Agile Operational Behavior for the Control-Level Devices in Plug&Produce Production Environments

Conference Paper · September 2019

CITATIONS
0

READS
103

5 authors, including:

Kirill Dorofeev
fortiss
11 PUBLICATIONS **29** CITATIONS
SEE PROFILE

Stefan Profanter
fortiss
18 PUBLICATIONS **119** CITATIONS
SEE PROFILE

Pedro Ferreira
Loughborough University
24 PUBLICATIONS **123** CITATIONS
SEE PROFILE

Alois Zoitl
Johannes Kepler University Linz
205 PUBLICATIONS **2,118** CITATIONS
SEE PROFILE

Agile Operational Behavior for the Control-Level Devices in Plug&Produce Production Environments

Kirill Dorofeev*, Stefan Profanter*, Jose Cabral*, Pedro Ferreira†, Alois Zoitl*
*fortiss GmbH, Munich, Germany
{dorofeev, profanter, cabral}@fortiss.org
†Wolfson School of Mechanical and Manufacturing Engineering Loughborough University, Loughborough, United Kingdom
p.ferreira@lboro.ac.uk
‡fortiss GmbH, Munich, Germany / Johannes Kepler University, Linz, Austria
alois.zoitl@jku.at

***Abstract*—The ongoing manufacturing systems transformation from mass production towards mass customization requires more flexible engineering solutions than the existing ones. The recently proposed control architectures target, among other plug-and-produce features, a reduction of configuration times. This is relevant for building a new production line as well as for faster reconfiguration when adding new hardware and product variants to an existing manufacturing line. This paper identifies operational requirements for such reconfiguration scenarios and proposes a way to implement them using the concept of a device adapter. The device adapter contains a device description and constantly updates it following the reconfiguration changes happening in a manufacturing system. This allows not only to detect the changes in the hardware, which appear in the production system, using the device discovery mechanisms but also automatically adapt the software. Preliminary tests have been performed on a demonstrator that shows both virtual and physical executions combined in a single system. The proposed solution supports automatic hardware and software reconfiguration on-the-fly without a need to stop and restart the whole production system.**

***Index Terms*—Reconfigurable Manufacturing Systems, Skill-Based Engineering, AutomationML, OPC UA**

I. Introduction

The fast-changing market forces the manufacturing companies to increase their production systems flexibility by introducing new technologies and architectures at all control levels. The emergent control architectures aim at providing flexibility by plug-and-produce features such as service discovery and automatic software composition [1]. The flexibility requirements can be divided into 1) product flexibility that enables production of a huge number of product variants 2) technology flexibility that eases the use of different technologies within the production process and 3) resource flexibility that empowers the easy application of different assets within production processes [2].

In traditional mass production, switching to a different product variant usually requires either stopping to reconfigure the equipment, or allocating specific production lines for each variant. However, a key part of smart-manufacturing is flexibility, allowing variants of a product to be made using the same equipment without the need to stop production and retooling. Flexibility includes the concepts of variable routing, where the products take the most efficient path through production machines at the current time, as well as order based production, where the parts are only made in quantities and variants specific to customer request.

This paper describes the set of operational requirements that should be fulfilled by the automation components on the field-device level in order to achieve the required flexibility and be capable of the reconfiguration on-the-fly. Further, we show how to implement the proposed concepts using Automation Markup Language (AutomationML) and OPC Unified Architecture (OPC UA). Furthermore, the running example is a fully-functional demonstrator that combines both physical and virtual executions and simulates the production of a complex product with multiple variants. A rich transportation network, where seven Automated Guided Vehicles (AGVs) ferry the products between the workstations over 32 different paths complements the demonstration complexity. While providing a clear benefit in reconfiguration effort and time, the performance comparison of our proof-of-concept implementation with the legacy systems did not show a big overhead in the production throughput.

II. Related Work

The key technologies that enable the reconfiguration in the production systems are the open-architecture control and modular machines, responsible for the reconfigurable software and hardware, respectively [3]. Mehrabi et al. [3] have also outlined the need for the methods enabling efficient and fast reuse of the automation components and their functionalities in the Reconfigurable Manufacturing Systems (RMSs). One such method is to semantically describe of the modules constituting a production system and their functionalities [4] and to catalog them. In process automation Module Type Packages (MTPs) are used to encapsulate the process modules functionalities and provide it in the form of service to the rest of the system [5]. The MTP manifest is modeled by means AutomationML that is defined in IEC 62714 [6]. The information from MTP can

The research leading to these results has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 680735, project openMOS (Open Dynamic Manufacturing Operating System for Smart Plug-and-Produce Automation Components).

be shared between the automation modules throughout the system by communicating it by means of OPC UA. OPC UA allows representing the content of the MTP manifest in the namespace of the module, enabling the modules discovery and interaction [7].

