Doc ID	18041001R03
Doc Creation Date	10 apr 2018
Doc Revision	02
Doc Revision Date	28 jun 2018
Doc Status	Released

Workpackage	Deliverable ID
WP6 Implementation and Integration of the I-MECH platform	**D6.2 Guideline of I-MECH methodology (first iteration)**

Summary

This document describes the I MECH methodology that serves as a (mandatory) guideline for specifying, developing, implementing, testing and deploying the various I-MECH building blocks.

Author	Hans Kuppens
Keywords	I-MECH methodology, I-MECH architecture, model based design, model simulation, code generation, controller testing

Coordinator	Sioux CCM
Tel.	0031 (0)40.263.5000
E-Mail	info@i-mech.eu
Internet	www.i-mech.eu

ECSEL Joint Undertaking
Electronic Components and Systems for European Leadership

Table of contents

List of Figures

List of tables

(Open) Issues & Actions

Open Issues (and related actions) that need central attention shall be part of a file called "IAL - Issues & Action List – Partners" which is can be found in the Goolge Drive Partner Zone.

ID	Description	Due date	Owner	IAL ID
OI-xxxxx				

Document Revision History

Revision	Status	Date	Author	Description of changes	IAL ID / Review ID
R01	Draft	17-may-2018	Hans Kuppens	Initiate	
R02	Proposal	19-jun-2018	Hans Kuppens	For review	
R03	Approved	26-jun-2018	Hans Kuppens	Review comments processed, submitted	

Contributors

Revision	Affiliation	Contributor	Description of work
R01	Technolution	Marc van Eert	

Document control

	Status	Draft	Proposal	Released					
	Revision	R01	R02	R03					
Reviewer Name	Role	Selection							
Marc van Eert	Reviewer Technolution	X	X	X					
Gijs van der Veen	Reviewer Nexperia		X	X					
Carlos Yurre	Reviewer Fagor		X	X					
John Buijs	WP6 leader Technolution			X					

File Locations

Via URL with a name that is equal to the document ID, you shall introduce a link to the location (either in Partner Zone or CIRCABC)

URL	Filename	Date
D2.3	18010801R03 D2.3 Overall Requirements on I-MECH reference platform	5-jan-2018

Literature

Ref	Name	Publisher	Year
[1]	N/A	N/A	N/A

Abbreviations & Definitions

Abbreviation	Description
MIL	Model-In-the-Loop
SIL	Software-In-the-Loop
PIL	Processor-In-the-Loop
HIL	Hardware-In-the-Loop

Definition	Description

1 Introduction

The I-MECH project, which is about “Intelligent Motion Control Platform for Smart Mechatronic Systems”, has been divided into 8 workpackages.

- **WP1** (project management) and **WP8** (dissemination, communication) cover the project process and not the technical aspects.
- **WP2** is about business requirements and defines a high level architecture, dividing the platform into 3 layers, but doesn’t cover much technical implementation details either.
- **WP3**, **WP4** and **WP5** elaborate and refine the requirements from WP2 for each layer separately, enabling the development of building block components.
- **WP7** lists a few existing pilots, demonstrators and use cases, on which selected building block implementations are integrated, tested and verified.

None of these work packages however addresses the fundamental topic how the project mission “novel intelligence” (which is refined in D2.3) has been implemented by the I-MECH platform. This is the subject of **WP6**, and this document describes the I-MECH methodology that serves as a (mandatory) guideline for specifying, developing, implementing, testing and deploying the various I-MECH building blocks.

2 Principles of model based techniques

2.1 Model Based Design

A good definition of model based design can be found on Wikipedia:

Model-Based Design (MBD) is a mathematical and visual method of addressing problems associated with designing complex control, signal processing and communication systems. It is used in many motion control, industrial equipment, aerospace, and automotive applications. Model-based design is a methodology applied in designing embedded software.

In model-based design of control systems, development is manifested in these four steps:
1. *modeling a plant,*
2. *analyzing and synthesizing a controller for the plant,*
3. *simulating the plant and controller, and*
4. *integrating all these phases by deploying the controller.*

[source: wikipedia]

More information about model based design principles and about systems engineering can be found on the extensive website https://www.incose.org/.

In the scope of I-MECH, which aims at augmented intelligence in the field of advanced motion control, model based design plays an essential role.

More specifically, the advanced motion control system can be seen from different viewpoints:

· model viewpoint:	the physics of the plant and the mathematics of the controller are modeled (covering above-mentioned steps 1-3)
· deployment viewpoint:	this view describes the physical sensors, actuators and computing units on which the controller is deployed
· test viewpoint:	this view describes the configurations and steps in which the controller is simulated, integrated and tested (see section 2.4 for different test configurations)

2.2 Modeling and simulating the plant and controller

A generic model viewpoint of a controller and plant is shown in Figure 1. In this figure, the controller model tries to 'manipulate' the plant, e.g. by applying a force [N]. The plant simulator model takes physical properties into account (mass, friction, damping, …) and calculates the resulting displacement [m] which in turn is measured by the controller.

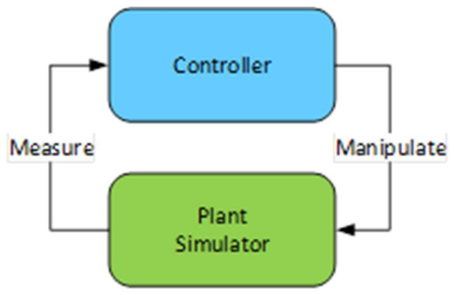

Figure 1: Generic model of a controller and plant

The process of modeling is an iterative process. First, the plant simulator model is drawn in more details and further refined until it sufficiently represents the real physical plant. This includes modeling the (expected) disturbances originating from external sources. Then the model of the controller is developed. This controller model aims to 'tame the physics' of the plant such that the measured values remain within acceptable limits.

In order to do so, the model developer has the following needs:

- A toolbox with components that can be used to compose and extend the controller and plant models
- A method for defining arbitrary testpoints inside the (whitebox) controller and plant models
- A method for defining arbitrary injection points inside the (whitebox) controller and plant models
- While running a simulation, a method for tracing the values of the testpoint signals and simultaneously injecting testdata at the injection points
- A toolbox with functions and methods to analyze and visualize the resulting time-traces from the simulations, and to compute the optimized controller parameters; possibly this is extended to a simulated system identification
- A method for changing (tuning) the controller parameters (auxiliary data); in this development stage the distinction between machine-specific and common parameters is not relevant yet but can be anticipated

The model developer will simulate all scenarios of the controller, and check if the simulated plant behaves as required. When the results are satisfactory, the model developer can proceed with the next steps – outlined in the sections below.

2.3 Automatic Code Generation

The previous step (modeling and simulating the plant and controller) results in models, which only ‘live’ inside a simulation tool. In the next step, these models must be transformed into executable lines of code. This step can be performed manually or automatically from the model.

Automatic code generation from the controller models gives quite a few significant benefits compared to handing over the models to a software engineer for manual coding:

- Functional equivalence between the models and the generated code
- Reduced development effort because no/little manual coding is needed
- Faster development iterations are possible, because automatic code generation is fast
- No coding errors due to human misinterpretations and mistakes

Apart from the models, also the controller-specific interface code can and should be generated automatically. The main advantage is that when an interface changes, the generated code on both sides of the interface is automatically updated and reflecting these changes.

It is not true that all the code can be generated automatically. Coding of the certain external interfaces will remain manual work, for example in the framework that provides the basis in which the generated code can run. In addition, integrating legacy and some common code into the models will partly remain manual work (e.g. S-Functions are used for integrating “external” C/C++ code into Simulink).

In most cases where motion is involved there will be a supervisory entity, responsible for system behavior. Such an entity, located in layer 3, sends commands to the motion controller and queries the status. Therefore, before starting with code generation, the model of the layer 2 controller (as shown in Figure 1) needs to be extended with an interface to layer 3, as shown in Figure 2. The resulting model is then ready for code generation.

