Deliverable: D2.1

Plug & Produce Automation Device Specification

Deliverable Responsible: fortiss, DE
Version: 1.4

28/06/2016

Dissemination level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (excluding the Commission Services)	

Project Information

Acronym	openMOS
Name	Open dynamic manufacturing operating system for smart plug-and-produce automation components
Theme	FOF-11-2015: Flexible production systems based on integrated tools for rapid reconfiguration of machinery and robots
Grant agreement	680735
Start date	1, October 2015
Duration	36 months

Contact Information

Company Name	fortiss - An-Institut Technische Universität München
Address	Guerickestr. 25 80805 München Germany
E-Mail	info@fortiss.org
Phone	+49 3603522 0
Fax	+49 3603522 50

Version Control

Version	Date	Change
0.1	24/05/2016	Initial creation of the document structure
0.2	08/06/2016	Merge the internal evaluation report in Enterprise Architect and Powerpoint files to this document
1.0	10/06/2016	Version to be reviewed by the consortium
1.1	17/06/2016	Harmonize the content (notational difference) by cross-referencing other documents
1.2	21/06/2016	Modification based on reviews from Loughborough
1.3	26/06/2016	Second modification based on reviews from Loughborough, Introsys, and Inotecuk
1.4	28/06/2016	Final version

List of Authors

Name	Role	Affiliation	Email
Chih-Hong Cheng	Author	fortiss, DE	cheng@fortiss.org
Kirill Dorofeev	Author	fortiss, DE	dorofeev@fortiss.org
Alois Zoitl	Author	fortiss, DE	zoitl@fortiss.org
Friedrich Durand	Contributor	Afag, CH	Friedrich.Durand@afag.com
Sven Hermann	Contributor	Asys, DE	Sven.Hermann@asys.de
Alessandro Mazzon	Contributor	Electrolux, IT	alessandro.mazzon@electrolux.it
Matteo Scagliola	Contributor	Electrolux, IT	matteo.scagliola@electrolux.com
Eugen Meister	Contributor	Elrest, DE	meister@elrest.de
Farhad Bidgol	Contributor	HSSMI, UK	farhad.bidgol@hssmi.org
Marco Peca	Contributor	HSSMI, UK	marko.peca@hssmi.org
Melanie Zimmer	Reviewer	HSSMI, UK	Melanie.zimmer@hssmi.org
Nelson Alves	Contributor, Reviewer	IntroSys, PT	nelson.alves@introsys.eu
Raquel Caldeira	Contributor, Reviewer	IntroSys, PT	raquel.caldeira@introsys.eu
Magno Guedes	Reviewer	IntroSys, PT	Magno.guedes@introsys.eu
João Dias Ferreira	Contributor	KTH, SE	jpdsf@kth.se
Mauro Onori	Contributor	KTH, SE	onori@iip.kth.se
Antonio Maffei	Contributor	KTH, SE	maffei@kth.se
Hakan Akillioglu	Contributor	KTH, SE	haaki@kth.se
Luis Ribeiro	Contributor	Linköping, SE	luis.ribeiro@liu.se
Harpal Singh	Contributor	Linköping, SE	harpal.singh@liu.se
Patrik Linder	Contributor	Linköping, SE	Patrik.linder@liu.se
Niels Lohse	Contributor, Reviewer	Loughborough, UK	N.Lohse@lboro.ac.uk
Pedro Ferreira	Contributor, Reviewer	Loughborough, UK	P.Ferreira@lboro.ac.uk
Ivo Pereira	Contributor, Reviewer	Loughborough, UK	i.pereira@lboro.ac.uk
Giuseppe Triggiani	Contributor	Masmec, IT	giuseppe.triggiani@masmec.com
Afifa Rahatulain	Contributor	SenseAir, SE	Afifa.Rahatulain@senseair.se
André Rocha	Contributor	UNINOVA, PT	andre.rocha@uninova.pt
Amit Eytan	Contributor	WePlus, IT	amit.eytan@we-plus.eu
Valerio Gentile	Contributor	WePlus, IT	valerio.gentile@we-plus.eu
Barry Auty	Contributor, Reviewer	Inotecuk, UK	barry.auty@inotecuk.com
Dale Read	Contributor, Reviewer	Inotecuk, UK	dale.read@inotecuk.com

Table of Contents

Table of Figures

1. Introduction

This document has been produced by partners in the openMOS consortium with the aim of generating a commonly agreed specification for "openMOS-enabled" automation devices supporting intelligent Plug-and-Produce (PnP) features. Such a specification is to be applied in devices for all openMOS demonstrators and to be adapted for future devices that will be integrated the eco-system facilitated by openMOS. In addition, the presented architecture is also designed with possibility to be used in applications outside the scope of openMOS.

The following is an extract from the openMOS description of work that states the objectives of the task that includes the production of this deliverable (whose description is underlined):

> ***"Task 2.1: Specifications of Smart Plug&Produce Device Adaptor Architecture [Fortiss, M1-M9]*** *Involved partners: fortiss, IntRoSys, Elrest, Xetics, Inotec, Afag, Lboro, LiU, Masmec, Asys*
>
> *This task aims to create a specification of the generic architecture for smart plug&produce devices. These smart plug&produce devices will have the capability to harmonise with the existing network of the machines/workstations when plugged in, i.e., they will communicate with the entire network of machines/workstations and when required by the system reintegrate itself without any conflicts, e.g., network address, into the architecture. More specifically, the following points will be addressed in context of smart plug&produce devices:*
> - *Device universal plug&produce architecture*
> - *Unique device addresses for identification of devices*
> - *Capability to making smart decisions according to the system requirements*
> - *Ability to communicate via network with the rest of the system about the requirement and skills*
>
> *… (text omitted) …*
>
> *At the end of this task a detailed specification of the architecture framework, including an UML-based description of the components, sequence diagrams, and other documentation will be available."*

1.1. Challenges and Proposed Methodology

We summarize difficulties that arise when considering the creation of such a specification.

- **[Maintaining technology independence]** There is a need to allow extensibility induced by existing diverse technologies. The designed architecture should guarantee that it is not technology restrictive, i.e., although in the demonstrator we may have concrete technology stack in mind, such a technology stack should not inhibit the use of other technologies.

- **[Satisfying demonstrator diversity]** Demonstrators being shown in openMOS vary in scale. On one extreme, there are Afag demonstrators whose goal is to show assembly of smart embedded components. On the other extreme, MASMEC demonstrators are designed to perform line-level integration.
- **[Understanding technology in-readiness]** During our initial investigation, we realized that existing demonstrators created in previous projects (e.g., IDEAS, OPAK) are not sufficient to be directly reused. As an example, consider an idealized scenario where a machine can act like a human to dynamically reason and create required production steps. To realize such a distributed intelligence, it is equivalent to equip an automatic reasoning engine on each device, in order to synthesize decisions during runtime (i.e., dynamically generating machine configurations from product specifications). By doing technology scouting, we found that such a demand can be unrealistic to be adopted quickly, as such technologies are only currently under active academic research (e.g., [3] [4]).

To overcome above difficulties, the openMOS consortium has organized numerous physical and online meetings, in order to generate architecture decisions that are commonly agreed. For architecture design, it is widely acknowledged that architecture decisions govern the design of architecture [5], where the detailed design of architecture is only realization / refinement of these decisions. Organizing these meetings helps the consortium to understand alternatives in architecture design and to create a coherent image over the architecture framework. In openMOS, the collected decisions range from communication (using communication API to hide individual communication technology e.g., OPC UA, DDS, or MQTT) to control (smart product can directly trigger production without a route to resource agent). Section 3 gives a detailed summary over such decisions.

1.2. Structure of the Report

After the introduction section, the rest of the report is structured as follows. To ease general understanding, Section 2 gives an overview regarding required activities to be done when realizing Plug-and-Produce concept. In Section 3, we summarize important architecture decisions that are related to intelligent Plug-and-Produce adaptor. In Section 4, we present detailed architecture design, where (1) static view is used to describe functional components (can be implemented using hardware or software) that are needed, and (2) dynamic view which describes, for each activity mentioned in Section 2, the logical behaviour over interacting components via UML sequence diagrams. In Section 5 we present concepts as cross-cutting concerns in the presented architecture such as security. Section 6 gives an initial result regarding feasibility testing (to reduce technical risks over realizability of the device adaptor) and lastly, in Section 7 we provide a summary and outline subsequent actions.

Contents of this report roughly follows the guideline proposed in the book *Software architecture in practice* (3rd Edition) [6], where one needs to document views and information beyond views (Fig. 18.4 in [6]). We refer readers to Table 1 for details. We additionally take the “concept” section mentioned in the Arc42 approach [7], to

describe some of the cross-cutting concerns to be investigated in the architecture description.

Table 1 - Mapping between contents suggested in [6] and the report

Content suggested by [6]	Content covered this report
Views	Section 4
Documentation roadmap How a view is documented	Section 1
System overview	Section 2
Mapping between views	Section 4, while we use Enterprise Architect model to ensure consistency among views
Rationale	Section 3
Directory e.g., acronym list	Appendix

2. High-level Workflow for Realizing Intelligent Plug-and-Produce

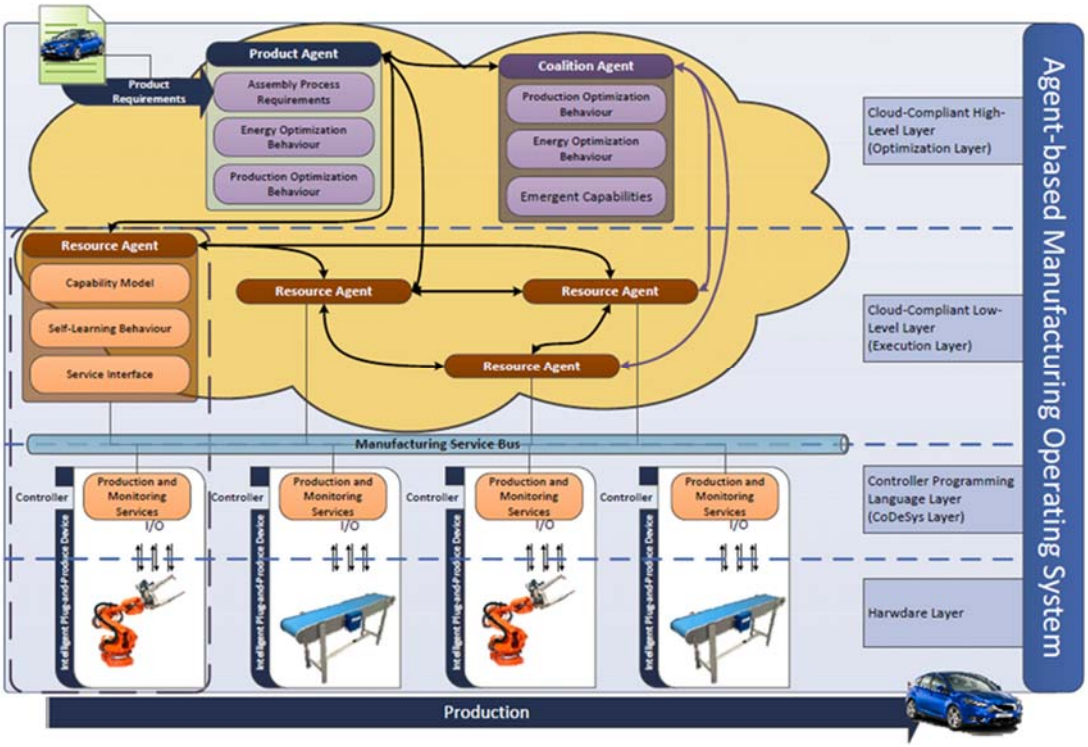

Figure 1 - An overview of agent-based manufacturing operating system (openMOS)

