Dynamically Wiring CPPS Software Architectures

Michael Mayrhofer
Pro2Future GmbH
Linz, Austria
michael.mayrhofer@pro2future.at

Christoph Mayr-Dorn
Johannes Kepler University
Linz, Austria
christoph.mayr-dorn@jku.at

Ouijdane Guiza
Pro2Future GmbH
Linz, Austria
ouijdane.guiza@pro2future.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

2021 22nd IEEE International Conference on Industrial Technology (ICIT) | 978-1-7281-5730-6/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICIT46573.2021.9453697

Abstract—**Modern cyber-physical production systems require a software-intensive integration at the shopfloor, such as machines, transport systems, worker assistance systems, and robots. Alongside the interaction of participants at the physical level exists another level of interactions at the software level: the distributed software architecture of the cyber-physical production system (CPPS). Contemporary CPPSs require frequent changes in communication paths and the signals communicated (i.e., rewiring) to flexibly adapt to changing production needs and environmental conditions. The integration of shopfloor participants and their wiring, hence, becomes a critical aspect of a dynamically adaptable production system. Here, incomplete and incorrect wiring is a major source of errors. To this end, this paper presents a light weight approach for wiring shopfloor participants that utilizes decentralized wiring information to extract/recover the distributed software architecture of the CPPS. We demonstrate the feasibility of our approach on a lab-scale production cell model.**

Index Terms—**software architecture; reconfiguration; cyber-physical production systems; adaptability; OPC UA**

I. Introduction

Modern production systems require the integration of shopfloor (factory) machines, transport systems, worker assistance systems, and robots. These production participants interact at the physical level and at the software level. Indeed, today's participants to production systems are distributed, interconnected, software-intensive systems: i.e., cyber-physical production systems (CPPS). Tomorrow's shopfloors need to be able to produce highly customized products (down to lot size one) while keeping costs similar to mass production. Doing so requires the dynamically reconfigurability of shopfloor participants.

Production engineers require considerable time setting up and reconfiguring a shopfloor, a procedure that often involves engineers from different disciplines and/or engineers responsible for different machines. Machine vendors often provide proprietary interfaces how their product's control software needs to be connected (i.e. the software components wired) to other shopfloor participants. Hence, an engineer (re)configuring a machine—respectively tool support—typically needs to have a good understanding of the internal machine configuration. However, because of this, quite often they only have a partial understanding of the overall shopfloor. Interoperability becomes a critical aspect as incomplete and incorrect configuration is a major source of errors and subsequently results in costly stand still times or even damage to the production cell.

The concept of virtual commissioning (configuration) aims to mitigate the threat of damage and lengthy down times [1] but doesn't address the complexity of the control software (re)wiring process. State of the art provides little support. Contemporary approaches and tools such as provided by the ArrowHead framework [2] presume a centralized approach for discovery, orchestrations, and description of shopfloor participants.

We argue in this paper that the (re)wiring of the control software of a CPPS requires a decentralized, lightweight approach to reflect the multi-vendor, multi-domain view on the shopfloor participants. An approach is needed that supports interoperability for systems-of-systems [3] whilst making use of existing standards for machine-to-machine communication.

Based on "traditional" software architecture description language-centric concepts (see e.g., [4]), we present in this paper a lightweight approach based on OPC Unified Architecture (OPC UA) [5], in Europe a de-facto protocol specification for industrial communication. While OPC UA specifies how information is modeled, accessed, and transmitted over the network, it provides no mechanism to specify interface-like information nor mechanisms that describe how components of different shopfloor participants may be wired up across each other.

Our approach allows easy online reconfiguration without interrupting the execution of control code. We further explain how to utilize decentralized wiring information to extract/recover the CPPS's distributed software architecture. Integration of this approach in a lab-scale production cell model demonstrates feasibility and benefits.

The remainder of the paper is structured as follows. Section II provides a motivating scenario that further outlines the challenges this paper addresses. Section III presents our approach for CPPS software architecture wiring, with the key wiring elements described in Section IV and the subsequent wiring procedure layed out in Section V. A use case demonstrates the feasibility of our approach in Section VI. Section VII then discusses related work, before we conclude this paper and provide an outlook on future work in Section VIII.

II. Motivating Scenario

A simplified production cell in our motivating scenario (inspired by one of our industry partner's production cells) is composed of different participants with the following basic capabilities: an Injection Molding Machine (IMM) with the capability of molding, two 3-axis robots with capabilities to move with a mounted picking unit in and out of a IMM as well as picking/releasing a molded object, and a conveyor belt that transports the parts to another production cell.

