clarity, we pair the presentation with a running example. Figure 9.2 shows an AutomationML description using IEC 62424-2 rules. The description represents a small segment of a plant where a robot moves different products from a source conveyor to one of three destination conveyors. The `InternalLink` objects are used to define the topology of the plant, connecting respectively the robot and the conveyors. In `MaterialInformation` class, Box object represents the MUs

[1] http://www.plm.automation.siemens.com/

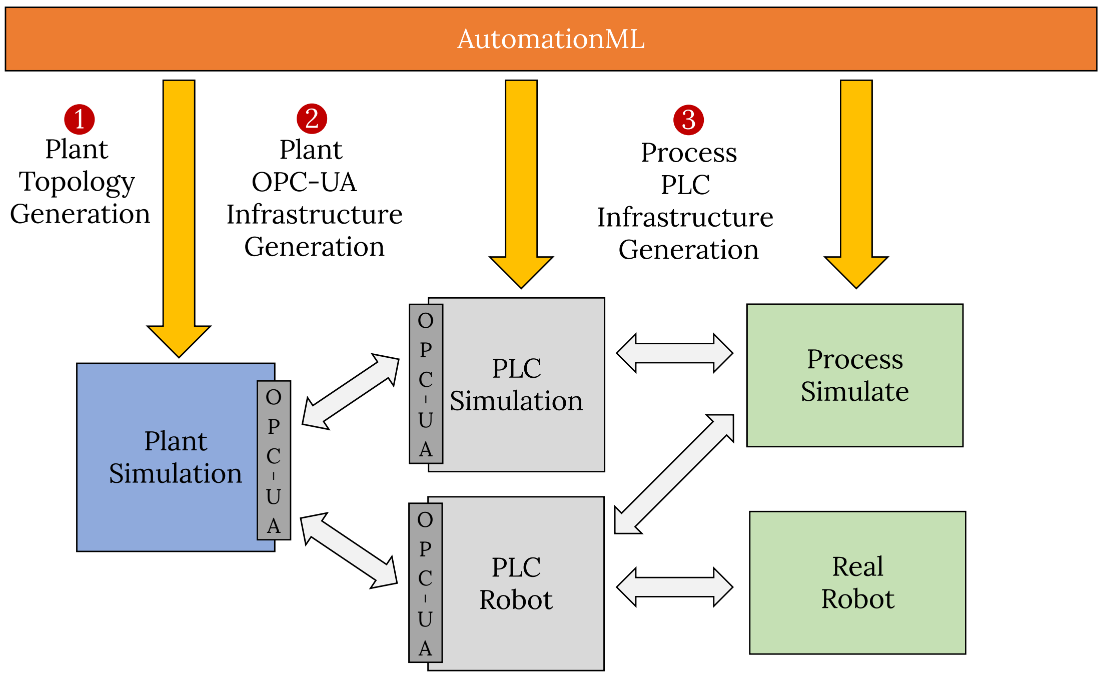

AutomationML
1 Plant Topology Generation
2 Plant OPC-UA Infrastructure Generation
3 Process PLC Infrastructure Generation
Plant Simulation
OPC-UA
PLC Simulation
OPC-UA
PLC Robot
OPC-UA
Process Simulate
Real Robot

Fig. 9.3: Overview of the steps that compose the generation of multi-level communication infrastructure.

that are moving through the plant. Each Box contains geometrical information (length, height, width, mass) and its destination (Conveyor 1, 2 or 3). The robot moves the boxes depending on the destination that is "marked" inside the object. The moving operation is affected by the mass of the box that is coded into 10 different masses. In particular, the robot represents the node where the multi-level is performed. As such, it provides a minimal while complete example for the proposed design methodology.

9.3 Methodology in Action

In this section, the entire design methodology is discussed with the use of the running example explained in the last section. Figure 9.3 shows the steps that compose the design methodology. The proposed approach starts with the AutomationML description reported in Figure 9.2 explained in 9.2. The entire flow is composed of the following steps:

- ① *Plant Topology Generation*
- ② *Plant OPC-UA Infrastructure Generation*
- ③ *Process PLC Infrastructure Generation*

The first step considers the generation of the plant model retrieving the information of the plant topology from the AutomationML description, then the following steps synthesized the communication infrastructure between the plant and process simulator.

Listing 9.1: Piece of AutomationML description related to the instantiation of a Conveyor.

```
 1 <InternalElement Name="Conveyor_1" ID="3b324033">
 2 
 3 <Attribute Name="Lenght" DataType="xs:string">
 4     <Value>5</Value>
 5 </Attribute>
 6 <Attribute Name="Speed" DataType="xs:string">
 7     <Value>5</Value>
 8 </Attribute>
 9 
10 <ExternalInterface Name="Input" ID="1e829f07"
11     RefBaseClassPath="AutomationMLInterfaceClassLib/
12     AutomationMLBaseInterface/Order">
13   <Attribute Name="Direction" DataType="xs:string">
14     <Value>In</Value>
15   </Attribute>
16 </ExternalInterface>
17 <ExternalInterface Name="Output" ID="41e42c8f"
18     RefBaseClassPath="AutomationMLInterfaceClassLib/
19     AutomationMLBaseInterface/Order">
20   <Attribute Name="Direction" DataType="xs:string">
21     <Value>Out</Value>
22   </Attribute>
23 </ExternalInterface>
24 
25 <RoleRequirements
26 RefBaseRoleClass="AutomationMLIEC62264RoleClassLib/
27 EquipmentModel/EquipmentClass/Conveyor">
28 </RoleRequirements>
29 
30 </InternalElement>
```

9.3.1 Plant Topology Generation

The AutomationML description has been enriched with additional information to better describe the topology of the plant. In particular, for each instantiated object two ports (`Input`, `Output`) have been added in order to code the direction of the link. For instance, it is easier to understand that the Robot has three output connections to the three conveyors(Figure 9.2). All the objects of the Siemens Plant Simulation MaterialFlow library have been mapped in the `RoleClassLib EquipmentModel` of AutomationML IEC 62424-2. In particular, some relevant attributes have been added to each object. For instance, the length and the speed for the conveyor object or the processing time for the Station object.

Listings 9.1 and Listing 9.2 report the AutomationML instantiation of the Conveyor_1 and the Robot. All the instantiated objects have a hexadecimal ID that unequivocally identifies them (Listings 9.1, 9.2 line 1). In listing 9.1, lines 3-8 show the two attributes related to the length and the speed of the conveyor. Listing 9.2, lines 3-25 represent the attributes needed to enable the multi-level simulation, that will be discussed in the next section.

Listing 9.3 reports the `InternalLinks` that encode the topology information of the plant. Lines 6-9 represent the link between the robot and the conveyor. `RefPartnerSideA` and

Listing 9.2: Piece of AutomationML description related to the instantiation of the Robot.

```
<InternalElement Name="Robot" ID="2dd0d757">

<Attribute Name="MultiLevel" DataType="xs:boolean">
  <Value>true</Value>
  <Constraint Name="MobileUnitDestinations">
  <NominalScaledType>
    <RequiredValue>Conveyor_1</RequiredValue>
    <RequiredValue>Conveyor_2</RequiredValue>
    <RequiredValue>Conveyor_3</RequiredValue>
  </NominalScaledType>
  </Constraint>
</Attribute>
<Attribute
  Name="Synchronization" DataType="xs:unsignedLong">
  <Value>10000</Value>
</Attribute>
<Attribute Name="PLCProtocol" DataType="xs:string">
  <Value>Kuka</Value>
</Attribute>
<Attribute Name="ProcTime" DataType="xs:string">
    <Value>5</Value>
</Attribute>
<Attribute Name="Operations" DataType="xs:string">
    <Value>3</Value>
</Attribute>

<RoleRequirements
RefBaseRoleClass="AutomationMLIEC62264RoleClassLib/
EquipmentModel/EquipmentClass/Manipulator">
</RoleRequirements>
<ExternalInterface Name="Output"
RefBaseClass="AutomationMLInterfaceClassLib/
AutomationMLBaseInterface/Order">
<Attribute Name="Direction" DataType="xs:string">
  <Value>Out</Value>
</Attribute>
</ExternalInterface>
...
</InternalElement>
```

Listing 9.3: Piece of the Plant AutomationML description that represents the topology of the plant. `InternalLink` are used to encode the physical connection between different equipment

```
<InternalLink
  Name="Link1"
  RefPartnerSideA="e3f29bb8:Output"
  RefPartnerSideB="2dd0d757:Input" />

<InternalLink
  Name="Link2"
  RefPartnerSideA="2dd0d757:Output"
  RefPartnerSideB="3b324033:Input" />
```

`RefPartnerSideB` represent respectively the source and the destination of the link and the attributes are the ID of the Robot and Conveyor.

Listing 9.4 shows the generated *Simtalk* method that contains the instantiation of all the objects of the plant for Siemens Plant Simulation. This method can be executed at the beginning of the simulation. Lines 2-4 represent the definition of the Box as a derived object of the MU Plant Simulation object. In particular, in line 3 is possible to see that the Box has been enriched by the attribute *Mass*. Other attributes (width, height, length) are provided by the MU superclass. Lines 10-11 show the instantiation of the robot of listing 9.2. Lines 13-15 show the instantiation of the conveyor presented in listing 9.1. In particular, lines 14-15 shows how the attribute length and speed are set in the conveyor object. Furthermore, lines 29-38 represent the entire topology of the plant, presented in AutomationML listing 9.3.

9.3.2 Plant OPC-UA Infrastructure Generation

In this section, the generation of the Plant OPC-UA communication infrastructure is discussed. The AutomationML description has been enriched with some attributes in order to identify and enable the multi-level simulation as mentioned in the previous section. Listing 9.2 lines 3-25 show the AutomationML attributes of the robot that are the following:

- `MultiLevel`: this boolean attribute is used to identify which is the multi-level node.
- `Synchronization`: This attribute is used to set how often Plant Simulation and Process Simulate have to synchronize. This parameter can be easily changed also during the simulation, in order to obtain an adaptable simulation. This number represents the number of MUs.
- `PLCProtocol`: This variable is used to uniquely identify the type of PLC protocol that uses the robot. Each vendor defines a specific PLC protocol. In this case, we are using a Kuka robot. The use of specific PLC protocol allows the integration of the real robot with Plant Simulation.
- `Operations`: It represents the number of operations that the robot can perform. In this case, the three operations represent the three different move actions to the conveyors.

Figure 9.3 shows two different actors used to perform communication and simulation between Siemens Plant Simulation and Process Simulate. This solution is necessary in order to better control the model and coordinate the two simulation tools. In particular, the separation of the two virtual PLCs allows also to connect the real robot, using in Plant Simulation real data coming from the field.

The communication between Plant Simulation and the two virtual PLCs is performed via OPC-UA protocol. Then, the two virtual PLCs are directly connected to Process Simulate. Figure 9.4 shows the sequence diagram of the communication protocol between Plant Simulation and Process Simulate via the two virtual PLC. This protocol is synthesized in *Simtalk* and it is associated with the Robot (Listing 9.4, line 11). The method is executed each time a new Box reaches the Robot.

The KUKA protocol is composed mainly of the following signals:

Listing 9.4: Simtalk Generated Method for the instantiation of the plant model.

```
var Pl := Models.Model
var Box := MUs.Part.derive(.Mus, "Box")
Box.createAttr("Mass", "Integer")
Box.Mass := 1

var Source_1:=Source.createObj(Pl,"Source")

var Conveyor_0:=Conveyor.createObj(Pl,"Conveyor_0")

var Robot:=Station.createObj(Pl,"Robot")
Robot.EntranceCtrl.load("Robot_KUKA_Protocol.txt")

var Conveyor_1:=Conveyor.createObj(Pl,"Conveyor_1")
Conveyor_1.length := 5
Conveyor_1.speed  := 5

var Conveyor_2:=Conveyor.createObj(Pl,"Conveyor_2")
Conveyor_2.length := 5
Conveyor_2.speed  := 5

var Conveyor_3:=Conveyor.createObj(Pl,"Conveyor_3")
Conveyor_3.length := 5
Conveyor_3.speed  := 5

var Drain_1:=Drain.createObj(Pl,"Drain_1")
var Drain_2:=Drain.createObj(Pl,"Drain_2")
var Drain_3:=Drain.createObj(Pl,"Drain_3")

Connector.connect(Source_1,Conveyor_0)
Connector.connect(Conveyor_0,Robot)

Connector.connect(Robot,Conveyor_1)
Connector.connect(Robot,Conveyor_2)
Connector.connect(Robot,Conveyor_3)

Connector.connect(Conveyor_1,Drain_1)
Connector.connect(Conveyor_2,Drain_2)
Connector.connect(Conveyor_3,Drain_3)

--Instantiation of OPC-UA Communication
...
```

- `App_start`
- `App_enable`
- `Robot_application`

The first part of the protocol represents the robot setup phase. First, the Box properties (width, lenght, mass) are sent to the Robot PLC. Then, the number of the total operations performed by the Robot node in Plant Simulation. This allows the Process model to synchronize its internal state with the Plant model, setting the correct wearing that could affect the processing time of the required operation. Then, the `app_enable` and `app_start` are rise up. `Robot_application`

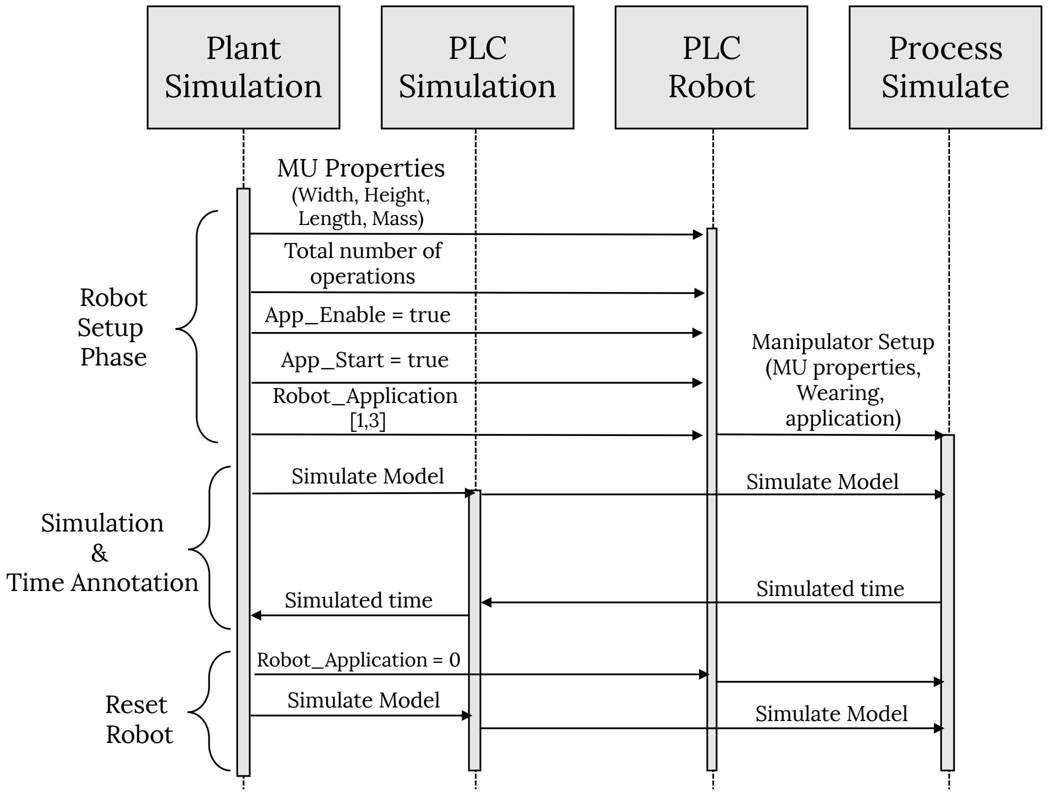


Fig. 9.4: Sequence Diagram of the communication protocol between Plant Simulation and Process Simulate.

signal represents the different move operation and it is set depending on the destination that is "attached" to the box (1, 2, or 3). The second part of the protocol is represented by the `simulate Model` signal that is sent to the PLC Simulation allowing to start the simulation of the process. At the end of the simulation, the simulated time is retrieved by the PLC Simulation and passed back to Plant Simulation. Plant simulation retrieves the timing annotation and sets it to the Robot Station. The Reset phase is necessary in order to bring back the robot to the initial position to perform another operation.

9.3.3 Process PLC Infrastructure Generation

This step represents the generation of the PLCs logics depending on the Robot protocol specified in the AutomationML description and reported in figure 9.4. The simulation of the virtual PLCs is performed by Siemens TIA Portal tool, part of Tecnomatix suite. Siemens TIA portal allows to program virtual PLC that exposes data also through an OPC-UA interface. The protocol is synthesized in a proprietary Siemens XML format, called PLC Openess, that is imported in Siemens TIA Portal.

In future, the PLC protocol will be described using PLC Open [114], then attached to the AutomationML description and finally synthesized directly in the target simulation environment.

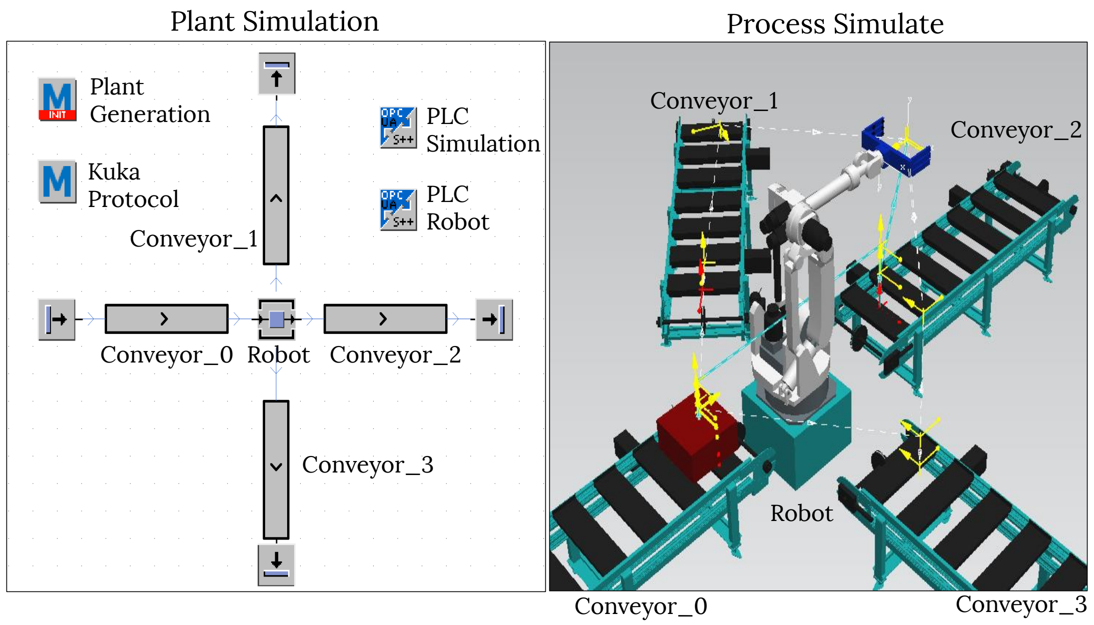


Fig. 9.5: Overview of the experimental setup generated with the design methodology.

9.4 Experiments

The methodology has been implemented as an automatic tool written in C++ using Xerces library [2]. The automatic tool parses the AutomationML description and generates all the necessary files to build the multi-level simulation infrastructure. The automatic tool has been tested using the discussed running example (Section 9.2). Figure 9.5 shows the obtained simulation environments, obtained with the design methodology.