To understand the role of a device adaptor and its interacting components, we reuse the generic overview diagram in Figure 1 (from the openMOS proposal [1]) to explain the scope. The **device adaptor** is the interacting element between the hardware being controlled and the **Manufacturing Service Bus (MSB)** which is in charge of horizontal and vertical communications. The **resource agent** in the **manufacturing cloud** is connected below with the corresponding device adaptor and can be understood as the virtual representation of the hardware. The resource agent is

connected above with product agent and coalition agent (see Deliverable 3.1 "Open Plug and Produce Architecture Specification" for further details) for orchestration and for understanding product information. Readers may realize that for implementing a Plug-and-Produce scenario, the underlying process is almost unavoidable to have interaction with all logical subsystems that appear in openMOS.

High-level activities that are needed to realize Plug-and-Produce are detailed in the following subsections, which can be separated into three phases.

2.1. Phase 1: Discovery

The discovery phase refers to the process starting from physical insertion of devices until the stage where the connected machine/device is ready to be configured for specific production tasks[1].

- **[Activity 1: Physical insertion and IP address assignment]** After physical insertion, to trigger any subsequent communication via network, a device should be assigned with an IP address, either statically or dynamically.
- **[Activity 2: MSB being informed by the existence of device]** After activity 1, the device adaptor is able to communicate. It should start the communication to the manufacturing service bus. Some issues are identified on this stage, e.g., the mechanism how the device is being informed about the IP of MSB controller, who initiates the communication, and how registration is done. Activities conducted in this stage are highly related to discovery services being implemented in technologies such as OPC UA.
- **[Activity 3: Updating the topological information in ontology]** A device being inserted in the production system should also be informed over the change / update of the topological information. Overall, the physical insertion will change the physical topology of the plant by adding more overlapping points, and enable the understanding of connectivity (i.e., a device such as conveyor belt can know its adjacent machines). Such a scenario can be explained in Figure 2, where one intelligent storage station (right) is placed next to the processing station (left). The integration creates new knowledge over the topology, as for the processing station, it should derive the knowledge where by transferring the object to CB01 (end of the lower conveyor belt), the object will be brought to the start position of the conveyor belt (CB02) in the storage station.

[1] These activities (from IP assignment to connection to resource agent) are summarized in openMOS Deliverable D3.1 as a single action called BroadcastPresence(), where in this report, we present a refinement realizing the underlying rationale and detailed component interaction.

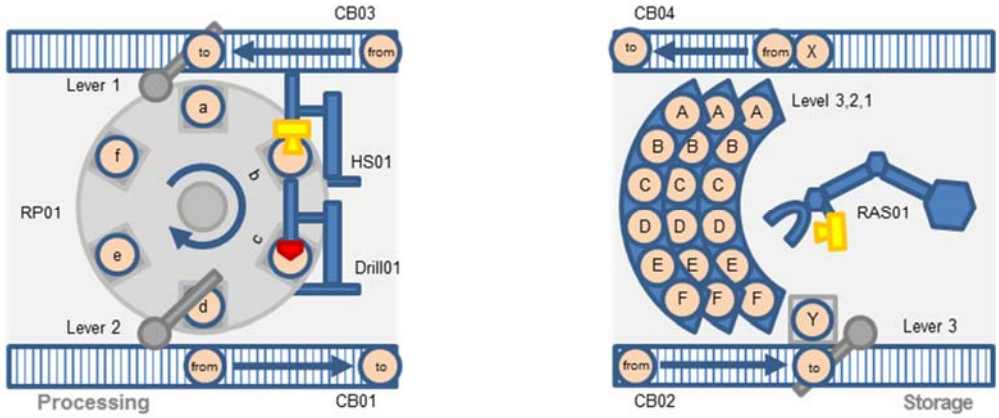

Figure 2 – The update of topological information when connecting two intelligent production stations

- **[Activity 4: Connection to the resource agent]** The goal of this step is to enable the device to be connected with its virtual part, i.e., the resource agent. Here again there are complexities such as how is the resource agent created. Reaching such a step is similar to the concept of creating an administration-shell (DE: Verwaltungsschale) in the German Industrie 4.0 concept [10].

2.2. Phase 2: Machine Configuration

The goal of this phase is for the machine / device being inserted be able to configure itself, such that it can perform product-specific production tasks. Overall, it is a process that transforms from product specification to machine configuration.

- **[Activity 5: Obtaining production configuration]** For each newly inserted machine, as in the openMOS framework the control of underlying hardware is done by skills (which provide an abstraction over concrete hardware functionalities, e.g., use "drill-a-hole (3cm, 2cm)" instead of directly setting the parameters for the motor), for the production of a specific product, it amounts to the execution of a dedicated recipe[2] and by setting corresponding parameters.

2.3. Phase 3: Production and Change-Configuration

- **[Activity 6: Triggering production tasks]** After all required skill-recipes are downloaded, the machine should start execution, via a continuous loop of workpiece manipulation and workpiece transfer.

[2] A recipe is a complex sequence (execution sequence or execution tree) of skills for underlying devices and inter-locking actions.

- **[Activity 7: Production changeover]** Perform job switch when new production tasks arrives.

3. Key Decisions Imposed on the Architecture Specifications

In terms of device adaptor and intelligent Plug-and-Produce, the openMOS consortium has agreed on the following decisions over the device adaptor architecture, to be either treated as guidelines for detailed specification or as assumptions that are guaranteed by the operating environment.

- **[Introducing the reactive control (orchestration) layer]** When observing the openMOS infrastructure diagram in Figure 1, the cloud-based control architecture seemed to imply that resource agents are responsible for all machine-level orchestration. However, as the reaction time from a resource agent issuing a command to receiving an acknowledgement from the device adaptor can be long (due to performance unpredictability for agents), the real-time feature that is often guaranteed on the line-level cannot be easily guaranteed via resource agents. Therefore, one important decision is to introduce an additional logical layer (called reactive layer) to handle real-time orchestration between machines.

  We use Figure 3 to explain the concept, where the manufacturing service bus stores the topological information. Assume that Machine 1 and Machine 3 are physically connected via Transport 2. After Machine 1 finishes producing product X, it just needs to inform that X is now ready to be transferred, and MSB will, based on the product ID and the topology, inform Transport 2 to do transport and subsequently, trigger Machine 3. The transform of control does not necessary involve resource agents.

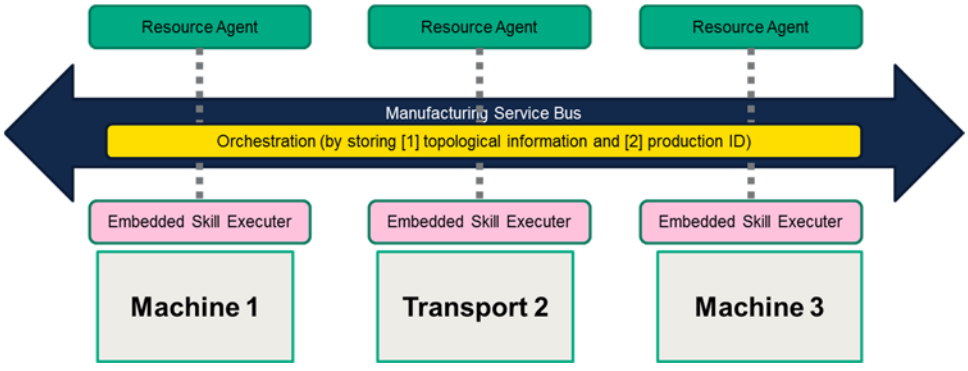

Figure 3 - Orchestration mechanism being integrated in MSB

- [Alternative architectural choices] There are many ways how such a logical concept can be implemented. Figure 4 shows an example where each unit realizes part of the choreography (distributed orchestration plan) to realize the reactive control. Figure 5 is an alternative implementation, where the

reactive control layer is relegated to a separate functional unit that is dedicated for machine-level orchestration. Both options are used in the demonstrator setup, where the first scenario is used on the machine-level (building lines out of machines), while the second scenario is used on the component-level (building machines out of components) – e.g., the use of CODESYS Application Composer can be matched to such a setup.

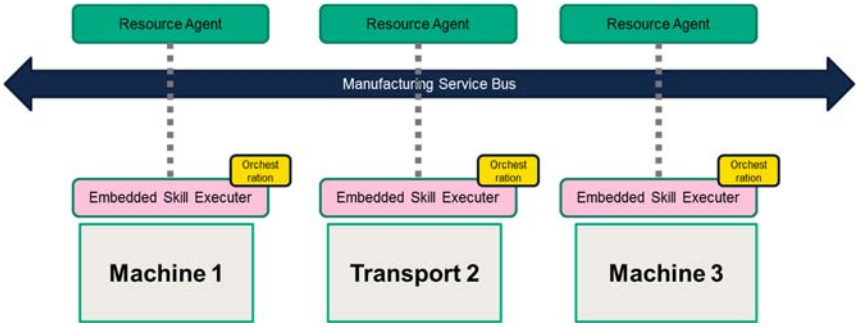

Figure 4 - Distributed orchestration

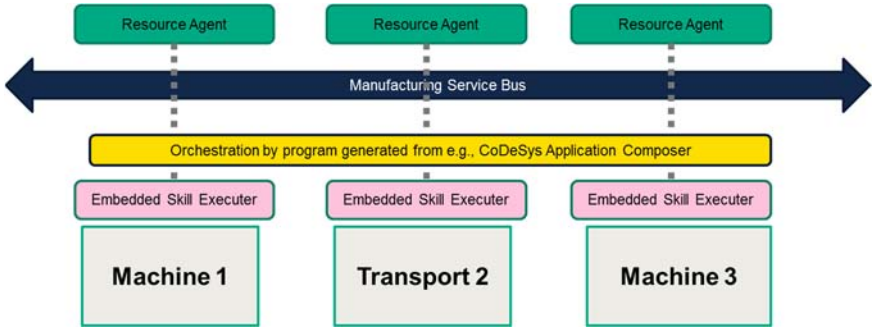

Figure 5 - Centralized orchestration in a separate functional unit

- **[Restriction over the role of resource agent]** In the architecture, following the above rationale, the duty of a resource agent will only be restricted to the deployment of skill recipe to the connected machine (in configuration mode) and the performance of required monitoring (in operation mode). The triggering of production, on the other hand, is done via a direct triggering from products to machines (i.e., device adaptor + embedded skill executor).
  Notice that the skill recipe of a machine being transferred from the resource agent is still generic (i.e., only macro-configuration), where detailed product-specific configuration (e.g., parameter or detailed-configuration) is done by operators in the loop, with the use of specific software component.
- **[Common communication API]** As the device adaptor should be prepared for integrating multiple communication protocols (e.g., MQTT, OPC UA, DDS), the ideal design pattern is to provide an abstraction over the communication API. By doing so, the high-level functional components only know that they are communicating with MSB via the communication API, and the realization of the API can be protocol specific. Such a design mechanism is commonly

available in SCADA software which needs integration of numerous communication protocols.

