The Implementation of Proactive Asset Administration Shells: Evaluation of Possibilities and Realization in an Order Driven Production

Sergej Grunau, Magnus Redeker, Denis Göllner and Lukasz Wisniewski

Abstract

A major benefit of Digital Twins is autonomous decision making. The concept of the Asset Administration Shell (AAS) enables an assets interaction in Industry 4.0 application scenarios and beyond. This article defines and validates implementation possibilities for proactive AASs by integrating their different types in an AAS infrastructure for an order driven production. The proactive AAS execute the VDI/VDE 2193-interaction protocol in a demonstrator. For this purpose suitable AAS submodels for production and storage are modeled.

Keywords

Industry 4.0 · Digital Twin · Asset Administration Shell (AAS) · Interaction · Semantic Interoperability · Bidding Procedure · Smart Manufacturing

S. Grunau (✉) · L. Wisniewski
Institute Industrial IT – inIT, OWL University of Applied Sciences and Arts, Lemgo, Germany
e-mail: sergej.grunau@th-owl.de; lukasz.wisniewski@th-owl.de

M. Redeker
Fraunhofer IOSB-INA, Lemgo, Germany
Fraunhofer Institute of Optronics, System Technologies and Image Exploitation, Lemgo, Germany
e-mail: magnus.redeker@iosb-ina.fraunhofer.de

D. Göllner
Lenze SE, Aerzen, Germany
e-mail: denis.goellner@lenze.com

J. Jasperneite, V. Lohweg (Hrsg.), *Kommunikation und Bildverarbeitung in der Automation*, Technologien für die intelligente Automation 14,
https://doi.org/10.1007/978-3-662-64283-2_10

1 Introduction

The high flexibility of intelligent manufacturing requires an increasing interaction between the system components as well as a higher ability to react to changing requirements [1]. The application scenario "order driven production" is one of the main Industry 4.0 (I4.0) scenarios defined by the German initiative "Platform Industrie 4.0" [2]. In future production lines the product itself, respectively its Asset Administration Shell (AAS), guides its way through the production.

An AAS is a digital representation of a physical or logical object – the asset – in an I4.0 system. It enables the properties and capabilities of assets to be described in a semantically unambiguous and machine-readable form. An I4.0 system consists of I4.0 components, comprising an asset and an AAS, that interact purposefully with each other [3].

Using these concepts can lead to a very flexible way of producing goods replacing rigid production processes. The AASs need the ability to interact with each other so that, for example, a product can negotiate and schedule each of its production steps directly with a production machine.

One of the main aspects of the it's OWL-research project "Technical Infrastructure for Digital Twins" (TeDZ), is to define and implement I4.0-use cases, [5]. This paper discusses different possibilities of implementing (proactive) AASs that can talk to each other in the specified language of VDI/VDE 2193, [7, 8]. For the practical implementation two assets – a product and a high-bay warehouse for storing the product – are used.

This paper is organized as follows: Sect. 2 briefly recaps the concept of proactive AASs as well as the VDI/VDE 2193-interaction protocol, Sect. 3 discusses possibilities to implement proactive AAS, Sect. 4 introduces a production infrastructure using proactive AASs as autonomous economic actors, Sect. 5 specifies in detail the invented Bidding-App implementing the bidding procedure, and, finally, Sect. 6 concludes this paper and gives an outlook on future work.

2 Types of AASs and the Bidding Procedure

This section briefly recaps the different types of AASs and the language they use to interact purposefully with each other.

2.1 The Types of AASs

In the document "Verwaltungsschale in der Praxis" [6], three different types of AAS are introduced:

Passive AAS in file format File as AASX-package, XML or JSON, which provides all information related to an asset. The structure of the AAS is specified in [3].

Reactive AAS Provides the same information content as the passive AAS in file format via an interface depending on the selected technology (HTTP-Rest, OPC UA, etc.) with a CRUD-oriented specification.

Proactive AAS Proactive AASs can take decisions and participate in protocol-based interactions like [7, 8] specifying an I4.0 language and an interaction pattern.