Inside Plant Simulation environment it is possible to notice the equipment that composes the plant, the two OPC-UA modules (PLC Simulation, PLC Robot), and the method needed to enable the communication (KUKA Protocol).

Inside Process Simulate environment, the robot has been modeled according to the three different moving actions defined in the AutomationML description. Furthermore, the robot has been modeled considering mechanical wear due to the number of operations performed, which affects the time required to perform the different operations.

When the simulation reaches a synchronization point, Plant Simulation requires to Process Simulate the timing annotations for all the available recipes.

In the experiment, the number of recipes is obtained from the combination of the mass of the box and the conveyor destination. Moreover, Process Simulate also provides an option to increase the simulation performance trying to optimize the model. Table 9.1 shows the time required by the robot to perform the moving operation depending on the mass of the box and the conveyor destination, in a normal situation.

The proposed experiments perform a set of simulations using a different number of synchronization points to underline the benefit of the proposed design methodology.

[2] https://xerces.apache.org/xerces-c/

Box Mass (kg)	Simulated Time (s)		
	Destination		
	Conveyor 1	Conveyor 2	Conveyor 3
1	9,91	8,76	8,64
2	10,12	8,54	8,64
3	10,21	8,65	8,88
4	10,39	8,83	8,70
5	11,01	8,96	8,96
6	11,54	9,26	9,30
7	12,64	9,73	9,49
8	14,43	10,93	10,39
9	18,57	12,78	12,04
10	29,19	19,70	17,72

Table 9.1: Simulated time of different robot operations considering the mass of the boxes.

1000000 MUs	Synchronization	Simulated Time		CPU Time
	Number of MUs	DD:HH:MM:SS	s	s
Plant Simulation	-	134:18:14:36	11643243	126
Multi-Level	500000	145:11:59:30	12571170	333
	200000	148:14:19:02	12838742	569
	100000	152:21:49:44	13211384	986
	50000	155:17:16:28	13454188	2293

Table 9.2: The table reports simulation of Plant and Multi-Level approach with a batch of 1 million MUs.

In Table 9.2 the simulation of 1 million MUs has been considered. The first row of the table represents the simulation of Plant Simulation, without considering the physical process. In particular, the processing time of the robot has been set statically in Plant Simulation using the average of all the moving actions presented in Table 9.1. The following rows of the table show the results obtained using the proposed methodology, with a different number of synchronization points. The first column reports how often the synchronization between Plant Simulation and Process Simulate carry out, based on the number of processed MUs. The simulated time describes the simulated time at the end of the entire simulation. In the last column, the time required to simulate the entire batch of 1 million MUs is reported. The results show that Plant Simulation has proved to be the fastest solution, but providing inaccurate prediction regarding

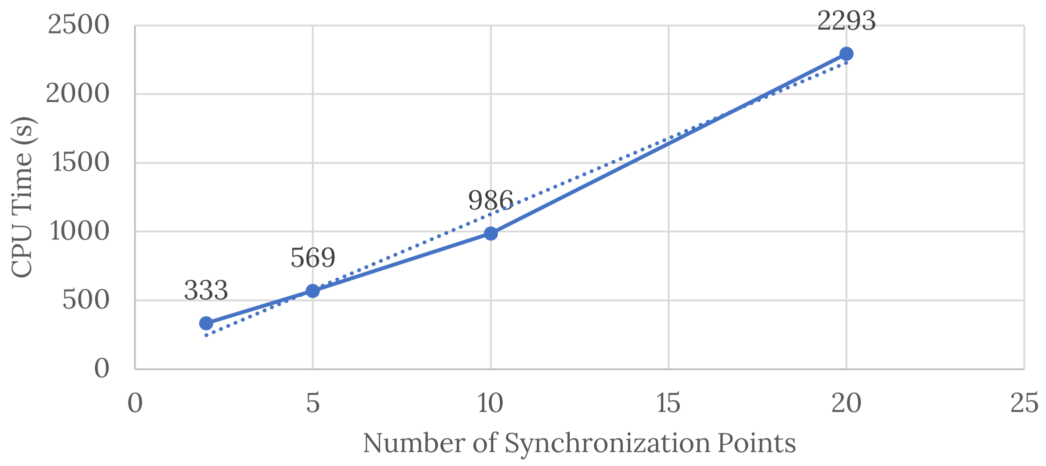


Fig. 9.6: Correlation of CPU Time and number of synchronization points.

the simulated time. Considering the Multi-Level approach, it is possible to notice that increasing the synchronization points, the simulation time is more accurate. This is due to the fact that the model of robot starts to introduce mechanical wearing and the operations require more time to be performed. The first row and the last row of the table clearly show that there is an important difference between the two simulated times (2 days circa). However, the Multi-level approach requires more computational effort than the only use of Plant Simulation, depending on how often the synchronization is set.

In figure 9.6 the number of synchronization points and CPU Time has been correlated. The relation between these two parameters demonstrates that the Multi-level approach in relation to the number of synchronization points is almost linear.

9.5 Conclusion

This paper proposed a novel design methodology to enable multi-level digital twins. In particular, the design methodology relies on the multi-level approach between a functional plant and kinematic process simulation. The proposed methodology enables accurate simulation of physical processes that can be used in a manufacturing plant model. The use of AutomationML neutral standard allows the adaptation of the entire design flow with different plant and process simulators. The methodology has been applied to specific industrial tools only to prove and validate the proposed approach. The obtained automatic solution can be adapted to different target environments starting from the same AutomationML description. Furthermore, the methodology has been explained with a real use case scenario that clearly shows the benefit of this approach.

Part V

From Real Data to Information

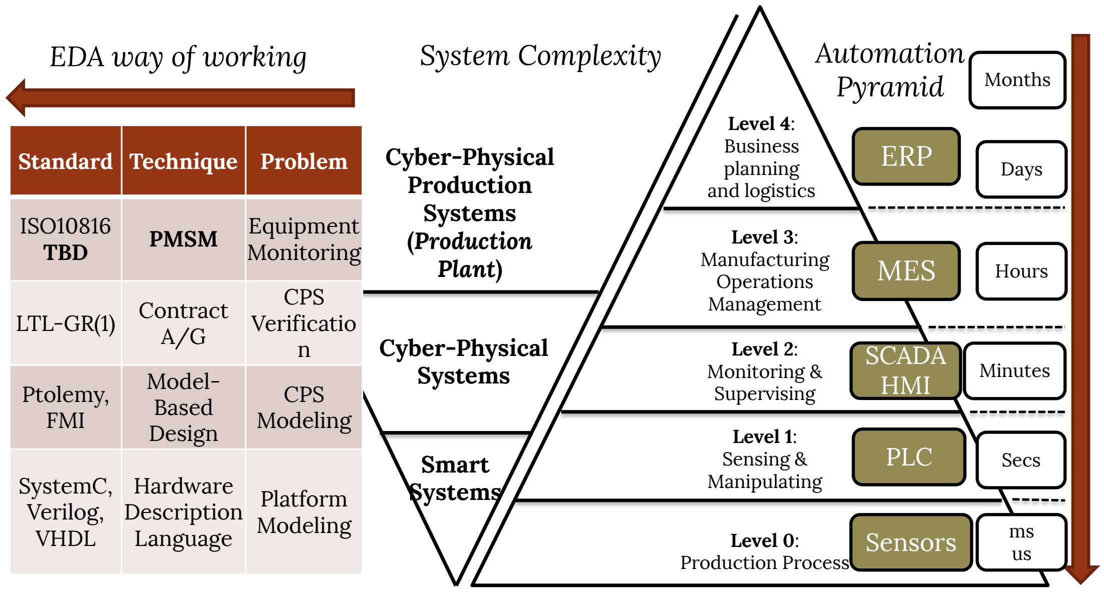
10

Industrial-IoT Data Analysis Exploiting Electronic Design Automation Techniques

10.1 Introduction

In the last decades, the number of Smart Systems dramatically increased. Thus, becoming a fundamental part of our life. Sensors can retrieve data from physical world and through digital processing, perform analysis. This process moves information from the physical domain to the digital domain creating the Digital Twin. The concept of a Digital Twin is not new, but in these years it becomes strictly correlated to manufacturing processes [1]. This concept can be seen as the modeling process of a physical phenomenon that refined or affected from real data. Thus, the Digital Twin represents the model, more or less abstracted, of real factory. The Digital Twin has the global view of the entire production line allowing data analysis verification and planning strategies targeted to maximize the production with real constraints coming from the real factory [4]. However, actually does not exist a reference development model that leads to the Digital Twin.

In the Electronic Design Automation (EDA) domain, the terms "model", "abstraction", "verification" are really familiar. For instance, the development processes of a IP-core must move through a set of refinement steps that mandatorily requires the use of these techniques. On the base of any EDA approach there is the process that starts modeling the problem, developing a solution technique based on formal models and languages and finally, when the solution strategy has been widely accepted, it generates standards. May we imagine to apply the same approach to Industry 4.0 problems by, hopefully, reusing the main results of the EDA evolution? This paper proposes a data analysis methodology based on EDA techniques to perform monitoring of a factory equipment. Figure 10.1 shows the strict relation between manufacturing automation and EDA techniques. In the right side, it is possible to see the *Automation Pyramid* that represents the reference hierarchical model of a manufacturing plant [115]. The *Automation Pyramid* moves from the production processes, at the lowest level, to the business and logistics that are on top of it. More in details, it is possible to see that all these levels have a different notion of time that depends on the operations they have to deal with. In the automation context, the *Automation Pyramid* is the mainly used paradigm for the identification of all the actors and them roles in the production plant. For this reasons, the EDA workflow must be coupled with this paradigm. The complexity of the corresponding system increases depending on how many


EDA way of working
System Complexity
Automation Pyramid
Cyber-Physical Production Systems (Production Plant)
Cyber-Physical Systems
Smart Systems
Level 4: Business planning and logistics
Level 3: Manufacturing Operations Management
Level 2: Monitoring & Supervising
Level 1: Sensing & Manipulating
Level 0: Production Process
ERP
MES
SCADA HMI
PLC
Sensors
Months
Days
Hours
Minutes
Secs
ms us

Standard	Technique	Problem
ISO10816 **TBD**	**PMSM**	Equipment Monitoring
LTL-GR(1)	Contract A/G	CPS Verificatio n
Ptolemy, FMI	Model-Based Design	CPS Modeling
SystemC, Verilog, VHDL	Hardware Description Language	Platform Modeling

Fig. 10.1: Overview of EDA Techniques with respect to ISA-95 Pyramid.

Table 10.1: Comparison of the principal measurements suitable for monitoring an industrial machine concerning the major condition monitoring standards. For each measurement are described the property that allows using of EDA techniques even for condition monitoring.

Property	Vibration	Oil Property	Acoustic Signal	Thermal Dissipation
Problem statement	ISO 7919, 10816, 13373	ISO 3734, 4406	ISO 22096	ISO 10880, 18251, 18434
Severity levels	ISO 10816	-	-	NFPA-70B
Measurement points	ISO 13373	-	ISO 3747	ASTM E1934, UNI 16714
P-F interval	weeks to month	month	weeks	weeks to month

variables need to be encapsulated in the model. For instance, Smart systems cover the lowest level of the complexity pyramid. CPSs describe the interaction within physical processes and digital components [116]. CPPS is the most complex system and it represents the Factory [117]. This is the most complex system in the manufacturing modeling domain.

The table on the left side of Figure 10.1 shows the usual "*EDA way of working*". The rightest column represents the starting problem, then on the left the emerged technique used to solve it and finally the generated standard. The lowest rows of the table represent problems, techniques and standards related to the electronic domain (*e.g.*, platform modeling, CPS modeling & verification).

However, in CPPSs context the reuse of EDA development process has been not yet explored.

The first row of the table analyzes the equipment monitoring problem that is specific of the automation domain. This paper proposes a novel approach, based on a EDA workflow, to solve this problem. The Predictive Maintenance State Machine (PMSM) emerged technique, based on EDA languages and FSM, is presented in Section 10.3. Section 10.2 discusses existing standard and what is missing for the standardization of PMSM technique. Section 10.4 reports some experiments for the validation of the approach. Finally, in Section 10.5 some conclusions are drawn.

10.2 IIoT Data Analysis

The observation of mechanical equipment is the mainly used technique to identify its health status. The healty status of a machine can be achieved through the observation of measurements coming from sensors. Research [118] at the State-of-the-Art was focused on the development process of IoT sensors (accelerometers, gyroscopes, pressure sensors, *etc.*) and summarized the most used sensors. Table 10.2 shows the principal sensors used in Industry 4.0. The interpretation of data sensors represents the primary issue related to the identification of the health status of equipment. In particular, it is difficult to understand when measurement can represent a significant deviation from the normal behavior of equipment. Some works tried to defined thresholds that can be used as guidelines to identify the severity of the measurement deviation.

The proposed methodology is derived from the application of EDA techniques, and it is supported by many international standards that formalize condition monitoring techniques in an industrial context. These standards represent the knowledge and experience on the industrial machineries of the committee that elaborated these indications. We tried to follow the same strategy, by identifying the main aspects that should be standardized to allow the application of EDA-inspired techniques to the monitoring problem. Such a problem is one of the most studied in Industry 4.0 and it is a the base of any predictive maintenance approach. The three main aspects necessary to implement the methodology are reported in Table 10.1 and concern the definition of:

- severity levels;
- measurement points;
- Potential-Failure (P-F) intervals.

This section starts with the definition of these three aspects then, the main variables used in condition monitoring are analysed and finally the proposed methodology is detailed.

10.2.1 Severity levels

It is fundamental to discretize the variables measured from the field, to reduce both the computational complexity to manage them and to set the critical thresholds. These critical thresholds are defined as severity levels for some variables used in condition monitoring. For example, for

Table 10.2: Example of sensors used in industry 4.0.

Measure	Vendor	Sensor Name	Description
Vibration	SignalLink	-	MEMS sensor
Vibration	LORD MicroStrain	G-Link-200	Wireless sensors
Vibration	Valmet	WVS-100	WLAN sensor
Temperature	RFMicron	RFM3250	Wireless sensor
Temperature	Yokogawa	YTA510	ISA100.11a sensor
Temperature	BB Smart	Wzzard	BLe sensor
Ultrasonic	Banner	Q45	Wireless sensor

the vibration variables, the ISO 10816 (*Mechanical vibration - Evaluation of machine vibration by measurements on non-rotating parts* [119]) standard defines various tables where the critical area is categorized for different kinds of machinery (see Figure 10.2). This standard defines four operating areas, with measures specified in mm/s for machines that operate in the range of 10 to 200 Hz (600 to 12000 rpm). In particular, the working conditions are subdivided from the first zone where vibration values are normal to the last one where a failure could occur at any time. This frequency range comprises various types of machinery *e.g.*, electric motors, pumps, compressor. Another standard deriving from a different domain, NFPA-70B (*Recommended practice for electrical equipment maintenance* [120]) defines various critical thresholds for thermal dissipation in electronic circuits. These severity levels once defined enable to set all the parameters necessary to run correctly the framework described in 10.3. In the case a recognized standard does not exist, it is necessary to define ad-hoc severity levels by following the way of working of the previously cited standards.

10.2.2 Measurement points

Once the severity levels have been defined, it is essential to define where the network of sensors must be positioned to perform the corresponding measurements. This point is crucial because an incorrectly located sensor could lead to a discrepancy in the prediction of the state of health of the machine. Various standards define a correct procedure to ensure the most possible correct measurements.

Concerning vibration measurements, the ISO 13372 standard specific for non-rotating parts of a machine specifies where sensors must be located on the machine side [121]. Rules are defined for the correct measurement of other variables, for example for acoustic emission and thermal dissipation [122, 123].

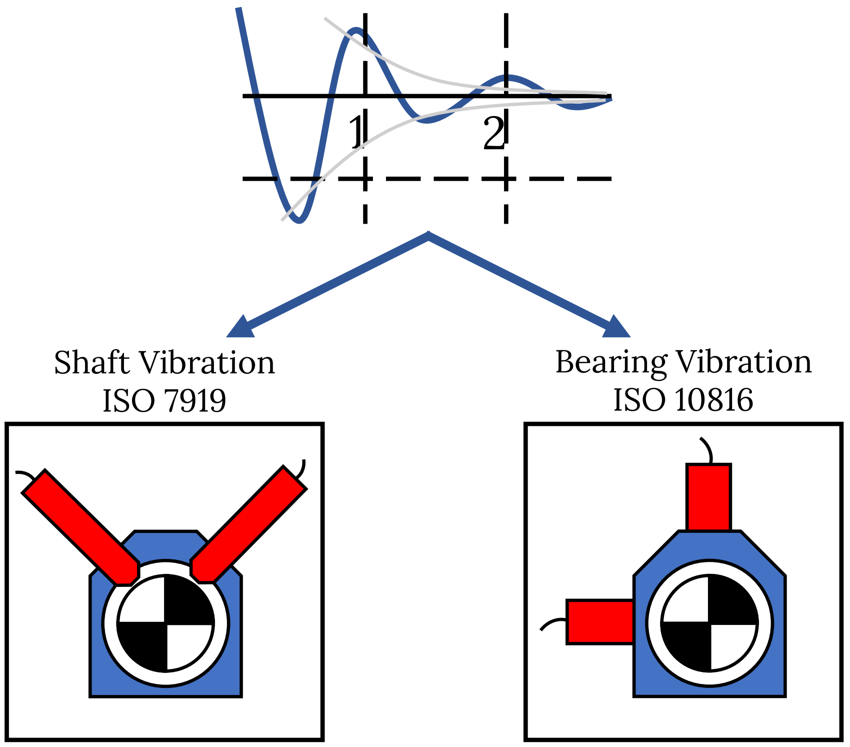


Fig. 10.2: Vibration standards.

10.2.3 P-F intervals

It is essential to prevent the breakage of an industrial machine by monitoring specific parameters. Often these breakages are not linked to the aging of the machinery but associated with a set of conditions that are no longer verified. The mapping between a fault and the cause that caused it is often 1: N, so different techniques of condition monitoring are used to prevent possible failures. Through the monitoring of different machine variables, it is possible to detect anomalies. When an anomaly is detected, the machine is in the condition in which a fault has already started to modify its functionality, even if this variation is not observable in this initial stage. It is therefore crucial the period between the detection of the fault and the point where the functional fault is in the act. This temporal interval is known in the literature as P-F interval [124] (see Figure 10.3). It is essential, therefore, to estimate in advance this interval with high accuracy. Each variable that can be monitored has its intrinsically associated P-F interval. For example, the P-F interval associated with vibration measurement and engine oil measurement is not the same. The P-F interval associated with vibration measurement is smaller than the P-F interval for engine oil measurement because, through advanced analysis of a machinery's engine oil, it is possible to detect in advance an anomaly.

10.2.4 Common variables in condition monitoring

Table 10.1 shows a comparison of the most common variables used for condition monitoring, with a focus on the principal standards cited for the aspects to be considered.

Vibration

Major studies in the field of condition monitoring are based on vibration analysis [125]. Vibration analysis is used to detect the energy emitted by a mechanical component as vibrations. Significant variations in vibration may indicate problems with machine wear or misalignment. Vibration-related parameters that can be measured are displacement, speed, and acceleration. Several analyses can be carried out on these parameters, *e.g.*, time waveform, broadband vibration, and spectrum analysis. For the vibration analysis, is the most used technique, many standards can be used for the definition of the main points of our methodology. For example, the family of standards ISO 10816 [119], ISO 7919 [126] and ISO 13373 [121] define the problem statement. They also outline the guidelines for the definition of severity levels and the points at which to perform the measurements.