A typical production situation of interest for our approach occurs when the IMM completes the moulding procedure. The Robot then moves in, grabs the object, moves out (signals the IMM to continue) and turns to place the object on the conveyor belt. The wiring and interoperability of the robots, the machine, and the conveyor belt is critical. For example, the robots need to receive a signal when the mold is ready to move in and grab the molded part from the IMM. The Robot also needs to know if the conveyor belt is in the right position and empty to receive the part.

In such a scenario several reasons and occasions for rewiring exist:

- Initial Commissioning: a production cell is assembled for the first time
- Product Variant-Induced Process Changes: For differently formed products, the robot might need to use different grippers. To keep the robot software independent from the gripper, the gripper requires its own controller. With every gripper change, the robot software has to connect to another gripper controller, resulting in rewiring.
- Layout changes: Production processes rely in the capabilities of multiple machines. Rewiring of part of the machines is necessary when individual machines are taken off-line for maintenance, replaced by newer ones, or when resources such as collaborative or lightweight robots are relocated for ghost shifts (i.e., the night or weekend), respectively to match demand.
- Dynamic Product Routing: When flexible transport systems such as AGVs take care of routing a product from one machine to the next, they will interact directly with each machine to synchronize product handover. Thus, an AGV will be rewired to each machine it services, thus has to be able to detect which synchronization protocol (from many) the machine supports.[1]

Following the current traditional wiring methods, several challenges are revealed, such as:

- Determining what are the required capabilities to completely integrate a shopfloor participant: When wiring capabilities, there is no precise specification on a software level about the required interoperable complementary capabilities on the other side of an interface. For example, a loading capability would require an unloading capability on the other machine. Thus, the conveyor belt loading capability needs to be wired to the unloading capability of the robotic arm.
- Missing or Incorrect Wiring: A missing connection (i.e., wire) between two participants might lead to production halt, as a participant keeps waiting for a never dispatched signal. An incorrect wire might cause the same: the signal is sent but doesn't arrive. For example, a robot is wrongly wired to another IMM, thus fails to move into the IMM to pick out a molded part, subsequently blocking further production. Incorrect wires might also cause broken equipment or products if a syntactically correct signal arrives to trigger an unexpected action. For example, in our cell, one robot places one part into the IMM for add-on moulding whereas the second robot retrieves the part after the moulding completes. If the free-signal intended for the first robot is also delivered to the second robot, the two robots will potentially collide when moving simultaneously into the IMM.
- Insufficient information hiding: vendors expose information/signals at various levels of the machine's subsystems, requiring expert knowledge which ones are relevant for wiring. Only a few standards exist (e.g., EUROMAP 77 based on OPC UA) that provide semantic models for intercommunication of the capabilities between machines and also these standards lack details how the wiring should be established.

[1]The AGV will most likely cache previous wiring details but—when using the rewiring approach—may react to machine updates (which may come with new synchronization protocol versions).

III. Approach

Our approach to flexible and lightweight wiring of cyber-physical production systems builds on the general idea to expose design, respectively, architectural information at runtime. Making design-time information available at runtime is not novel by itself, but hasn't been realized in a lightweight manner, for wiring up systems dynamically, in the context of industrial environments making use of OPC UA (see our discussion on related work in Section VII).

The key concepts that achieve light-weightness are:
- Independence from a heavyweight modeling language as we provide a simple meta model just for describing capabilities (see Subsection IV-A) without having to describe the participating systems in full detail in a language such as UML/SysML.
- Exposing information on required and provided capabilities at arbitrary locations in the OPC UA nodeset, conceptually similar to using microformats to provide meta data on HTML elements.
- A minimal interface to supply shopfloor participants with (updated) wiring information.

Figure 1 displays the distinction between design-time/deploy-time (i.e., designing the shopfloor participants and starting them on site but without becoming operational, steps a to d) and run-time/commissioning time (i.e., integrating/wiring individual participants to become operational as well as production time, steps 1 to 6).