Although the AAS meta model is precisely specified in [3], it does not clarify at which point a proactive AAS makes decisions. On the other hand, [11] presents a concept and structure of a proactive AAS as a combination of passive and active behavior. Parts of the active component are an interaction manager and a messenger. The interaction manager uses state machines to implement various semantic interaction protocols for calling up the necessary decision algorithms. The messenger is an interface that handles the transport of messages.

2.2 The VDI/VDE 2193-Interaction Protocol

The bidding procedure specified in [7, 8] is used for the interaction of AASs. It consists of two parts. The first part defines the structure of exchanged messages. They contain a frame and an interaction element. The frame includes mainly the IDs of the communication participants, the purpose of a message and the role (service requester or provider) of the sender. An interaction element is a property-based description of a service. The interaction protocol is described in the second part of [7, 8] and defines the process of how messages are exchanged and how to react.

3 Implementation of Proactive AASs

Today's state of the art does not provide a clear solution on how to implement a proactive AAS. Therefore, this section proposes two basic types of proactive AAS implementation that is preceded with a summary of requirements.

3.1 Requirements for Proactive AASs

The following requirements for proactive AASs were derived from the smart production showcase described in detail in Sect. 4:

1. Possibility to deliver proactive and reactive AAS-parts as one single AAS together with an Asset.
2. Effortless integration and execution after receiving a proactive AAS.
3. Asset data is stored exclusively in the reactive part, so that proactive parts may act upon the same asset status.

4. Possibility to integrate proactive parts into the "Operation" SubmodelElement of standardized AAS meta model [4].
5. Possibility to apply high security standards.

3.2 Type 1: Proactive Part as AAS-Server Functionality

Implementing the proactive part of an AAS as an AAS Operation in the server, on which the reactive part is deployed, is the first possible solution (Fig. 1a). Comparable to methods in an OPC UA-server, an AAS Operation must be implemented as part of the AAS-Server code before the actual AAS-Server can be started. For example, the BaSyx SDK, that implements AAS Operations as lambda functions, follows this approach [13]. AAS Operations contain the proactive parts. They are directly connected to the reactive part containing the asset data.

While this approach simplifies the implementation of proactive AASs, it significantly complicates the subsequent distribution and integration of new proactive parts adding to an existing reactive or proactive AAS. Consequently, it is well suitable for proprietary proactive parts like the price determination for a proposed service and also parts that do not change over time.

3.3 Type 2: AAS-Application Outside the AAS-Server

Implementing a proactive part of an AAS as a separate application outside the server, on which the reactive part is deployed, is the second possible solution for implementing

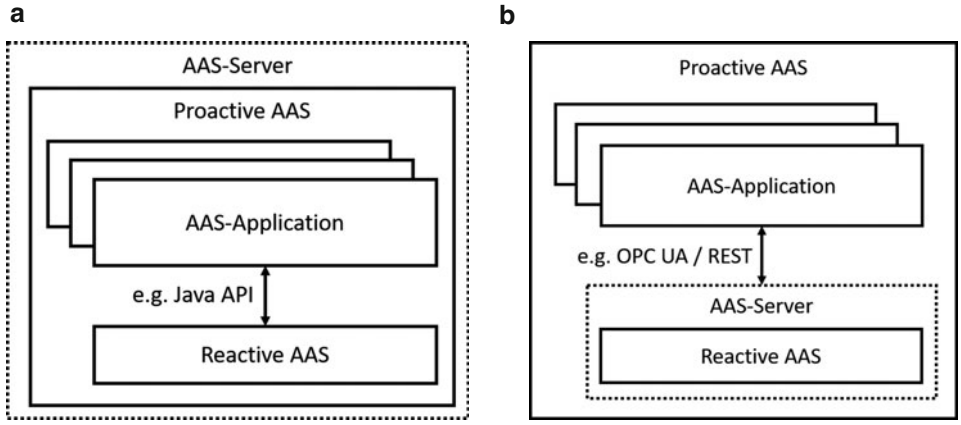

Fig. 1 Two basic types of implementations of proactive AASs. (**a**) Type 1: Proactive part of the AAS as part of server functionality. (**b**) Type 2: Proactive part of the AAS as separate application

proactive AASs. Such an application communicates with the corresponding reactive AAS via the API of the AAS-server, like REST or OPC UA. This kind of implementation is presented in detail in Fig. 1b. Via an AAS Registry the application can connect to a specific AAS and its Submodels and provide the proactive functionality for a particular asset.