Oil property

The analysis of the engine oil can be carried out on different types of machine's oil. Through the oil component analysis it is possible to detect a degradation of the machine, a contamination or a oil consistency different from the requirements. Different analysis can be performed on the oil, the main ones are ferrography, infrared and ultraviolet spectroscopy analysis and particle counter [127]. However, these analyses can mainly be carried out in the laboratory and not on board the machine and involve significant additional costs compared to monitoring other variables. For the analysis of engine oil standards as ISO 3734 [128] and ISO 4406 [129] define only the problem statement. It is a very complex technique and requires laboratory analysis, no standardized levels of severity or measurement points have been established.

Acoustic emission

The analysis of acoustic signals allows to carry out analyses related to degradation and stress of industrial machinery [130]. For the analysis of acoustic emissions standards such as ISO 22096 [131] define the problem statement. No standard defines severity levels to determine critical thresholds in measurements while the ISO 3747 [122] standard defines guidelines to perform the measurements.

Thermal dissipation

The analysis of the thermal dissipation allows to identify possible failures related to the temperature of a component [132]. Significant changes in temperature can indicate both mechanical problems such as excessive mechanical stress that could be caused either by a faulty bearing or by inadequate lubrication or electrical problems in the system itself. Temperature measurements can refer to measurements made outside or inside the machine. Various sensors allow

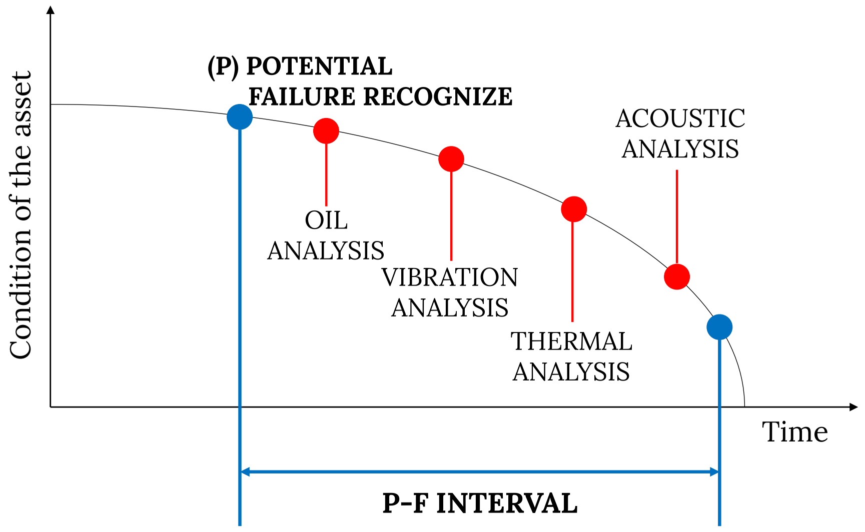


Fig. 10.3: P-F interval - Intervals between the point at which a potential fault is detected and the point at which the fault occurs.

to measures temperature, *e.g.*, thermocouples, Resistance Temperature Detectors (RTDs), and infrared thermography camera.

The advantage of using a thermography camera is that it is portable and allows to observe the temperature of a very large area. For the analysis of thermal dissipation, several standards can be useful to define the main points of our methodology. For example, the family of standards ISO 10880 [133], ISO 18251 [134] and ISO 18434 [135] define the problem statement. A formalization of severity levels it is defined in the standard NFPA-70B [120] that defines critical thresholds for thermal dissipation within electronic circuits while standards as ASTM E1934 [136] and UNI EN 16714-3 [137] define guidelines to perform measurements.

10.3 Predictive Maintenance

The proposed methodology describes a novel technique to translate raw data coming from different sensors in severity levels. The role of these severity levels is to measure the quantity of deviations of the equipment for a determined measurement. In the context of a production line, any equipment usually performs a finite amount of operations depending on the products that are produced. For instance, let us consider a factory that produces three different products. The input domain of equipment depends on the production recipes of the products. More in details, an equipment could have a set of real parameters that allows identifying a large number of setups. The same equipment in a production line uses a limited number of setup depending on the production recipes. This assumption allows to reduce the number of inputs of the machine and allows to identify the normal behavior of the equipment for each production recipe.

The normal behavior is compared to the actual behavior of the equipment depending on the current product. Ideally, a deviation represents a measurement that is diverging respect to a ref-

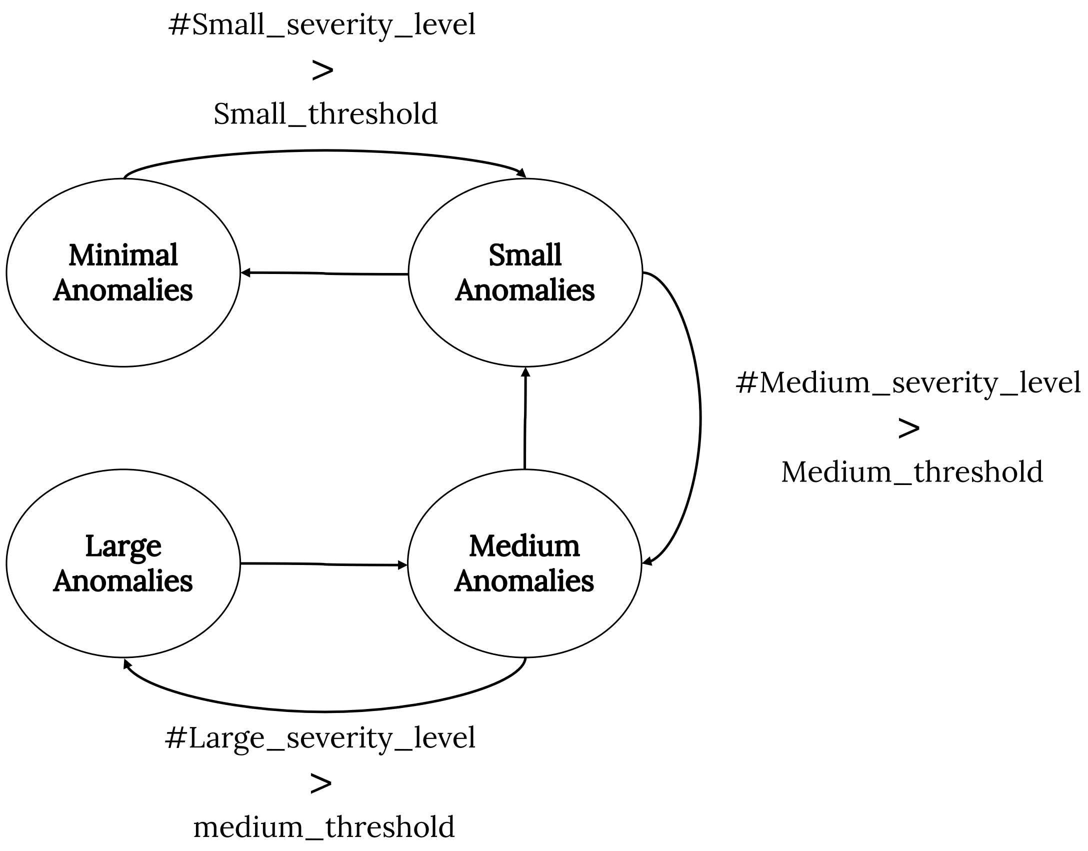


Fig. 10.4: Overview of a generic PMSM. States represents the deviation observed from the Normal Behavior.

erence signal or crossing a threshold. The identification of a normal behavior allows obtaining a fixed threshold with simply comparing the actual behavior with the normal behavior. This is a normal filtering technique used for instance in Audio signal processing (i.e. mobile phone noise reduction), where the reference signal is used in opposition to the current signal. The obtained result represents the global deviation of the actual behavior respect to the normal behavior. This deviation allows setting a fixed threshold that is used to capture deviations from the normal behavior. Moreover, the threshold is used also to avoid deviations caused by sensing errors and capture only real deviations.

Figure 10.5 depictes the main components of the framework that have been implemented as an OPC Unified Architecture (OPC UA) client.

Let us introduce an actor called *Severity Level Classifier*. This actor performs three operations:

- Storing normal behavior traces for each product;
- Filtering actual behavior with normal behavior trace comparison;
- Translating deviation into severity level.

The first action is the learning phase where the *Severity Level Classifier* stores the reference traces for each product needed from the filtering phase. The filtering phase is the same men-

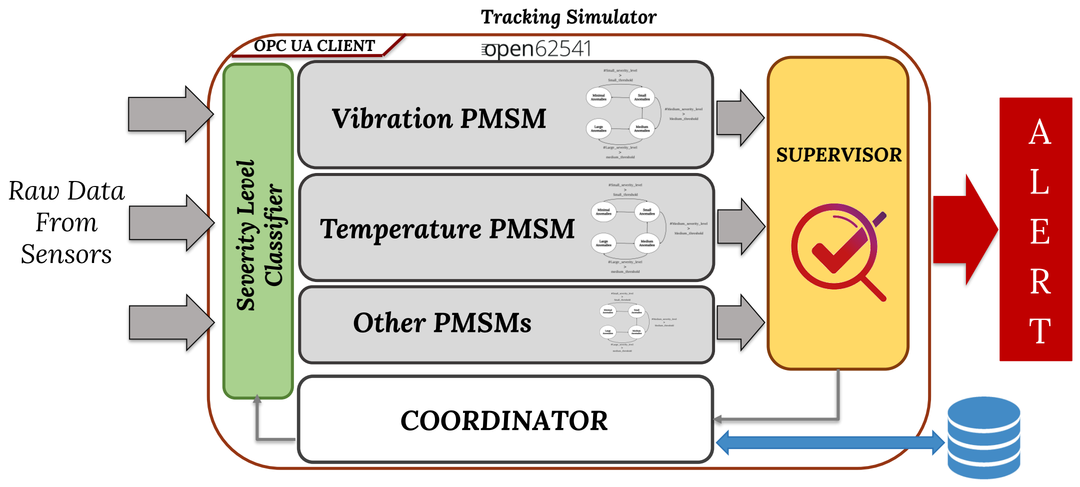


Fig. 10.5: Overview of the framework implemented as an OPC UA Client.

tioned before. The third action is performed by considering the number of deviations that have crossed the fixed threshold. When a deviation is detected, the *Severity Level Classifier* increments an internal counter. When the comparison is completed, this actor maps the counter value into a severity level.

This approach defines the following severity levels:

- `Minimal severity`;
- `Small severity`;
- `Medium severity`;
- `Large severity`.

More in details, when the internal counter of the *Severity Level Classifier* is minimal it represents a minimal deviation from the normal behavior of the equipment. It means that the equipment health status is good. The *Severity Level Classifier* maps this counter into the `Minimal Severity` level. In another case, the counter could be higher. This means that the equipment is deviating for an important amount of time than its normal behavior. In this case, the Classifier will map the counter into `Large Severity` level.

The goal of predictive maintenance is to anticipate equipment failures in order to optimize production and plan maintenance strategies. In this scenario, an equipment maintainer can establish the status of an equipment by observing it through measurements of vibrations, temperature, energy consumption, *etc.* When these measurements are deviating from a normal behavior the equipment could perform inaccurate operations or stopped for a failure. With the analysis of these measurements, it is possible to define the state of health of the equipment.

This paper introduces a new method to achieve Predictive Maintenance, called PMSM. A PMSM is a finite state machine representing the health status of equipment for a certain observable measurement. Formally, a PMSM is a transition system that depends only on inputs. The output of a PMSM is its actual state. A PMSM is composed of four states:

- `Minimal Anomalies`;

- `Small Anomalies`;
- `Medium Anomalies`;
- `Large Anomalies`.

Each state represents the health status of the observed measurement.

For instance, let us assume the PMSM in Figure 10.4 represents the status of Vibration measurement of a bearing that could be a critical component of an equipment. In this case, the `Minimal Anomalies` state is used to represent minimal deviations of vibrations from the normal behavior of the bearing. The `Large Anomalies` state represents an important deviation of vibrations that could lead to failure. Inputs for a PMSM are the severity levels. Each time the equipment makes an operation the vibration deviations are translated into severity levels.

In a real factory, a maintainer tries to classify the health status of an equipment by observing its measurements. For instance, the maintainer could identify states like "equipment working", "equipment deteriorating", "maintenance requires" and "broken". This is usually performed manually with personal knowledge of the maintainer acquired with the experience. In the last Section, PMSM has been presented with the role to monitor the deviations of observable measurements. This section introduces a new actor called *Supervisor*. The role of this actor is to monitor all the PMSMs in order to raise alert when the equipment needs maintenance.

Moreover, it also notifies when an imminent failure of the equipment is incoming. The *Supervisor* is defined as a finite state machine that analyses the current state of all PMSMs and expose its current state as output. The internal states of the *Supervisor* are:

- `Good`;
- `Maintenance`;
- `Imminent Failure`.

The initial state of the *Supervisor* is the `Good` state and it is used to represents the normal behavior of the equipment. The `Maintenance` state represents the state in which the *Supervisor* detects `Medium Anomalies` state from PMSMs for a certain time. The `Imminent Failure` state is used to raise an alert when the deviations of the observed measurements are compromising the equipment. Transitions between a state to another are defined depending on the physical properties of the equipment and on the number of PMSMs available. It is important to clarify that the main contribution of this paper concerns by the structure definition of a customizable framework.

10.4 Experimental Validation

This Section explains the methodology implementation and its application to a model of a real use case scenario. More in details, the methodology has been implemented with the use of two support technologies (OPC UA, FMI).

10.4.1 Experiment Setup

Now the experiment setup necessary to validate the methodology is described:

OPC UA Client

The two main components of the methodology: PMSMs and *Supervisor* are described with VHDL, an HDL. This allows defining into a single description the finite state machine and the control for each state of the FSM. Once defined these components into a VHDL description it is necessary a further step to proceed with the integration into the OPC UA Client.

HIFSuite is a tool that provides a set of tools for the manipulation of Hardware models [44]. In this work, HIFSuite has been used to obtain C++ code starting from Hardware models [44]. It defines a proprietary format called HIF (Heterogeneous Intermediate Format). The framework provides a front-end tool that translates Hardware models into HIF format. Then, manipulation tools can be applied to these translated models. In particular, the framework provides two manipulation tools to perform abstraction [20]. Finally, a back-end tool can be used to generate the C++ code that represents the abstracted Hardware Model.

The obtained C++ abstracted models of PMSMs and *Supervisor* are easily integrated into the OPC UA Client. Furthermore, into the OPC UA Client, a Coordinator is responsible for the product-accurate execution of all the components. This means that the frequency of execution of PMSM and *Supervisor* is correlated with the frequency of production of each piece. The OPC UA Client received the severity levels and processed it into the core.

OPC UA Server

The *Severity Level Classifier* it is modeled using VHDL, an HDL. It is translated into C++ code with the HIF Suite with the same toolchain used for PMSM and *Supervisor*. This module is integrated into an OPC UA Server, this makes it possible to abstract from the machine. If the real machine is available, the *Severity Level Classifier* will be integrated into the machine proprietary OPC UA Server. Otherwise, an abstract OPC UA will be modeled, as it happens in this experiment necessary to validate the framework. This OPC UA Server incorporates the *Severity Level Classifier* and the library that allow simulating the model of the abstract machine (see Section 10.4.2) with the FMI interface.

10.4.2 Abstract model of a real machine

A Driveline Two-Speed Transmission System is used as a reference model of a real machine. This model represents a critical subsystem of manufacturing equipment. In particular, the drive shaft is a critical subsystem of manufacturing equipment. This manufacturing equipment produces three different products: A, B, C. The critical subsystem received different inputs that allow modeling the product production. Mathworks Simulink © is used to model the critical subsystem. The transmission system consists of a torque drive, drive shaft, clutch and high and low gears connected to an output shaft.The model incorporates two sensors: vibration and temperature sensors. Vibration sensors monitor the casing vibrations. The casing model translates the shaft angular displacement to a linear displacement on the casing. Temperature is used to monitor the oil temperature of the casing. The counters are implemented in the module *Severity Level Classifier* into the OPC UA Server.

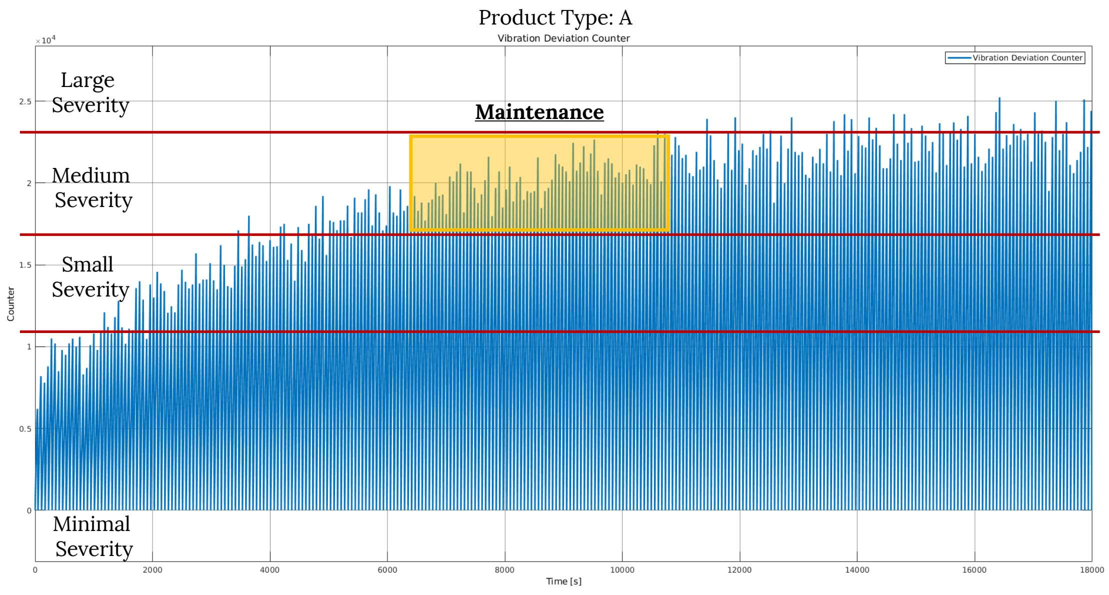


Fig. 10.6: Vibration deviation counter, product type A, series: 300 pieces.

10.4.3 Methodology validation with a mutation analysis technique

In this Section, the correctness of the methodology is shown. In particular, the Structure validation mode of PMSM is considered. The technique used is derived from the mutation analysis used in software testing [138]. The abstract model of a real machine used as a reference has been faulty. The fault injected is a gear tooth fault that is part of the transmission shaft. This fault is modeled by injecting a disturbance torque at a fixed position in the rotation of the drive shaft. The magnitude of the disturbance is controlled externally during the simulation. This faulty model is simulated into the OPC UA Server. The simulation step is set with the same frequency of a real sensor. In this context, the point of breakage is set by analyzing the simulation results at 17500s of simulation. In a real context, this point of breakage is set by analyzing the machine's documentation, in particular, Mean Time To Failure (MTTF) and Mean Time To Repair (MTTR), or Failure Mode and Effect Analysis (FMEA) report.

Figure 10.7 and Figure 10.6 shows the simulation results of 18000s, equivalent to the production of 300 pieces for the temperature and vibration sensors. In particular, the measurements are related to the counters of the *Severity Level Classifier*. Figure 10.7 shows the numbers of deviation of the oil temperature with respect to a normal behavior. The *Supervisor* detects correctly the anomalies caused by the faulty model, and the internal state of the *Supervisor* switch from a healthy state to a maintenance state before the broken point of the machinery.