The control software reconfiguration can be achieved by employing the agent technology [8] that is successfully applied for reconfiguration scenarios on the system level such as overall scheduling and planning. However, the current challenges in production flexibility include such application scenarios that can be only solved by moving the intelligence away from system level to the production stations [9]. If production stations of different vendors shall interact seamlessly, the use of standardized and secure communication protocols, information models, and functional specifications is essential for the implementation of this use case. Therefore, there is a need to have a reconfiguration mechanism also at the lower component system level. As the changes that occur at the component level, the whole system, as well as other interconnected systems, should be able to reconfigure themselves. The information about the current layout and operational state should be available at any time.

III. Problem Motivation

In order to sustain in the market conditions that are characterized by aggressive competition, the manufacturing systems must be upgradable and allow easy integration of the new functionalities and features. For dealing with such kind of scenarios serves a paradigm of RMSs was proposed [10]. RMS is a system that consists of hardware and software modules that can be quickly rearranged or replaced. Reconfiguration enables, in general case, adding, removing, or modifying specific process capabilities, controls, software, or machine structure. A RMS is characterized by 1) reduction of time for launching new systems and reconfiguring existing systems, and 2) the rapid manufacturing modification and quick integration of new technology and/or new functions into existing systems [3]. We put even more strict requirements on the production system under consideration. While producing different product variants, the system should never shut down completely, allowing to deal with product customization and various plug&produce devices on-the-fly. Additionally, to have a lot of redundancy in the production scenarios, the system can have a complex transportation network connecting different workstations constituting the system.

In order to show the operational mechanism for such RMS on the device level we developed a demonstrator, which is used as a running example in this paper. The demonstrator originates from the Open Dynamic Manufacturing Operating System for Smart Plug-and-Produce Automation Components (openMOS) research project. The main goals of the project were: 1) embedding plug-and-produce capabilities into automation devices, robots and machines, 2) enabling vertical and horizontal connectivity between plug&produce automation components and higher level control and business functions, and 3) creating an easily extendable and adaptable manufacturing operating system (MOS) that permits the easy introduction of new products, work orders and changes in the equipment and allows easy deployment of optimization and changeover management strategies. To develop and test most of the proposed features, as well as to examine the scalability of the solution, a demonstrator was developed that implements an ”openMOS-enabled” simulation environment. At its most basic, the demonstration has the following key requirements, derived from the use-case scenarios from multiple intelligent manufacturing scenarios among different industry sectors (electronics, white goods, automotive):

- Scalability - One feature that required testing is how the developed Device Adapter (DA) cope with a large number of workstations. Thus, openMOS should function with hundreds of devices, which consequently requires large-scale testing. This allows us to add and remove a large number of different workstations in the system on-the-fly.
- Plug&Produce - Associated with scalability, one of the key tenets of openMOS is the ability for equipment to be added or removed from the production system, without manual reconfiguration and system restart. This requires sufficient numbers of each workstation type for production so that there always remains a possibility to pick an alternative path to conclude the production. The workstations themselves must also be isolated from each other, to allow them to be removed individually.
- Transportation - For plug&produce to be viable, material flow within the system must not be workstation dependent. Should a workstation be added or removed, it must not only appear within the supply chain but must also be connected to the transportation network. Therefore a flexible transportation system is required to enable the concepts.
- Product Variation - An essential requirement was to have many workstations, which could be used to provide the multiple production lines required to allow product variations. Therefore, the end product has been chosen specifically to include multiple variants. It is a generic electronic device assembled from four components: two casing parts and two electronic internals shown in Fig. 1. In addition to the components, a gluing process is executed to hold the parts together once assembled. The product casing, internals, and gluing process can all be varied, allowing for individual products to be customized greatly, with 18 variants overall.
- Optimization - The agent–based control system for openMOS includes an optimizer which attempts to make the overall system more efficient. For this optimizer to work, the stations must have tunable parameters, which can be modified to alter overall material flow without stopping the system.

IV. Concept

A reconfigurable manufacturing system is designed to cater to the situations where both system productivity and its ability to deal with a rapid change in a structure are of great

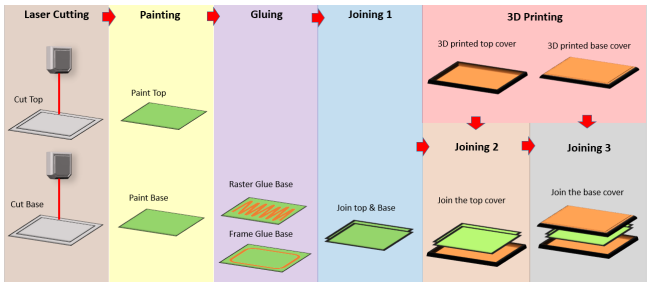

Fig. 1. Production Process

importance [11]. A manufacturing system is composed of equipment, including transport and workstations, containing multiple sub-components. We propose an implementation of such a reconfigurable manufacturing system by utilizing a concept of a skill [12] [13] as a way to normalize the functionality of each device in a production system and provide a generic interface to execute a production task. In the previous paper, we introduced a concept of a DA [14], a software wrapper that allows a device to be seamlessly plugged in into an agile production environment. Each system component has its capabilities modeled in the form of a skill. The proposed system architecture [15] is shown in Fig. 2, where the low-level devices being responsible for executing a production are connected to their cloud representation, which main task can be seen as the overall production optimization. This is achieved by constantly monitoring the data gathered from the devices via the DAs.