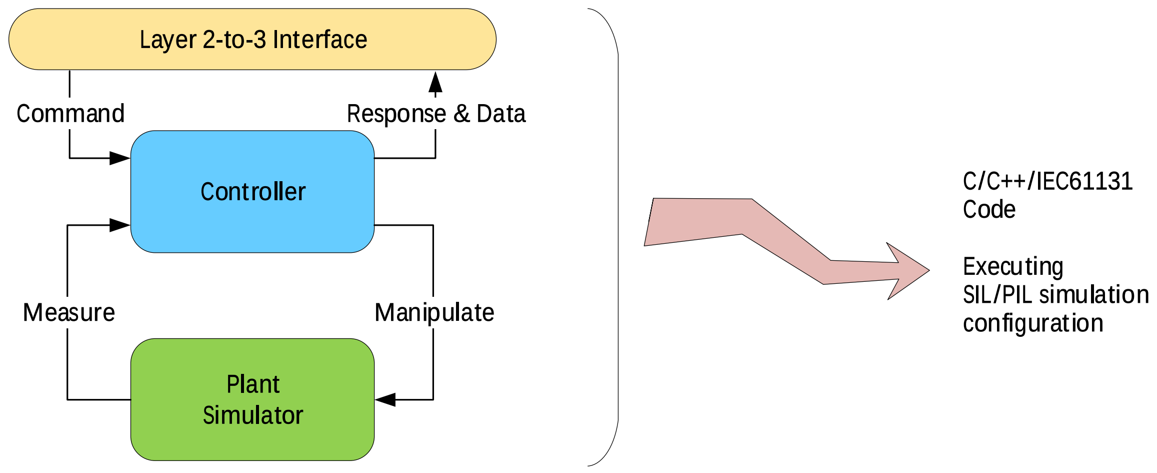

Figure 2: Generating code for simulation

Note: SIL/PIL is explained in section 2.4 below.

When the resulting generated C/C++/IEC61131 code is compiled, an executable is obtained that implements the same behavior as the model in Figure 1, including the plant simulator. This executable can run on the desired target platform.

Subsequently, a 2nd model can be created. In this model, the plant simulator is replaced by model elements that represent the interface of the SW to HW physics transducers (sensors and actuators).

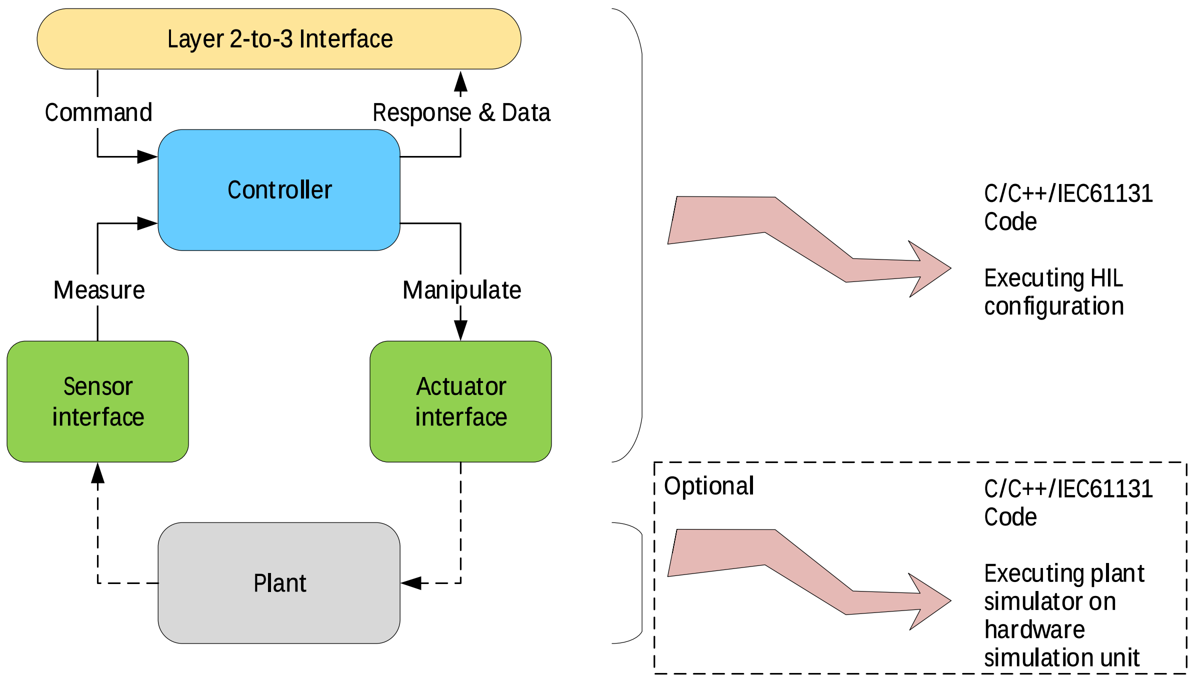

Figure 3: Generate code for hardware-in-the-loop configuration

Note: HIL is explained in section 2.4 below.

When the resulting generated C/C++/IEC61131 code is compiled, an executable is obtained that implements the same behavior as the model in Figure 2, as far as the controller and layer 3 interface are concerned. The plant simulator is missing; this executable now includes the access to the transducers.

When executing these executables, the model developer has still the same needs as outlined in section 2.2 before. The executable (within its execution environment) thus needs to provide support for:

- A method for defining arbitrary testpoints inside the (whitebox) controller
- A method for defining arbitrary injection points inside the (whitebox) controller
- While running the executable, a method for tracing the values of the testpoint signals and simultaneously injecting testdata at the injection points
- A method for changing (tuning) the controller parameters (auxiliary data)

At the same time, the aforementioned toolbox (with functions and methods to analyze and visualize the resulting time-traces from the simulations, and to compute the optimized controller parameters) must also process the data obtained from the executions.

Altogether, the true external interfaces are broader and more comprehensive than what is modeled explicitly as an external interface in Figure 2 and Figure 3: Parameters and signals (testpoints) are external interfaces too, implicitly, and access should be possible from applications, scripts and tools in layer 3.

2.4 Model Based Testing

The basis of Model Based Design is that in any stage of a project the behavior of the modeled controller and plant can be simulated and tested. The following definitions are public property:

MIL: **Model-in-the-loop:** The model (controller + plant) is simulated within the modeling environment.

SIL: **Software-in-the-loop:** Code is generated from the model (controller + plant) and executed, however this code is not necessarily real-time and not necessarily run on the target platform.

PIL: **Processor-in-the-loop:** Code is generated from the model (controller + plant), optimized for and executed on the target platform. The code is running in real-time but still includes the plant simulator.

HIL: **Hardware-in-the-loop:** Code is generated from the model (controller only), optimized for and executed on the target platform in its final configuration. Depending on the project constraints, the target platform is attached to either:

- a hardware simulation unit (code is generated from the plant model)
- an experimental hardware test setup (functional equivalent to the final plant)
- a standalone module of the final plant (in a safe environment)
- the final plant (under safe operating conditions)

2.4.1 The Functional Mock-up Unit (FMU)

In the ITEA2 Modelisar project the FMI (Functional Mock-up Interface) was defined (https://fmi-standard.org/).

A component which implements the FMI is called a FMU (Functional Mock-up Unit). A FMU contains:

- Description of interface data (in a XML file)
- Functionality (C-code or binary)
- *[optional]* Solver (for co-simulation)

A FMU often contains a model of a plant (with sufficiently required granularity) for simulation purposes. With FMI it is possible to combine multiple FMUs (from different vendors) and use them together in different simulation tools. Many simulation tools (including AMESim and Simulink) support FMI.

Similarly, a FMU may also contain a controller model. Thus it is possible to model the controller and plant with different modeling tools, and use the FMU technology to combine them into a single simulation model within the preferred simulation tool.

With respect to code generation, it is important to notice that the functionality of the FMU should be provided in either source C-code or as a binary for all used platforms (including the target platform of the controller).

3 Model Based Design in the scope of the I-MECH architecture

3.1 The I-MECH reference architecture

The reference architecture is obtained from D2.3/D2.4 and shown in the figure below.