The major advantage of this type is its independence of AAS-servers simplifying significantly the distribution of proactive AASs. Another benefit is that an application can be deployed once and multiple instances of it can activate multiple reactive AASs. On the other hand, a disadvantage might be the effort related to managing of a large number of separate applications that is needed in a factory production line.

3.4 Future Possibility: JSON-Function Description

As pointed out in Sect. 2.1, the AAS concept provides the possibility to distribute an AAS as a .aasx zip file containing a JSON-information model. An advantageous feature would be the possibility of exchanging proactive parts as .json code within the JSON-information model. Although it is not recommended for security reasons, arbitrary code might get executed, the possibility to embed JavaScript code into JSON exists.

This possibility is not realizable in currently available AAS-Server implementations, which must first be further developed so that they are able to securely parse JSON-function descriptions and provide actual functionality.

3.5 Selection of the Appropriate Type and Their Coexistence

On the one hand, there are functionalities like the so called “bidding procedure”, that are standardized together with their “I4.0 language”. In the future, every asset in I4.0 production lines as well as the produced products will need to support this concept. Type 2, the separate AAS-application, is suitable for this kind of functionality. It can be deployed once and an instance of it can be executed for each asset in the production.

On the other hand, there are proprietary functions such as asset-specific algorithms, that single machine providers create just for their asset. These functions have to be provided in a manufacturer specific manner. The authors guess that they will be implemented as type 1, that is as part of the AAS server, which is hosting the reactive AAS. This server would be supplied by the producer of the machine with reactive and proactive parts already contained.

Please note, that both implementation types of a proactive AAS can coexist. Furthermore, a separate app of type 2 can start an AAS-operation of type 1 and vice versa. The following Sect. 4 presents an infrastructure for order driven production integrating both types of proactive AAS.

4 Infrastructure in an Order Driven Production System

This section presents an infrastructure of an order driven production system using proactive AASs of production facilities and products as autonomous economic actors. They control, schedule and document the production of the products. Figure 2 depicts, for the sake of simplicity, only an extract of this infrastructure, focusing on the production's final step: the storage of the manufactured product. The processing of the preceding production steps follows the same scheme as the final step.

The infrastructure extract consists of a Manufacturing Execution System (MES), AAS-Servers, an AAS-Registry-Server, a MQTT-Broker, two proactive AASs of two assets: product and storage. MES, AAS-Servers, AAS-Registry-Server, storage (asset) and the proactive Storage-AAS are always operating.