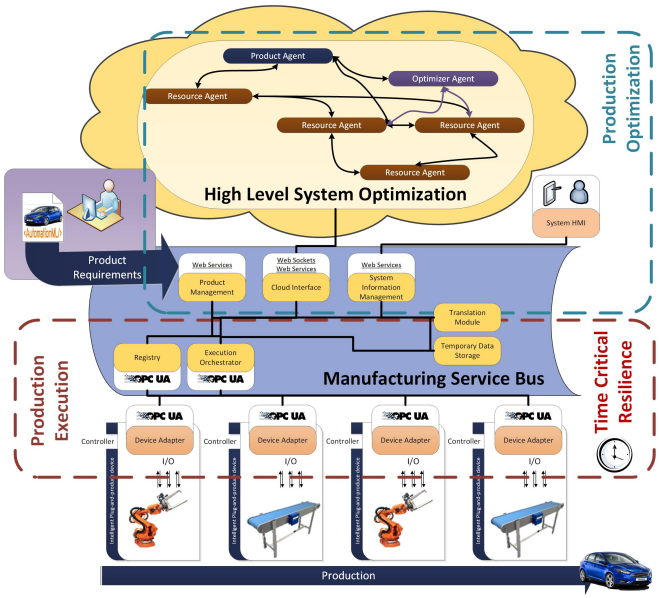

Fig. 2. System architecture

The device level requires atomic devices which are aggregated up to workstation level, the highest level of granularity. This means all devices internal to a workstation will require communication with the ability to expose their skills to the upper level. This communication will be critical for the performance of the system, as it implies the ability to execute skills, and therefore requires a higher level of priority. Each device has a self-description that follows a common semantic model and provides, among other data, the information about its capabilities [16].

On the cloud level of the solution, there are two aspects to consider: the cyber representations and the device data. The cyber representations are achieved through the agents, which are the counterparts for the highest level resources, namely the resource agent (workstation) and the transport agent. The agents require a direct connection to their physical counterparts, which will provide the ability to monitor the system and explore potential optimizations. These agents are expected to interact with other agents to achieve their goals. It is important to note that agent to agent communication will not go through the service bus but directly between the agents within the cloud. The device data for every device in a system is stored in the cloud. This data can then be accessed by the agents on the cloud to inform their decision-making process.

The Manufacturing Service Bus (MSB), the middleware layer, provides a uniform communication mean for the whole system, enabling a mechanism for an easy component discovery, configuration, and interoperability.

We define four operational phases to achieve a running Plug&Produce system [17]: Discovery, Configuration, Production, and Reconfiguration. In this paper, we concentrate on the Reconfiguration phase and the tasks that arise when a new product is introduced into such a system or a workstation/module is either plugged in or unplugged. In response to such changes the system should be capable of performing a job change when new production task arrives, or if there is a change in communication parameters, or if there is a change in hardware setup. This allows rapid integration, adaptation of different devices and assembly stations.

The skills are defined in a device description file that serves as an input for a DA: each adapter is automatically generated out of its description. The skills can be later matched with product definitions and, afterward, a skill sequence, needed for production, can be derived. Skills vary from atomic (elementary skills, executed by single backend device) to composite (an ordered set of atomic or other lower-level composite skills).

The execution of skills is performed by skill recipes. A recipe is a parameterized executable instance of a skill that fulfills a certain skill requirement. The process of matching the skill requirements defined by the product to skills offered by the equipment will result in the creation of a skill recipe [14].

Additionally, each DA has an execution table [16] that stores the recipe execution sequence. The execution table also contains recipe-product relations that define the set of instructions to be executed for each specific product arriving at a workstation. This serves as the basis for improvements by the cloud optimization methods that may update the execution tables and, thus, change the execution logic on-the-fly.

The following list of DA operational requirements was defined for the reconfiguration phase:

- **Semantic description** of all elements should not only contain the hardware structure of a connected component and a list of its capabilities, but it also should be re-adjustable to accommodate the changes, occurring in a system. This self-description is stored in the DA in form of a file and follows a generally accepted component description model. The proposed solution utilizes AutomationML as a modeling language where the engineering information is stored following object-oriented principles. Such a description contains physical and logical production system components as data objects encapsulating the different relevant information aspects [18]. The device description follows the common semantic model and describes not only the hierarchical structure of a workstation or a transport unit but also their skills, which can then be executed in order to fulfill a production task. A device adapter self-description is an aggregation of all lower-level components descriptions that constitute a workstation.
- **(Un)plugging a workstation/module** requires the corresponding DA and MSB adjustments including reflecting the changes that happened in the self-description of a workstation/module for traceability.
- **DA standalone operation** should be guaranteed even if the connection to the optimization cloud is lost. The production must not be stopped even if it is not run optimally. The cloud, whenever it restores the connection, can reconfigure the production scenario afterward.
- **Switching a module from one workstation to another** should be possible in runtime. In this case, the system after discovering a change readjusts itself.
- **Creation of new recipes for atomic/composite skills, editing recipes, and execution tables**. It should be possible to create new recipes or editing existing ones in order to either execute new production sequences, e.g. in the case of a new product or parameterize the existing sequences, e.g. to produce another product variant.
- **DA orchestration of composite skills** should enable a mechanism of on-the-fly reconfiguration of the skill sequences.
- **Runtime production adjustments** can be either demanded by MSB or by the low-level device after executing some process step depending on its outcome. An example of the adjustments coming from the high-level could be a hardware change in the system setup so that the products need to be re-routed to another working station to execute the whole process. An example of a later case could be a product that should be processed differently depending on a test result done by a workstation.
- **Skill interlocking** should provide a mechanism to buffer the skill calls and execute in a queue, one after another. This should be used when a composite skill sequence contains a long-running process during the execution of which the next recipe call can be already started. Consider, for example, an AGV, which task is to bring the products between a laser cutting, painting and gluing stations. Assuming that the painting process takes some time and is defined as a long-running, an AGV skills execution logic can be modeled as follows (see Fig. 3):
  - The AGV receives a call to bring a product from laser-cutting station to a painting station and then to a gluing station.
  - While it executes the transport task between a laser cutting and painting station, the next product arrives, demanding the same AGV to bring it along the same path.
  - After the first product arrives at the painting station, the AGV is able to start the second product execution, while the first part is painting. It comes back to the laser cutting station, gets the second pruct and transports it to the painting station. After arriving at painting station the AGV unloads the second product for painting and loads the first one, once it is ready.
  - Finally, the AGV finishes the task for the first product and then gets back to finish the second one.

  This allows having a more flexible execution logic, which at the end saves time and costs, and deals with complex products.
- **Scalability**. The system should be scalable and cater to the setups consisting of virtual and physical stations running together.

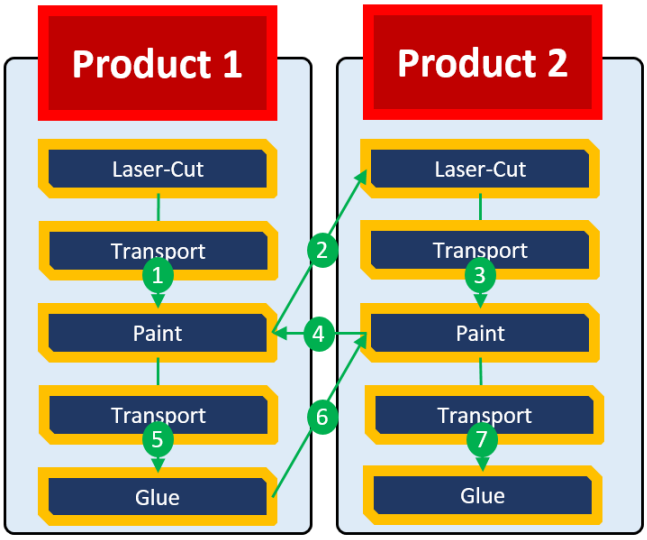

Fig. 3. Skills interlocking example

V. Implementation

The developed demonstrator, introduced in Section III, consists of eighteen simulated (four laser cutting, three painting, two 3Dprinting, two gluing, six joining and one labeling) and a real (gluing) workstations as well as seven simulated AGVs (see Fig. 4). The AGVs can bring the material between any two stations, without the restrictions of a fixed system, such as a conveyor belt. Each workstation has its DA, responsible for the system description and the corresponding skill execution. All

AGVs are controlled on the contrary by one DA introducing a device management layer. This demonstrates an ability to control multiple resources from single DA.

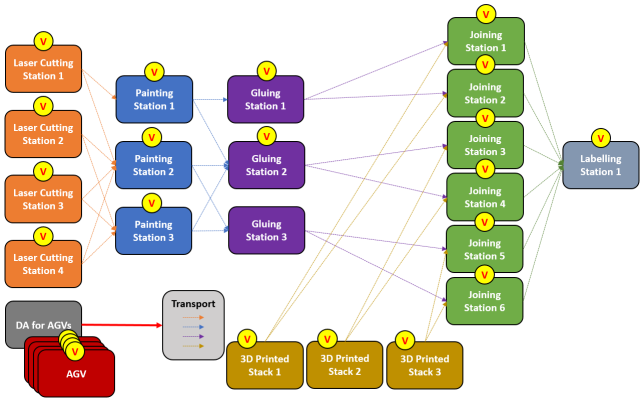

Fig. 4. Demonstrator Layout

All the communications between the workstations and the MSB are done via OPC UA. The modeling capabilities of OPC UA allow the devices to communicate the information contained in the device description AutomationML files with each other in the runtime. Each DA after starting up, reads its own self-description file and generates an OPC UA server out of it, following the OPC UA Companion specification "AutomationML for OPC UA" [19] [18].