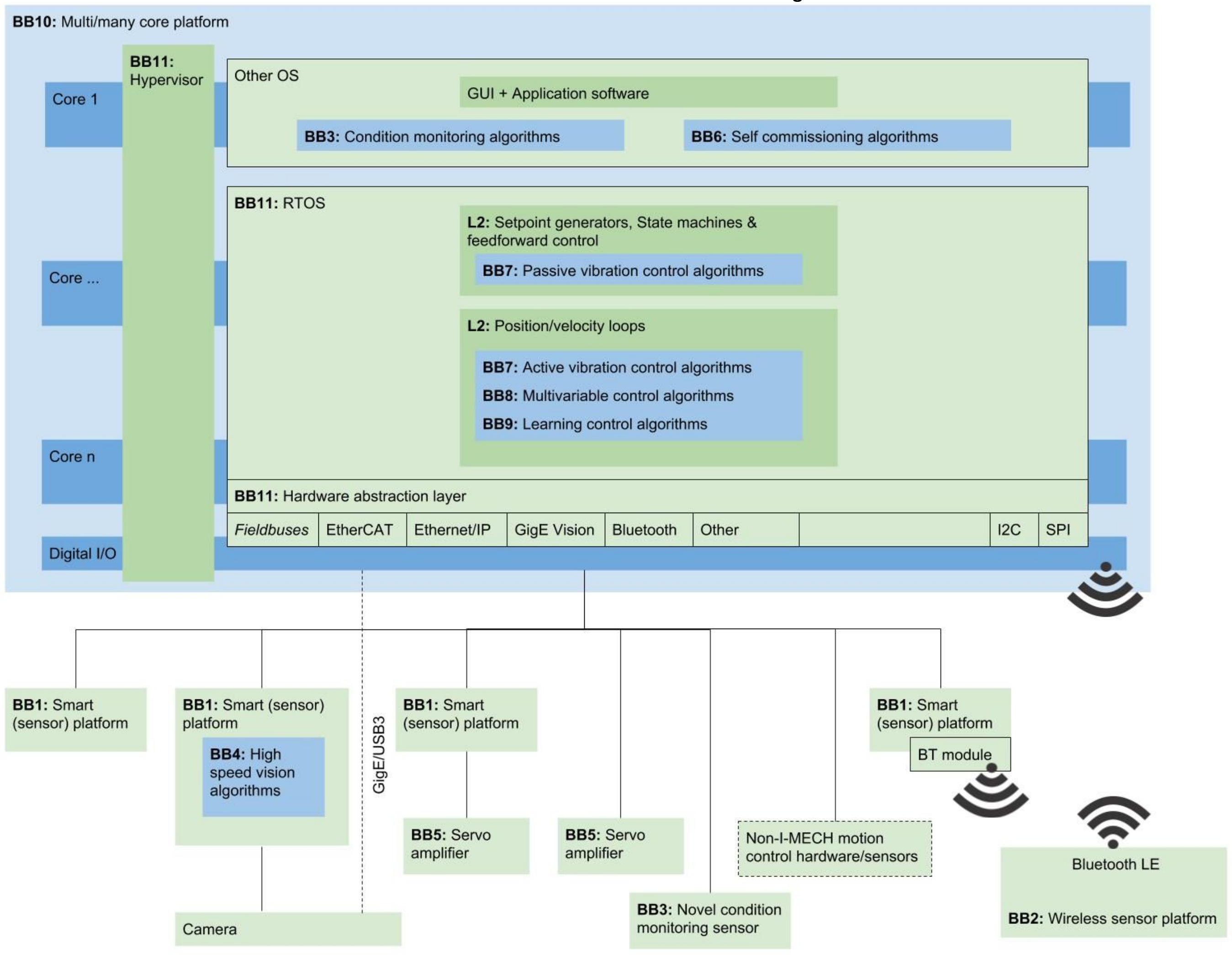
BB10: Multi/many core platform
BB11: Hypervisor
Core 1
Core ...
Core n
Digital I/O
Other OS
GUI + Application software
BB3: Condition monitoring algorithms
BB6: Self commissioning algorithms
BB11: RTOS
L2: Setpoint generators, State machines & feedforward control
BB7: Passive vibration control algorithms
L2: Position/velocity loops
BB7: Active vibration control algorithms
BB8: Multivariable control algorithms
BB9: Learning control algorithms
BB11: Hardware abstraction layer
Fieldbuses
EtherCAT
Ethernet/IP
GigE Vision
Bluetooth
Other
I2C
SPI
BB1: Smart (sensor) platform
BB1: Smart (sensor) platform
BB4: High speed vision algorithms
GigE/USB3
BB1: Smart (sensor) platform
BB1: Smart (sensor) platform
BT module
BB5: Servo amplifier
BB5: Servo amplifier
Non-I-MECH motion control hardware/sensors
Bluetooth LE
BB2: Wireless sensor platform
BB3: Novel condition monitoring sensor
Camera

Figure 4: I-MECH reference architecture (source: D2.3/D2.4, slightly modified)

This reference architecture maps pretty well on the model based design paradigm described in the previous chapter.

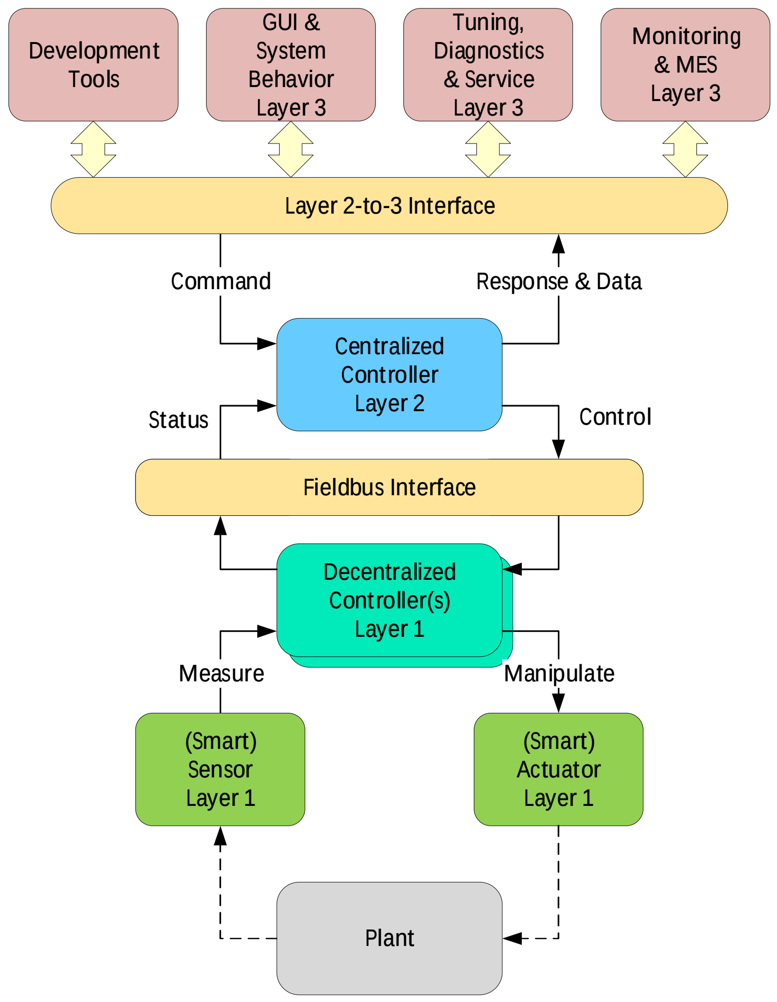

Figure 5: Generic model of an I-MECH controller

In Figure 5, the decentralized controller(s) are optional and may be absent, in which case all sensors and actuators are controlled centrally via the fieldbus. In section 3.3 the centralized versus decentralized control paradigm is further elaborated.

Figure 5 shows the HIL configuration. When the plant simulator model is used (the PIL simulation or HIL simulation with a hardware simulation unit), a setup is obtained that can also be used in combination with Layer 3 software. This allows particularly for integration and testing of layer 2 software with layer 3 software in the absence of the plant hardware. Only the computing platform is needed, including the network infrastructure and optionally the hardware simulation unit setup.

3.1.1 Access to parameters

When executing a simulation or running a real controller of an I-MECH model as shown in Figure 5, the various applications in layer 3 need to have access to all kinds of “parameters” of the building block components that are found in all layers.