  - With analogous methodology, one can also introduce OS-level API that makes the implementation independent of the underlying OS. Currently, having such an OS-level API is not considered by the consortium, as the corresponding technical debt can be ignored, compared to other urgent problems that need to be resolved.

- **[All communication via MSB]** All interactions between the embedded control functions, agent platform and cloud data persistence functions should be managed by the Manufacturing Service Bus. Also, the integration between lower level Plug-and-Produce devices and any work station internal orchestrators should be handled via the MSB (when possible). This simplifies the design, as there is no need to handle communication varieties that may arise due to system scale (e.g., workstation level or line level), and the only variety comes from the different protocols realized in the network.
- **[Restricted intelligence without knowledge reasoning]** In the architecture, it is assumed that device adaptors do not need to be equipped with complex reasoning capabilities, such as recipe generation (which is commonly an action sequence or a decision tree). The reasoning done on the Plug-and-Produce adaptor is restricted to simple parameter checking (e.g., range check), in order to reject infeasibilities. Notice, however, that such a constraint does not restrict the use of typing (i.e., device categories), which might be needed when considering scenarios such as replacing a machine with a functionally equivalent one.
- **[The use of deployment configuration]** To cover a wide range of demonstrator scenarios, it is decided to have a deployment configuration. The deployment configuration is a logical concept that specifies, for every step in the Plug-and-Produce setup, whether it is done before deployment or during run-time. Such a description can be stored as a separate file (similar to makefile) to be automated or as a guideline (checklist) to assist operators when installing the new equipment.

4. Detailed Device Architecture Specification

Based on the high-level activities (in Section 2) and agreed architecture constraints, this section presents the detailed Plug-and-Produce automation device adaptor specification, where the static view (Section 4.1) presents the logical / functional decomposition, and the runtime view (Section 4.2) presents how components in static view interacts, in order to realize activities mentioned in Section 2.

All diagrams (e.g., UML sequence diagrams) presented in this section is taken from a model created using the Enterprise Architect tool[3]. Methodologically, we use static view to construct logical components, whereas all instances shown in the runtime

[3] Sparx Enterprise Architect: http://www.sparxsystems.com.au/products/ea/

view are instantiations over components in the static view. This provides an easy way to maintain consistency among multiple views in the model.

4.1. Static (Component) View

4.1.1. Tier-1 Decomposition

Figure 6 gives an overview over the logical decomposition of a Plug-and-Produce device adaptor. The dashed line surrounding multiple logical components (**Discovery, Production Configuration, Execution, Equipment Module Service, MSB communication, Utilities**) indicates the subsystem boundary, or alternatively, the parts that are integrated to the actual device. Things such as Manufacturing Service Bus Controller or Product Agent are outside of the scope, meaning that we can view the underlying mechanism as a black box.

The architecture in Figure 6 is categorized into three layers. The first layer (Discovery, Production Configuration, Execution) is for key functional applications. These applications are connected to service functions (layer 2: Equipment Module Service, MSB Communication), and service functions may call primitive utility functions, including basic functions such as extracting fields in the deployment configuration file.

For the ease of understanding, the dependencies between these components and OS-level APIs are omitted, e.g., extracting fields in the deployment configuration file, when being implemented in C, requires stdio.h for file I/O. Such dependencies to basic libraries are not listed.

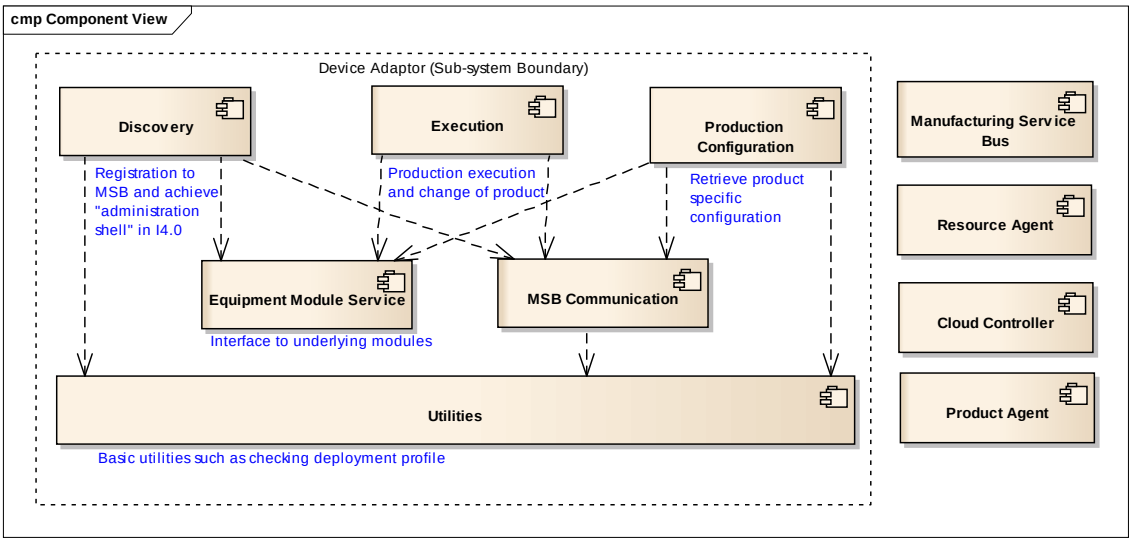

Figure 6 - First level component view for intelligent PnP device

We give an overview of the features of each component.

- Component "**Discovery"** contains member functions to realize activities from finding MSB controller to establishing the connection to the resource agent.
- Component "**Production Configuration**" contains member functions to realize the installation of product-specific recipes and to perform suitable negotiation.
- Component "**Execution**" contains member functions to trigger production and to switch production together with the features to send required monitoring information to the resource agent.
- Component "**Equipment Module Service**" is in charge of all interactions with the underlying elements being controlled. It also provides a unified wrapper to encapsulate machine-specific actions to high-level skills in order to be coherent with the description specified in the recipe.
- Component "**MSB Communication**" provides a wrapper for *Discovery*, *Production Configuration*, and *Execution* to communicate with the agent or other machines, without handling low-level communication details specific to the technology.
- Component "**Utilities**" provides basic utilities that can be used by other components, such as retrieving the info from the deployment configuration and OS-level API.

4.1.2. Tier-2 Decomposition

Currently, only "Execution" and "Equipment Module Service" are further refined to sub-components.

For "Execution", as shown in Figure 7, it is further refined into "Interpret Recipe" and "Recipe Management", where sub-component "Interpret Recipe" is similar to a scripting engine to, based on the received product ID, trigger and execute the corresponding skill recipe with concrete production parameters associated with the product. Sub-component "Recipe Management" is used as a placeholder to store all recipes and parameters, and to examine if stored parameters are valid (e.g., parameters do not invoke security flaws such as injection attack).

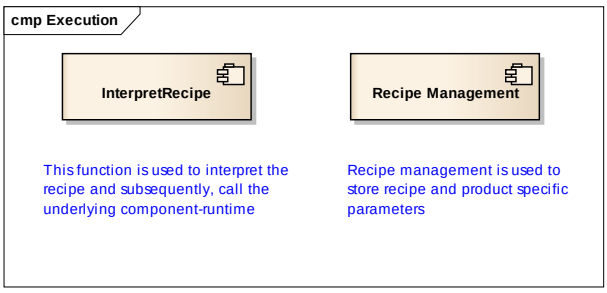

Figure 7 - Tier-2 Decomposition of "Execution"

For “Equipment Module Service”, as shown in Figure 8, it is further refined into “Equipment Module Runtime” and “Equipment Module Info Retrieval”, where “Runtime” provides a centralized interface to execute one skill in the recipe, and “Info Retrieval” is used to extract required information for configuration and KPI / log extraction, to be sent to MSB or to the resource agent.[4]

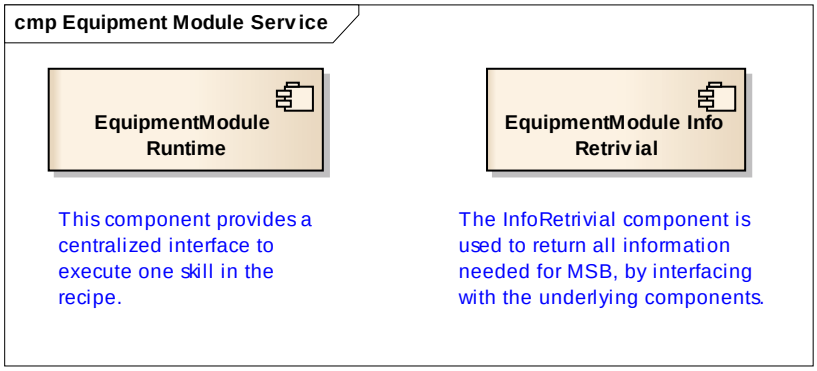

Figure 8 - Tier-2 Decomposition of "Equipment Module Service"

4.1.3. Application Programming Interfaces (APIs)

In this section, we provide a brief summary over important APIs that are used in above mentioned components. It is only used as a reference for completeness considerations; readers can omit this subsection and directly understand how these APIs are used by a walk-through over the run-time scenarios described in Section 4.2.

Component / Sub-Component	**Function description**
MSB Communication	AsyncSend(String destination, String info) • Usage: This function performs an abstraction over protocol-specific transmission, in order to communicate with other entities in the network. This allows vendors to implement protocol independent logic. AsyncReceive(String port, String info) • Usage: This function allows retrieving the item in the agreed port, where a port can be semantically embed device description and message type.