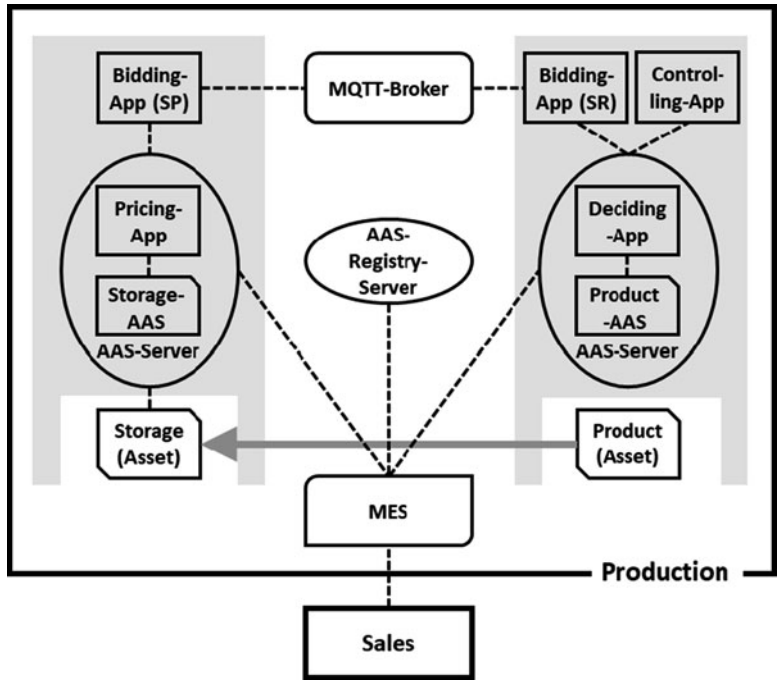

Fig. 2 Infrastructure-extract of the production system using proactive AASs of production facilities and products as autonomous economic actors. The proactive AASs are indicated by dark gray AAS symbols: each consisting of a reactive AAS in an AAS-Server in combination with AAS-Apps (Controlling-, Bidding-, Pricing, Deciding-App). Dashed lines indicate inter-component connections. For the sake of clarity, the connections from the AAS-Apps to the AAS-Registry are not depicted. The asset "Storage" fetches instructions from its AAS in the server. The asset "Product" on the other hand, is offline and not connected with its AAS. In the focused final step of production, the product instance, the manufacturing of which is already completed, must be stored in the storage. For this purpose, the product instance's proactive AAS executes the bidding procedure via the MQTT-broker

4.1 The Initialization of a Production Process

The MES receives orders from the sales department to manufacture products. In addition, sales sends a passive AAS in file format of each product instance to be manufactured.

Whenever the MES receives an order from sales to produce a product instance, it uploads the corresponding AASX-file to an AAS-server and registers this reactive AAS in the AAS-Registry. Furthermore, it adds Submodels containing all necessary information for the execution of the production process to the reactive AAS. This includes the Submodels of template ProductionSteps (Table 1), BiddingProcedureConfig (Table 2), and, for each step, ProductionStep (Table 3). Finally, the MES starts an instance of the Controlling-App, which ultimately controls the production of this one product instance.

Please note, that the mentioned Submodels are only added by the MES at the beginning of the production, since it is the production department's function to manage the production. This becomes even more relevant in case that sales and production are departments of separate companies.

Table 1 Submodel of template ProductionSteps. RefEl = ReferenceElement

idShort	Type	Value (example)
overallStatus	Property	started
numberOfSteps	Property	3
statusStep1	Property	completed
refStep1	RefEl	SMProductionStep1
statusStep2	Property	started
refStep2	RefEl	SMProductionStep2
statusStep3	Property	waiting
refStep3	RefEl	SMProductionStep3

Table 2 Submodel of template BiddingProcedureConfig. RefEl = ReferenceElement

idShort	Type	Value (example)
mqttUrl	Property	https://smartfactory-owl.de/.../mqttbroker
mqttPort	Property	3000
mqttUser	Property	pi-32657
mqttPw	Property	pi-052617022400
biddingAppUrl	Property	https://smartfactory-owl.de/.../bidding.exe
biddingMode	Property	serviceRequester
biddingPricingApp	Operation	null
biddingDecidingApp	Operation	decision.app

Table 3 Submodel of template ProductionStep. SubmElColl = SumbodelElementCollection, RefEl = ReferenceElement

idShort	Type	Value (example)
reqCapability	Property	store
biddingStatus	Property	started
biddingCall	SubmElColl	InteractionElement
biddingProposals	SubmElColl	InteractionElementSet
biddingDecisionStatus	Property	waiting
biddingDecisionRef	RefEl	biddingProposal
executionStatus	Property	waiting
executionStart	Property	null
executionEnd	Property	null

4.2 The Execution of a Production Process: The Proactive AASs

The proactive AAS of a product instance consists of the instance's reactive AAS in an AAS-Server in combination with instances of the Controlling-, Bidding- and Deciding-App. On the other hand, the proactive AAS of a production facility consists of the facility's reactive AAS and instances of Bidding- and Pricing-App.

The Controlling-App On start up, the ID of the AAS of the product instance whose production it controls is passed to the Controlling-App. From the AAS-Registry it retrieves the reactive AAS's endpoint in the AAS-Server. From the Submodel ProductionSteps (Table 1) it periodically queries the status of the production. When the production of a product instance is completed and it is stored in the storage, the Controlling-App terminates itself. On the contrary, if the production is not yet completed and

- if the status of none of the steps is *started*, it starts the first step in *waiting* and activates an instance of the Bidding-App passing as a parameter the IDs of the Submodel BiddingProcedureConfig and of the one Submodel ProductionStep that is referenced for that step,
- if the status of one step is *started* and
  - if both status of the bidding procedure and of the actual execution in the referenced Submodel of template ProductionStep, which are respectively edited by the Bidding-App and the selected production facility, are *completed*, it finishes the step,
  - otherwise, it checks again in the next period.