Some parameters are “static” and immutable (determined during model design, when the code of the controller is compiled), while other parameters are “dynamic” and can be modified during runtime execution of the controller. This depends on the nature of the parameters involved.

Typically, parameters are used for (but not limited to):
- System configuration

  - Static configuration (related to fixed system hardware setup)
  - Dynamic configuration (i.e. calibration, tuning, but also 'adaptive' system hardware setup)
- System initialization
- System control from application layer, system behavior (incl. trajectory commands)
- System excitation (e.g. noise injection, error injection)

Evidently, all these parameters need to be transmitted through the layer interfaces shown in Figure 5.

3.1.2 Access to signals

When executing a simulation or running a real controller of an I-MECH model as shown in Figure 5, the various applications in layer 3 need to have access to all kinds of "signals" of the building block components that are found in all layers. These signals include input, output, auxiliary and internal[1] data.

Some signals are "required" and access must always be available, while other signals are "optional" and access can be (temporarily) switched on/off during runtime execution of the controller. This depends on the nature and need of the signals involved.

Generally the signals can be retrieved in the following ways:
- Snapshot, a single signal value is queried
- Single-shot trace, a time-series of signal values is obtained during a limited time interval (with trigger options)
- Continuous trace, a time-series of signal values is obtained continuously

Every signal value is provided with a time stamp. The above must be extendable for multiple simultaneous signals, in which case all values are sampled simultaneously at the same time stamp (if the hardware capabilities support this) or at the same controller execution step (in which case the time of the controller is provided).

The continuous trace demands the highest communication performance and data bandwidth, since the signal acquisition and data streaming take place in real time, as opposed to the snapshot and single-shot trace where the signal acquisition can be buffered locally and transmitted at a lower data rate.

It is up to the system designer to choose the proper signal acquisition approach within the constraints of the system performance capabilities. As a rule, all signals can be accessed in all possible ways, not restricted by the I-MECH architecture implementation but in accordance to the system designer's choice.

Typically, signals are used for (but not limited to):
- Get system state
- Get system snapshot, signal tracing, monitoring, data acquisition
- Report events and notifications
- Report errors and warnings

Evidently, all these signals need to be transmitted through the layer interfaces shown in Figure 5.

3.1.3 RAMI 4.0, Administration Shell and OPC UA

RAMI 4.0 (Reference Architectural Model for Industrie 4.0) is an approach for smart and connected factory systems. It takes into account the factory topology (hierarchy and networks), layering architecture (from physical things to organization and business processes) and the product life cycle.

In order to communicate and share information, the **Administration Shell** is the standardized digital representation of each manufacturing system within such a factory.

OPC UA is an extensible, secure and platform independent framework that enables the communication between the Administration Shells.

[1] It is strongly suggested to

See https://www.plattform-i40.de/I40/Redaktion/EN/Downloads/Publikation/hm-2018-trilaterale-coop.pdf and https://opcfoundation.org/about/opc-technologies/opc-ua/ for more details.

For I-MECH, the building blocks will also have an Administration Shell that represent the building block characteristics (such as control, status, configuration, electronic data sheet, self-reflection and etcetera) in the factory environment. This Administration Shell “hooks up” at layer 3, and connects to the underlying BB components in layers 1, 2 and 3. Also, the Administration Shells of the various BB’s will coexist with the Administration Shell of the entire I-MECH controller.

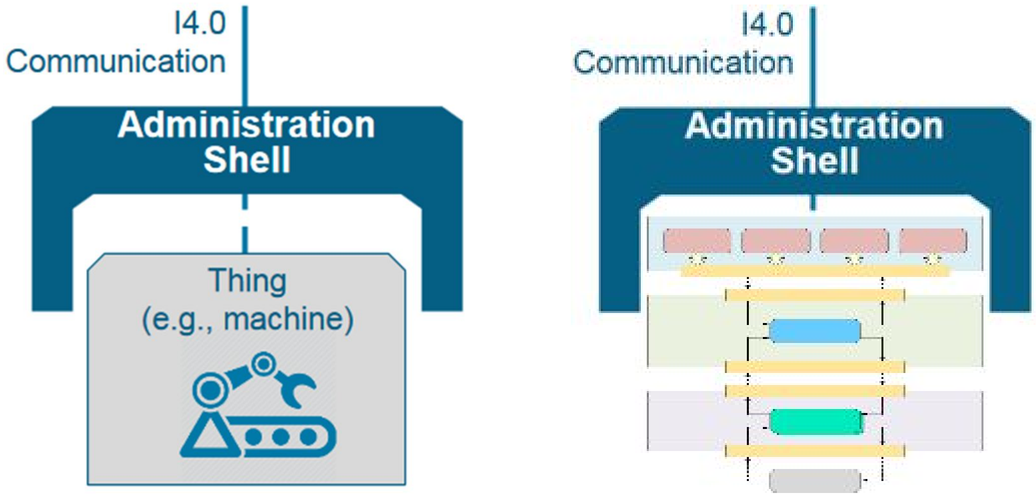

Figure 6: Administration Shell of an I-MECH controller or I-MECH building block

3.2 Location of building block components

An I-MECH building block is not necessarily a single entity, but is typically a set of software and hardware components with distinct functions and roles (including simulators), which altogether fulfill the objectives of the applicable building block. Therefore, a building block can be represented by various model components that each have a dedicated function within the composed controller models. The building blocks components are not restricted to a single layer, but generally are distributed among the layers 1, 2 and 3. See Figure 7 for an example.

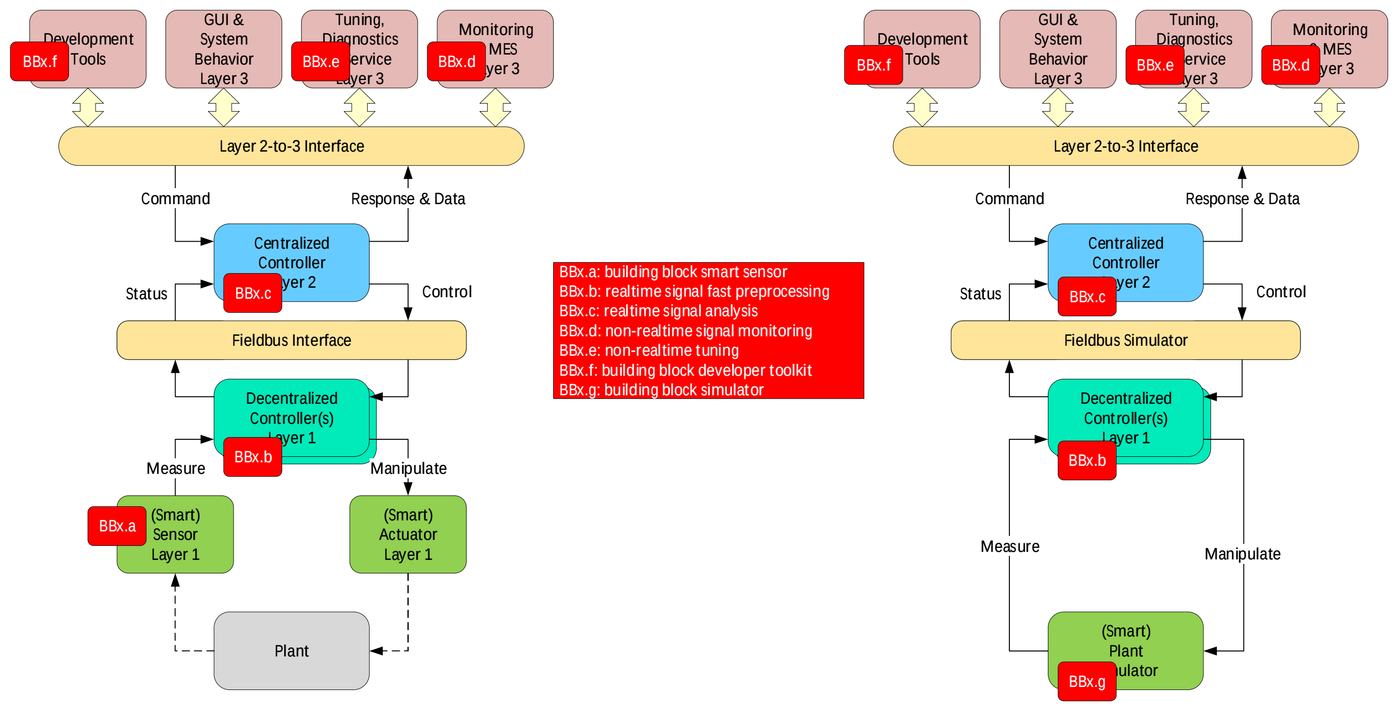
Development Tools
BBx.f
GUI & System Behavior Layer 3
Tuning, Diagnostics Service Layer 3
BBx.e
Monitoring MES Layer 3
BBx.d
Layer 2-to-3 Interface
Command
Response & Data
Centralized Controller Layer 2
BBx.c
Status
Control
Fieldbus Interface
Decentralized Controller(s) Layer 1
BBx.b
Measure
Manipulate
(Smart) Sensor Layer 1
BBx.a
(Smart) Actuator Layer 1
Plant
BBx.a: building block smart sensor
BBx.b: realtime signal fast preprocessing
BBx.c: realtime signal analysis
BBx.d: non-realtime signal monitoring
BBx.e: non-realtime tuning
BBx.f: building block developer toolkit
BBx.g: building block simulator
Fieldbus Simulator
(Smart) Plant Simulator
BBx.g