[4] Notice that the communication with the underlying component can either be networked or no-networked (similar to direct control over I/O variables). For networked communication to the underlying component, it is also suggested to be linked via the manufacturing service bus, implying that there is also package dependency between “Equipment Module Service” and “MSB Communication”. In this report, such a scenario is not considered for the sake of clarity, and to also only consider common cases.

<table>
<tr><td colspan="2"></td><td>SyncMsgSend(String destination, String info)
• Usage: A wrapper over AsyncSend(), AsyncReceive(), to provide callers a locking mechanism.

RegisterToMSB(String info)
• Usage: Perform registration to MSB controller, such that the MSB controller is aware of the device

RetrieveConfig(String field)
• Usage: A wrapper for the configurator to access configuration passed from the MSB, without handling detailed receive information.

Protocol specific (here we omit details, as the function name is designed to have clear meaning over the feature):
• OPC-UA
o GetEndPoint()
o MsgSendOPCUA(): Depending on concrete message type being sent, and depending on the concrete programming API, it may further be refined to other APIs that are specific to the concrete technology stack.
• MQTT
o MsgSendMQTT()
• DDS
o MsgSendDDS()</td></tr>
<tr><td colspan="2">Utilities</td><td>GetConfiguration(String item)
• Usage: This function allows retrieving the corresponding field from the tabular (or XML) description, with keyword “item”.
• Return: The configuration specified in the deployment configuration. NULL if such a field is not specified.</td></tr>
<tr><td colspan="2">Discovery</td><td>LoadMSBServerFromFile()
• Usage: Retrieve MSB controller information, if the information is by default stored internally in the device adaptor.</td></tr>
<tr><td colspan="2">Production Configuration</td><td>ConfigProduction()
• Usage: A centralized functional entry for configuring machines via connecting to the resource agent</td></tr>
<tr><td>Execution</td><td>Intercept Recipe</td><td>GetProductInformation()
• Usage: Retrieve the associated product ID (product type), for the product to be produced.

FindCorresondingParameter(String skillName, String[] pars)
• Usage: For a given skill, and all parameters associated with the recipe of a product, find corresponding skill-specific parameters</td></tr>
</table>

	Recipe Management	SetMachineRecipe(String recipeInfo) • Usage: Store the recipe (recipeInfo) in the system, such that it can later be reused. ConfigParameter() • Usage: Centralized access point for triggering the process of configuring product-specific parameters. Notice that this function should always be executed after the recipe is installed (i.e., after function SetMachineRecipe()) RetrieveParameter(String productID) • Usage: During production time, retrieve product-specific parameters.
Equipment Module Service	Equipment Module Runtime	ExecuteSkill(String skillName, String[] parameters) • Usage: Execute the skill with associated parameters.
	Equipment Module Info Retrieval	GetDataModel() • Usage: This function is used to retrieve the underlying data model of the connected component.

Notice that in the openMOS Deliverable D3.1, there are actions that directly perform a cross-talk between MSB controller and the device adaptor. There is no conflict, however, as cross-talks such as “DeployRecipe()” from MSB to device adaptor or “DeploymentComplete()” from device adaptor to MSB can always be realized using underlying message mechanism AsyncRecv(“DEPLOY_RECIPE”), and AsyncSend(“DEPLOYMENT_COMPLETE”). Precisely, the function name can be moved into parameters of a message being sent.

4.2. Runtime View

In this section, we detail the interaction of above mentioned components, for each activity being mentioned.

4.2.1. Retrieving the information of Manufacturing Service Bus (MSB)

[Background] The starting phase of Plug-and-Produce is the physical insertion, which brings power and network connectivity. Once when the device is assigned with an IP, it is still needed to retrieve information concerning “how can I communicate to the MSB?”. In result, this amounts to the process of informing a newly inserted device the IP or the URI for the Manufacturing Service Bus.

[Illustration] The corresponding sequence diagram is shown in Figure 9.

[Mechanism]

1. The Discovery component starts to understand where to find such a piece of information, by invoking the utility function GetConfiguration() in the Utilities component, allowing to retrieve information from the deployment configuration file. By feeding the function with keyword "IS_MSB_IP_STORED", the result can be used to infer whether such information is already stored in the device adaptor.
    - **[Notice]** As checking if any piece of information is pre-stored in the configuration file is done in many cases, after this scenario the checking via function call GetConfiguration() might be omitted. By doing so, it increases the clarity of reading.
2. When the IP is stored statically on the device adaptor (isStatic = true), then it is only required to load MSB controller IP (or URI) from the deployment description. The Discovery component then invokes the function LoadMSBServerFromFile(), which internally again calls GetConfiguration().
3. Otherwise (isStatic = false), there is a need to obtain such an information via sending messages to the network. The device adaptor then invokes the function SyncMsgSend("BROADCAST", "GET_MSB_IP"), which is a function for synchronous message sending. SyncMsgSend() is internally translated to AsyncMsgSend() on the low-level using appropriate locking mechanisms. The message means that it is broadcasted to the complete network, in order to retrieve the IP of the manufacturing service bus controller.[5]
4. For the component "MSB Communication", it reacts differently based on the protocol type (it also calls the Utilities to check if such information pre-exists). If the communication medium is OPC UA, then the initial point for accessing MSB Controller is equivalent to the IP of the discovery server.
    - Such information should be responded by the AutoDiscovery component.
    - If the device adaptor does not have any knowledge whether MSB is operated using OPC UA, MQTT, or DDS (i.e., msbType = UNKNOWN), it tries all possibilities in sequence (the mechanism is omitted in the figure, to reduce complexity of the diagram).

[A note on realizing the AutoDiscovery mechanism] The communication to the AutoDiscovery unit for the Manufacturing Service Bus can again be protocol specific. E.g., when the MSB uses zeroconf service based on Multicast DNS Service Discovery, in the actual implementation, the device adaptor needs to trigger a zeroconf listener to get the required info. On the contrary, when zeroconf is based on WS-Discovery (Web-Service Dynamic Discovery), the protocol uses IP multicast address 239.255.255.250. Therefore, the actual processing in the component "MSB Communication" should have two dimensions for case splits: (1) the protocol of MSB (OPC UA, DDS, MQTT) and (2) the protocol of zeroconf (DNS-based or WS). Here such a refinement based on actual implementation of zeroconf is not explicitly stated in the diagram.

[5] If an MSB controller continuously broadcasts its IP, then such a step can be avoided. However, a continuous broadcasting from MSB controller can be a waste of bandwidth.

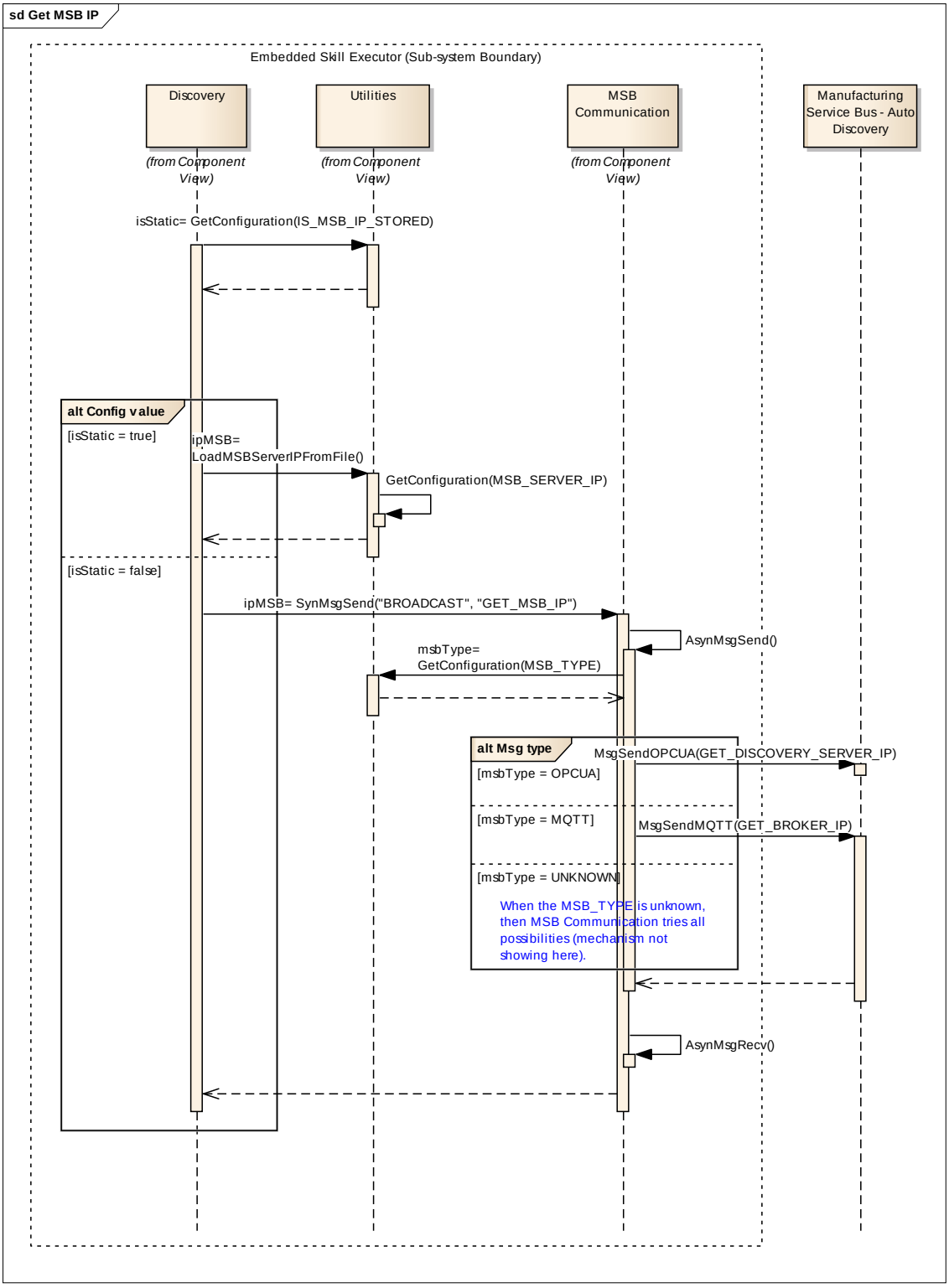

Figure 9 - Retrieving the info regarding MSB Controller

4.2.2. Registration to the Manufacturing Service Bus

[Background] When the IP for the MSB controller is known, the subsequent step is to register the device to the MSB.

[Illustration] The corresponding sequence diagram is shown in Figure 10.