The Bidding-App The Bidding-App implements interactions of AASs following the Bidding Procedure from [7, 8] (see Sect. 2.2), where it can perform both sides: service requester (SR) and service provider (SP). As SP it proposes a service to a SR if its associated asset is capable and available to meet the request. Contrarily, as SR it tries to find a convenient SP for the service its associated asset needs. Please find a detailed description in Sect. 5.

Table 4 SubmodelElement-Collection of template InteractionElement

idShort	Type	Value (example)
callHeight	Property	15
callWidth	Property	10
callLenght	Property	20
proposalPrice	Property	5
proposalStart	Property	2020-10-29T09:00:00
proposalEnd	Property	2020-10-29T17:00:00

The Pricing- and the Deciding-App Pricing- and Deciding-App are both implemented as server functionality. An instance of the Pricing-App is invoked by the Bidding-App acting as a service provider in order to price the service that is proposed to the requester. In case of the storage, for example, the calculation is based on the expected energy consumption for storing and retrieving the product instance. On the service requester side, after expiry of the proposal period, the Bidding-App activates an instance of the Deciding-App for selecting the most suitable service-proposal weighting the proposed prices, start and end times. Both, Pricing- and Deciding-App, insert the result of their calculation into the predetermined SubmodelElement (PricingApp: proposalPrice in Table 4; DecidingApp biddingDecisionRef in Table 3), whose reference was passed to them as a starting parameter, and then terminate themselves.

4.3 The Completion of a Production Process

The MES periodically retrieves status information from the reactive AASs in the AAS-Servers of the product instances to be produced. When it determines that the production of an instance is completed, the MES downloads the corresponding AAS from the server into the original AASX-file. It erases the AAS as well as the Registry-entry from the respective servers. Finally, the MES sends the AASX-file in combination with a production confirmation back to the sales department.

5 The Bidding-App: Detailed Specification

This section specifies in detail the Bidding-App (in the following simply referred to as *App*) presented in Sect. 4. It was implemented in Java using the Eclipse BaSyx SDK [13], which implements the AAS-meta model and provides functions like the (de-)serialization of passive AASs (file format).

The requirements for the App to function are first determined with respect to the interaction protocol from the bidding procedure (Sect. 2.2) and the infrastructure from Sect. 4. Furthermore, the required Submodels which the App needs for configuration are

explained. Finally, the functionality of the App is described and its interoperability with a proactive AAS from an external project is shown.

5.1 Requirements

As described in Sect. 3, the App is not an information carrier, but it must be able to create, read, update and delete information in a reactive AAS if required. The reactive AASs are provided by AAS-Servers implemented with technologies such as HTTP or OPC UA. When reactive AAS and App interact, they are in a server-client relationship (vertical communication). When two proactive AASs interact, they are in a peer-to-peer relationship (horizontal communication). This horizontal message exchange is done via MQTT as specified in [7, 8]. Therefore, the App must implement an MQTT-client.

The App must implement both roles in the bidding process. One is that of the service requester (SR), who requests a service, for example a product requesting its storage. The second is the service provider (SP), who offers services, for example a warehouse that can store products.

Finally, the App should be executable on all Operating Systems used in the I4.0-context.

5.2 Required Submodels

The Submodel *BiddingProcedureConfig* (see Table 2) is required to configure the App. It contains properties for creating and configuring the MQTT client. The Submodel also contains two AAS-Operations (type 1) for price calculation (SP side) and decision (SR side).

In the example from Sect. 4 the SubmodelElementCollection *InteractionElement* from Table 4 contains the interaction element that describes the service. The storage (SP) needs the dimensions of the object to check whether it fits into its storage bins. The properties price, start time and end time are information for the SP. The product (SR) uses this information to decide which offer to accept.

The data elements in the Submodel *ProductionStep* that are necessary for the bidding procedure are the following:

- biddingStatus: Status of the bidding process.
- biddingCall: Interaction element of the bidding procedure.
- biddingProposals: Received proposals.
- biddingDecisionStatus: Status of the selection operation.
- biddingDecisionRef: Reference to the selected proposal.

5.3 Procedure

The execution of the App is shown with the activity diagram in Fig. 3: Initialization, SR and SP.