Figure 7: Example locations of building block components

This section describes how and where building block components may appear in the models when the model based techniques are applied.

Building blocks can have different topologies, not limited to a single appearance.

Table 1: Components of BB's and their locations

Building Blocks	Typical BB components
BB1/BB10/BB11	Model components that provide entry points to the HAL or OS specific features.
BB2	Model components that provide communication and data exchange with wireless sensors.
BB4	Generally the image processing is expected to take place in FPGA, and the model based techniques described in this chapter are less usable – at least they fall in a different scope and technology domain. However, some front end interface to this BB4 is still needed, and model components will be provided for that purpose.
BB5	Depending on how this BB5 is employed, it will either have some simple front end interface in layer 2 (at the simple side of the spectrum) or a more detailed front end interface in case the servo amplifier is modeled itself, possibly in conjunction with BB1.
BB3/6/7/8/9	These building blocks will have model components that are typically used in the layer 2 controller, but possibly also in the layer 1 decentralized controllers. In either case, these building block components need to be accessed, configured and controlled from matching building block components in layer 3.

3.3 Centralized vs decentralized control paradigm

The I-MECH methodology and architecture provide support for both the centralized and the decentralized control paradigms. It is up to the application architect to decide on the best solution.

In the centralized control paradigm, all control algorithms run on a single computing unit. All sensor values are collected centrally and available to all controllers. Also data exchange between controllers is straightforward.

In the decentralized control paradigm, additional computing units are added to the fieldbus. Typically, these decentralized controllers can run at higher sample rates and achieve less sensor-to-actuator delays. It is possible to reduce data bandwidth consumption or EMI disturbances.

Examples of decentralized controllers:
BB1: Smart sensor modeling (especially with actuating capabilities)
BB4: Vision sensor modeling
BB5: Servo amplifier modeling (current, velocity and position loops; feedforwards)

3.3.1 Decentralized controller modeling

When modeling a decentralized controller, special attention must be paid to the internal parameters and signals of the decentralized modules. Depending on the needs of either the developer or the layer 3 software, these parameters and signals must be made available at layer 3, which means that these need to be communicated via fieldbus between layer 1 and layer 2. This requires explicit modeling of these parameters in the fieldbus (whereas the layer 2 internal parameters are already implicitly available to layer 3, as required in section 2.3).

When setting up a decentralized controller, including a model of the fieldbus is initially not mandatory. In the 1st iteration, the fieldbus can be left out, while in the 2nd iteration the fieldbus effects can be simulated. Finally, the fieldbus simulator will be replaced by the separate master and slave interface components. This iterative approach becomes more apparent when the decentralized controller is simulated and tested as described in section 3.4.2 and shown in Figure 10.

3.4 Automatic Code generation

3.4.1 Generating code for executables

As described in section 2.3, automatic code generation is an important step of model based engineering.

In the first place, this applies to generated C/C++/IEC61131 code that can be compiled to an executable running on the centralized controller (layer 2), and *optionally* to additional executables running on the decentralized controllers (layer 1).

Furthermore, from the models it is possible to generate code for the Administration Shell (OPC UA servers), to support the development of business applications (layer 3 and higher).

Finally, in cases where OPC UA is not needed, it is possible to generate the applicable interfaces (in multiple programming languages), e.g. for development tools, the GUIs and so on (layer 3).

In many cases, the generated code has an interface to (and must be tailored to) the OS and to the hardware (sensors and actuators, via a HAL).

Layer 3 has a general purpose operating system (GP OS: e.g. Windows, Linux), layer 1+2 is real-time (RTOS: e.g. Erika3) with the access to the hardware components. The generated code for the centralized controller (layer 2) will be deployed on BB10/BB1, while the generated code for the optional decentralized controller (layer 1) will be deployed on BB1.

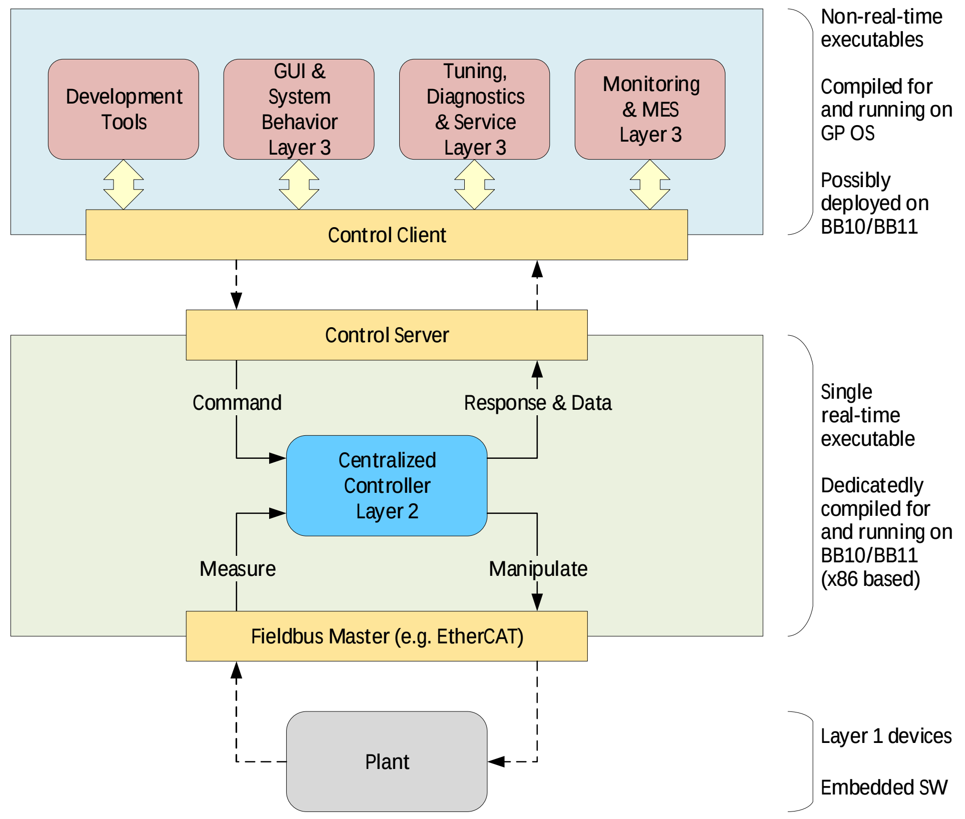
Development Tools
GUI & System Behavior Layer 3
Tuning, Diagnostics & Service Layer 3
Monitoring & MES Layer 3
Control Client
Non-real-time executables
Compiled for and running on GP OS
Possibly deployed on BB10/BB11
Control Server
Command
Response & Data
Centralized Controller Layer 2
Measure
Manipulate
Fieldbus Master (e.g. EtherCAT)
Single real-time executable
Dedicatedly compiled for and running on BB10/BB11 (x86 based)
Plant
Layer 1 devices
Embedded SW