[Mechanism]

1. After checking the device configuration file, when a registration is needed during run-time, the Discovery component prepares required information and triggers function RegisterToMSB().
    - Here the required information for sending includes the following:
        - Name
        - Type
        - The list of service points being supported for connection
        - Quality attributes as specialized identifiers
2. Internally, the component MSB Communication proceeds differently based on the underlying communication protocol.
    - For OPC UA, the registration amounts to the registration to the discovery server, which is handled by two actions:
        - The first action “endpoint := MsgSendOPCUA(IpMSB, GET_END_POINT)” is to trigger a request to get all end points. It then can select one end-point that has (1) a compatible transport protocol and (2) a security profile that is compatible.
        - MsgSendOPCUA(endpoint, REGISTER_SERVER, info). With the decision of one particular end-point, it can now send the Register Service request message.
    - For MQTT, the publish-subscribe mechanism requires that the first connection from the client (i.e., device adaptor) to the MQTT broker (the implementation of MSB) to be a CONNECT packet, while the client will receive a CONNACK (connection acknowledgement) package from the MQTT broker. Details over the connection mechanism for MQTT can be found in the standard [2].

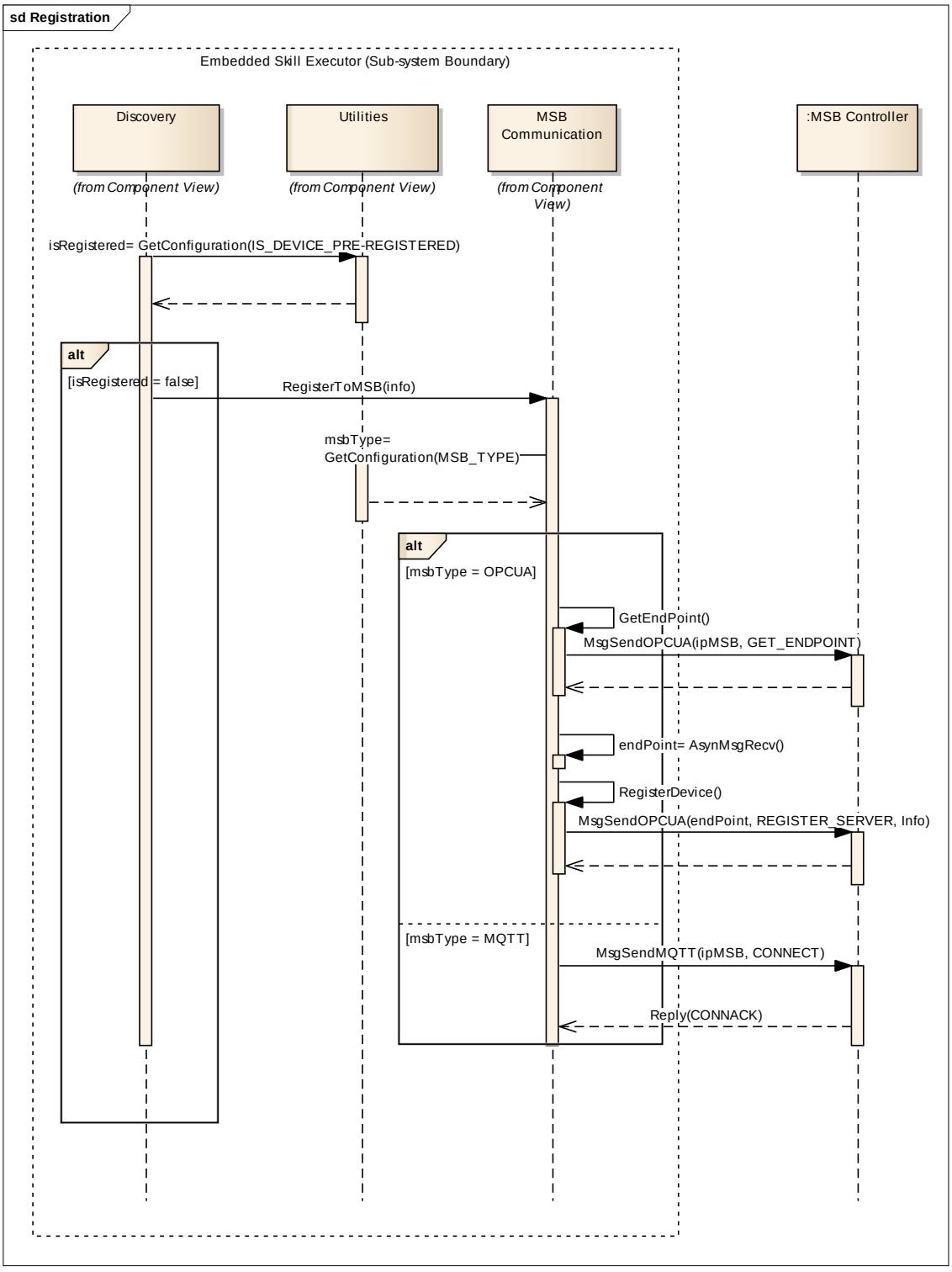

Figure 10 - Device registration to MSB

4.2.3. Update the ontological and topological information

[Background] Whenever a component is inserted, the topological information should be updated such that existing components that are connected with the new component know who it is being connected. The update of topological information also implies that the ontological structure (device tree) maintained in the Manufacturing Service Bus is updated.

[Mechanism] In previous meetings, it was decided by the openMOS consortium that for the first release, the update of ontological information is handled by the MSB and in the cloud. This implies the scenario in Figure 3, where each inserted device only needs to be informed over the product identifier that needs to be processed or transferred. The duty of a workstation, however, is to maintain the topological structure of its controlled devices.

4.2.4. Resource agent creation and connection

[Background] The new device, after being inserted, should be connected to a resource agent which is the virtual representation of itself. Here, the presented scenario is for the case where the corresponding resource agent needs to be created and connected, and the action is initiated after the device is being introduced.

[Illustration] The corresponding sequence diagram is shown in Figure 11.

[Mechanism]

1. This diagram considers steps where the creation and connection of agents are all done during run-time, with initiation from device adaptor.
2. First, examine if the creation of resource agent is supported by the MSB via the internal function isAgentHelpedByMSB(), which triggers a call to MSB via SynMsgSend(MSB, HELP_AGENT_CREATE). After that, component “MSB communication” then triggers corresponding message sending, depending on the protocol.

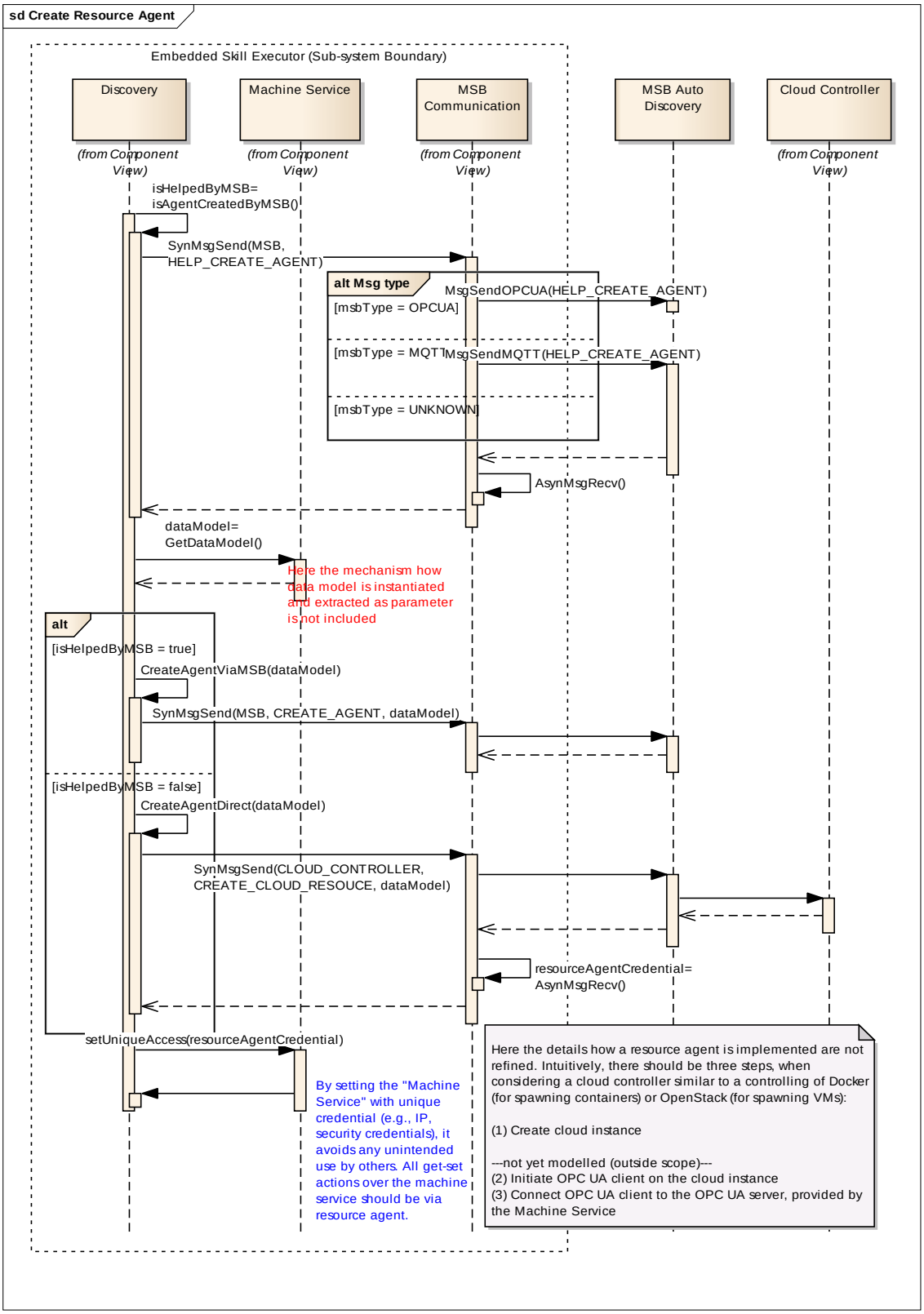

Figure 11 - Connect to resource agent (with restricted access)

3. Depending on the response, the Discovery component always prepares the data required for creating such an agent (by querying the Equipment Module Service to understand the nested product information), and proceeds with one of the following tasks.
    a. If MSB can help creating (i.e., MSB can help managing all required information such as credentials and digital signatures – the actual creation is still done on the cloud side), then prepare all key information required to the MSB, such that it generates an agent.
    b. Otherwise (in rare cases), it is the duty of the device adaptor to create such an agent. In this case, the device adaptor communicates with the cloud controller (again via MSB) to create an agent. Notice that details how an agent is created after cloud-controller receives the message is not further refined; these technology details are included in the part of design of manufacturing cloud.
4. Lastly, the Discovery unit configures the "Equipment Module Service" component, such that the update of recipe is uniquely available by the resource agent (via a unique IP, credential, or digital signature). Such a restriction also prevents any unintended access from other internal devices.