A. Semantic description of all elements

The self-description file for each device requires common terminologies and concepts for both the Role Class and Interface libraries of the AutomationML file [20]. The first step for defining the self-description file is to create the generic device description in the system unit class library and initialize the available skill types and interfaces. Once this is done, one can simply create an instance of a generic device in the Instance Hierarchy to define the specific device. This node is communicated to the next hierarchical level where it is aggregated, and so on until reaching the highest level of granularity (workstation or transport). It is also important to note that a device description has approved recipes that can be used to fulfill requirements, from both products and composite skills.

Each skill has a state machine that identifies if the skill is either in $Ready$, $Executing$ or $Error$ state. The MSB, before triggering a skill, checks if the DA is ready for execution and, if so, triggers the production. All the information, defined in AutomationML description is passed up to the cloud-based system towards having a virtual representation of an I4.0-compliant component.

Overall, this enables an automatic generation of the executable skills that can be triggered via OPC UA out of the AutomationML system description.

B. (Un)plugging a workstation/module

Re-configuration requires the detection of newly plugged in devices and downtime of already registered devices. To be able to detect new devices on the network, it is required to implement a discovery mechanism for OPC UA servers. The OPC UA specification [21] describes the necessary OPC UA services for the automatic discovery of other OPC UA servers on the network. The concept of using Local Discovery Servers with Multicast Extension (LDS-ME) was previously described in [22].

LDS-ME provides basic functionality to find other OPC UA servers on the network using multicast DNS [23]. As soon as a new device is plugged in, it announces itself on the network. Within seconds all other LDS-ME servers on the network are notified. We are using this announced information to forward it to the DA implementation, which can handle new devices correspondingly.

One major issue in this implementation is the detection of device unplugging or shutdown. If a device is suddenly offline, the DA on the other end should be able to discover this new state and either redirect tasks to other devices or notify the MSB about the missing device. A device may become offline due to different reasons. It may simply be shut off in a graceful way by, e.g., using the device's user interface. In this case, the device is still able to unregister itself from other DAs. If the device has a hard failure or the network connection is interrupted, the device can not send an unregister message, therefore other devices need some kind of heartbeat implementation to detect this downtime. The OPC UA specification defines a timeout of 10 minutes: if a device does not send a new register request every 10 minutes, it should be removed from the known devices. This interval of 10 minutes can be quite high for specific use-cases. Therefore we implemented an extension to the OPC UA discovery process which is able to detect a device downtime within less than a second.

This is achieved by implementing an additional heartbeat communication channel via User Datagram Protocol (UDP). As soon as a device registered itself with another DA, an additional heartbeat connection is opened, where both devices send a heartbeat every 250 milliseconds. It is necessary to use UDP since Transmission Control Protocol (TCP) communication is bound to a specific connection where messages are automatically cached by the operating system. If one of the two ends does not receive a new heartbeat message within 600 milliseconds, the network connection is defined as broken. In this case, the OPC UA discovery process is notified about the lost connection. The DA implementation will then be notified through the same API as if the device would unregister itself. Reusing the OPC UA discovery interface allows us to transparently implement the heartbeat and adapt it to the requirements of the corresponding network connection.

C. Switching a module from one workstation to another

A heartbeat mechanism described above can be used for example in the following scenario: if a module that provides an auxiliary functionality is shared between two workstations, its presence at one workstation and absence at another can be automatically detected. For instance, in our demonstrator,