Figure 8: Code generation for centralized controller

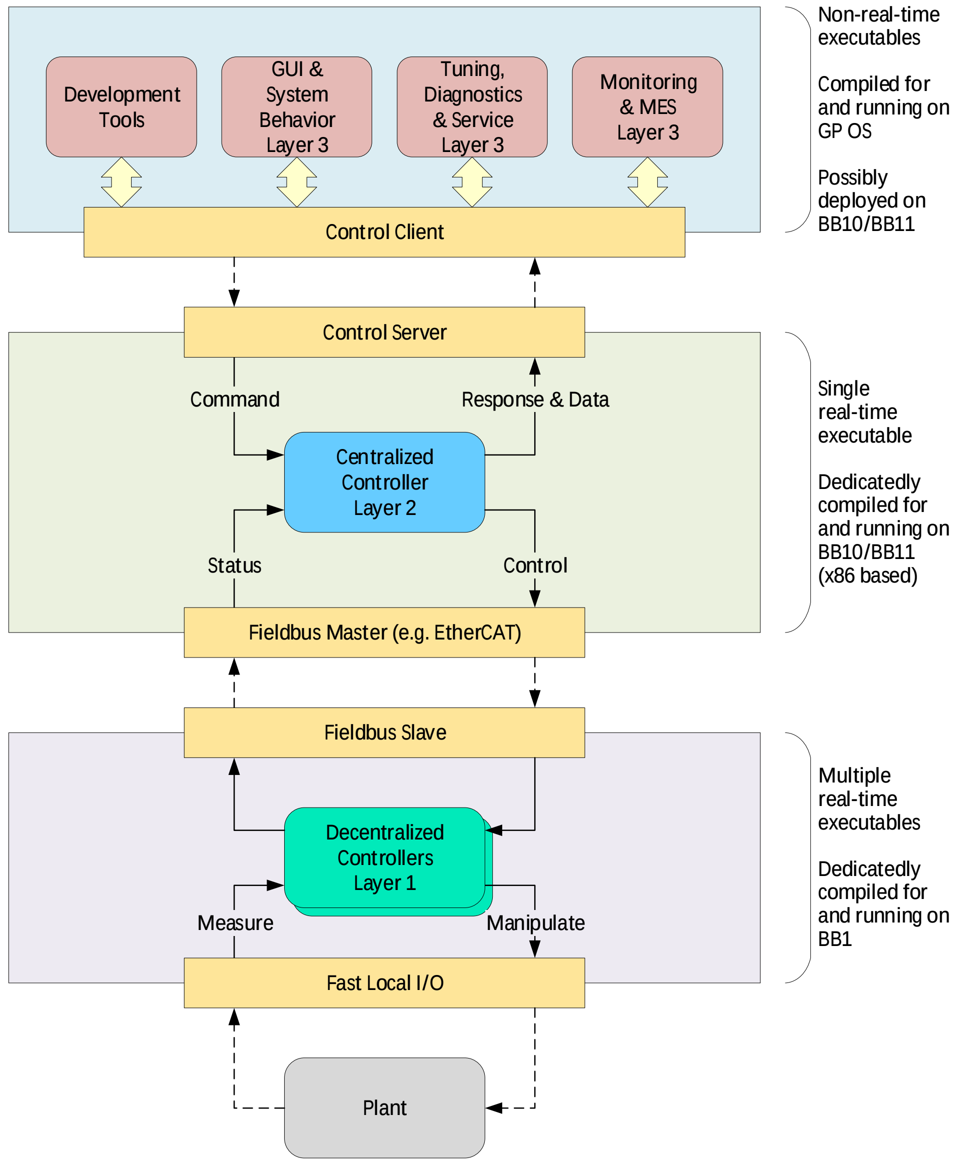
Development Tools
GUI & System Behavior Layer 3
Tuning, Diagnostics & Service Layer 3
Monitoring & MES Layer 3
Control Client
Non-real-time executables
Compiled for and running on GP OS
Possibly deployed on BB10/BB11
Control Server
Command
Response & Data
Centralized Controller Layer 2
Status
Control
Fieldbus Master (e.g. EtherCAT)
Single real-time executable
Dedicatedly compiled for and running on BB10/BB11 (x86 based)
Fieldbus Slave
Decentralized Controllers Layer 1
Measure
Manipulate
Fast Local I/O
Multiple real-time executables
Dedicatedly compiled for and running on BB1
Plant

Figure 9: Code generation for centralized and decentralized controllers

3.4.2 Generating code variants for simulation and testing

During the development process, there is the need of generating code for different successive simulation and testing configurations.

These configurations are:

- PIL: Central computing unit only (without fieldbus simulator)
- PIL: Central computing unit only (with fieldbus simulator)
- PIL: Including decentral computing units (implies fieldbus-in-the-loop)
- HIL: Including the hardware (test) setup or a hardware simulation unit

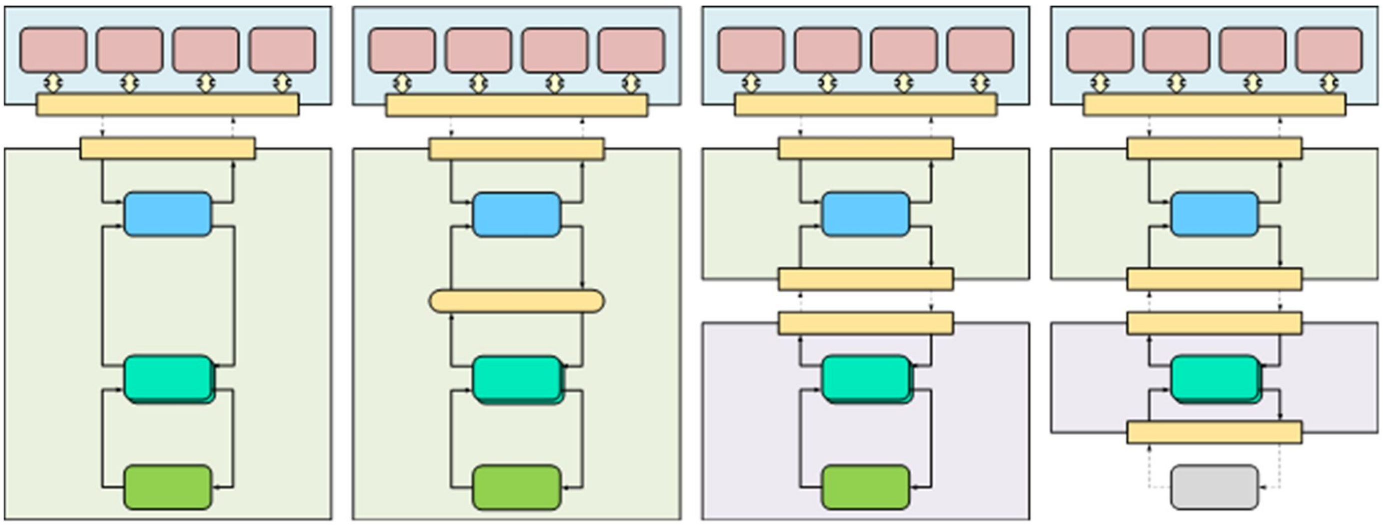
Figure 10: Code generation steps

Figure 10 shows these code generation steps (for the decentralized control paradigm).

3.5 Engineering programming language

During the development of a new system (or maintaining existing systems with an update) there are many system specific tests and calibrations that need to be performed. These tests and calibrations are typically performed by an application/integration engineer (who is not the end-user) with only “basic” programming skills and requiring more functionality than the end-user. To facilitate this integration work, an engineering programming language software interface is required. This allows the engineer to access the necessary functionality from the software layer in his engineering language environment of choice (typically Matlab or Python). This interface should provide typical functions to control the system and acquire the necessary data, which can directly be analyzed and visualized in the engineering language environment.

Typical functionality of this engineering programming language covers:

- System configuration
- System initialization
- System I/O functions
- Single axis control
    - Axis state
    - Motion trajectory generation
- Multi-axis or virtual axis control
- System monitoring
- Signal acquisition and system excitation
- File I/O functions

The engineering programming language defines and implements an interface in layer 3, with which all the components and building blocks can be accessed (via the communcation channel between the control client and the control server).