Initialization When the App is executed, the ID of the current Submodel *BiddingProcedureConfig* and the Registry-URL are given as starting arguments. From the Registry the App queries the endpoint of this Submodel. Depending on the type of the endpoint the App creates a client (HTTP, OPC UA) and connects to it. Afterwards the App creates and configures an MQTT client with the Submodel's parameters. Finally, it checks the bidding mode to execute: SR or SP.

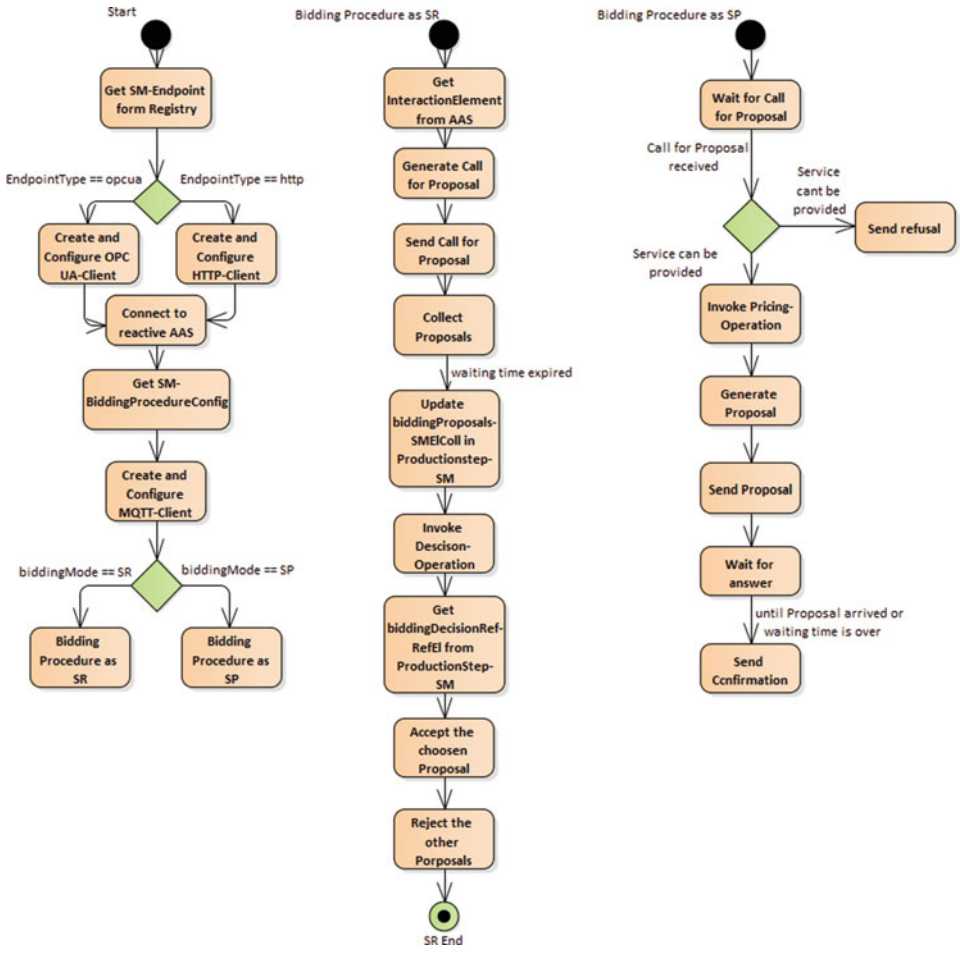

Fig. 3 Activity diagram of the App with the initialization (left), the SR activity (middle) and SP activity (right)

SR-mode In SR mode, the App generates and publishes a call for proposal containing frame and interaction element. It stores all incoming proposals. When the fixed waiting time is expired, the App invokes the decision operation for a selection. The App accepts the selected proposal and rejects the remainder.

SP-mode In SP mode, the App checks incoming calls for proposals and generates, if capability and availability permits it, a proposal invoking the pricing operation for the pricing of the service. If an acceptance of the proposal is received, the App generates and sends a confirmation.