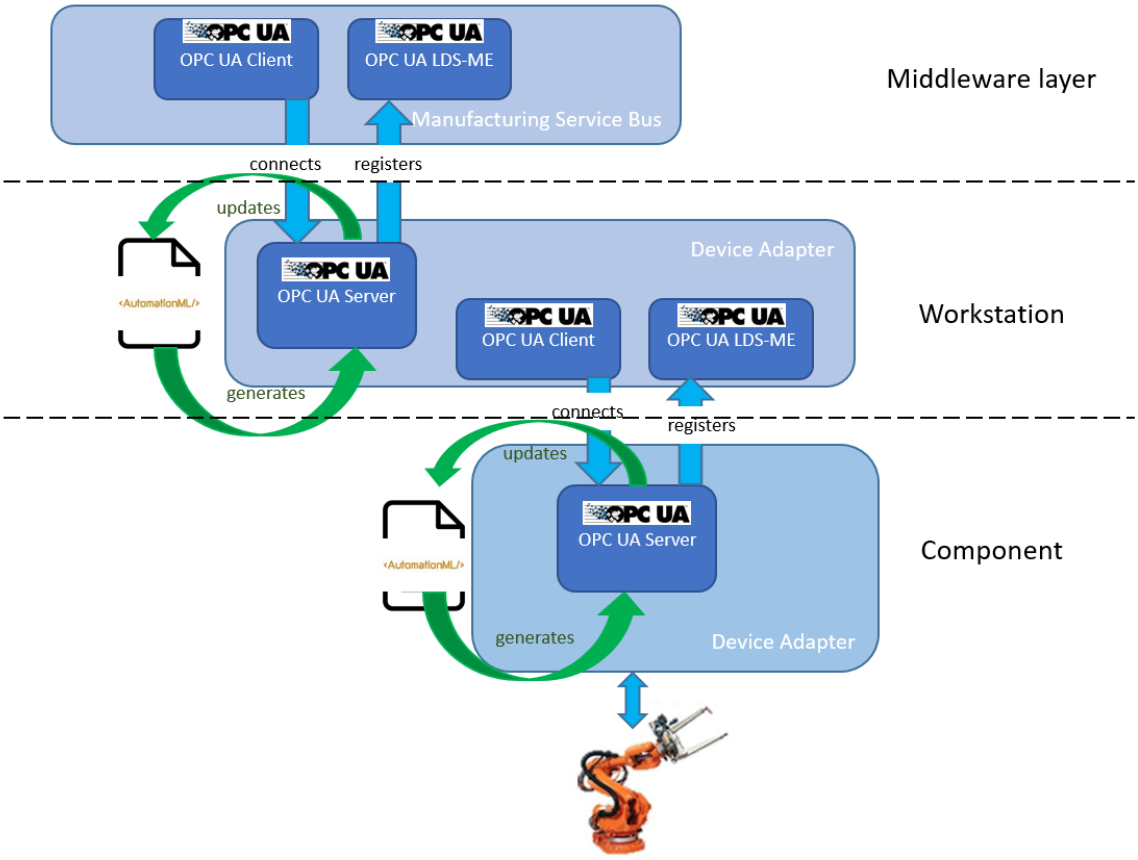

Fig. 5. Device adapter functionality to support dynamic reconfiguration

a painting station can have a quality checking camera that can be connected either to one or another virtual gluing workstation. It monitors only one workstation at the same time, therefore, whenever, it is connected to a workstation, this workstation provides not only *Gluing* skill but also *GluingWithQualityCheck* skill, whereas another gluing workstation is only capable of *Gluing*. Whenever the LDS-ME server of the workstation DA detects a newly registered camera server, it extends its description with the skills defined in the camera's self-description file and communicates the new capabilities to the MSB by triggering it, notifying the changes done in the workstation setup (see Fig. 5). The MSB re-browses the OPC UA namespace of the DA and gets the information about newly available skills. Reversely, if the device is unplugged after the heartbeat detects it, it is communicated to the higher-level OPC UA LDS-ME server, the device is unregistered and its functionality is removed from the corresponding workstation. The MSB is notified again and, after re-browsing the workstation DA namespace, it adjusts correspondingly.

D. Creation of new recipes for atomic/composite skills, editing recipes and execution tables

The device description for a workstation, which is created during the Engineering Phase contains a predefined set of the atomic skills, provided by the devices that constitute a workstation and some composite skills for the production processes known at the engineering time. However, agile manufacturing demands the mechanisms that allow to rapidly react to the changes in the production tasks, happening while the system is in operation. That requires the scenarios where either the existing production sequences are changed or new composite skills are introduced. This includes the parameter editing of existing sequences, for example, to execute a new product variant (having a new case color). Moreover, it is possible to create new combinations of skills to produce a completely new product, for example, an electronic device without a casing (in our demonstrator this will mean omitting the 3DPrinting skills). In this case it is essentially important to keep the documentation up-to-date, saving all new parameters and newly created skills. Therefore, once a new composite skill or an edited skill is sent to a DA either via the system Human-Machine Interface (HMI) or from the higher-level Manufacturing Execution System, the DA is updating its AML description, writing back new parameters and/or new composite skill steps. Afterward, it generates a new OPC UA server out of it exactly as it does it during the startup and notifies the MSB about the changes being made so that the upper control level can re-browse the new data and get the executed updates. The overall process is shown in Fig. 5.

By doing it that way, the information is kept up-to-date on each level of the running system. If the module fails for some reason, after its restarting, it will appear in the system with the last working configuration, without need to parameterize the module again manually.

E. DA standalone operation

After the MSB gets the initial process configuration from the cloud or system HMI, it deploys the configuration down to the device adapters. If then the connection to the cloud is broken, the DA continues its operation executing the existing recipes and ensuring the system operation even without the cloud connection. Thus, the agent cloud can be seen optional for the system operation, however, its presence is crucial for the system optimization and constant improvement.

F. Runtime production adjustments

This is a mechanism needed for the on-the-fly recipes configurations when the whole composite recipe should be changed in response to the process outcome during the execution. As an example for this use-case serves a quality check scenario, when a camera at the gluing station detects a defect in a gluing process, the gluing station DA gets this information and should be able to change the following sequence of the composite recipe execution in order to stop the production of the affected part. To implement such a scenario, the execution tables contain a list of possible next recipes to execute. The DA, based on the process outcome, can then choose the next recipe to execute from this list and notify the MSB about the next process steps by writing the chosen recipe to the $NextRecipeToExecute$ column in the execution table (see Fig. 6). This allows us to deal with the product quality checks scenarios, namely if during the production the product fails some quality test and could be then sent again to the previous workstations in order to be reworked if that is possible.