Individual vendors (through market power) or standardization bodies define a set of capabilities (e.g., the set of EUROMAP[2] standards for the plastics and rubber machinery industry). At design-time engineers at the individual vendors then implement that capability (a), and ensure that their system

[2]https://www.euromap.org/

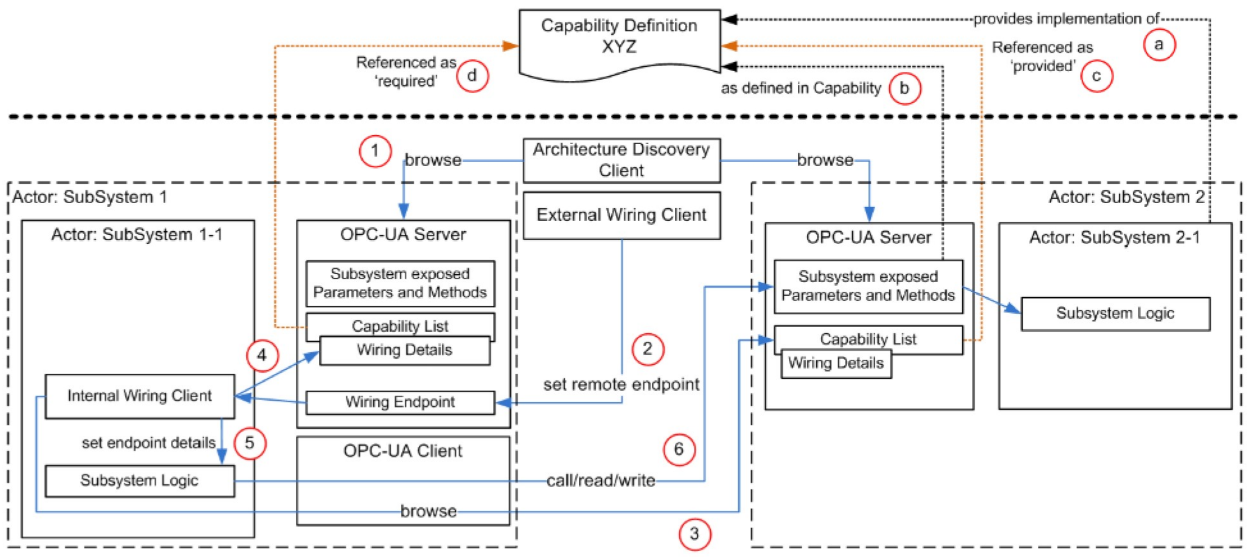


Fig. 1. CPPS Architecture Wiring Approach. The figure displays the main building blocks and their relations using no particular modeling language.

(i.e., actor), when deployed, exposes the capability associated parameters, methods etc. via OPC UA (b). The actor then additionally signals the ability to provide the capability via a reference (c). Other actors signal the requirement to use a particular capability (d) in the same manner. Steps (b) to (d) take effect upon deployment of an actor at the shopfloor but before wiring/commissioning.

At runtime a discovery client (be it as part of an engineer's setup tool or as part of a shopfloor's autoconfiguration mechanism) browses the shopfloor's OPC UA servers merely for capabilities and wiring endpoints (1). The *External Wiring Client* engages with an actor's *Wiring Endpoint* merely to inform it about the location (i.e., server and actor node id) of the required capability's remote counter part (2). The actor's *Internal Wiring Client* thus knows where at the remote OPC UA server to locate the capability's defined methods, parameters etc. (3). It then sets the wiring details (4) and informs the actor's subsystem logic (5) how to connect via OPC UA (6) to the required functionality. Figure 1 doesn't show the interactions for setting the wiring details on the providing actor (i.e., Subsystem 2-1) for sake of clarity.

We outline the detailed discovery and wiring procedure below. Note that the methods, techniques, algorithms, or models for establishing which required capability should be wired exactly to which provided capability (i.e., instance) is outside the scope of this paper as this is a very domain and context-dependent aspect.

IV. Elements for Wiring CPPS

A. Capabilities and Actors

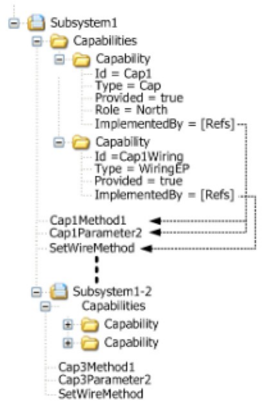
Our approach is based on a core meta-model developed in multiple applied research projects in $Pro^2Future$. The two central, and tightly linked concepts in this meta-model are *Capabilities* and *Actors*. *Capabilities* most closely resembles interface descriptions in traditional architecture description languages such as xADL [4] *Actors* are systems which provide or require capabilities (see *Capability Instances* below) for their production tasks—conceptually similar to components in software architecture, which specify via ports which interfaces (i.e., capabilities) they realize/implement, and which ones they require. The wiring in our approach then describes, only at runtime, which ports become connected.

In CPPS, capabilities range from representing simple activities such as moving a robotic arm, to complex activities such as executing a full production cycle in an IMM. Similar to interfaces, a capability describes what to do (with what input and expected output), but not how to achieve the represented activity, thus hiding actor internal realization details.