4.2.5. Configuring the underlying machine

[Background] After the connection to the resource agent is established, the device still needs proper configuration to receive recipes and to receive product-specific parameters.[6]

[Illustration] The corresponding sequence diagram is shown in Figure 12.

[Mechanism]

1. The "Production Configuration" component starts the work by informing the MSB and the Resource Agent (via message "READY_RETRIEVE_RECIPE"), and waits until a set of recipes specialized for the machine. Each recipe is either a skill or a sequence of skills, being parameterized. E.g., the below recipe (Recipe X) is parameterized based on three parameters (a, b, c).

Recipe X
Probe(a) Rotate(b) Drill(c)

[6] Compared to the deliverable D3.1, here we present a more refined sequence by considering different product variants can be produced by the same recipe, while using different parameters.

2. After receiving recipes, then the device adaptor sends a message (CONFIRM_RECIPE_DEPLOYMENT) saying that it is able to retrieve product specific parameters. Parameters are provided via the Service Engineer via the Resource Agent, as previously we identify the resource agent to be the single point of access. Alternatively, it can be directly send to the device adaptor, but it may impose additional security risks. E.g., for a particular product Y, it may require to use recipe X with parameters (a=0cm, b=30 degree, c=1cm).
3. After product specific parameters are given, the intelligent machine is ready to produce products, either in trial run or in batch production. It notifies the MSB via the message “CONFIRM_PARAMETER_DEPLOYMENT”.
4. If trial production is needed, it can be realized by first switching to the production mode (cf. Section 4.2.7), execute a production run (cf. Section 4.2.6), examine the validity of trial production, and subsequently, send a notification to the MSB via “CONFIRM_PARAMETER_DEPLOYMENT”.

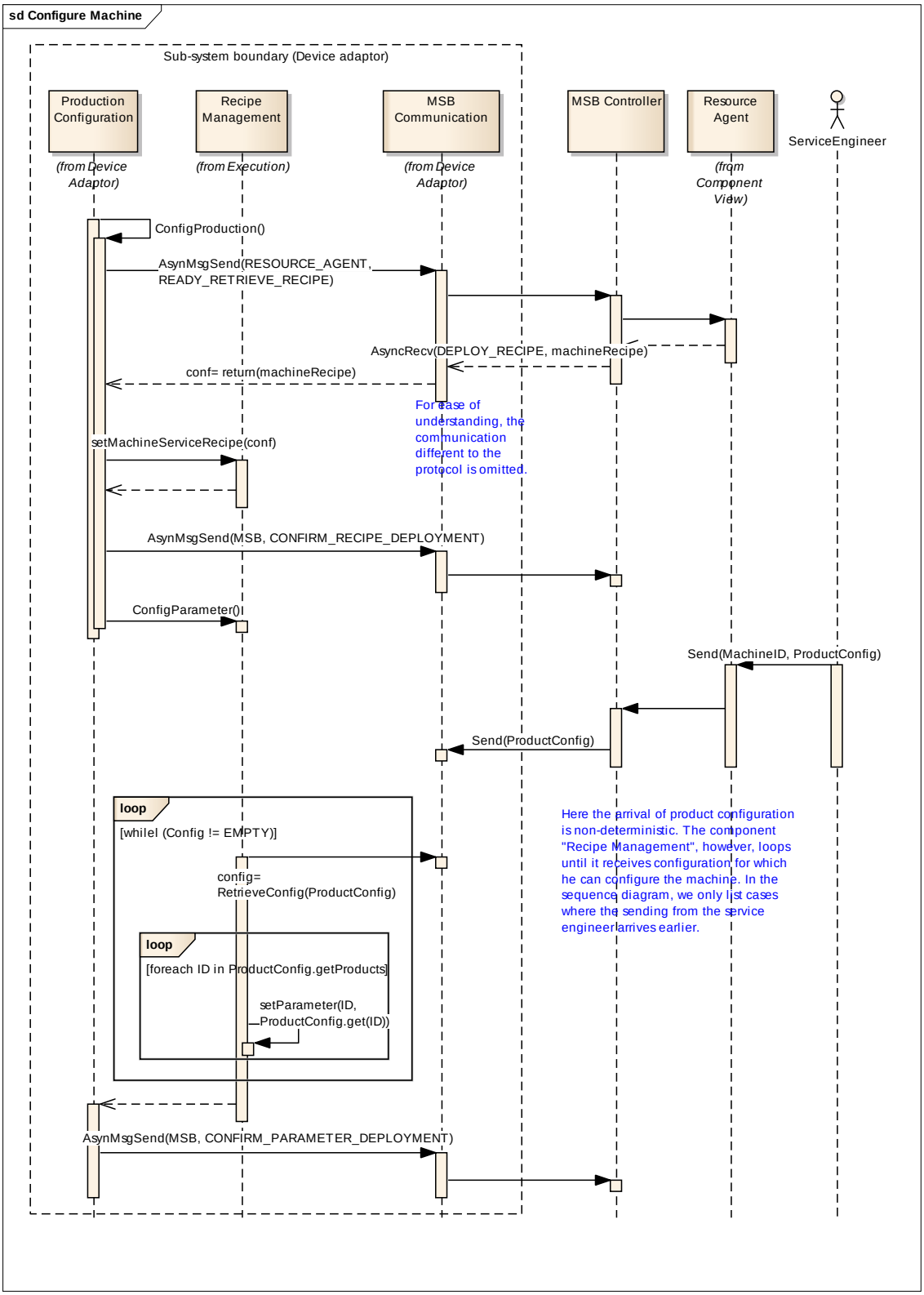

Figure 12 - Storing recipes and product-specific parameters

4.2.6. Production and forwarding control to other components

[Background] After configuration, the intelligent production unit is able to receive concrete production tasks. This diagram details the underlying mechanism how information is transferred within the unit.

[Illustration] The corresponding sequence diagram is shown in Figure 13.

[Mechanism]

1. The component "InterpretRecipe" (a subcomponent of Execution) is running as a thread such that it can take production instructions continuously[7]. It first loops until it detects a production need, which is updated to the variable "productID" via function call GetProductionInstruction().
2. When productID is not NULL, then productID stores the type of product currently being located on the machine. "Interpret Recipe" then calls "Recipe Management" to retrieve the corresponding recipe and product parameters, via function RetrieveRecipe() and RetrieveParameter().
3. With corresponding parameters and skill-recipe, component InterpretRecipe then iterates through each skill enclosed in the recipe, find the required parameters, and send to the underlying functional interface "EquipmentModuleRuntime" to execute the skill.

Recipe X	Product Y
Probe(a) Rotate(b) Drill(c)	Use X, with a=0cm, b=30 degree, c=1cm

   For the above mentioned example, InterpretRecipe first executes Probe(0cm) to EquipmentModuleRuntime_1 which is in charge of executing Probe(), and subsequently Rotate(30 degree) and Drill(1cm) to other EquipmentModuleRuntime that are in charge of these skills.
4. When all skills in the recipe are executed, InterpretRecipe resets productID to NULL (to avoid duplicate execution), informs ResourceAgent concerning all production KPIs during this production task, and finally, informs MSB about the switch of control.
5. In the sequence diagram, the MSB then, based on the topological information it knows, informs the corresponding transport unit for workpiece transfer, such that further processing over the workpiece on other machines.

[7] Therefore, in the UML sequence diagram, InterpretRecipe should be again surrounded by an infinite loop, to describe that it can take production instructions forever. Here for simplicity (to avoid showing nested loops) it is not drawn.

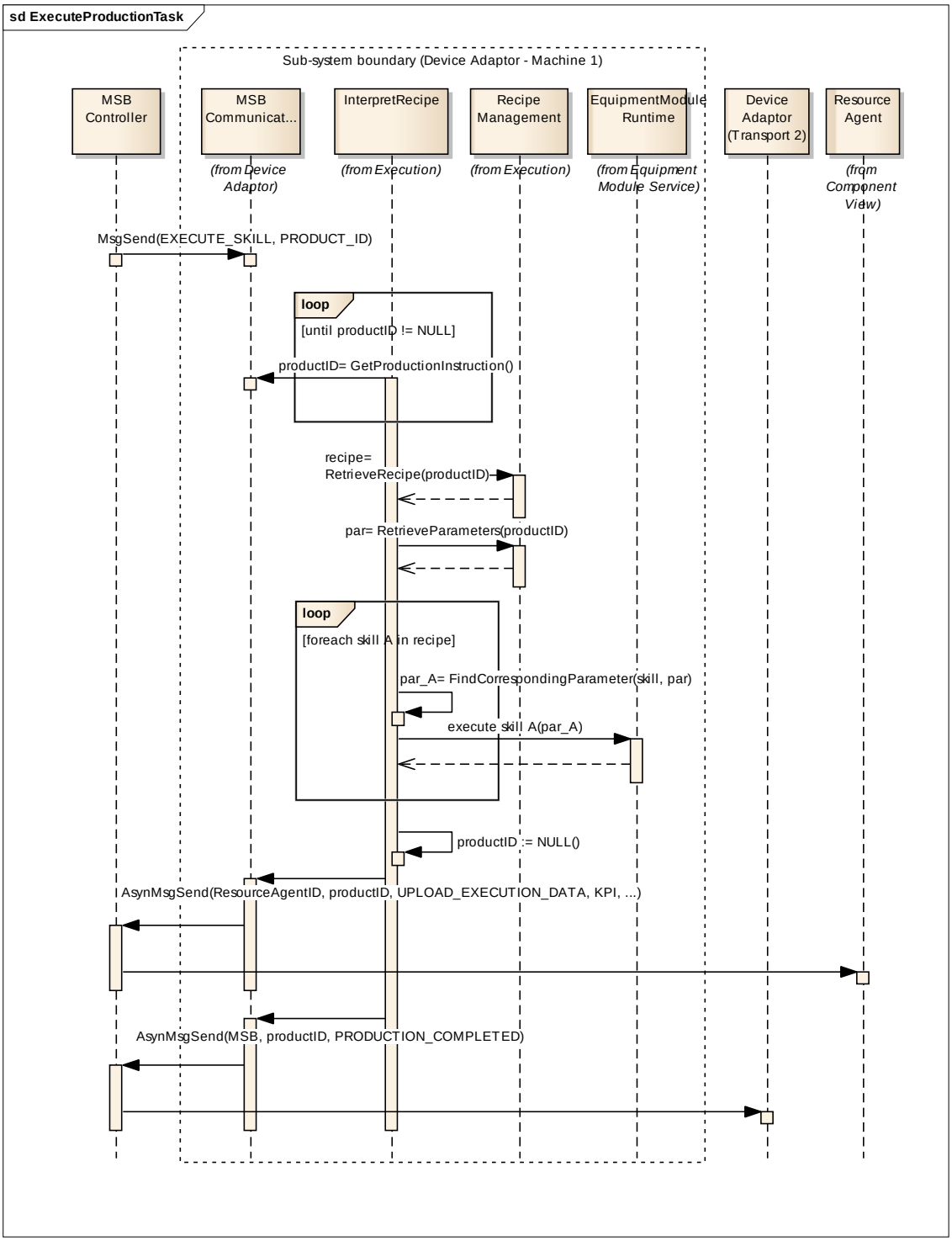

Figure 13 - Execute production instruction

4.2.7. Change of production

[Background] For an intelligent production unit, the device adaptor is already designed to process multiple products. This implies that the change of production from A to B, where recipes and parameters of A and B are already stored, does not require any reconfiguration. The only change that requires new configuration is the introduction of new products, which may require loading of new product and recipe configurations.