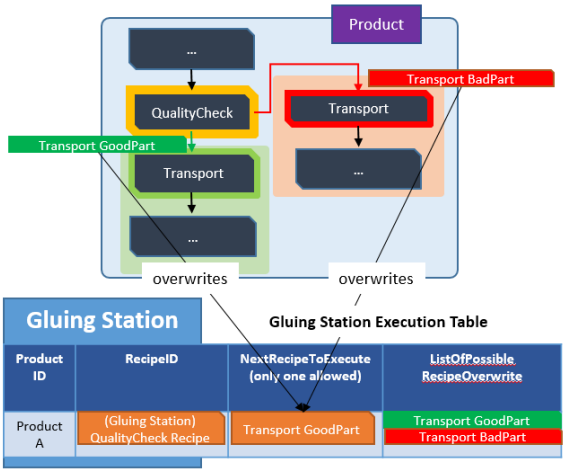

Fig. 6. Runtime adjustments using device adapter execution table.

G. Skill interlocking

As it is described earlier in Section IV requires a semaphore-like mechanism for the skill execution logic. The skill thread using the same resource must wait until the semaphore's value is positive, then change the semaphore's value by subtracting one from it. When it is finished the thread changes the semaphore's value by adding one to it. To realize this, we add a separate $LongRunningProcess$ state to the skill state machine that identifies a long-running process in a composite skill. While executing such a lingering skill, the other resources, responsible for the skill executions, can be released for executing another skill calls. We add a buffer of the skill calls, where we store the limit number of the skill calls that are arriving, while the other skill instance is already executing, blocking the immediate execution of the next call. Additionally, we add a semaphore variable that is controlling the number of skills using the resource. In the example in Fig. 3 the overall interlocking functionality works as follows:

- $Product1$ triggers the execution of the task recipe. The task consists out of laser-cutting, then transporting the product to the painting station, then painting, transporting the product to the gluing station and, finally, gluing. For simplicity, we assume that in this example there is only one AGV, which is responsible for transporting the products between the stations.
- When $Product1$ is executing the recipe, it decrements the semaphore value, setting it to zero, identifying that it will block for its execution the AGV resource.
- While $Product1$ is executing, the next call for $Product2$ arrives, but, since there is no free resource available and the semaphore value is not positive, this call is saved in a buffer queue.
- When $Product1$ arrives at the painting station, which is defined as a long-running skill, the state machine of the highest-level $Task$ skill is set to the $LongRunningProcess$ state and releases all other resources, responsible for this composite skill execution, incrementing their semaphore values.
- After the AGV is released, it reads the next skill call, which is stored in the buffer queue and executes the $Product2$, bringing it from the laser-cutting to the painting station. $Product2$ takes the semaphore and releases it as soon as the product arrives at the painting station.
- As $Product1$ is ready to leave the painting station, it blocks the AGV again by decrementing the semaphore. Then the process is finished for the $Product1$ at the gluing station and AGV brings the second product to finish the executed task for it as well.

This interlocking mechanism is especially useful for transport skills as it is shown in Fig. 3. Note, that we additionally restrict the interlocking to be executed only if the product types that require the same skill execution is the same to avoid bottlenecks in the skill queues. That means that interlocking is only possible for the products of the same types that need the same skill sequences to be executed. The type check is

done via the $ProductTypeIdentifier$ that is specified in the product description AutomationML file.

H. Scalability

The demonstrator includes 20 DAs running in parallel and communicating with each other over the MSB. Even though in this case all DAs are running on a similar hardware platform, the other project demonstrators successfully proved the concept of running the same wrapper code on top of multiple various control-level devices [14].

VI. Conclusion

In this paper, we demonstrated how a device adapter, a software wrapper that enables an integration of the automation devices – both brown- and green-field – into the agile production environments. We have put the strong operational requirements in order to achieve the required level of flexibility in the use-case scenarios. As a result, we have implemented the demonstrator that simulates an agile production line consisting of a relatively large number of the workstations, having a complex transportation system and producing a complex product with multiple variants. We have shown that the DA supports the overall system flexibility on the component level by keeping the system description updated, allowing dynamical device discovery mechanism, reattaching the modules and switching them between different workstations, supporting software reconfiguration, runtime product adjustments, and complex execution scenarios. It is worth to mention that the performance tests comparing the openMOS approach with the legacy systems showed that the overhead in the production throughput is not more than 1.5% [15] of the production cycle.

We have implemented the DA using the AutomationML for the engineering phase and OPC UA for the runtime phase. The prototype implementation is open sourced[1]. It contains not only the DA source code but also the examples of the AutomationML models for several workstations.

References

[1] V. Vyatkin, "Software engineering in industrial automation: State-of-the-art review," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1234–1249, Aug 2013.

[2] D. Wünsch, A. Lüder, and M. Heinze, *Flexibility and Re-configurability in Manufacturing by Means of Distributed Automation Systems an Overview*, 2010.

[3] M. Mehrabi, A. Ulsoy, and Y. Koren, "Reconfigurable manufacturing systems: Key to future manufacturing," *Journal of Intelligent Manufacturing*, vol. 11, 08 2000.