10.4.4 Experiment results

Experiments have been run on a 64-bit server with 3.40 GHz cores and 32GB of RAM and running on Ubuntu 18.04 Linux OS. All the results are related to a simulation of 18000s, equivalent

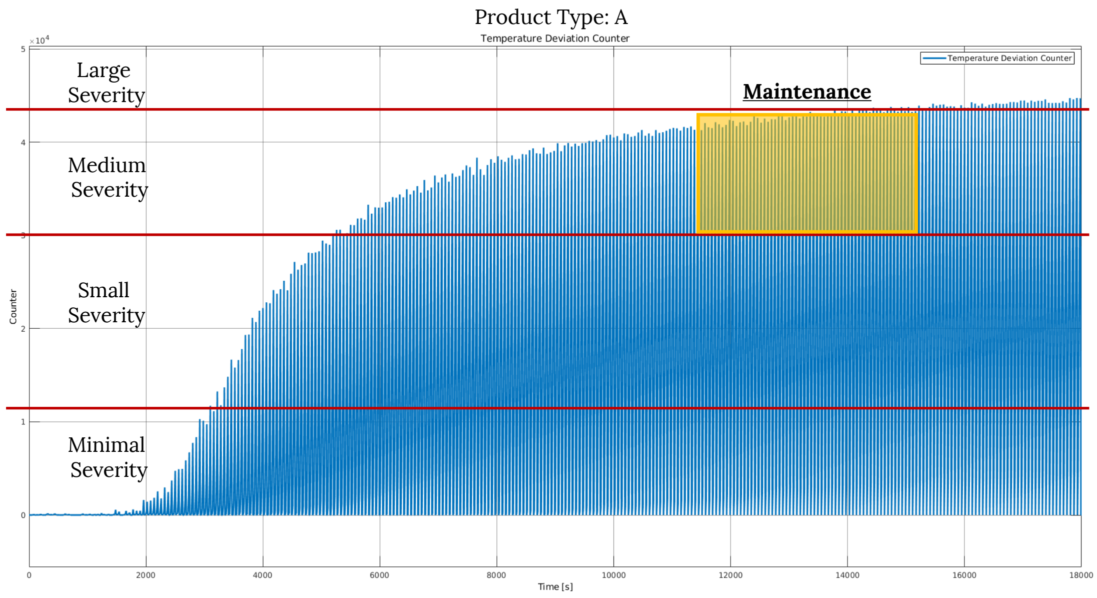


Fig. 10.7: Temperature deviation counter, product type A, series: 300 pieces.

to the production of 300 pieces with the point of breakage set to 17500s. In this Section, two observable measurements are related to inferred early the anomalies in the abstract model of a real machine 10.4.2. The approach has been validated with three different experiments based on the observation of measurements.

In the first experiment, the framework observes only the Temperature of the Casing, while in the second experiment considers only vibration measurements related to casing Vibration. The third experiment relies on the observation of the vibration and the temperature. The *Severity Level Classifier* recorded the normal behaviour related to the three different products A, B, C. The transition thresholds of the temperature PMSM and the *Supervisor* has been manually set according to the deviations of the model. In this case, the *Supervisor* transition between `Good State` to `Maintenance State` was approximated. More in details it means that the *Supervisor* needs to be sure to raise an alert, only a significant deviation is detected. The same process has been done for the second experiment but considering Vibration measurements.

In the third experiment both the measurements have been considered, Vibration and Temperature. By combining these two measurements of a specific critical component of a most complex machine it is possible to switch early the internal state of the *Supervisor* to the maintenance state. For example, considering the production of 300 pieces of type A, the temperature deviations from a nominal behavior is shown in Figure 10.7. This measure combined with the Vibration measure related to the deviations from a nominal behavior (see Figure 10.6) it is possible to predict early the anomalies associated with the monitored critical component. To predict early anomalies, the *Supervisor* used to achieve the experimental results in Sectionwith one PMSM was adapted. Considering two PMSMs, one for monitoring temperature and one for vibration the *Supervisor* must allow considering both. To combine the information of the two PMSMs, the behavior of internal state machines of the *Supervisor* is changed. Considering the situation with medium variations respect to the nominal behavior for the temperature are detected, and the state of the *Supervisor* is in the state of Good health. When variations are also detected for the vibrations, the *Supervisor* will change its internal state to a maintenance state before respect to the case of one variable monitored.

Table 10.3 shows the results of the faulty model related to the simulation of 18000 sec and considering the production of 300 pieces of type A, B and C. It shows the simulation time in which the framework recognizes potential failure. This point of potential failure is related to the internal states of the *Supervisor*. In particular, the *Supervisor* switches to the Maintenance State in correspondence with this timing point. In the first experiment, considering product A, the observation of the oil Temperature has shown that the *Supervisor* moves from Good State to Maintenance state after 8300s. The same experiment with Product C has produced a Maintenance alert after 8000s. This is due to the fact that Product C induced more mechanical stress than Product A.

Table 10.3: Experimental results with different productions in relation to different observed measurements in a simulation of 18000s. The results represent the time necessary to arrive in the `Maintenance State`.

Monitoring	Product Type		
	A	B	C
Temperature	8300s	8200s	8000s
Vibration	8000s	7900s	7800s
Temperature & Vibration	6500s	6100s	6000s

The observation of the vibrations in the second experiment has shown that in all of the three productions, the *Supervisor* detect anomalies before the first experiment. It is possible to observe that a mechanical degradation of the casing starts to generate vibrations, and the temperature of the casing starts to increase due to vibrations that will affect the oil temperature. The combination of both the measurements allows detecting anomalies before the observation of only one measure. In fact, considering the production of A, the *Supervisor* switched to the `Maintenance State` after 6500 seconds.

10.5 Conclusions

This paper applied the typical "*EDA way of working*" to the industrial automation context, in particular to the predictive maintenance problem. Standards have been identified as basis of the proposed technique that uses modeling formats of the EDA domain. Automation and EDA domains have been thus merged to perform data analysis of IIoT sensors.

11

The Design of a Digital-Twin for Predictive Maintenance

11.1 Introduction

Nowadays terms like Smart Factory, Digital Twin, Industry 4.0 [1] announce the dawn of a new era for intelligent automation. Computers should be able to monitor a real plant and make decisions to optimize productions and reducing costs [4]. This requires to analyze data from the real plant and to integrate them into a model representing the factory usually called Digital Twin. The concept of a Digital Twin is not new, but in these years it becomes strictly correlated to manufacturing processes [1]. The Digital Twin has the global view of the entire production line allowing data analysis and strategies planning targeted to maximize the production with real constraints coming from the real factory [4]. However, actually there is not a reference development model that leads to the Digital Twin. One of the most critical point of this problem regards the health status analysis of equipments and reflect this information to the related model. The encapsulation of this information inside the model of a Digital Twin is usually a hard challenge that requires a lot of time spent to refine the entire model. For instance, these information could be retrieved from technical knowledge of vendors, equipment maintainers or data analysis coming from sensors. All the mentioned strategies are not easily and quickly adaptable to the equipment model.

In literature, many works tried to solve this issue by proposing ad-hoc solutions for specific cases. However, these solutions do not define a standard way-of-working to face the problem. Authors of [139] have been discussed some concepts to engage the problem with a higher level of abstraction. The authors proposed the reuse of techniques coming from the EDA domain adapted to the Automation domain. The EDA way-of-working tried to define a unique path that starts by formalizing the problem, moves to the solution by analysing design automation techniques and finally defines new standards [140–142]. In this paper, we will apply this approach with a particular focus on the health status analysis of an equipment that can be reused in different scenarios. In particular, a novel approach that leads to the definition of equipment health status based on observable measurements, and finite state machines.

Figure 11.1 summarizes the contribution of this work. The entire flow starts from the data-generation that can come from sensors of a critical component in a real equipment or a model (vibrations, temperature and acoustic emissions). First, the vibration data are collected and in-

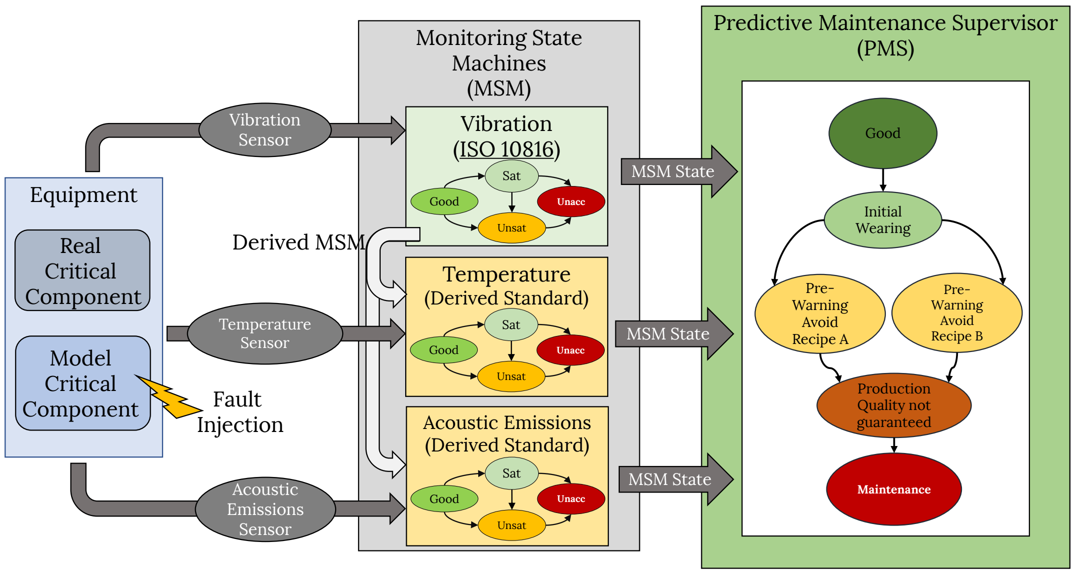

Monitoring State Machines (MSM)
Predictive Maintenance Supervisor (PMS)
Vibration Sensor
Vibration (ISO 10816)
Sat
Good
Unacc
Unsat
MSM State
Equipment
Real Critical Component
Model Critical Component
Derived MSM
Temperature Sensor
Temperature (Derived Standard)
Fault Injection
Acoustic Emissions Sensor
Acoustic Emissions (Derived Standard)
Good
Initial Wearing
Pre-Warning Avoid Recipe A
Pre-Warning Avoid Recipe B
Production Quality not guaranteed
Maintenance

Fig. 11.1: Overview of the proposed approach.

tegrated into a finite state machine called Monitoring State Machine (MSM) that defines the severity level, based on ISO-10816, of the actual measurement. Then, the comparison between the *vibration MSM* and data coming from other sensors (temperature, acoustic emissions) allows defining MSM for other observable measurements. Finally, Predictive Maintenance Supervisor (PMS) is the actor that monitors all the MSMs and defines the health status of the critical component of the equipment under monitoring. The use of a model is fundamental to refine the PMS, but more important, it can be used to perform predictive maintenance via simulation through faults injection on the model. Moreover, the identification of the health status of the equipment allows to reflect this information in the model and obtaining the Digital-Twin.

The main contributions of the proposed solution are:

- Definition of a design flow for predictive maintenance oriented Digital-Twin;
- **Flexibility** to be adapted to other equipment;
- **Reuse** of existing standard (ISO-10816) to define new standards for other observable measurements;
- Evolution from condition-based to **predictive maintenance via simulation** with a unique framework.

The paper is organized as follow: Section 11.2 presents the necessary background and the technologies uses to build the framework. Section 11.3 discusses about the definition of severity levels, starting from sensor raw data and MSMs. In Section 11.6 the PMS is explained, while Section 11.8 reports the use of the entire framework in condition-base mode and predictive maintenance reporting experimental results, finally, Section 11.9 reports conclusions and possible future works.

Machine			Class I Small Machines	Class II Medium Machines	Class III Large Rigid Foundation	Class IV Large Soft Foundation
	in/s	mm/s				
Vibration Velocity Vrms	0.01	0.28	🟩	🟩	🟩	🟩
	0.02	0.45	🟩	🟩	🟩	🟩
	0.03	0.71	🟩	🟩 GOOD	🟩	🟩
	0.04	1.12	🟨	🟩	🟩	🟩
	0.07	1.80	🟨	🟨	🟩	🟩
	0.11	2.80	🟧	🟨 SATISFACTORY	🟨	🟩
	0.18	4.50	🟧	🟧	🟨	🟨
	0.28	7.10	🟥	🟧 UNSATISFACTORY	🟧	🟨
	0.44	11.20	🟥	🟥	🟧	🟧
	0.70	18.00	🟥	🟥	🟥	🟧
	1.10	28.00	🟥	🟥 UNACCEPTABLE	🟥	🟥
	1.77	45.90	🟥	🟥	🟥	🟥

Table 11.1: ISO 10816: Definition of vibration severity levels with respect to the machine class.

11.2 Background & State of the art

11.2.1 Technologies

This section gives an overview of OPC UA [143] and FMI [21] that are the mainly technologies used in this work.

OPC UA represents a de-facto standard in industrial automation; it has been developed by the OPC Foundation. The goal of the standard is to unify the communications between different actors of a manufacturing plant. In this context, the OPC UA standard enables communications between the different levels of the automation pyramid, both horizontally (Machine to machine), and vertically (Machine to ERP and viceversa). Moreover, the standard defines a set of profiles. An OPC UA server that implements a certain profile allows to use a set of specific functions defined by the specific profile. For instance, the `Historical Data Access` profile allows retrieving Historical Data that are stored in the OPC UA server. Exposed data of each server are visible in a special structure called `Information Model`. OPC UA allows structuring the exposed data with the use of Object-Oriented paradigm through an XML Schema defined by the standard.

FMI is a standard born from MODELISAR cooperative in 2008 [58]. The goal of the standard is to define a clear and usable interface for dynamic models. This promising standard allows exchanging models described in different languages enabling the cooperation of heterogeneous models, avoiding co-simulation techniques. A Functional Mock-up Unit (FMU) represents the basic block of the standard. The standard is based on the separation of interface and functionality. The FMI defines a set of C-APIs that wraps the model allowing to interact

	Machine			**Vibration Severity Level**	**Temperature Severity Level**
Measurements	Vibration	Temperature	Acoustic Emission		
	0.28	...	...	🟩	🟩
	0.45	...	...	🟩	🟩
	0.71	...	...	**GOOD**	
	1.12	...	...	🟩	🟩
	1.80	...	...	🟩	🟩
	2.80	...	...	**SATISFACTORY**	
	4.50	...	...	🟨	🟨
	7.10	...	...	**UNSATISFACTORY**	
	11.20	...	...	🟧	🟧
	18.00	...	...	🟥	🟥
	28.00	...	...	**UNACCEPTABLE**	
	45.90	...	...	🟥	🟥

Table 11.2: Definition of severity levels for other observable measurements. In this case temperature and acoustic emissions are considered.

with it. More in details, these functions define how to set or get a value from the interface of the model and how to simulate it. Then, the wrapped model is compiled as a shared library. The interface of the model is described through an XML file, called `modelDescription.xml`, with a well-defined XML schema. A FMU is a zip file that contains the obtained shared library and the *modelDescription.xml*. When a target environment imports a FMU, it can easily identify its interface from the XML file and interact with it through the C-APIs.

11.2.2 Related Work

The identification and prediction of faults of an equipment has been studied in a multitude of researches [144–146]. Two principals classes of maintenance have been identified: time-based and condition monitoring. The main difference between these two classes depends on the maintenance strategy. The first relies on periodical maintenance cycles, while condition monitoring relies on the status of the equipment. Concerning time-based maintenance, failure models have been defined to estimate the failure percentage of equipment. However, these failure models are not accurate enough. In fact, the precision of these models detect only 11% of all the failures [147].

In [148] the authors analyze the correlation between vibration and wear debris that can occur from mechanical wear. Yaqub et al. [149] proposed an interesting approach based on the classification of vibrations into severity levels. This is a promising approach, but it is strictly related to vibration analysis.

The combination of different observable measurements has been studied in [150]. The paper explains how the combination of vibration and temperature of a mechanical system allows to better detect the health state of the system.

In [151] a tracking simulator is presented, which relies on the estimation of the dynamic process through data coming from the OPC UA Server of the physical process. This work shows an approach to refine a virtual model of the equipment, but does not consider the correlation of different observable measurements (vibrations, temperature, *etc*).

11.3 Modeling The Problem

11.4 From Sensor Data to Severity Levels

The observation of a mechanical equipment is the mainly used technique to identify its health status. This can be achieved through the observation of measurements coming from sensors. A lot of researches [118] have focused on the development process of IoT sensors (accelerometers, gyroscopes, pressure sensors, *etc.*). Some works tried to defined thresholds that can be used as guidelines to identify the severity of the measurement deviation. For example, the International Organization of Standardization (ISO) has proposed a standardization for the vibration measurements of mechanical parts. This standard tries to define a set of severity levels combining vibrations with the class of machines [119].

The standard establishes general procedures for the measurement and evaluation of mechanical vibration of machines, as measured on non-rotating parts(*i.e.* casing of a motor).

The $V_{r.m.s.}$ (vibration velocity root mean square) is obtained with the following equation:

$$V_{r.m.s.} = \sqrt{\frac{1}{T}\int_0^T v^2(t)dt} \tag{11.1}$$

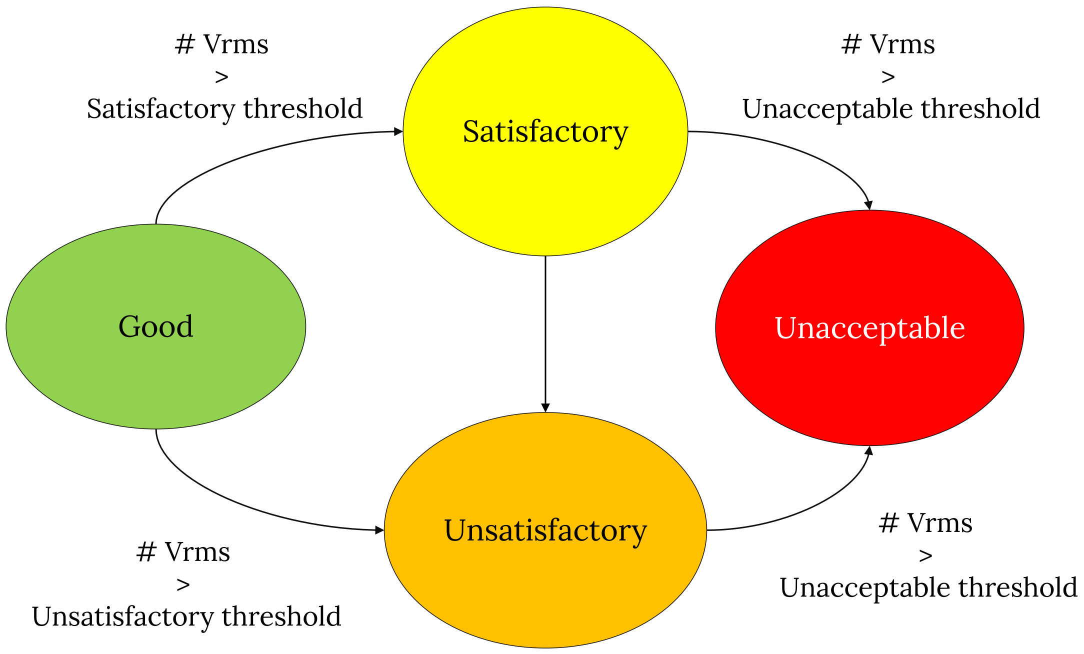
T is the sampling time and it is correlated with the vibration sensor frequency. v is the time-dependent vibration velocity and $V_{r.m.s.}$ is the corresponding r.m.s. velocity.

Table 11.1 reports the classification of the ISO-10816. It considers four class machines depending mainly on the generated power.