For this purpose, a state-machine diagram capturing various modes is presented.

[Illustration] The corresponding state-machine diagram is shown in Figure 14.

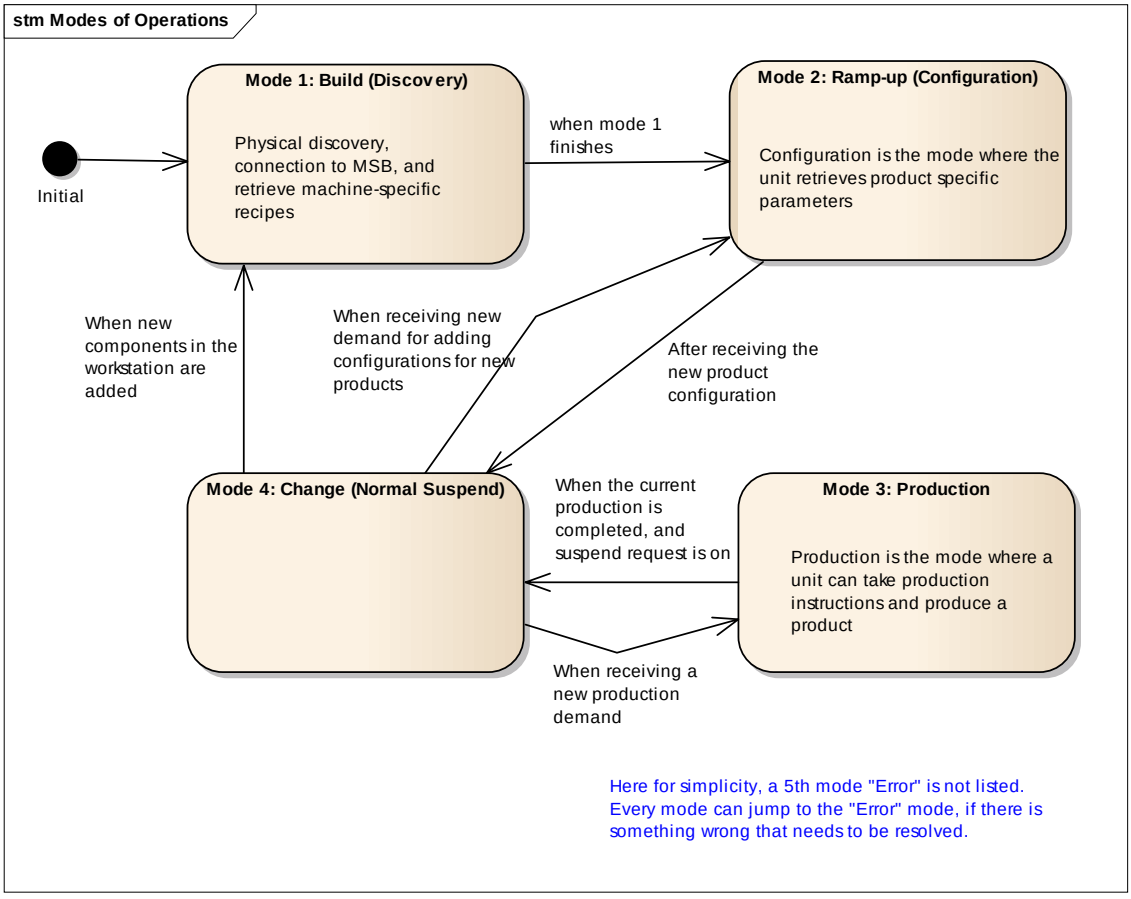

Figure 14 - Operating modes in the device adaptor

[Mechanism] Mode 1, 2 and 3 matches the previously mentioned three phases.

- Mode 1 covers activities described from Section 4.2.1 to the first part of Section 4.2.5 (until recipe is downloaded and stored).
- Mode 2 covers activities described in second part of Section 4.2.5 (download product-specific parameters).
- Mode 3 covers activities described in Section 4.2.6.

Additionally, we introduce an additional mode “Normal Suspend”, to be an intermediate mode between configuration (Mode 2) and execution (Mode 3). By doing so, whenever there is a request to update the system during run-time, the machine will jump to mode “Normal Suspend” **only when the current production task is completed**. This ensures that every production is an atomic action, under the normal operation. It can also be used as a change-up scenario, such that one can install new equipment.

Apart from above mentioned four modes, the last mode “Error” is not listed, for the ease of clarity. Essentially, the “Error” mode is used to involve human in the loop under erroneous scenarios. When errors are resolved, the operator can decide to move back to any of the operation modes.

4.2.8. Request module details

[Background] For administration purposes, it is sometimes required to perform a query from MSB or resource agent over details. This is covered by the deliverable D3.1 as “Request_Module_Details()” or “Request_Workstation_Information()”.

[Mechanism] Such mechanism is easily realized by a look-up table regarding where to fetch the data. If OPC UA is used, the search of information can also be found by a querying over the underlying information model. Due to its simplicity, the underlying sequence diagram is omitted.

4.3. Deployment View

In actual deployment, the Device Adaptor specified in this document requires is either installed on an embedded operating system with corresponding network protocol stack, or it is directly compiled with the network stack and installed on the machine. Apart from Figure 1, in Figure 15 we show a deployment view focusing on internals of the intelligent production unit.

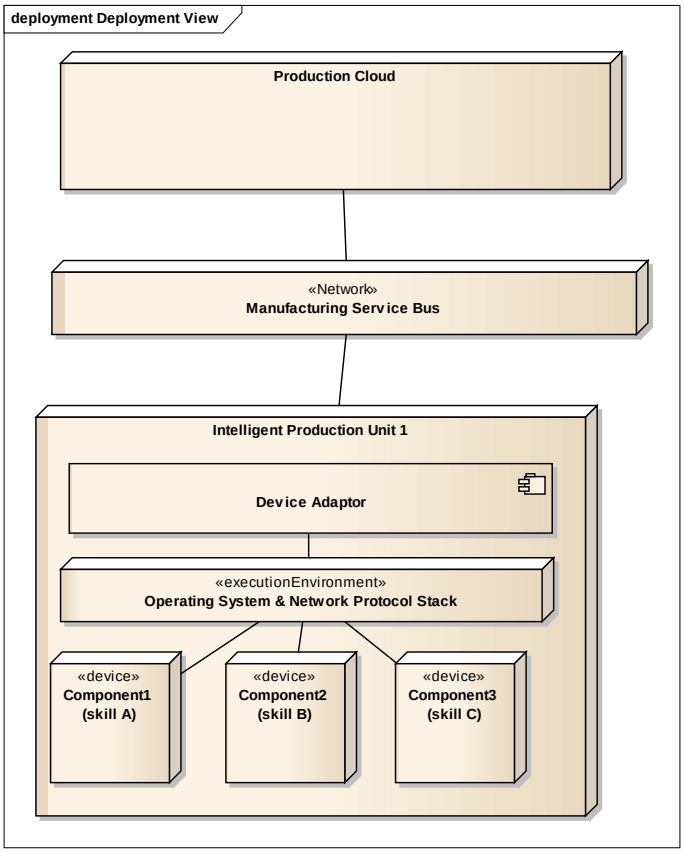

Figure 15 - Deployment view

5. Concepts

This section briefly summarizes frequently-discussed cross-cutting concerns that are exposed on the designed architecture.

5.1. Flow of Control

The flow of control is governed by the state-machine diagram in Figure 14 - Operating modes in the device adaptor.

5.2. Recurring or Generic Structure and Patterns

The overall system (openMOS) employs a client-server architecture, while in the design of device adaptor, layers are used to regulate dependencies among components.

5.3. Exception and Error Handling

In Figure 14, all operating modes are designed to have jumps to the error mode. Such a design allows a centralized component to handle all machine-specific errors

or exception during the process. Having such an error mode is also suggested by approaches such as the PackML standard[8].

5.4. Logging and Tracing

The logging and tracing (e.g., KPI info) are sent to the resource agent (see Figure 13 for info), as individual machines can be of limited resource to store all logs and traces that are generated during production.

5.5. Security

Security is commonly considered as a system-level property and it is not the intention of this report to cover the complete security design of the whole system. However, from the device adaptor perspective, the basic security goal contains two parts: (1) Parameters and recipes that are stored in the machine should not be accessed without authorization, as they are considered to be company assets. (2) It is not allowed to have unauthorized triggering of machines. If a machine has entered a safe mode where an engineer enters and performs maintenance jobs, malicious triggering of machines can be a security-induced safety hazard.

In the current setup the overall system is still considered to be operated without external access. Towards futuristic changes, we still perform a security analysis based on the threat modelling framework suggested by Microsoft Security Development Lifecycle (SDL) [8], which is also used by National Institute of Standards and Technology (NIST) as the starting point for threat modelling [9]. The proposed mitigation plans for counter-measuring threats should be implemented in the futuristic products where the overall system is exposed to the world-wide internet.

From the perspective of device adaptor, the attack surface[9] is the network interface to the MSB, as well as the access via the underlying OS (e.g., the device adaptor may be running under embedded OS such as Linux). Based on the following two assumptions, Table 2 lists all threats that need to be considered[10].

Table 2 - Security threats for the device adaptor and mitigation plans

Threat description	Strategies which address each threat
Spoofing of MSB or agent	Mitigation: Device adaptor should be equipped with mechanisms to understand digital signature Transfer: The MSB should also support digital signature (this also requires that MSB needs to, prior to deployment, know the digital signature of the device adaptor)

[8] Packaging Machine Language (PackML): http://omac.org/workgroups/packaging-workgroup/
[9] Source from wikipedia (http://en.wikipedia.org/wiki/Attack_surface): The attack surface of a software environment is the sum of the different points (the "attack vectors") where an unauthorized user (the "attacker") can try to enter data to or extract data from an environment.
[10] SDL has identified many potential scenarios that can induce threats. Here we omit those that are less important (i.e., “Accept the risk” or “Transfer the risk to 3rd party - MSB”).