Capabilities are not necessarily limited to representing physical activities. A capability may also represent purely software-centric activities (i.e. services) such as planning an optimal route between two machines, sending an alert to a foreman, or updating production statistics in an ERP system.

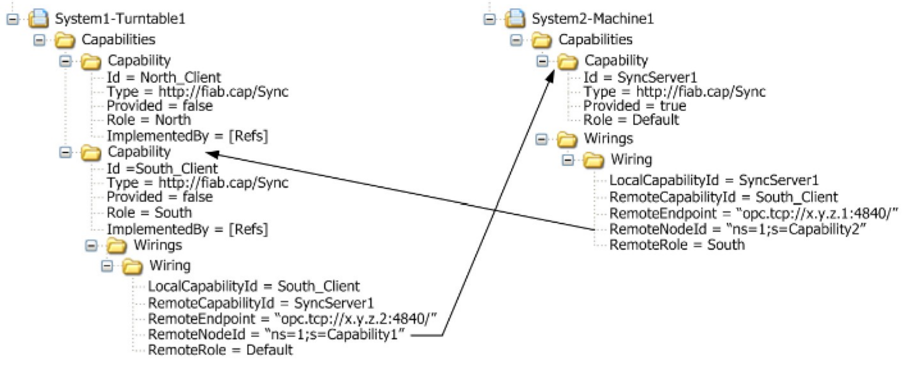
Capabilities may define input parameters to configure the desired result—from simple values like the position of a robotic gripper, up to an xml-encoded set of configuration parameters—and output parameters describing the result. In our approach, atomic capabilities include OPC UA method calls, OPC UA readable and writable parameters, OPC UA events, and OPC UA programs. Depending on such a binding, other actors make use of a capability by sending an event using a publish subscribe protocol, invoking a method, or writing to a parameter. Currently efforts to standardise the interfaces and modeled information are undertaken, however these standards don't specify the wiring procedure.

We foresee that capabilities are published similar to XML schema documents, thus identifyable via an URI. The next subsections outline how actors make use of capability definitions and how they expose capability instances via OPC UA.



```
Subsystem1
  Capabilities
    Capability
      Id = Cap1
      Type = Cap
      Provided = true
      Role = North
      ImplementedBy = [Refs]
    Capability
      Id =Cap1Wiring
      Type = WiringEP
      Provided = true
      ImplementedBy = [Refs]
  Cap1Method1
  Cap1Parameter2
  SetWireMethod
  Subsystem1-2
    Capabilities
      Capability
      Capability
    Cap3Method1
    Cap3Parameter2
    SetWireMethod
```

Fig. 2. Pseudo OPC UA node set hierarchy showing capabilities



```
System1-Turntable1
  Capabilities
    Capability
      Id = North_Client
      Type = http://fiab.cap/Sync
      Provided = false
      Role = North
      ImplementedBy = [Refs]
    Capability
      Id =South_Client
      Type = http://fiab.cap/Sync
      Provided = false
      Role = South
      ImplementedBy = [Refs]
      Wirings
        Wiring
          LocalCapabilityId = South_Client
          RemoteCapabilityId = SyncServer1
          RemoteEndpoint = "opc.tcp://x.y.z.2:4840/"
          RemoteNodeId = "ns=1;s=Capability1"
          RemoteRole = Default

System2-Machine1
  Capabilities
    Capability
      Id = SyncServer1
      Type = http://fiab.cap/Sync
      Provided = true
      Role = Default
      Wirings
        Wiring
          LocalCapabilityId = SyncServer1
          RemoteCapabilityId = South_Client
          RemoteEndpoint = "opc.tcp://x.y.z.1:4840/"
          RemoteNodeId = "ns=1;s=Capability2"
          RemoteRole = South
```

Fig. 3. Wiring example (Pseudo OPC UA node set)

B. Capability Instances

Actors may provide or require the same capability *multiple times*. A turntable (actor) as part of a flexible transport system may receive and distribute pallets from four sides, with machines, robots, or other transport systems placed at these four side. The turntable needs to synchronize un/loading actions with each neighboring actor separately to ensure a successful handover of a pallet. Such synchronization is typically defined as a capability to ensure that actors from different vendors can seamlessly interact using an interoperable interface. The capability then may consist of a state machine describing the synchronization protocol between the two neighboring actors, the methods to trigger transitions in the protocol state machine, and events to monitor the current protocol state. A turntable actor would then provide a separate synchronization capability for each side. A single neighboring actor then only needs to monitor/interact with its assigned (i.e., wired) capability without any further knowledge needed how many other capabilities (i.e., pallet un/loading sides) the turntable actor exposes or what state their synchronization protocols are in. Internally the turntable actor may want to distinguish among its synchronisation capabilities by roles with labels such as "NORTH", "WEST", "SOUTH", and "SOUTH".