5.4 Evaluation of the App

The project "Administrative Shell Networked" [14] provides a test bed in which AASs can interact with each other using the bidding procedure. The test bed contains a SR that sends a Call for Proposal every minute requesting for the service boring [9]. For the evaluation of the App an AAS for a virtual drilling machine was created which is capable of executing exactly this requested service. The interaction between the drilling machine AAS and the SR worked until the confirmation/informing step proving that the App works properly in this case. A complete test of the App follows.

6 Conclusion

In order to realise an order driven production exemplary implemented in the SmartFactoryOWL in Lemgo, Germany, this paper presents an infrastructure including reactive and proactive Asset Administration Shells (AASs). The proactive AASs act as economic autonomous actors, that are capable of taking decisions and interacting purposefully with each other with the specified VDI/VDE 2193-interaction protocol.

Two types of implementations are developed and validated in this paper. In both cases, a reactive AAS in an AAS-Server is completed with a proactive part. Type 1 implements the proactive part as AAS-Server functionality. On the contrary, type 2 implements it as an AAS-application outside the AAS-Server.

While type 1's implementation is straightforward, its distribution to partner's in a value chain is complicated. Consequently, it is well suited for proprietary functionality like the pricing of services or other functionalities that do not change over time. On the contrary, major advantage of type 2 is its independence of AAS-Servers simplifying significantly its distribution and instantiation. An application can be deployed once in a system and each instance of it can activate one AAS. Thus, type 2 is the proper solution for global functions.

For a fully functional order driven production system, a variety of proactive applications is necessary: a Controlling-, Bidding-, Deciding- and Pricing-App were presented and

implemented. Each of these applications needs specific AAS Submodels for parameterization. Asset data is stored exclusively in the reactive part, so that all proactive parts act upon the same asset status.

These proactive applications together with reactive AASs in AAS-Servers as well as the communication components build the infrastructure of the order driven production in the mentioned demonstrator.

Acknowledgements The research and development project "Technical Infrastructure for Digital Twins" (TeDZ) is funded by the Ministry of Economic Affairs, Innovation, Digitalisation and Energy (MWIDE) of the State of North Rhine-Westphalia within the Leading-Edge Cluster "Intelligent Technical Systems OstWestfalenLippe (it's OWL)" and managed by the Project Management Agency Jülich (PTJ). The authors are responsible for the content of this publication.

References

1. Qin, Jian, Ying Liu, and Roger Grosvenor. "A categorical framework of manufacturing for industry 4.0 and beyond." Procedia cirp 52 (2016): 173-178.
2. Anderl, R., et al. "Fortschreibung der Anwendungsszenarien der Plattform Industrie 4.0." (2016).
3. Plattform Industrie 4.0: Struktur der Verwaltungsschale – Fortentwicklung des Referenzmodells für die Industrie 4.0-Komponente. Berlin: BMWi, 2016.
4. Plattform Industrie 4.0: Details of the Asset Administration Shell. Part 1 – The exchange of information between partners in the value chain of Industrie 4.0. BMWi. Berlin, 2018.
5. it's OWL. URL: www.its-owl.de, Execessdate 28.02.2018
6. Plattform Industrie 4.0: Verwaltungsschale in der Praxis – Wie definiere ich Teilmodelle, beispielhafte Teilmodelle und Interaktion zwischen Verwaltungsschalen (Version 1.0). Berlin: BMWi, 2020.
7. VDI/VDE 2193 Blatt 1: Sprache für I4.0-Komponenten," Düsseldorf: VDI, 2019
8. VDI/VDE 2193 Blatt 2: Sprache für I4.0-Komponenten. Interaktionsprotokoll für Ausschreibungsverfahren. Düsseldorf: VDI, 2019
9. Belyaev A., Diedrich C.: Specification "Demonstrator I4.0-Language"
10. Plattform Industrie 4.0. URL: https://www.plattform-i40.de, accessed: 28.02.2018
11. Belyaev A., Diedrich C.: Aktive Verwaltungsschale von I4.0-Komponenten. 2019
12. Bidding-App. URL: https://gitlab.com/itsowl-tedz/bidding-app
13. Eclipse BaSyx. URL: https://wiki.eclipse.org/BaSyx, accessed: 28.02.2018
14. VWS vernetzt. URL: http://vwsvernetzt.de/, accessed: 26.07.2020