- *Class I*: Small machines that generates up to 15kW output;
- *Class II*: Medium-sized machines, without special foundations, with an output up to 75kW;
- *Class III*: Large prime-movers with rotating masses mounted on rigid and heavy foundations;
- *Class IV*: Large prime-movers and other machines with rotating masses with output greater than 10MW (*i.e.* turbogenerator, gas turbine).

The standard defines four different states: *good*, *satisfactory*, *unsatisfactory* and *unacceptable*.

- *Good*: The vibration of newly commissioned machines normally falls within this zone;
- *Satisfactory*: Machines with vibration within this zone are normally considered acceptable for unrestricted longterm operation;



# Vrms > Satisfactory threshold
# Vrms > Unacceptable threshold
Satisfactory
Good
Unacceptable
Unsatisfactory
# Vrms > Unsatisfactory threshold
# Vrms > Unacceptable threshold

Fig. 11.2: Overview of a generic MSM. The states reflect the severity levels defined by ISO-10816.

- *Unsatisfactory*: Machines with vibration within this zone are normally considered unsatisfactory for long-term continuous operation. Generally, the machine may be operated for a limited period in this condition until a suitable opportunity arises for remedial action;
- *Unacceptable*: Vibration values within this zone are normally considered to be of sufficient severity to cause damage to the machine.

In particular, it is important to notice that with the same $V_{r.m.s.}$ the different machine classes could have different states. For instance, let consider the $V_{r.m.s.}$ range between 7.10 and 11.20 mm/s. A large machine of *class IV* with this range is in a state of *satisfactory* and a small machine of *class I* is in an *unacceptable* state.

The mentioned standard is applied only for vibration measurements. Actually, there is no other standard that defines severity levels for other observable measurements (*i.e.* temperature, acoustic emissions, power consumption).

This paper shows a novel approach to define severity levels for other observable measurements relying on ISO-10816 existing standard. More in details, the approach combines vibrations with other observable measurements and retrieves the related states mapped from ISO-10816. This can be applied if and only if other observable measurements are strictly related to a particular component of the considered machine. Table 11.2 shows the theoretical application of this novel approach with a machine of *class I*.

11.5 Monitoring State Machine (MSM)

The goal of predictive maintenance is to anticipate equipment failures to optimize production and plan maintenance strategies. In this scenario, an equipment maintainer can establish the status of equipment by observing it through measurements of vibrations, temperature, power consumption, etc. When these measurements are deviating from a normal behavior the equipment could perform inaccurate operations or stopped for a failure. With the analysis of these measurements, it is possible to define the state of health of the equipment.

An MSM is a state machine that monitors one observable measurement of an equipmentra critical component and translates raw data coming from sensors into severity levels representing its health status (see figure 11.2). This work relies on the concept of severity levels defined by ISO-10816 for vibration measurements and adapting it to other observable measurements. An MSM is composed of four states explained in section 11.3 (good, satisfactory, unsatisfactory, unacceptable) defined by the ISO-10816 standard. The use of ISO-10816 coupled to a finite state machine guarantees persistent deviation detection is avoiding transient events. Transient events could occur from sensing errors leading to wrong equipment diagnosis. A persistent deviation can be detected from an MSM with a transition, between a state to another, triggered when a severity level occurs for a certain amount of times. The transition is defined with a threshold that can be set during the tuning phase of the MSM. The tuning phase requires a model of the critical component of the equipment that will be faulted. This phase will be explained in section 11.7. Figure 11.2 represents a *vibration MSM*. Let assume that the *vibration MSM* is in good state. The *vibration MSM* will move to satisfactory state only after a persistent detection of satisfactory severity level, defined by the satisfactory threshold.

Unfortunately, there are no similar standards of ISO 10816 for other observable measurements. The definition of MSMs for other observable measurements of the same critical component is performed via the correlation of vibration severity levels with other observable measurements (see table 11.2). For instance, in figure 11.2 the correlation of vibration severity levels, temperature and acoustic emissions of the same critical component allows having more information about its health status and defines the state range for each measurement.

11.6 Developed Technique

11.6.1 Predictive Maintenance Supervisor

In the previous section, MSM has been presented with the role to monitor the deviations of observable measurements of the same critical component. This section introduces a new actor called Predictive Maintenance Supervisor (PMS). The role of this actor is monitoring all the MSMs to detect anomalies and to raise alerts regarding health status. The PMS is defined as a finite state machine that analyses the current state of the available MSMs. It is modeled as a transition system where the output of the automaton is its current state.

The inputs of the PMS are represented with the tuple $< R, M >$. R represents the class of recipes that are produced by the equipment. For instance, the equipment produces two different

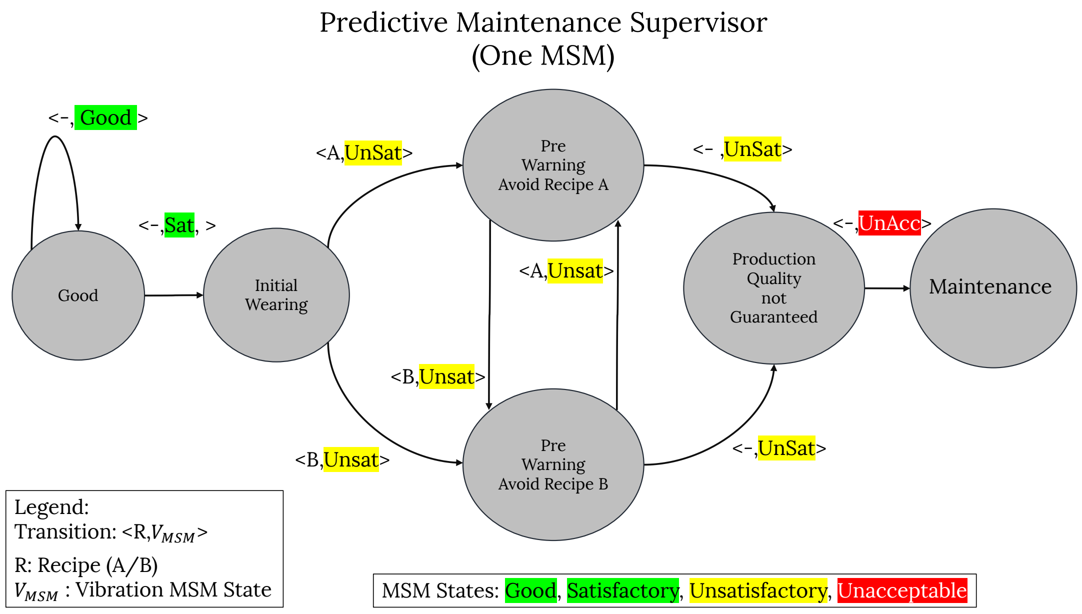

Predictive Maintenance Supervisor
(One MSM)
<-,Good>
<-,Sat, >
Good
Initial Wearing
<A,UnSat>
Pre Warning Avoid Recipe A
<- ,UnSat>
<A,Unsat>
<B,Unsat>
<B,Unsat>
Pre Warning Avoid Recipe B
<-,UnSat>
Production Quality not Guaranteed
<-,UnAcc>
Maintenance
Legend:
Transition: <R,V_{MSM}>
R: Recipe (A/B)
V_{MSM} : Vibration MSM State
MSM States: Good, Satisfactory, Unsatisfactory, Unacceptable

Fig. 11.3: Overview of a Predictive Maintenance Supervisor monitoring one MSM (*vibration MSM*).

products with two different recipes A, B that are different in terms of energy consumptions and mechanical stress of the critical component. This input is necessary to better model the PMS allowing to catch more information about the real health status of the equipment. M represents the class of the available MSMs monitored by the PMS. In particular, the PMS takes as inputs the internal state of the MSMs (good, satisfactory, unsatisfactory, unacceptable). Figures 11.3 and 11.4 show two different structures of PMS. Figure 11.3 shows the structure of a basic PMS that monitors only one MSM, which detects vibrations severity levels. In this basic PMS it possible to notice the following states: *good, initial wearing, avoid recipe A, avoid recipe B, production quality not guaranteed, maintenance* that reflect the health status of the critical component. The execution of the PMS starts from *good* state. The PMS will stay in good state as long *vibration MSM* is in *good* state regardless of the recipes (transition $< -, Good >$). When the *vibration MSM* detects a persistent deviation coming from the vibration sensor it will move to *satisfactory* state. This implies a transition for the PMS from *good* to *initial wearing* state (transition $< -, Sat >$). When the critical component has a significant deviation the MSM moves to unsatisfactory state. From the PMS point of view this means that the critical component is generating significant vibrations with the actual recipe. In this case, the PMS moves into *avoid recipe A or B* state, depending on the actual recipe. This is due to the fact that the two recipes are different in terms of mechanical fatigue and this is reflected in the critical component in terms of vibration severity. For instance, let assume that recipe A generates more mechanical stress than recipe B. If the equipment generates more vibration with recipe A the PMS notifies that recipe A should be avoided (transition $< A, Unsat >$). This can improve the use of the equipment, instead of going directly to maintenance reducing undesired equipment downtime. When the PMS detects unsatisfactory severity level from the *vibration MSM*,

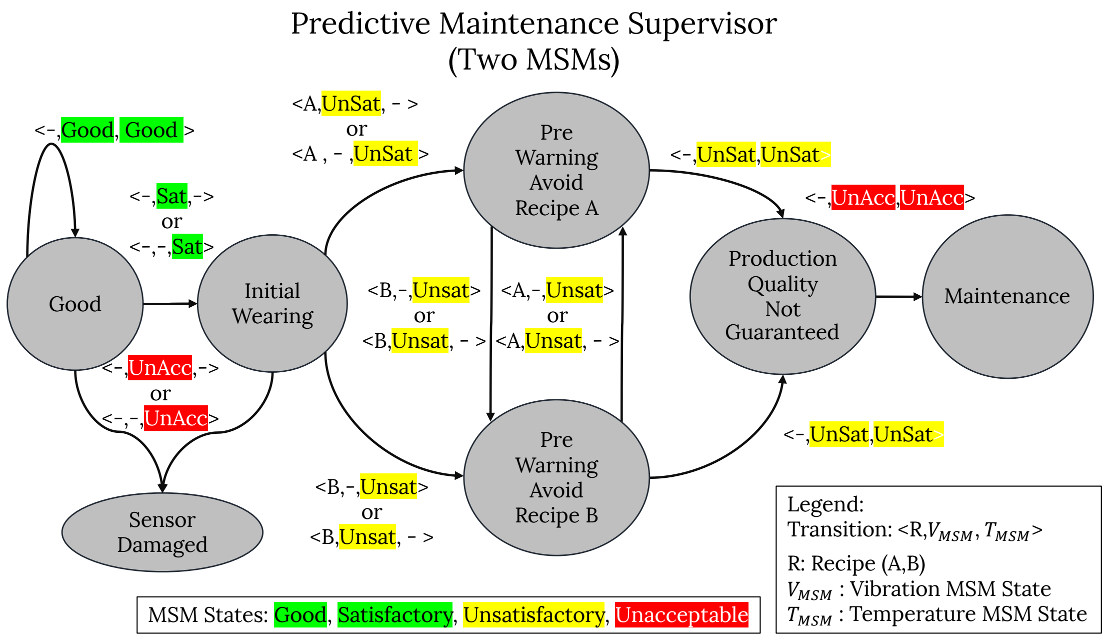

Predictive Maintenance Supervisor
(Two MSMs)
<-,Good, Good >
<-,Sat,-> or <-,-,Sat>
<A,UnSat, - > or <A , - ,UnSat >
<-,UnSat,UnSat>
<-,UnAcc,UnAcc>
Good
Initial Wearing
Pre Warning Avoid Recipe A
Production Quality Not Guaranteed
Maintenance
<B,-,Unsat> or <B,Unsat, - >
<A,-,Unsat> or <A,Unsat, - >
<-,UnAcc,-> or <-,-,UnAcc>
<-,UnSat,UnSat>
Sensor Damaged
<B,-,Unsat> or <B,Unsat, - >
Pre Warning Avoid Recipe B
Legend:
Transition: <R,V_{MSM}, T_{MSM}>
R: Recipe (A,B)
V_{MSM} : Vibration MSM State
T_{MSM} : Temperature MSM State
MSM States: Good, Satisfactory, Unsatisfactory, Unacceptable

Fig. 11.4: Overview of a Predictive Maintenance Supervisor monitoring two MSMs (vibration and temperature).

for both the recipes A and B, it moves to *production quality not guaranteed* state (transitions $< A, Unsat >, < B, Unsat >$). This means that the critical component has shown a continuous and important deviation in all the possible recipes. The PMS can not detect the quality of the production because it does not have enough information related to the output products from the equipment. In this case, the PMS does not move directly to *maintenance* state in order to avoid equipment downtime. From a production planning perspective, this is an important information that could help to define different quality levels of the products. For instance, products that are produced when the equipment is *production quality not guaranteed* state could be still good, but requiring additional visual or technical analysis. The *maintenance* state is reached from the previous state when the PMS detects unacceptable severity levels from MSM.

Figure 11.4 shows the structure of a PMS that is monitoring two MSMs, vibration and temperature. In section 11.5 the definition of severity levels starting from ISO-10816, for other MSMs have been discussed. The structure of this PMS with a second MSM allows the PMS to retrieve additional information than the previous PMS (figure 11.3. The two MSMs are considered part of class M in the tuple $< R, M >$ used to represent the PMS state transitions. In particular, a new state called *sensor damaged* has been added that represents sensors anomalies that can be detected from the PMS. Let consider the PMS in *good* or *initial wearing* state and an anomaly in the vibration sensor. The vibration sensor is reporting an important and persistent untrue deviation that is translated into unacceptable severity level from the *vibration MSM*. The PMS detects this deviation from *vibration MSM* and a good severity level from *temperature MSM*. In this case, the PMS moves to *sensor damaged* state.

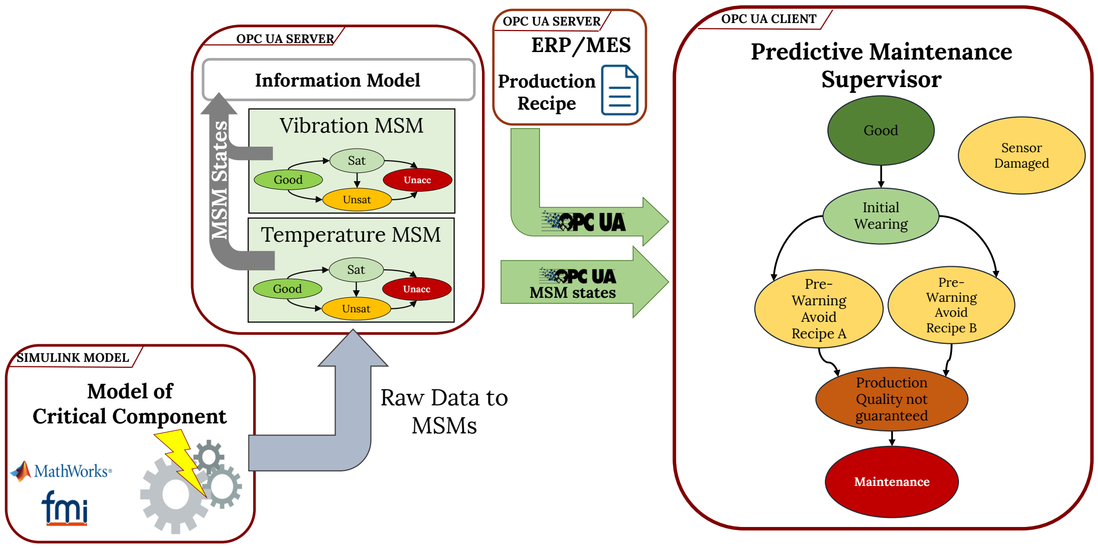


Fig. 11.5: Overview of the entire framework in validation mode.

11.7 Alternative Use of MSM and PSM

This section discusses the alternative use of the framework composed of MSM and PMS in different scenarios. The entire framework has been wrapped in an OPC UA client and server in order to be easily integrated into a production plant environment and retrieves the necessary data in the different following scenarios (see Figure 11.5). In particular, all the MSMs were integrated into the OPC UA client and the PMS on the server-side.

11.7.1 *Structure Validation Mode*

In this scenario, we assume that a model of the equipment critical component is available. Clearly, the model is an abstraction of the real equipment and it can not be precise as the real measurements coming from the equipment sensors. However, the model is used to preliminary define new standards for the observable measurements and then set the internal thresholds of all the MSMs (see section 11.5). The information model of the OPC UA server exposes only the MSMs states that are needed for the PMS. For instance, if vibration and temperature are the observed measurements, the information model will contain only the *vibration* and *temperature MSM* actual states. The OPC UA client, that contains the PMS, retrieves the severity levels from the OPC UA server information model. Moreover, the validation phase requires to fault the model in order to check if the framework detects the deviation of the faulted model.

11.7.2 *Condition-based-Maintenance Continuous Mode*

The framework is integrated into the real equipment after the validation phase. The OPC UA server that contains the MSMs is connected to sensors that retrieve data from the real equipment.

The framework in the OPC UA client monitors the real equipment and raises alerts when an anomaly is detected during the production. In this scenario, the framework is used to perform Condition-based Maintenance. This is due to the fact that the framework receives data regarding the actual product. The framework can predict the health status of the equipment when the "symptoms" are just detected.

11.7.3 *Predictive Maintenance Structure Training Mode*

This mode represents a preparatory phase for the *Predictive Maintenance Mode*. This phase is performed concurrently with the *Condition-based-Maintenance Continuous Mode*. The framework monitors the equipment and stores raw data related to the severity levels and the type of product that is produced. This tuple is stored in a local or remote database.

11.7.4 *Predictive Maintenance Mode*

A Factory can produce different types of products that can affect the health status of the equipment depending on the physical process required from the product recipe. Let us assume that a Factory produces three different products *A, B, C*. For instance, product *A* requires less mechanical stress than product *B* or product *C*. The goal of the Framework in Predictive Maintenance Mode is to predict if a production batch can be produced without anomalies. The information of the production plans can be retrieved from ERP or MES that cover the highest levels of the Automation Pyramid.

The framework in the OPC UA client retrieves this information from the ERP/MES OPC UA server. The production plans contain the type of products (*A, B, C*) and the numbers of products to produce. The combination of production plans and the historical data stored in the *Predictive Maintenance Structure Training Mode* allows using the framework offline to perform prediction related to the specific production plan. The framework simulates the MSMs, using the local raw data stored in the database.

11.8 Methodology Application

This section explains the methodology application and the experimental results.

11.8.1 Experimental Setup

The methodology has been validated with the use of two support technologies: OPC UA and FMI, both explained in Section 11.2. The two principal components that compose the framework of the experiments are the OPC UA client and server (see figure 11.5). In particular, the two actors have been developed in two different FPGA.