3.6 Automatic model generation

So far, it is assumed that the I-MECH model itself is the basis and starting point for model based engineering. However, there are development approaches to be considered in which parts of the model are automatically generated or configured on the basis of some “supermodel”. For example, think of a “factory description file” containing a “factory model” from which the interfaces and context of an I-MECH model are extracted.

For the I-MECH building blocks, it is important to realize that certain model aspects need to be imported from 'other' tooling, e.g. via XML files. Although this is not an anticipated and mandatory I-MECH requirement, an I-MECH building block yet needs to be flexible and compliant to such a way-of-working, irrespective of whether this import is performed manually or automatically. A standardized method for this WoW doesn't exist yet, and is not within the scope of the I-MECH project.

3.7 I-MECH intelligence functions

The architecture in this chapter (Figure 5) has been mainly described from a physical or deployment viewpoint. A functional viewpoint of an I-MECH system is shown in Figure 11.

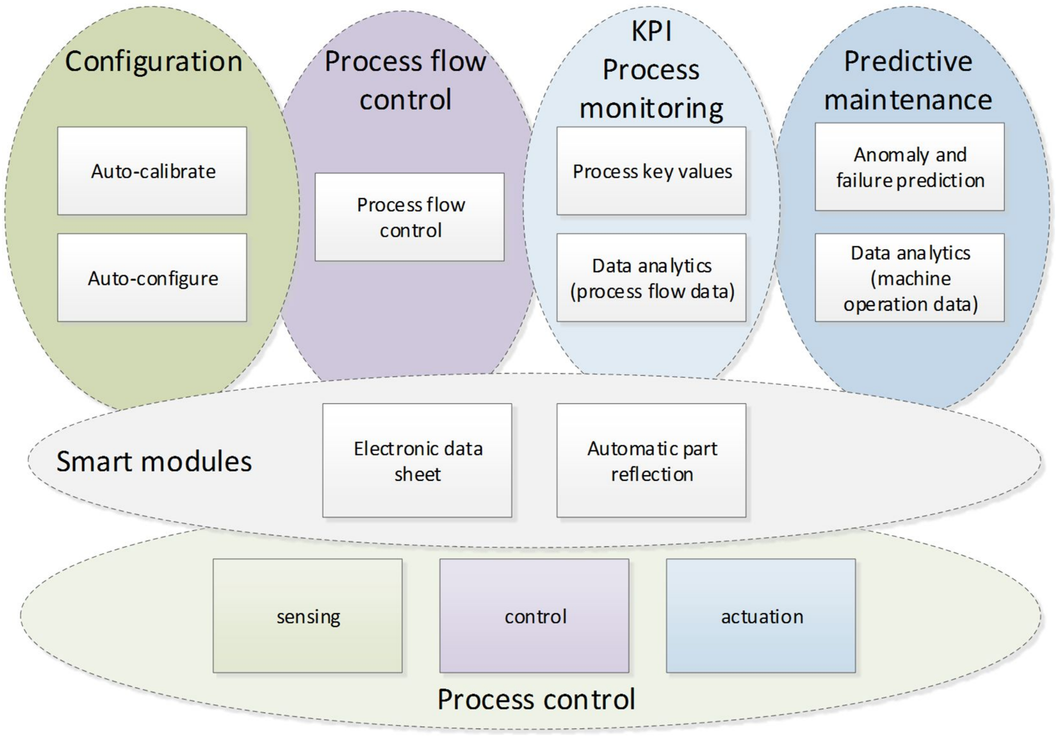

Figure 11: Functional view of an I-MECH system

The functional basis of an I-MECH system is process control, in which modules become smart modules when they are "self-descriptive". At the factory level, an I-MECH system provides advanced functionality in the four main pillars: Process flow control, process monitoring, configuration and predictive maintenance.

How these intelligence functions are mapped onto the architecture is elaborated in Figure 12.

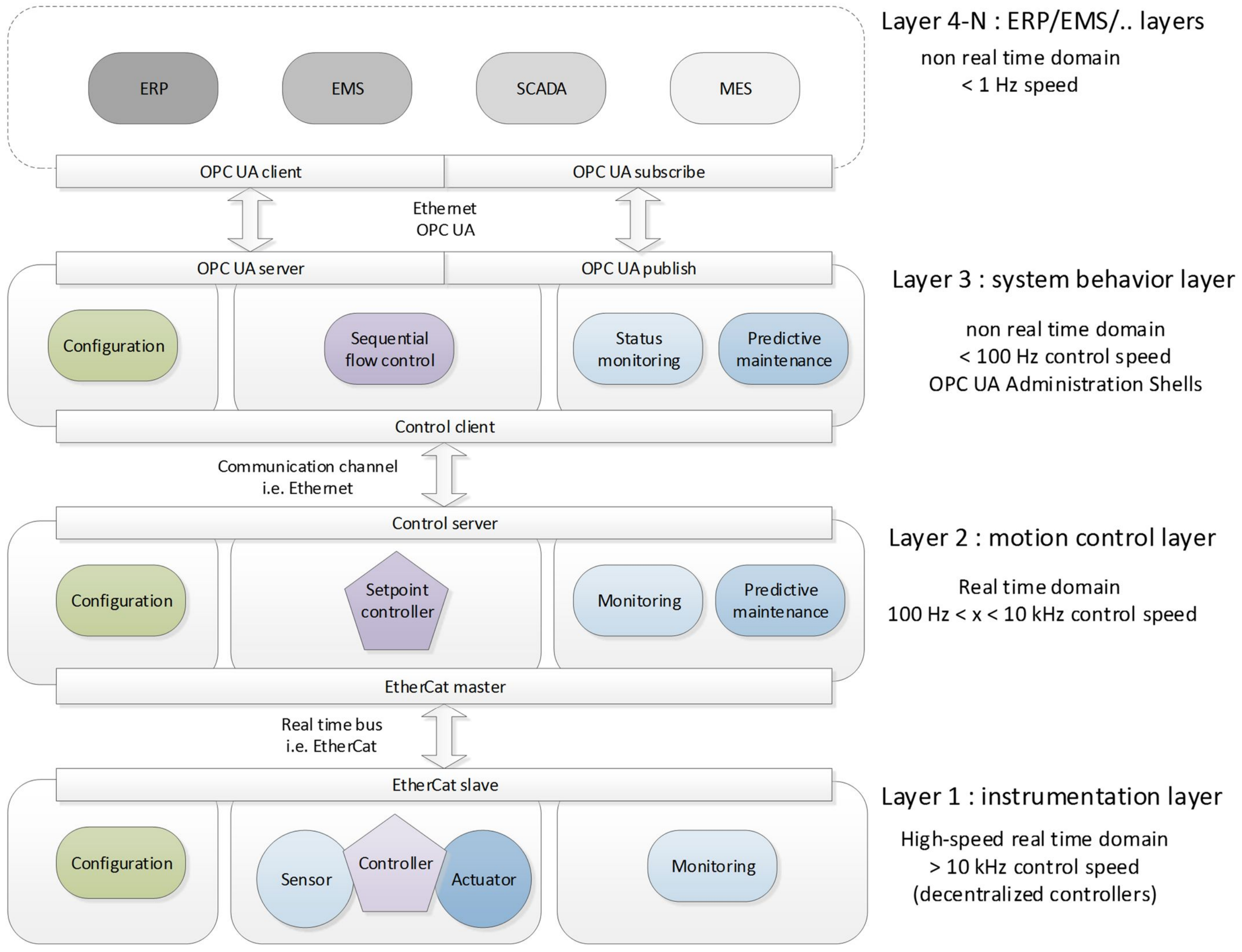

Figure 12: Functional mapping to the layers of an I-MECH system for configuration and operation

As Figure 12 shows, the intelligence functionality is vertically distributed among the various layers, and the applicable data is transferred via the available communication channels.

For the EtherCAT fieldbus, fast data signals (process data) are transmitted via PDO messages, while slow data signals and static information can be transmitted using the acyclic CoE and FoE message protocols (or alternatively via EoE or AoE).