[4] A. Perzylo, S. Profanter, M. Rickert, and A. Knoll, "OPC UA NodeSet Ontologies as a Pillar of Representing Semantic Digital Twins of Manufacturing Resources," in *Proceedings of the IEEE International Conference on Emerging Technologies And Factory Automation (ETFA)*, Sep. 2019.

[5] J. Bernshausen, A. Haller, T. Holm, M. Hoernicke, M. Obst, and J. Ladiges, "Namur Modul Type Package Definition," *atp magazin*, vol. 58, no. 01-02, pp. 72–81, 2016.

[6] International Electrotechnical Commission (IEC, "Engineering data exchange format for use in industrial automation systems engineeringAutomation Markup LanguagePart 1: Architecture and general requirements, , International Standard, Rev. 2.0, IEC 62714-1," International Electrotechnical Commission (IEC, Tech. Rep., 2018.

[7] S. Wassilew, L. Urbas, J. Ladiges, A. Fay, and T. Holm, "Transformation of the NAMUR MTP to OPC UA to allow plug and produce for modular process automation," 09 2016, pp. 1–9.

[8] W. Lepuschitz, A. Zoitl, M. Valle, and M. Merdan, "Toward self-reconfiguration of manufacturing systems using automation agents," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 41, pp. 52 – 69, 02 2011.

[9] "Structure of the Administration Shell. Continuation of the Development of the Reference Model for the Industrie 4.0 Component." accessed: 2019-03-20. [Online]. Available: https://www.plattform-i40.de/I40/Redaktion/EN/Downloads/Publikation/structure-of-the-administration-shell.pdf?__blob=publicationFile&v=5

[10] Y. Koren, U. Heisel, F. Jovane, T. Moriwaki, G. Pritschow, G. Ulsoy, and H. Van Brussel, "Reconfigurable manufacturing systems," *CIRP annals*, vol. 48, no. 2, pp. 527–540, 1999.

[11] Y. Koren, *The global manufacturing revolution. Product-Process-Business Integration and Reconfigurable Systems*, 2010.

[12] J. Pfommer, M. Schleipen, and J. Beyerer, "Configuration model for evolvable assembly systems," in *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2014.

[13] P. Ferreira and N. Lohse, "Configuration model for evolvable assembly systems," in *4th CIRP Conference On Assembly Technologies And Systems*, 2012.

[14] K. Dorofeev, C.-H. Cheng, M. Guedes, P. Ferreira, S. Profanter, and A. Zoitl, "Device Adapter Concept towards Enabling Plug&Produce Production Environments," in *Proceedings of the IEEE International Conference on Emerging Technologies And Factory Automation (ETFA)*, Sep. 2017.

[15] F. Miranda, R. Martins, K. Dorofeev, V. Gentile, P. Ferreira, and M. Guedes, "Towards a Common Manufacturing Service Bus to Enable Flexible Plug-and-Produce Automation," in *ISR 2018; 50th International Symposium on Robotics*, Jun. 2018.

[16] P. Danny, P. Ferreira, K. Dorofeev, and N. Lohse, "An Event-Based AutomationML Model for the Process Execution of Plug-and-Produce Assembly Systems," in *IEEE 16th International Conference of Industrial Informatics (INDIN)*, Jul. 2018.

[17] T. Arai, Y. Aiyama, Y. Maeda, M. Sugi, and J. Ota, "Agile assembly system by plug and produce," *CIRP Annals-Manufacturing Technology*, Dec. 2000.

[18] M. Schleipen, A. Lüder, O. Sauer, H. Flatt, and J. Jasperneite, "Requirements and concept for Plug-and-Work," *Automatisierungstechnik*, vol. 63(10), pp. 801–820, Jun. 2015.

[19] "Companion Specification AutomationML for OPC UA," accessed on 2019-03-25. [Online]. Available: https://opcfoundation.org/news/opc-foundation-news/bridging-the-gap-between-communication-and-semantics-for-industrie-4-0-companion-specification-automationml-for-opc-ua/

[20] "Whitepaper AutomationML Part 1 - Architecture and general requirements." [Online]. Available: {https://www.automationml.org/o.red/uploads/dateien/1417686950-AutomationML%20Whitepaper%20Part%201%20-%20AutomationML%20Architecture%20v2_Oct2014.pdf/}

[21] OPC Foundation, "OPC UA Specification Part 12: Discovery," OPC Foundation, Tech. Rep., 2015. [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-12-discovery/

[22] S. Profanter, K. Dorofeev, A. Zoitl, and A. Knoll, "OPC UA for Plug & Produce: Automatic Device Discovery using LDS-ME," in *Proceedings of the IEEE International Conference on Emerging Technologies And Factory Automation (ETFA)*, Limassol, Cyprus, Sep. 2017.

[23] S. Cheshire and M. Krochmal, "Multicast DNS," Internet Requests for Comments, RFC Editor, RFC 6762, February 2013, accessed on 2019-03-25. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6762.txt

[1] https://github.com/openMOS/deviceAdapter