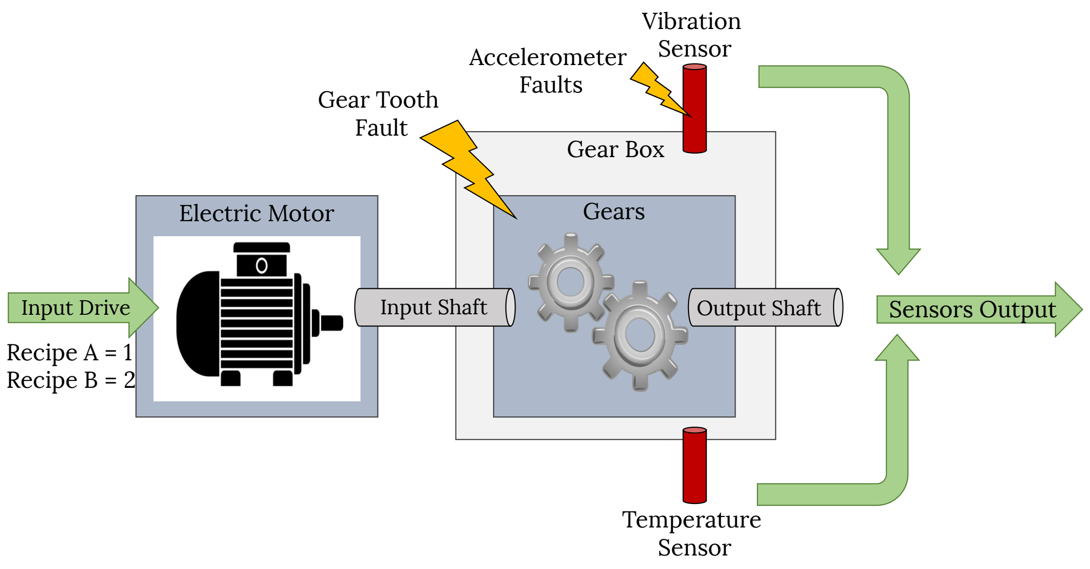


Fig. 11.6: Overview of the transmission system. The gears represent the critical component of the system. The model can be faulted in two different zone: gear tooth and the vibration sensor.

OPC UA Client

The client has been developed in python with the use of FreeOpcUa python library[1] on a PYNQ-Z1 FPGA. This choice comes from the necessity to have a specific description that can be synthesized to an FPGA to obtain a compact reactive device minimizing power consumption. The PMS has been described using a Hardware description language (Verilog) and then synthesized into the FPGA that provides python API to perform data exchange with synthesized bitstream[2].

OPC UA Server

The server has been developed in a second FPGA using the same structure presented for the client. All the MSMs have been developed in Verilog and then synthesized on the FPGA. Then, the FREEopcUa python library has been used to develop the OPC UA server information model where all the MSM actual states are exposed.

The OPC UA server retrieves data directly from real sensors attached to the critical component of the equipment or from a model of the critical component that is not integrated into the OPC UA server. The model is often complex, requiring computational effort that can not be computable by a small device. The server is connected remotely to the model via a socket connection. The model is exported using the FMI standard as an FMU to be as flexible as possible. This allows having more control over the model that can be easily reallocated to other remote resources or integrated with other environments.

[1] http://freeopcua.github.io/
[2] http://http://www.pynq.io/

a) Severity Levels Correlation

Tooth Fault Level (%)	Recipe A		Recipe B	
	Vibration Vrms (mm/s)	Temperature (°C)	Vibration Vrms (mm/s)	Temperature (°C)
0	0.714	19.85	1	20,50
5	0.946	20.65	1.66	21.45
10	1.157	22.15	1.877	23.55
20	1.58	26.35	2.3	29.45
30	2.018	32.25	2.743	36.65
40	2.416	40.85	3.151	46.95
50	2.845	51.05	3.586	58.65
60	3.301	62.95	4.048	71.75
70	3.791	76.55	4.543	86.16
80	4.326	91.45	5.081	101.65
90	4.952	108.15	5.721	118.55
95	6.736	133.05	8.483	159.55
98	10.57	200.25	11.59	237.85
100	12.67	268.05	13.24	289.75

Legend

🟩 Good	🟧 Unsatisfactory
🟨 Satisfactory	🟥 Unacceptable

b) Temperature Severity Levels

Tooth Fault Gain (%)	Temperature	Severity Level Range
	(°C)	
0	19.85	≤ 20,50
	20,50	
5	20,65	20,51-40,85
	21.45	
10	22.15	
	23.55	
20	26.35	
	29.45	
30	32.25	
	36.65	
40	40.85	
	46.95	40,86 - 133.05
50	51.05	
	58.65	
60	62.95	
	71.75	
70	76.55	
	86.16	
80	91.45	
	101.65	
90	108.15	
	118.55	
95	133.05	
	159.55	>133.05
98	200.25	
	237.85	
100	268.05	
	289.75	

Table 11.3: Definition of new standard for temperature MSM: a) reports the experiments for recipe A and B with different tooth fault levels, with respect to ISO 10816 severity levels. b) shows the obtained severity level ranges for temperature MSM.

11.8.2 Abstract Model of a Real Machine

A driveline two-speed transmission system is used as a model of a critical component of an equipment (see figure 11.6). Mathworks Simulink © is used to model the critical component.

The model consists of an electric motor, a gear system contained in a gearbox, and two shafts. The electric motors is connected to the gearbox via the input shaft. The gearbox contains two gears with different dimensions that are connected respectively to the input and output shaft. In particular, the gears in the gearbox represent the critical subsystem of the manufacturing equipment. The model incorporates two sensors to monitor vibration and temperature. The two sensors are attached to the surface of the gearbox monitoring the vibration displacements and the external temperature.

The model can be faulted in two parts of the critical component. The tooth of the gear and the vibration sensor can be faulted to simulate the wearing of the gears or sensors anomalies. The gear tooth fault is driven by an integer input with a range from 0 to 100 that represents the percentage of fault. The parameter affects the gear tooth that will increase the gear vibrations. The vibration sensors can be faulted with two parameters: gain and offset. Gain parameter can be used to simulate an amplification or attenuation of the output sensors data and offset allows to add an offset to these data. With this two parameters it is possible to model the principal effects caused by sensor anomalies. The model is driven by a single real input that defines the output speed of the electric motor. The manufacturing equipment produces two different products classified with the recipes A, B that are represented by two different speed of the electric motor, part of the critical component. Both the recipes requires 40 seconds to be executed by the critical component, but generates different mechanical stress for the gears.

11.8.3 Structure Validation Mode in Action

This section describes the experiments used to set and validate the correctness of the methodology. In particular, the Structure validation mode of MSM is considered (see figure 11.5). Two different experiments have been considered:

- One MSM (vibration);
- Two MSMs (vibration and temperature).

For the first experiment, only the vibration sensor is considered. In this case, the setup phase is required only to set the MSM internal thresholds to detect the persistent deviation of vibrations. The severity levels used are those defined by ISO 10186 standard (see Table 11.1). The tooth fault has been manually injected following the wearing curve of a real gear system. Figure 11.3 shows the PMS used in this experiment that was able to detect the deviation related to the gear tooth fault. After the setup phase, we tried to inject faults in the vibration sensor and the PMS was not able to detect the sensor anomalies, as we expected. The PMS does not have enough information to clearly understand if the anomalies are coming from the sensor or the critical component.

In the second experiment, a second MSM related to the temperature has been introduced. In this case, the setup phase requires first to define a new standard for *temperature MSM*. Tables 11.3 a) and b) show the obtained severity level ranges for the temperature MSM. The PMS used in this experiment is the one presented in Section 11.6.1 (see Figure 11.4). Once the new standard has been derived and the thresholds for both the MSMs have been set the PMS has been validated. The same wearing curve of a real gear system has been used to validate the methodology. The Supervisor has been capable to detect gear tooth anomalies moving in the different states. Then, the vibration sensor has been faulted as in the previous experiment. In this case, the PMS was able to detect the sensor anomalies moving to *sensor damaged* state.

11.8.4 Experimental results

In this Section, two observable measurements, vibration and temperature, have been considered in different scenarios. Moreover, the framework has been tested with the PMS monitoring one or two MSMs (see Figures 11.3, 11.4).

The approach has been validated with two different production recipes (A, B), faulting the critical component and vibration sensor. The PMS, monitoring only one MSM, has been able to detect the persistent deviation of the critical component with the different recipes, rising alerts regarding the health status.

Table 11.4 reports the properties that the PMS can analyze in relation to the number of monitored MSMs. The complexity of the PMS strictly depends on two factors: the number of monitored MSMs and the number of recipes that the equipment has to deal with. It is clear that, following the proposed PMS structure, the number of PMS states increases linearly with the number of recipes. Moreover, with only one MSM the complexity is low and requires less time to set up the thresholds and the PMS. With only one MSM it is necessary to have a vibration sensor in order to use the ISO 10816 standard.

Properties	PMS		
	Number of monitored MSM		
	1	2	> 2
Persistent deviation	√	√	√
Sensor damaged	X	√	√
Specific sensor damaged	X	X	√
Complexity	Low	Medium	Medium-High

Table 11.4: Experimental results related to the properties of the PMS.

Considering two MSMs (vibration and temperature) the PMS is more precise and able to detect more information regarding the sensor status. In this case, it is possible to detect deviation due to sensor anomalies. The complexity of the PMS increases due to the fact that with more MSMs there are more transitions to take into account. Furthermore, the definition of a new standard is required to set up the derivated MSM.

The third column of Table 11.4 reports the best scenario for the PMS in terms of anomalies detection. In particular, with more than two sensors it is possible to precisely detect which sensor is damaged. The complexity of the PMS increases due to the number of transitions and requires a lot of time spent to retrieve the new severity levels for other observable measurements. Each scenario requires to be analyzed to define which is the best trade-off solution with the proposed framework.

11.8.5 Standards

The main goal of the *EDA way of working* is the generation of new standards based on the developed techniques. The entire framework works with the standardization of measurements, in particular on the definition of thresholds and the correlated health status. As discussed in section 11.4, some works tried to define a classification of vibrations [119]. Another work analyzed temperature of electrical equipment defining temperature classes related to the health status of the circuit [152]. At the moment, no other standards has been defined for the acoustic emissions, power consumption, liquid pressure of mechanical equipment.

11.9 Conclusions

This paper proposes a modular and scalable framework to monitor and predict maintenance cycles. In particular, the entire framework is oriented in the detection of the current health status of an equipment. This important information can be used to reflect the detected health status in the model allowing to obtain the Digital-Twin of the equipment. The use of the OPC UA standard enables a flexible solution for the integration in a manufacturing plant with respect to the automation pyramid. Moreover, the use of the framework in different scenarios has been proposed and the use of FMI standard permits to initialize the framework with synthetic data coming from a model.

To follow the "*EDA way of working*", we formalized at first the problem, we then identified an automation technique to design it and we used some standards (*e.g.*, ISO-10816) while we will contribute to define new standards as future work.

Experiments showed the benefits of this approach, in particular the flexibility of the framework with different physical measurements. Moreover, the observation of multiple measurements with the advanced PMS turned out to be the best solution to have a global view of the health status of the equipment. The combination of multiple observable measurements enables to detect anomalies before the observation of single measurement.

Part VI

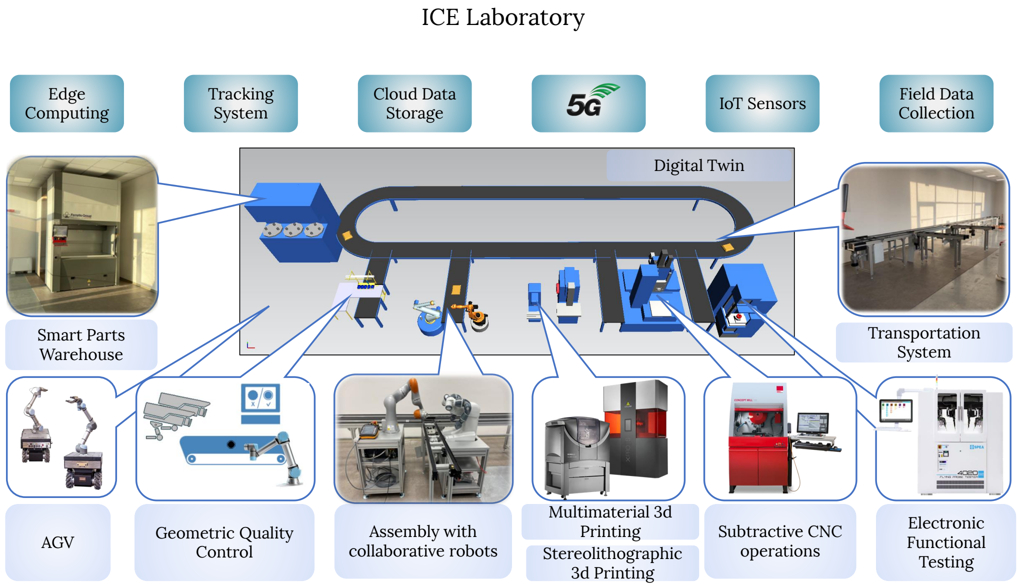
Unified Example & Conclusions

12

Industrial Computer Engineering Laboratory & Conclusions

All the presented methodologies has been tested in a real use case scenario in the context of the Industrial Computer Engeneering Laboratory (ICE Lab). This laboratory is under development with a national project "Dipartimenti di Eccellenza 2018-2022" financed by the Ministry of Education, Universities and Research (MIUR). The ICE Lab is an innovative industrial laboratory representing a production where multiple research groups of the Department of Computer Science of Verona are involved. Figure 12.1 shows the structure of the laboratory with all the equipment that composes the entire production line. The laboratory is composed by a set of heterogeneous equipment used in real industries, trying to cover all the steps needed to perform a real production. The line is composed of the following equipment:

- Pallet transportation system, used to move the products;
- Vertical Smart warehouse;
- Assembly station with two collaborative manipulators;



ICE Laboratory
Edge Computing
Tracking System
Cloud Data Storage
5G
IoT Sensors
Field Data Collection
Digital Twin
Smart Parts Warehouse
Transportation System
AGV
Geometric Quality Control
Assembly with collaborative robots
Multimaterial 3d Printing
Stereolithographic 3d Printing
Subtractive CNC operations
Electronic Functional Testing

Fig. 12.1: Overview of the ICE Lab used as case study for all the presented methodologies.

- Two 3D Printer with different printing techniques;
- Industrial Drilling machine;
- Video quality control cell;

The vision behind this lab aims to obtain a production line that is capable to adapt itself based on external events in order to maximize the production. This is possible with the Digital Twin, that allows to perform analysis and predictions regarding the future state of the line.

The methodologies presented in this thesis aim to define a set of rules in order to obtain the digital twin tested in some equipment of the production line.

In part II have been discussed a methodology to the definition of synthetic OPC UA node that can virtually represents a new equipment connected to the line in order to see the effects with data coming from the field.

Part III discussed how to deal with heterogeneous models. The methodologies has been tested focusing on a particular part of the transportation system and trying to mime a new equipment connected to the line.

The Multi-Level approaches, discussed in part- IV, have been tested in the Assembly station where a manipulator has been considered as case study.

Part V reports a technique to transform real data coming from field into information. In my vision, this represents the joining link between a model and the real world, needed to obtain a real Digital Twin. Furthermore, it represents the worthy open-problem in the context of Industry-4.0. This methodology has been proved with data coming from an electrical engine of the transportation system.

There are several future works for each part that have been discussed.

Considering the Homogeneous and Heterogeneous modeling methodologies, I will investigate on AutomationML language in order to extend this language to support different domains and obtain an unique description of the heterogeneous model. Moreover, the smart coordinator,discussed in the Heterogeneous modeling methodologies, can be extended also for the physical part in order to obtain an optimzed simulation strategy based on the context of the involved models. Furthermore, currently investigation on the interaction between smart coordination and Multi-level approaches are considered. Integration between multi-level approach with "real data to information" methodologies will be considered.

The global view of the entire work aims to integrate all the presented methodologies in order to obtain a Digital Twin with an optimized coordinator that considers multiple description of the same model, with different level of details, in order to optimize the simulation and obtain a multi-level smart coordinator that considers also real data coming from sensors.

13

Published Contributions

The work carried on to develop this thesis led to a total of X publications.
The methodologies for Heterogeneous Modeling and Simulation discussed in Part III, have been presented in:

- Centomo, S., Lora, M., Portaluri, A., Stefanni, F., Fummi, F.,
  "Automatic generation of cycle-accurate Simulink blocks from HDL IPs"
  in Proceedings of 2017 Forum on Specification and Design Languages (FDL), pp. 1-8
- Centomo, S., Lora, M., Portaluri, A., Stefanni, F., Fummi, F.,
  "Automatic Integration of HDL IPs in Simulink using FMI and S-Function Interfaces"
  in Languages, Design Methods, and Tools for Electronic System Design. Lecture Notes in Electrical Engineering, vol 530.(Springer)
- Centomo, S., Lora, M., Fummi, F.,
  "Transaction-level functional mockup units for cyber-physical virtual platforms"
  in Proceedings of 2018 Forum on Specification and Design Languages (FDL), pp. 1-8
- Centomo, S., Lora, M., Fummi, F.,
  "Generation of Functional Mockup Units for Transactional Cyber-Physical Virtual Platforms"
  in Languages, Design Methods, and Tools for Electronic System Design. Lecture Notes in Electrical Engineering, vol 611.(Springer)
- Centomo, S., Panato, M., Fummi, F.,
  "Cyber-Physical Systems Integration in a Production Line Simulator"
  in Proceedings of 2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), pp. 237-242
- , Lora, M.,Centomo, S., Quaglia, D., Fummi, F.,
  "Automatic Integration of Cycle-Accurate Models into Cyber-Physical Virtual Platform"
  in Proceedings of 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 676-681

The methodologies for Muti-Level Modeling and Simulation discussed in Part IV, have been presented in:

- Centomo, S., Fraccaroli, E., Panato, M.,
  "From Multi-Level to Abstract-Based Simulation of a Production Line"
  in Proceedings of 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1253-1256
- Centomo, S., Avogaro, A., Panato, M., Tadiello, M., Fummi, F.,
  "A Design Methodology for Multi-Level Digital Twin"
  in Proceedings of 2021 IEEE International Conference on Industrial Technology (ICIT). pp. (to appear)

The methodologies of "From Real Data to Information" discussed in Part V, have been presented in:

- Dall'Ora, N., Centomo, S., Fummi, F.,
  "Industrial-IoT Data Analysis Exploiting Electronic Design Automation Techniques"
  in Proceedings of 2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI). pp. 103-109
- Centomo, S., Dall'Ora, N., Fummi, F.,
  "The Design of a Digital-Twin for Predictive Maintenance"
  in Proceedings of 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). pp. 1781-1788

References

1. R. Drath and A. Horch, “Industrie 4.0: Hit or hype? [industry forum],” *IEEE Industrial Electronics Magazine*, vol. 8, no. 2, pp. 56–58, jun 2014.
2. D. Mourtzis, M. Doukas, and D. Bernidaki, “Simulation in manufacturing: Review and challenges,” *Procedia CIRP*, vol. 25, pp. 213–229, 2014.
3. S. Centomo, M. Panato, and F. Fummi, “Cyber-physical systems integration in a production line simulator,” in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, oct 2018.
4. J. Vachalek *et al.*, “The digital twin of an industrial production line within the industry 4.0 concept,” in *2017 21st International Conference on Process Control (PC)*. IEEE, jun 2017.
5. “Simulation software survey,” 2017. [Online]. Available: https://www.informs.org/ORMS-Today/OR-MS-Today-Software-Surveys/Simulation-Software-Survey
6. S. Makris and K. Alexopoulos, “AutomationML server-A prototype data management system for multi disciplinary production engineering,” in *Procedia CIRP*, 2012.
7. G. N. Schroeder, C. Steinmetz, C. E. Pereira, and D. B. Espindola, “Digital twin data modeling with automationml and a communication methodology for data exchange,” *IFAC-PapersOnLine*, vol. 49, no. 30, pp. 12 – 17, 2016, 4th IFAC Symposium on Telematics Applications TA 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405896316325538
8. M. Sabou, F. Ekaputra, O. Kovalenko, and S. Biffl, “Supporting the engineering of cyber-physical production systems with the AutomationML analyzer,” in *2016 1st International Workshop on Cyber-Physical Production Systems, CPPS 2016*, 2016.
9. P. Novák, F. J. Ekaputra, and S. Biffl, “Generation of Simulation Models in MATLAB-Simulink Based on AutomationML Plant Description,” *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 7613–7620, 2017. [Online]. Available: https://doi.org/10.1016/j.ifacol.2017.08.1027
10. W. Mahnke, S. H. Leitner, and M. Damm, *OPC unified architecture*, 2009.
11. VID/VDE, “Reference Architecture Model Industrie 4.0 (RAMI4.0),” *Igarss 2014*, 2015.
12. (2020) Opc-ua specification. [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture
13. (2020) Opc-ua specification: Part 2 security model. [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-2-security-model/
14. (2020) Opc-ua specification: Part 1 overview and concepts. [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-1-overview-and-concepts/
15. R. Drath, A. Luder, J. Peschke, and L. Hundt, “Automationml-the glue for seamless automation engineering,” in *2008 IEEE International Conference on Emerging Technologies and Factory Automation*. IEEE, 2008, pp. 616–623.
16. (2020) open62541. [Online]. Available: https://open62541.org/
17. J. C. Jensen, D. H. Chang, and E. A. Lee, “A model-based design methodology for cyber-physical systems,” in *Proc. of IWCMC 2011*, pp. 1666–1671.
18. Mathworks, “Matlab Simulink,” http://www.mathworks.com/products/simulink.html.