The communication channel between control server (layer 2) and control client (layer3) realizes a further dissemination of the intelligence information and functionality.

The OPC UA Administration Shells (see section 3.1.3) publish the intelligence information and functionality to the factory layers.

4 I-MECH Methodology

4.1 Matlab/Simulink as the model based design tool

Matlab/Simulink is chosen as the preferred model based design tool, because it is quite versatile in the field of motion control algorithm development, which includes simulation and code generation. Many companies seem to agree that MATLAB/Simulink is the de facto standard for these purposes. However, there must always exist an alternative way that makes it possible to disseminate I-MECH compliant results without being bound to a specific vendor (i.e. The Mathworks). Having said this, Matlab/Simulink will be used as the primary steppingstone tool.

With Matlab/Simulink, it is possible to:
- Create controllers, import building block parts from managed component libraries
- Include models of the plant and then simulate the controllers
- Generate code from the controllers, tailored to a dedicated RTOS and processing hardware platform
- Generate code for layer 3 applications, for interfacing to the running real-time controllers
- Collect, visualize and analyze data obtained from the running controllers

All with the support of customizable Matlab scripts.

Interfaces to Simulink blocks can be defined as Simulink buses. This applies to both input and output signals, as well as block parameters[2] (auxiliary input data, run-time configurable) and auxiliary output data.

Simulink also provides a few mechanisms for IP protection, when sharing Simulink blocks with end users.
- Protected Simulink blocks restrict access of end users regarding viewing, simulating and code generation.
- Furthermore, it is possible to convert Simulink blocks to S-function blocks with precompiled static libraries that can be shared with end users.
- Static libraries are a good means to share IP protected contents anyway, also when these are not generated from Simulink block source code.

Similar methods exist for Matlab functions requiring IP protection.
- Protected Matlab function files obfuscate the source code (but does not encrypt them)
- Compile Matlab functions into binary format
- Compile non-Matlab functions as MEX files (for use in other Matlab functions)

4.2 In scope of the product creation process

A central aspect of new product development (NPD) is product design. The process of a product design is often executed following a phased way-of-working model. In Table 2 the potential role of the I-MECH methodology is put forward for the successive PCP phases. This table serves as a generic guidance, and can be tailored freely to the specific project and system characteristics.

Table 2: I-MECH during the product creation process phases

PCP phase	I-MECH methodology activity	Benefits
Requirements	Identify principal characteristics of the plant, and derive the principal requirements for the controller.	Early capture of SMART plant characteristics and SMART controller requirements.
Concept	Consider and choose the (initial) controller architecture. Define and execute the main use case scenario (MIL simulation), with low granularity and only the “happy flow”.	Principal requirements are implemented and simulated in a model. First embryonic feedback on concept.

[2] For convenience, an encapsulating Simulink subsystem can provide the constant (not run-time but compile-time configurable) parameters in a mask, internally merge these parameters into the defined Simulink bus and wire this bus to the parameter port of the encapsulated Simulink block.

PCP phase	I-MECH methodology activity	Benefits
System design	While the system design evolves, the models of the plant and controller evolve at the same pace. The main use case scenario is extended with the most important secondary and alternative scenarios, including the critical exceptions. The granularity of the controller architecture and the MIL simulations is increased.	The completeness of the system is secured in the completeness of the model. Feedback on the system regarding the system behavior as well as the capability to deal with exceptional situations.
Component design	The complete system is now extended towards complete components with detailed behavior and high granularity. Where necessary, detailed MIL simulations are executed – possibly using FMUs. BB components are added to the models in order to configure and tune the controller to meet the performance specifications in simulation mode.	Feedback on the performance of the system is obtained by reliable and detailed simulations. This includes initial configuration and tuning of the controller.
Realization	In this phase, the final controller architecture is developed, including the final models of the controller and the plant. All necessary BB components are integrated into the realized design. All use case scenarios and the critical and major exceptions are implemented and simulated. An optional step is to execute SIL simulations, again possibly including FMUs.	Detailed feedback on the controller functionality and performance. Most requirements can be tested in simulation mode.
System Integration	On the target computing platform, execute the real-time PIL simulations (most likely excluding FMUs). Connect to the PIL simulation using the engineering programming language and the BB components in layer 3. Connect to the PIL simulation using the factory software.	Integrated controller with factory software (in layer 3). Integrated controller with developer/configuration/tuning scripts and BB components in layer 3.
	On the target hardware with a (partial) field bus, execute the real-time HIL simulations (connected with a hardware simulation unit or with a “safe” plant).	Tested controller with developer/configuration/tuning scripts and BB components in layer 3.
	Gradually connect the entire system hardware to the controller. Perform detailed hardware diagnosis using the engineering programming language.	Tested controller with developer/configuration/tuning scripts and BB components in layer 3. Tested performance of the hardware.
Verification	Run verification tests.	Most tests have already been addressed in previous PCP phase(s).
Factory Integration	Integrate system into the factory. Perform hardware and software integration.	Most of software integration has already been performed in previous PCP phase(s) with PIL simulation on the target computing platform.
Validation	Run validation tests.	Most of software validation has already been performed in previous PCP phase(s) with PIL simulation on the target computing platform.

PCP phase	I-MECH methodology activity	Benefits
Maintenance (process monitoring, failure prediction, service-repair-recalibrate-upgrade scenarios)	Developer: · Use engineering programming language for online maintenance · Collect data for offline maintenance Operation: · Use BB components in layer 3 for all maintenance activities	Adequate maintenance software for every situation.

One may notice from Table 2 that a W-model is applied for the development lifecycle. As soon as a requirement is refined towards a specification, the requirement is tested using a simulation configuration as soon as possible, providing the earliest feedback as possible. The more detailed the specifications get, the more detailed the simulations get as well.

4.3 In scope of the industrial stakeholders

The tasks of the **system engineer**, who is applying the I-MECH methodology, are:

- Make adequate choice among reference architectures (as a guide).
- Add increasing granularity to controller and plant models.
- Embed and run detailed (co-)simulations.
- Compose the controller model from a library of building block components.
- Perform early system integration and testing, based on MIL and SIL simulation configurations.

The tasks of the **control system integrator**, who is applying the I-MECH methodology, are:

- Perform pre-system integration and testing, based on PIL and HIL simulation configurations.
- Perform system integration and factory integration, using the tools and BB components in layer 3.

The tasks of the **manufacturing and maintenance end user**, who is applying the I-MECH methodology, are:

- Perform (re)configuration, using the tools and BB components in layer 3, including the OPC UA administration shell.
- Embed the motion control system into the process flow control of the factory.
- Perform process monitoring and predictive maintenance, using the tools and BB components in layer 3.

5 Verifiable I-MECH methods

Workpackage WP6 aims at integration, evaluation and validation of all the I-MECH technical developments, including the pilots set up in WP7.

It does so by identifying the highest common factors (hcf's) of all BB's on the one hand, and defining test cases for them to be executed on one or more of the pilots. Generally, these test cases are unambiguous because these hcf's have SMART functional requirements originating from the pilots.

On the other hand, the I-MECH project strives to provide a reference motion control platform where besides the functional performance also the non-functional properties, such as usability, easy reconfigurability and traceability, are crucial. Therefore, also the effectiveness of the I-MECH methodology (including the reference architecture and delivered BB's) will be measured and analyzed in WP6.

The topics to be tested in WP6 cover:

- Great usability of the platform and its tools at every complexity level
- Flexibility and composability in creating tailored motion control systems
- Coherence and compatibility between I-MECH components
- State-of-the-art performance (on the pilots in WP7)
- Model Based Design: Early development feedback, via simulations
- Automatic code and interface generation from models
- Adequate tooling, for architects, developers, integrators, testers and for operators, service engineers and plant maintenance personnel, satisfying all their needs
- Easy reconfiguration and system update (includes auto-tuning and self-commissioning)
- Clear and transparent monitoring and diagnostics of the motion control system state

END OF DOCUMENT

Acknowledgement
This project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737453. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation program and Netherlands, Czech Republic, Latvia, Spain, Greece, Portugal, Belgium, Italy, France and Ireland.