File spoofing or tempering – when the system is plugged-out from system, malicious user can download or change the recipe or parameter	Mitigation: When network is disabled, disable untrusted local modification of the recipe or parameter, preferably via access control list (ACL).
An attacker can try one credential after another	Mitigation: Enable maximum trial of password
Attacker can reuse password	Mitigation: Password management (e.g., periodic change of password to reduce the risk)
Device adaptor is shipped with a default admin password	Mitigation: Enforce that by shipping, the password is always generated randomly.
Racing to create a file	Mitigation: Guarantee that the “MSB Communication” does not have write access to components in “Execution” and “Production Configuration". This is already guaranteed by our static design.
Repudiation: dispute due to change of logs	Transfer: The record of logging stored in the resource agent should only be modifiable by 3rd party, i.e., independent to machine builder (who creates a machine) and product builder (who designs a recipe).
Network information disclosure	Mitigation: Device adaptor should be equipped with HTTPS communicating capability. For other proprietary protocols corresponding security hardening should be done.
Resource flooding - network	Mitigation: Close all other ports to disable un-specified connection (to avoid TCP flooding) Transfer: MSB should detect and resolve the network flooding
Injection attack (malicious parameters can trigger additional skills in the skill recipe)	Mitigation: all parameters should be examined for the validity (e.g., integer value)

6. Validating the Architecture Design and Some Follow-up Development Plans

6.1. Architecture Validation

The described architecture is preliminarily validated using a simple demonstrator using mDNS mechanism for device auto discovery, and to test multi-directional transfer between the conceptual MSB and the device adaptor. The setup (shown in Figure 16) is the following:

- The virtual machine represents the MSB and it runs a zeroconf service, implementing the Multicast DNS Service discovery, announcing the devices

in the network about its IP address and protocol type to be used to connect to MSB controller,

- The PC represents the device itself; it also runs a zeroconf service that allows to listen to the network and get the information about the MSB controller.

When the device is added to the network, it is checked whether the MSB type and address are preconfigured and if that is the case the device uses that configuration to start the communication with the MSB Controller. Otherwise, if there is no MSB controller address preconfigured and stored in the configuration file, the discovery process is triggered. The mechanism how to deal with different MSB protocols is described in Section 4.2.1 and is implemented in demonstrator. As soon as the MSB controller IP address and type are discovered, the registration to MSB process is executed.

During the registration phase the internal mechanism behaves differently, based on the MSB protocol (currently in demonstrator there are MQTT and OPC UA implemented). In the MQTT case, the device is sending “CONNECT” message and waits for “CONNACK” (connection acknowledged) message from the MSB controller. In OPC UA case the device executes GetEndpoints() method and gets Endpoint description as a response (OPC UA server acting as service discovery). Then the device can execute RegisterServer() methods to register itself to MSB. Once the server is registered, the MSB (OPC UA client part) finds an endpoint that meets its requirements for compatible transport, security policy and security mode and starts the connection process by issuing Open Secure Channel request.

After the discovery and registration parts are successfully finished, the device and MSB can communicate so that the device can publish its skills and subscribe for parameters and recipes. That allows MSB to store the recipes and configure product-specific parameters and to get acknowledgements whenever the device has finished the defined actions.

[Note] Resource-constrained embedded systems need to have an ability to announce themselves on the Multicast Subnet with a basic Multicast Extension. This requires a small subset of an mDNS Responder that announces the device and responds to mDNS probes. The device does not need to provide the full functionality of zeroconf services (e.g., caching and address resolution), used in the demonstrator.

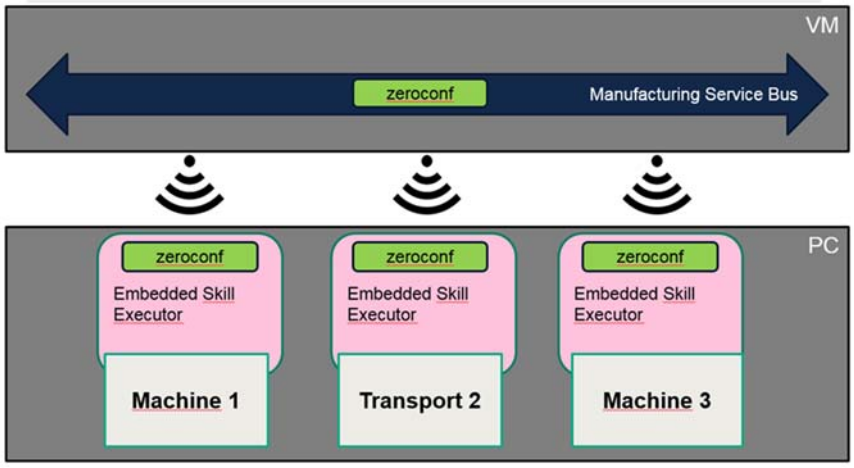

Figure 16 – Architecture validation - autodiscovery and bidirectional transfer

6.2. Follow-up Development Plans as Realization Plans

Based on the proposed architecture, openMOS partners have created subsequent development plans with more refined protocol specific details.

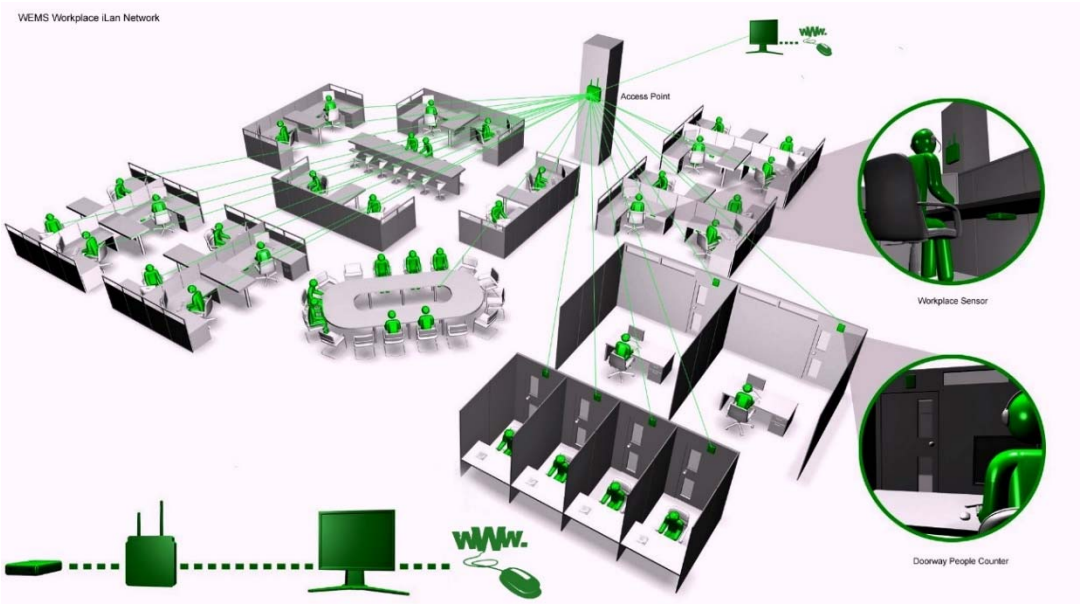

Figure 17 - Example of numerous RF transmitters communicating to single RF receiver (access point)

For example, Inotec will provide the RF plug-and-produce sensors for the openMOS project based on the architecture design. The device adaptor will be integrated to a RF receiver (access point, see Figure 17 for details), such that it can communicate with hundreds of sensor RF transmitters beneath, and communicate with the MSB. In this setup, the access point uses on the ISM RF band (850-950MhZ) to offer

connection to connected “device equipment” (i.e., without MSB), while the physical connection to the MSB (via MQTT protocol) will be based on Wi-Fi, Ethernet or GPRS.

7. Concluding Remarks

In this report we presented the architecture description of the device adaptor to support Plug-and-Produce. We first summarized important architecture decisions and the underlying rationale. We then provided detailed description over component diagrams (for logical view) and run-time sequence diagrams (for run-time view). The validity of the presented architecture description is evaluated via a prototype implementation. This avoids commonly seen pitfalls in architecture design where the gap between design and technological stack is huge. As production parameters are considered to be crucial and should be protected, we also give a brief description over the threat model being created and subsequently, the set of mechanisms that need to be provided to ensure security.

The architecture being designed can still evolve during time, to incorporate new features and to reduce inefficiency caused by initial design. The openMOS consortium has prepared themselves for foreseeable changes via using the professional tool Enterprise Architect to create a design model that accompanies this report. Doing so makes it possible to maintain the integrity of multiple views in the architecture.

8. References

[1] “openMOS - Description of Work”, Proposal 680735 for the EU Horizon 2020 framework programme, The openMOS consortium, 2015.
[2] OASIS MQTT standard 3.1 (http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html)
[3] N. Keddis, G. Kainz, and A. Zoitl. Capability-based planning and scheduling for adaptable manufacturing systems. In Proceedings of the 19th Conference Emerging Technology and Factory Automation (ETFA), pages 1–8. IEEE, 2014.
[4] C.-H. Cheng, M. Geisinger, H. Ruess, C. Buckl, and A. Knoll. MGSyn: Automatic Synthesis for Industrial Automation. In Proceedings of the 24th International Conference on Computer Aided Verification (CAV), volume 7358 of LNCS, pages 658-664. Springer, 2012.
[5] A Jansen and J Bosch. Software architecture as a set of architectural design decisions. 5th Working IEEE/IFIP Conference on Software Architecture (WICSA), pages 109-120, IEEE, 2005.
[6] L. Bass, P. Clements, R. Kazman. Software architecture in practice (3rd Edition), Addison-Wesley, 2012 (ISBN: 978-0321815736)

[7] Arc42 Template - the template for software architecture documentation and communication. Available at http://arc42.org/
[8] Microsoft Security Development Lifecycle http://www.microsoft.com/en-us/sdl/
[9] G. Martins, S. Bhatia, X. Kousoukos, K. Stouffer, C. Y. Tang, and R. Candell. Towards a Systematic Threat Modelling Approach for Cyber-physical Systems. 2nd National Symposium on Resilient Critical Infrastructure (ISRCS 2015).
[10] Industrie 4.0: Die Industrie 4.0-Komponente. Zentralverband Elektrotechnik- und Elektronikindustrie e.V. Available at http://www.zvei.org/Downloads/Automation/Industrie%204.0_Komponente_Download.pdf

Appendix: Acronyms

Acronyms	Full description
MSB	Manufacturing Service Bus
OPC UA	Open Platform Communications (OPC) Unified Architecture
MQTT	MQ Telemetry Transport
DDS	Data Distribution Services
PnP	Plug-and-Produce
SDL	Security Development Lifecycle