19. W. Chang, D. Roy, L. Zhang, and S. Chakraborty, “Model-based design of resource-efficient automotive control software,” in *Proc. of IEEE/ACM ICCAD, 2016*, pp. 1–8.
20. S. Vinco, V. Guarnieri, and F. Fummi, “Code Manipulation for Virtual Platform Integration,” *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2694–2708, 2016.
21. T. Blochwitz *et al.*, “Functional mockup interface 2.0: The standard for tool independent exchange of simulation models,” in *Proc. of MODELICA Conference 2012*, 2012, pp. 173–184.
22. R. Malone, B. Friedland, J. Herrold, and D. Fogarty, “Insights from Large Scale Model Based Systems Engineering at Boeing,” in *Proc. of INCOSE International Symposium 2016*, vol. 26, no. 1, pp. 542–555.
23. M. Bajaj, D. Zwemer, R. Yntema, A. Phung, A. Kumar, A. Dwivedi, and M. Waikar, “MBSE++–Foundations for Extended Model-Based Systems Engineering Across System Lifecycle,” in *INCOSE International Symposium 2016*, vol. 26, no. 1, pp. 2429–2445.
24. F. Fummi, M. Lora, F. Stefanni, D. Trachanis, J. Vanhese, and S. Vinco, “Moving from Co-Simulation to Simulation for Effective Smart Systems Design,” in *Proc. of ACM/IEEE DATE*, 2014, pp. 1–4.
25. W. Li, X. Zhang, and H. Li, “Co-simulation platforms for co-design of networked control systems: An overview,” *Control Engineering Practice*, vol. 23, pp. 44–56, 2014.
26. D. Quaglia, R. Muradore, R. Bragantini, and P. Fiorini, “A SystemC/Matlab co-simulation tool for networked control systems,” *Simulation Modelling Practice and Theory*, vol. 23, pp. 71–86, 2012.
27. M. S. Hasan, H. Yu, A. Carrington, and T. Yang, “Co-simulation of wireless networked control systems over mobile ad hoc network using SIMULINK and OPNET,” *IET communications*, vol. 3, no. 8, pp. 1297–1310, 2009.
28. F. Bouchhima, M. Briere, G. Nicolescu, M. Abid, and E. Aboulhamid, “A SystemC/Simulink co-simulation framework for continuous/discrete-events simulation,” in *Proc. of IEEE BMAS 2006*, pp. 1–6.
29. Y. Nakamoto, I. Abe, T. Osaki, H. Terada, and Y. Moriyama, “Toward integrated virtual execution platform for large-scale distributed embedded systems,” in *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 2008, pp. 317–322.
30. H. Yan, T. Wang, C. L. i, and H. Zhang, “Functional reliability simulation analysis for electronic throttle control system based on saber-simulink co-simulation,” in *2015 Prognostics and System Health Management Conference (PHM)*. IEEE, oct 2015. [Online]. Available: https://doi.org/10.1109/phm.2015.7380023
31. Y. Wang, K. Li, H. Zhou, S. Deng, J. Xu, and J. Liu, “Dynamic analysis and co-simulation ADAMS-SIMULINK for a space manipulator joint,” in *2015 International Conference on Fluid Power and Mechatronics (FPM)*. IEEE, aug 2015. [Online]. Available: https://doi.org/10.1109/fpm.2015.7337258
32. W. Hanini and M. Ayadi, “PSpice and simulink co-simulation for diode and AC-DC converter using SLPS interface software,” in *2017 18th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*. IEEE, dec 2017. [Online]. Available: https://doi.org/10.1109/sta.2017.8314910
33. T. Peter and S. Wenzel, “Coupled simulation of energy and material flow using plant simulation and MATLAB simulink,” *SNE Simulation Notes Europe*, vol. 27, no. 2, pp. 105–113, jun 2017.
34. R. Kawahara, D. Dotan, T. Sakairi, K. Ono, H. Nakamura, A. Kirshin, S. Hirose, and H. Ishikawa, “Verification of embedded system’s specification using collaborative simulation of SysML and simulink models,” in *2009 International Conference on Model-Based Systems Engineering*. IEEE, mar 2009. [Online]. Available: https://doi.org/10.1109/mbse.2009.5031716
35. S. Tudoret, S. Nadjm-Tehrani, A. Benveniste, and J.-E. StrÃűmberg, “Co-simulation of hybrid systems: Signal-simulink,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1926, pp. 134–151, 2000, cited By 9.
36. Y.-S. Kung, N. V. Quynh, N. T. Hieu, C.-C. Huang, and L.-C. Huang, “Simulink/modelsim co-simulation and FPGA realization of speed control IC for PMSM drive,” *Procedia Engineering*, vol. 23, pp. 718–727, 2011. [Online]. Available: https://doi.org/10.1016/j.proeng.2011.11.2571
37. M. Lora, R. Muradore, D. Quaglia, and F. Fummi, “Simulation alternatives for the verification of networked cyber-physical systems,” *Microprocessors and Microsystems*, vol. 39, no. 8, pp. 843–853, 2015.
38. R. Görgen, J. Oetjens, and W. Nebel, “Transformation of event-driven HDL blocks for native integration into time-driven system models,” in *Proc. of the IEEE/ECSI FDL 2012*, pp. 152–159.

39. M. Lora, E. Fraccaroli, and F. Fummi, “Virtual prototyping of smart systems through automatic abstraction and mixed-signal scheduling,” in *Proc. of IEEE/ACM ASP-DAC 2017*, pp. 232–237.
40. E. Fraccaroli, M. Lora, S. Vinco, D. Quaglia, and F. Fummi, “Integration of mixed-signal components into virtual platforms for holistic simulation of smart systems,” in *Proc. of IEEE/ACM DATE*, 2016, pp. 1–6).
41. F. Cremona, M. Lohstroh, D. Broman, E. A. Lee, M. Masin, and S. Tripakis, “Hybrid co-simulation: it’s about time,” *Software & Systems Modeling*, nov 2017. [Online]. Available: https://doi.org/10.1007/s10270-017-0633-6
42. N. Bombieri, F. Fummi, and G. Pravadelli, “Automatic abstraction of RTL IPs into equivalent TLM descriptions,” *IEEE Transactions on Computers*, vol. 60, no. 12, pp. 1730–1743, 2011.
43. ARM. Carbon Model Studio. http://carbondesignsystems.com/.
44. N. Bombieri *et al.*, “Hifsuite: tools for hdl code conversion and manipulation,” *EURASIP Journal on Embedded Systems*, vol. 2010, no. 1, pp. 1–20, 2010.
45. M. Lora, S. Centomo, D. Quaglia, and F. Fummi, “Automatic integration of cycle-accurate descriptions with continuous-time models for cyber-physical virtual platforms,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, mar 2018. [Online]. Available: https://doi.org/10.23919/date.2018.8342095
46. S. Centomo, M. Lora, and F. Fummi, “Transaction-level Functional Mockup Units for Cyber-Physical Virtual Platforms,” in *Proc. of ECSI/IEEE FDL*, 2018, pp. 1–8.
47. S. Centomo, M. Panato, and F. Fummi, “Cyber-Physical Systems Integration in a Production Line Simulator,” in *Proc. of IEEE VLSI-SoC 2018*, 2018, pp. 1–6.
48. R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, “Cyber-Physical Systems: The Next Computing Revolution,” in *Proc. of the 47th Design Automation Conference*. ACM, 2010, pp. 731–736.
49. S. W. Golomb, “Mathematical models: Uses and limitations,” *IEEE Transactions on Reliability*, vol. 20, no. 3, pp. 130–131, 1971.
50. E. A. Lee, “Fundamental Limits of Cyber-Physical Systems Modeling,” *ACM Transactions on Cyber-Physical Systems*, vol. 1, no. 1, p. 3, 2017.
51. M. Lora, S. Vinco, and F. Fummi, “Translation, abstraction and integration for effective smart system design,” *IEEE Transactions on Computers*, 2019.
52. F. Fummi, M. Lora, F. Stefanni, D. Trachanis, J. Vanhese, and S. Vinco, “Moving from Co-Simulation to Simulation for Effective Smart Systems Design,” in *Proc. of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 286.
53. M. Lora, S. Centomo, D. Quaglia, and F. Fummi, “Automatic Integration of Cycle-accurate Descriptions with Continuous-time Models for Cyber-Physical Virtual Platforms,” in *Proc. of ACM/IEEE DATE 2018*, 2018, pp. 1–6.
54. S. Tripakis, “Bridging the semantic gap between heterogeneous modeling formalisms and fmi,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*. IEEE, 2015, pp. 60–69.
55. S. Centomo, M. Lora, A. Portaluri, F. Stefanni, and F. Fummi, “Automatic Generation of Cycle-Accurate Simulink Blocks from HDL IPs,” in *Proc. of ECSI/IEEE Forum on Specification & Design Languages 2017 (FDL 17)*, 2017, pp. 1–8.
56. L. Cai and D. Gajski, “Transaction level modeling: an overview,” in *Proc. of the 1st IEEE/ACM/IFIP CODES-ISSS*. ACM, 2003, pp. 19–24.
57. S. Centomo, M. Lora, and F. Fummi, “Transaction-level functional mockup units for cyber-physical virtual platforms,” in *2018 Forum on Specification & Design Languages (FDL)*. IEEE, 2018, pp. 1–8.
58. MODELISAR Consortisuum, Modelica Association *et al.*, “Functional Mock-up Interface for Model Exchange and Co-Simulation – Version 2.0,” *Available from https://www.fmi-standard.org*.
59. D. Broman, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, “Requirements for Hybrid Cosimulation Standards,” in *Proc. of the 18th International Conference on Hybrid Systems: Computation and Control*. ACM, 2015, pp. 179–188.
60. S. Centomo, M. Lora, A. Portaluri, F. Stefanni, and F. Fummi, “Automatic Integration of HDL IPs in Simulink Using FMI and S-Function Interfaces,” in *Languages, Design Methods, and Tools for Electronic System Design: Selected Contributions from FDL 2017*, D. Große, S. Vinco, and H. Patel, Eds. Springer International Publishing, 2019, pp. 1–23.

61. G. Liboni, J. Deantoni, A. Portaluri, D. Quaglia, and R. De Simone, "Beyond Time-Triggered Co-simulation of Cyber-Physical Systems for Performance and Accuracy Improvements," in *Proc. of Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2018.
62. D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate Composition of FMUs for Co-Simulation," in *Proc. of the Eleventh ACM International Conference on Embedded Software*, 2013, p. 2.
63. S. Centomo, J. Deantoni, and R. De Simone, "Using SystemC Cyber Models in an FMI Co-Simulation Environment: Results and Proposed FMI Enhancements," in *Proc. of Euromicro Conference on Digital System Design (DSD)*. IEEE, 2016, pp. 318–325.
64. N. Bombieri, F. Fummi, and G. Pravadelli, "Automatic Abstraction of RTL IPs into Equivalent TLM Descriptions," *IEEE Transactions on Computers*, vol. 60, no. 12, pp. 1730–1743, 2011.
65. P. J. Mosterman and J. Zander, "Industry 4.0 as a cyber-physical system study," *Software & Systems Modeling*, vol. 15, no. 1, pp. 17–29, oct 2015. [Online]. Available: https://doi.org/10.1007/s10270-015-0493-x
66. D. Pfeifer, J. Valvano, and A. Gerstlauer, "SimConnect and SimTalk for distributed cyber-physical system simulation," *SIMULATION*, vol. 89, no. 10, pp. 1254–1271, mar 2013.
67. T. Peter and S. Wenzel, "Coupled simulation of energy and material flow using plant simulation and MATLAB simulink," *SNE Simulation Notes Europe*, vol. 27, no. 2, pp. 105–113, jun 2017.
68. P. Fritzson *et al.*, "OpenModelica - a free open-source environment for system modeling, simulation, and teaching," in *2006 IEEE Conference on Computer-Aided Control Systems Design*. IEEE, oct 2006.
69. Siemens, "Tecnomatrix, Plant Simulation."
70. IEEE, "IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows."
71. OSTATIC, "VHDLC."
72. Veripool, "Verilator."
73. B. Bailey and G. Martin, "Codesign Experiences Based on a Virtual Platform," in *ESL Models and their Application*. Springer, 2010, pp. 273–308.
74. F. Rosa, L. Ost, R. Reis, and G. Sassatelli, "Instruction-driven timing CPU model for efficient embedded software development using OVP," in *Proc. of IEEE ICECS 2013*, pp. 855–858.
75. D. Roy, L. Zhang, W. Chang, D. Goswami, and S. Chakraborty, "Multi-Objective Co-Optimization of FlexRay-Based Distributed Control Systems," in *Proc. of IEEE RTAS 2016*, pp. 1–12.
76. S. Tripakis, "Bridging the semantic gap between heterogeneous modeling formalisms and FMI," in *Proc. of IEEE SAMOS 2015*, pp. 60–69.
77. W. Mueller, M. Becker, A. Elfeky, and A. DiPasquale, "Virtual Prototyping of Cyber-Physical Systems," in *Proc. of ASPDAC 2012*, pp. 219–226.
78. M. Lora, S. Vinco, E. Fraccaroli, D. Quaglia, and F. Fummi, "Analog Models Manipulation for Effective Integration in Smart System Virtual Platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.
79. E. Fraccaroli, M. Lora, and F. Fummi, "Automatic abstraction of multi-discipline analog models for efficient functional simulation," in *Proc. of IEEE/ACM DATE 2017*, pp. 662–665.
80. A. Naderlinger, "Simulating Preemptive Scheduling with Timing-aware Blocks in Simulink," in *Proc. of ACM/IEEE DATE 2017*.
81. F. Pecheux, C. Lallement, and A. Vachoux, "VHDL-AMS and Verilog-AMS as alternative hardware description languages for efficient modeling of multidiscipline systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 2, pp. 204–225, feb 2005.
82. S. Centomo, J. Deantoni, and R. de Simone, "Using SystemC Cyber Models in an FMI Co-Simulation Environment: Results and Proposed FMI Enhancements," in *Proc. of Euromicro DSD 2016*, pp. 1–8.
83. S. Centomo, M. Lora, A. Portaluri, F. Stefanni, and F. Fummi, "Automatic generation of cycle-accurate simulink blocks from hdl ips," in *Proc. of ECSI/IEEE FDL 2017*, pp. 1–8.
84. D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of FMUs for co-simulation," in *Proc. of ACM EMSOFT 2013*.
85. E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing*, 2008, pp. 363–369.

86. J. Ceng, W. Sheng, J. Castrillon, A. Stulova, R. Leupers, G. Ascheid, and H. Meyr, "A high-level virtual platform for early MPSoC software development," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2009, pp. 11–20.
87. M. Lora, S. Vinco, and F. Fummi, "Translation, Abstraction and Integration for Effective Smart System Design," *IEEE Transactions on Computers*, 2019.
88. C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation: Using Ptolemy II*. Ptolemy. org Berkeley, CA, USA, 2014.
89. P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE (special issue on CPS)*, vol. 100, no. 1, pp. 13 – 28, January 2012.
90. R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov, "System-on-chip: Reuse and integration," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1050–1069, 2006.
91. C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, "Co-simulation: State of the art," *arXiv preprint arXiv:1702.00686*, 2017.
92. D. Broman, E. A. Lee, S. Tripakis, and M. Törngren, "Viewpoints, formalisms, languages, and tools for cyber-physical systems," in *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, 2012, pp. 49–54.
93. V. Berman, "Standards: The P1685 IP-XACT IP Metadata Standard," *IEEE Design & Test of Computers*, vol. 23, no. 4, pp. 316–317, apr 2006, http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1683721.
94. M. Hause *et al.*, "The sysml modelling language," in *Fifteenth European Systems Engineering Conference*, vol. 9, 2006, pp. 1–12.
95. R. Nane, S. van Haastregt, T. Stefanov, B. Kienhuis, V. M. Sima, and K. Bertels, "Ip-xact extensions for reconfigurable computing," in *ASAP 2011-22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 2011, pp. 215–218.
96. S. Vinco, M. Lora, E. Macii, and M. Poncino, "IP-XACT for smart systems design: extensions for the integration of functional and extra-functional models," in *Proc. of ECSI/IEEE FDL 16*, 2016, pp. 1–8.
97. A. Kamppi, L. Matilainen, J.-M. Määttä, E. Salminen, and T. D. Hämäläinen, "Extending ip-xact to embedded system hw/sw integration," in *2013 International Symposium on System on Chip (SoC)*. IEEE, 2013, pp. 1–8.
98. T. P. Perry, R. L. Walke, R. Payne, S. Petko, and K. Benkrid, "Ip-xact extensions for ip interoperability guarantees and software model generation," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2012, pp. 429–436.
99. W. Kruijtzer, P. Van Der Wolf, E. De Kock, J. Stuyt, W. Ecker, A. Mayer, S. Hustin, C. Amerijckx, S. De Paoli, and E. Vaumorin, "Industrial ip integration flows based on ip-xact standards," in *2008 Design, Automation and Test in Europe*. IEEE, 2008, pp. 32–37.
100. E. Salminen, T. D. Hämäläinen, and M. Hännikäinen, "Applying ip-xact in product data management," in *2011 International Symposium on System on Chip (SoC)*. IEEE, 2011, pp. 86–91.
101. G. Borriello, "Formalized timing diagrams," in *Proceedings The European Conference on Design Automation*. IEEE, 1992, pp. 372–377.
102. EDALab. HIFSuite. http://www.hifsuite.com/.
103. C. Andersson, J. Åkesson, and C. Führer, *Pyfmi: A python package for simulation of coupled dynamic models with the functional mock-up interface*. Centre for Mathematical Sciences, Lund University Lund, 2016.
104. P. Reynolds, A. Natrajan, and S. Srinivasan, "Consistency maintenance in multiresolution simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 3, pp. 368–392, jul 1997.
105. D. Huber and W. Dangelmaier, "A method for simulation state mapping between discrete event material flow models of different level of detail," in *2013 IEEE International Systems Conference (SysCon)*. IEEE, 2011, pp. 2877–2886.
106. M. Lora, S. Vinco, E. Fraccaroli, D. Quaglia, and F. Fummi, "Analog models manipulation for effective integration in smart system virtual platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 2, pp. 378–391, feb 2018.
107. N. Bombieri, F. Fummi, and G. Pravadelli, "Automatic abstraction of RTL IPs into equivalent TLM descriptions," *IEEE Transactions on Computers*, vol. 60, no. 12, pp. 1730–1743, dec 2011.

108. P. Cicconi, A. C. Russo, M. Germani, M. Prist, E. Pallotta, and A. Monteriu, "Cyber-physical system integration for industry 4.0: Modelling and simulation of an induction heating process for aluminium-steel molds in footwear soles manufacturing," in *2017 IEEE 3rd International Forum on Research and Technologies for Society and Industry (RTSI)*. IEEE, sep 2017.
109. T. Peter and S. Wenzel, "Coupled simulation of energy and material flow using plant simulation and matlab simulink," *Simulation Notes Europe Special Issue" Simulation in Production and Logistics: Impact of Energetic Factors*, 2017.
110. S. Centomo, E. Fraccaroli, and M. Panato, "From multi-level to abstract-based simulation of a production line," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1253–1256.
111. G. N. Schroeder, C. Steinmetz, C. E. Pereira, and D. B. Espindola, "Digital twin data modeling with automationml and a communication methodology for data exchange," *IFAC-PapersOnLine*, vol. 49, no. 30, pp. 12–17, 2016.
112. M. Schleipen, R. Drath, and O. Sauer, "The system-independent data exchange format CAEX for supporting an automatic configuration of a production monitoring and control system," in *2008 IEEE International Symposium on Industrial Electronics*. IEEE, Jun. 2008.
113. B. Wally, C. Huemer, A. Mazak, and M. Wimmer, "Iec 62264-2 for automationml," in *Proc. 5th Autom. ML User Conf.*, 2018, pp. 1–7.
114. E. van der Wal, "Plcopen," *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, p. 25, 2009.
115. A. ANSI, "Isa-95.00. 01: Enterprise-control system integration–part 1: Models and terminology," *Washington, DC: American National Standards Institute*, 2010.
116. E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, may 2008, pp. 363–369. [Online]. Available: https://doi.org/10.1109/isorc.2008.25
117. L. Monostori, "Cyber-physical production systems: Roots, expectations and r&d challenges," *Procedia Cirp*, vol. 17, pp. 9–13, 2014.
118. D. Catenazzo, B. OrFlynn, and M. Walsh, "On the use of wireless sensor networks in preventative maintenance for industry 4.0," in *2018 12th International Conference on Sensing Technology (ICST)*. IEEE, Dec. 2018.
119. ISO, *ISO 10816:2014 Mechanical vibration – Evaluation of machine vibration by measurements on non-rotating parts*. pub-ISO, 2014.
120. NFPA, *NFPA70B:2019 Recommended Practice for Electrical Equipment Maintenance*. NFPA, 2019, available in electronic form for online purchase at https://www.nfpa.org/NEC/electrical-codes-and-standards/NFPA-70B?code=70B.
121. ISO, *ISO 13373:2002 Condition monitoring and diagnostics of machines – Vibration condition monitoring*. pub-ISO, 2002, available in electronic form for online purchase at https://www.iso.org/standard/21831.html.
122. ——, *ISO 3747:2011 Acoustics - Determination of sound power levels and sound energy levels of noise sources using sound pressure*. pub-ISO, 2011, available in electronic form for online purchase at http://store.uni.com/catalogo/index.php/uni-en-iso-3747-2011.html.
123. UNI, *UNI EN 16714 Non-destructive testing - Thermographic testing*. pub-ISO, 2016, available in electronic form for online purchase at http://store.uni.com/catalogo/index.php/uni-en-16714-2-2016.html.
124. A. B. of Shipping Incorporated by Act of Legislature of the State of New York 1862, *Equipment Condition Monitoring Techniques*, 2016.
125. E. P. Carden and P. Fanning, "Vibration based condition monitoring: A review," *Structural Health Monitoring: An International Journal*, vol. 3, no. 4, pp. 355–377, Dec. 2004.
126. ISO, *ISO 7919:2009 Mechanical vibration – Mechanical vibration – Evaluation of machine vibration by measurements on rotating shafts*. pub-ISO, 2009, available in electronic form for online purchase at http://www.iso.org/standard/50527.
127. J. Zhu, D. He, and E. Bechhoefer, "Survey of lubrication oil condition monitoring, diagnostics, and prognostics techniques and systems," *Journal of chemical science and technology*, vol. 2, no. 3, pp. 100–115, 2013.
128. ISO, *ISO 3734:1997 Petroleum products – Determination of water and sediment in residual fuel oils*. pub-ISO, 1997, available in electronic form for online purchase at https://www.iso.org/standard/9221.html.

129. ——, *ISO 4406:2017 Hydraulic fluid power – Fluids – Method for coding the level of contamination by solid particles*. pub-ISO, 2017, available in electronic form for online purchase at https://www.iso.org/standard/72618.html.
130. S. M. A. Al-Obaidi, M. S. Leong, R. R. Hamzah, and A. M. Abdelrhman, "A review of acoustic emission technique for machinery condition monitoring: Defects detection & diagnostic," *Applied Mechanics and Materials*, vol. 229-231, pp. 1476–1480, Nov. 2012.
131. ISO, *ISO 22096:2007 Condition monitoring and diagnostics of machines – Acoustic emission*. pub-ISO, 2007, available in electronic form for online purchase at http://store.uni.com/catalogo/index.php/iso-22096-2007.html.
132. S. Bagavathiappan, B. Lahiri, T. Saravanan, J. Philip, and T. Jayakumar, "Infrared thermography for condition monitoring – a review," *Infrared Physics & Technology*, vol. 60, pp. 35–55, Sep. 2013.
133. ISO, *ISO 10880:2017 Non-destructive testing – Infrared thermographic testing*. pub-ISO, 2017, available in electronic form for online purchase at https://www.iso.org/standard/61881.html.
134. ——, *ISO 18251-1:2017 Non-destructive testing – Infrared thermography*. pub-ISO, 2017, available in electronic form for online purchase at https://www.iso.org/standard/61882.html.
135. ——, *ISO 18434-1:2008 Condition monitoring and diagnostics of machines – Thermography*. pub-ISO, 2008, available in electronic form for online purchase at https://www.iso.org/standard/41648.html.
136. ASTM, *ASTM E1934 - 99a Standard Guide for Examining Electrical and Mechanical Equipment with Infrared Thermography*. ASTM, 2018, available in electronic form for online purchase at https://www.astm.org/Standards/E1934.htm.
137. UNI, *UNI EN 16714-3 Non-destructive testing - Thermographic testing*. UNI, 2016, available in electronic form for online purchase at http://store.uni.com/catalogo/index.php/uni-en-16714-3-2016.html.
138. N. Bombieri, F. Fummi, and G. Pravadelli, "A mutation model for the systemc tlm 2.0 communication interfaces," in *Proceedings of the conference on Design, automation and test in Europe*. ACM, 2008, pp. 396–401.
139. N. DallâĂŹOra, S. Centomo, and F. Fummi, "Industrial-iot data analysis exploiting electronic design automation techniques," in *2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI)*. IEEE, 2019, pp. 103–109.
140. D. J. Pagliari, M. Poncino, and E. Macii, "Energy-efficient digital processing via approximate computing," in *Smart Systems Integration and Simulation*. Springer, 2016, pp. 55–89.
141. E. Fraccaroli, M. Lora, S. Vinco, D. Quaglia, and F. Fummi, "Integration of mixed-signal components into virtual platforms for holistic simulation of smart systems," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1586–1591.
142. Y. Chen, D. Jahier Pagliari, E. Macii, and M. Poncino, "Battery-aware design exploration of scheduling policies for multi-sensor devices," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, 2018, pp. 201–206.
143. W. Mahnke, S.-H. Leitner, and M. Damm, *OPC Unified Architecture*. Springer Berlin Heidelberg, 2009.
144. D. Goyal and B. Pabla, "Condition based maintenance of machine tools—a review," *CIRP Journal of Manufacturing Science and Technology*, vol. 10, pp. 24–35, Aug. 2015.
145. R. K. M. President and C. of Integrated Systems Inc., *An Introduction to Predictive Maintenance (Plant Engineering)*. Butterworth-Heinemann, 2002.
146. H. M. Hashemian, "State-of-the-art predictive maintenance techniques," *IEEE Transactions on Instrumentation and Measurement*, vol. 60, no. 1, pp. 226–236, jan 2011.
147. F. Nowlan and H. Heap, *Reliability-centered Maintenance*. Dolby Access Press, 1978.
148. Z. Peng and N. Kessissoglou, "An integrated approach to fault diagnosis of machinery using wear debris and vibration analysis," *Wear*, vol. 255, no. 7-12, pp. 1221–1232, Aug. 2003.
149. M. F. Yaqub, I. Gondal, and J. Kamruzzaman, "Machine fault severity estimation based on adaptive wavelet nodes selection and SVM," in *2011 IEEE International Conference on Mechatronics and Automation*. IEEE, aug 2011.
150. A. D. Nembhard, J. K. Sinha, A. J. Pinkerton, and K. Elbhbah, "Combined vibration and thermal analysis for the condition monitoring of rotating machinery," *Structural Health Monitoring*, vol. 13, no. 3, pp. 281–295, 2014.

151. G. S. Martinez, T. Karhela, R. Ruusu, T. Lackman, and V. Vyatkin, “Towards a systematic path for dynamic simulation to plant operation: OPC UA-enabled model adaptation method for tracking simulation,” in *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*. IEEE, Oct. 2017.
152. N. NFPA, “70b: Recommended practice for electrical equipment maintenance, quincy, massachusetts,” 2006.

List of Figures

1.1 Role of Digital Twin in Industry 4.0. 4
1.2 Overview of the proposed methodologies. Circles represents the aspects discussed in this thesis. 5

2.1 Overview of the proposed methodology. 10
2.2 OPC UA stack example 12
2.3 Example of a plant topology in AML. 14
2.4 The final result of SimulinkRoleClassLib 17
2.5 Attributes of the Simulink Constant component 18
2.6 The Constant RoleClass 18
2.7 The Constant RoleClass only mapped attribute 18
2.8 Attributes of the Simulink Gain component 19
2.9 The Gain RoleClass only mapped attribute 19
2.10 The Gain RoleClass 19
2.11 The Gain input port attributes 19
2.12 How the ports gets named for a Simulink component 20
2.13 Attributes of the Simscape Disk Friction Clutch component 20
2.14 The Disk Friction Clutch RoleClass 20
2.15 The Disk Friction Clutch mapped attributes 21
2.16 Port names for Simscape components 21
2.17 The result of the script reported in listing 2.1 22
2.18 Structure of the wrapper routine. 23
2.19 The simple model scheme. 24
2.20 OPC UA client main dialog. 25
2.21 Model variable - Server variable connections. 25
2.22 The Brake and the Gear signals for the Station. 26
2.23 The ICE lab Digital Twin. 27
2.24 Recipe 1. 28
2.25 Recipe 2. 28
2.26 Recipe 3. 28

2.27 Recipe 4. . . . 28
2.28 The four recipes input signals . . . 28
2.29 Recipe 1 Output . . . 29
2.30 Recipe 2 Output . . . 29
2.31 Recipe 3 Output. . . . 29
2.32 Recipe 4 Output. . . . 29
2.33 Output of the four recipes. . . . 29

3.1 Main steps of the proposed methodology. The boxes represent the different files involved and generated during the different steps of the methodology. Green boxes identifies models at the Register Transfer Level of Abstraction; orange boxes represents abstracted models; blue boxes represents interface-specific files. The different transformation steps of the methodology are represented by the arrows and their corresponding labels. Checked arrows represent steps reused from the methodology in [20], solid arrows represents novel steps. For automation reasons, the flow is built on top of HIFSuite [44], and the different manipulations are performed on the Heterogeneous Intermediate Format (HIF) provided by the suite. . . . 40

4.1 Overview of the contribution. . . . 54
4.2 Statechart representation of the coordinator algorithm for a Functional Mock-up Unit (FMU). . . . 56
4.3 Overview of the proposed approach, and comparison with the state-of-the-art methodology presented in [55]. . . . 60
4.4 Scheme of the *Smart Master Algorithm* with the *FMU Coordinator* of Transaction-Level FMUs. . . . 63
4.5 Trend of the simulation overhead using the Smart Master Algorithm with respect to the protocol latency. . . . 66
4.6 Scalability of the Smart Master Algorithm with respect to the number of FMUs. . 66

5.1 Overview of the CPS integration in a production line simulator . . . 72
5.2 Example of an OpenModelica Model . . . 78
5.3 Overview of proposed production line in Plant Simulation . . . 79
5.4 Overview of bending machine CPS . . . 80
5.5 Number of correct metal sheets . . . 82

6.1 Structure of the target simulation environment. . . . 84
6.2 Flow diagram implemented by HDL simulators. . . . 85
6.3 Statechart representation of HDL simulation. . . . 85
6.4 HDL simulation statechart after abstraction. . . . 85
6.5 Execution schemes involved in the RTL to cycle-accurate abstraction and translation methodology. . . . 85

6.6 Statechart representation of the calling sequence of Co-simulation C functions to simulate an initialized FMU. 86
6.7 Naïve structure of Simulink model with multiple computational components. . . . 89
6.8 89
6.9 89
6.10 State charts representing the FSMs used to manage synchronization locally to (a) FMUs and (b) S-functions. 89
6.11 Structure of the simulation environment combining the continuous-time model of the physical plant and a computational infrastructure composed by multiple FMUs. 90
6.12 Architecture of the CPS used as case study. Colors represents the original modeling languages of the components: blue for Verilog, yellow for VHDL, green for Simulink, while red closed source Hardware Description Language (HDL) Intellectual Propertys (IPs). 92
6.13 Scalability of the proposed approaches. 94
6.14 Time evolution of the ideal Simulink model. 95
6.15 HW-aware model evolution. 95
6.16 Time evolution of the model after the architectural optimization. 95
6.17 Time evolution of the model using the Network Interface. 95
6.18 Time evolution of the model after the SW optimization. 95
6.19 Time evolution of the CPS model in the different phases of SW design exploration and optimization. 95

7.1 Target simulation environment produced by this work. A cyber-physical virtual platforms allowing the simulation of complex Cyber-Physical System (CPS), while allowing integration of previously designed IP components. The cyber-physical virtual platforms must allow running control software for validation purposes. Architectural and protocol informations are used to automatically generate standardized interfaces, and the system-level coordinator enabling heterogenous co-simulation. 98
7.2 Statechart of the Functional Mock-up Interface (FMI) standard's master algorithm. 101
7.3 Overview of the Contribution. The paper proposes to exploit system-level information to build FMI-based alternatives to build cyber-physical virtual platforms overcoming the limitation of typical cycle-accurate co-simulation elying on the co-execution of HDL and dynamical systems simulators. From the bottom, up to the top, co-simulation environment exploits an increasing amount of system-level information and, consiquently, improving the simulation performance. 105
7.4 Case study used throughout the paper. 106
7.5 Crypto APB Timing Diagram. 114
7.6 Uart APB Timing Diagram. 114

7.7 Sensor's Timing Diagrams 114
7.8 Timing Diagram of platform components. 114
7.9 Overview of the smart coordinator obtained mixing System-level information, with all the execution phases. 115

8.1 Overview of the contribution of the work. The figure shows the *Multi-level* and *Abstract-Based* approaches for the CPS integration in a production line simulator. 124
8.2 Multi-level Common Interface and switching actions 125
8.3 Overview of the Experiment Setup. 128
8.4 Simplifed and Detailed Physical Models. 129

9.1 Overview of the proposed multi-level design methodology, starting from AutomationML neutral description.The methodology automatically generates the model of the plant, the process skeleton and the communication infrastructure. 134
9.2 AutomationML description of Plant running example using IEC62264-2 [113]. . 136
9.3 Overview of the steps that compose the generation of multi-level communication infrastructure. 138
9.4 Sequence Diagram of the communication protocol between Plant Simulation and Process Simulate. 143
9.5 Overview of the experimental setup generated with the design methodology. 144
9.6 Correlation of CPU Time and number of synchronization points. 146

10.1 Overview of Electronic Design Automation (EDA) Techniques with respect to ISA-95 Pyramid. 150
10.2 Vibration standards. 153
10.3 P-F interval - Intervals between the point at which a potential fault is detected and the point at which the fault occurs. 155
10.4 Overview of a generic Predictive Maintenance State Machine (PMSM). States represents the deviation observed from the Normal Behavior. 156
10.5 Overview of the framework implemented as an OPC Unified Architecture (OPC UA) Client. 157
10.6 Vibration deviation counter, product type A, series: 300 pieces. 160
10.7 Temperature deviation counter, product type A, series: 300 pieces. 161

11.1 Overview of the proposed approach. 164
11.2 Overview of a generic Monitoring State Machine (MSM). The states reflect the severity levels defined by ISO-10816. 168
11.3 Overview of a Predictive Maintenance Supervisor monitoring one MSM (*vibration MSM*). 170
11.4 Overview of a Predictive Maintenance Supervisor monitoring two MSMs (vibration and temperature). 171
11.5 Overview of the entire framework in validation mode. 172

11.6 Overview of the transmission system. The gears represent the critical component of the system. The model can be faulted in two different zone: gear tooth and the vibration sensor. 174

12.1 Overview of the Industrial Computer Engeneering Laboratory (ICE Lab) used as case study for all the presented methodologies. 181

List of Tables

3.1 Mapping of HDL events onto FMI and S-Functions primitives. 40
3.2 Mapping of HDL data-types to FMI and Simulink. 41
3.3 Characteristics of the benchmarks and time required for the automatic code generation. 48
3.4 Results obtained on the set of benchmarks. 49

4.1 Execution time of FMUs simulation using trivial Master Algorithm, with different number of iterations. 65
4.2 Execution Time Comparison of Normal Master Algorithm and Smart Master Algorithm with different protocol latencies. In all the scenarios, 10 million clock cycles of the system have been simulated. 65

5.1 Properties evaluated by the different simulation line versions 80
5.2 Execution Time comparison of different approaches. 80
5.3 Simulation times and percentage of errors of the proposed approaches, respect to the real bending machine. 81

6.1 Mapping of HDLs simulation events onto the primitives offered by the FMI standard and S-Functions. 88
6.2 Comparison between feature of the state-of-the-art [76] methodology and the proposed approaches. 92
6.3 Simulation overhead necessary to simulate one minute of the system by using different modeling alternatives. 93

7.1 Mapping of HDL events onto FMI primitives. 107
7.2 Mapping of HDL data-types to the FMI Standard. 108
7.3 Summary of the obtained experimental results. It compares the results considering both the two proposed techniques exploiting system-level information (*i.e.*, cycle-accurate and smart coordinator), and the two realized implementations (*i.e.*, based on PyFMI and based on FMI4CPP). The reference co-simulation environment is based on QuestaSim and Simulink. 117

8.1 Simulation times of bending operations for Mobile Units in the two different approaches. 127
8.2 Times needed to simulate one second in the two different approaches. 128

9.1 Simulated time of different robot operations considering the mass of the boxes. . . 145
9.2 The table reports simulation of Plant and Multi-Level approach with a batch of 1 million MUs. 145

10.1 Comparison of the principal measurements suitable for monitoring an industrial machine concerning the major condition monitoring standards. For each measurement are described the property that allows using of EDA techniques even for condition monitoring. 150
10.2 Example of sensors used in industry 4.0. 152
10.3 Experimental results with different productions in relation to different observed measurements in a simulation of 18000s. The results represent the time necessary to arrive in the `Maintenance State`. 162

11.1 ISO 10816: Definition of vibration severity levels with respect to the machine class. 165
11.2 Definition of severity levels for other observable measurements. In this case temperature and acoustic emissions are considered. 166
11.3 Definition of new standard for temperature MSM: a) reports the experiments for recipe A and B with different tooth fault levels, with respect to ISO 10816 severity levels. b) shows the obtained severity level ranges for temperature MSM. 175
11.4 Experimental results related to the properties of the PMS. 177