C. Exposing required and provided capabilities

In order for actors to become wired up to other actors, they need to expose what capabilities they provide and which they require. To this end, we use the OPC UA node set's hierarchical structure to represent a shopfloor participant's actor hierarchy (see Figure 2). Specifically, an OPC UA server may contain at any hierarchy level a *Capabilities* folder node. We then consider the parent node of this folder an implicit actor, the parent node's id becoming the actor's id. Further *Capabilities* folder nodes further down in the hierarchy represent child actors. A *Capabilities* folder lists all *Capability* references that this actors provides or requires. A *Capability* node consists of following properties:

- Id: (required) host-wide unique identifier of this capability instance.
- Type: (required) URI referencing the capability definitions (similar how XML identifies schema documents).
- Provided: (required) flag for specifying whether the capability is provided (true) or required (false).
- Role: (optional) actor-centric specific role identifier that may be non-unique across the server, useful when more than one capability instance of the same type is required/provided (see above).
- ImplementedBy: (optional) list of references to OPC UA nodes to identify–in the case of multiple provided capabilities of the same type–which parameters and methods (having the same OPC UA browse name) belong to a particular capability instance.

Adding such design information doesn't interfere with a system's OPC UA node set as the exposed information is purely additional, merely identifying and marking up those parts in an OPC UA node set hierarchy that are relevant for wiring. Once the wiring is complete, one could even remove all these OPC UA nodes without affecting the systems functionality (at the cost of not being able to dynamically discover and rewire capabilities any longer).

This meta-model and grounding in OPC UA serves as a foundation for the wiring procedure outlined next.

V. Runtime CPPS Architecture Wiring

A. Decentralized Architecture Recovery/Discovery

An engineer responsible for setting up or rewiring systems on the shopfloor applies an *Architecture Discovery Client* which browses the OPC UA servers on the shopfloor (Figure 1 (1)).[3] The list of servers to browse is available via cached information, an out-of-band mechanism, or by contacting a local discovery server (LDS); all these mechanisms are outside the scope of this paper. Instead of exhaustively (and slowly) browsing top-down through the complete OPC UA node set, the Architecture Discovery Client may simply search for nodes with the *Capabilities* browse name and retrieve all child nodes.

[3]We developed a prototype architecture discovery client described in Section VI

This results in a flat list with entries of type *Capability*. For each capability, the client then navigates towards the root to establish for the underlying machine the capabilities', respectively actors', hierarchical structure. The primary output for each machine is thus a tree of actors, each requiring and/or providing a set of capabilities (each a leaf node in the tree).

For shop floor software architecture recovery, the *Architecture Discovery Client* additionally inspects the discovered *Capability* nodes for *Wiring* nodes and establishes the opposite end of a wiring by extracting the Remote Endpoint and Remote NodeId properties. The client then augments the actor/capability tree with the extracted wirings between the capability nodes (see Figure 3). Hence the engineer can inspect which actors on the shop floor are already set up to communicate with each other, and which required capabilities still need wiring up.

Note that the *Internal Wiring Client* is also described by its respective capability definition (not shown in Figure 1 for sake for clarity but displayed in Figure 2 for the wiring capability). Hence the Architecture Discovery Client also detects the ability of an actor to be dynamically wired up.

B. Executing Wiring Information

Each actor that intends to support dynamic wiring needs to expose a *Wiring Endpoint* identified by the Wiring capability (see for example Figure 2). The Wiring capability defines a single method to provide the wiring details. The *Internal Wiring Client* behind the wiring endpoint is responsible for all required and provided capabilities of its actor, in Figure 3, for example, responsible for SyncNorth and SyncSouth capabilities.

When calling the wiring endpoint (2), the *External Wiring Client* provides following wiring information (based on the example in Figure 3):

- Local Capability Id: (required) which capability out of many is wired, e.g., "South_Client".
- Remote Capability Id: (required) which remote capability requires or provides the counterpart to the local capability, e.g., "SyncServer1".
- Remote Endpoint: (required) host at which the OPC UA capability node is exposed, e.g., "opc.tcp://x.y.z.2:4840/".
- Remote NodeId: (required) node id that identifies the remote capability's root node, e.g., "ns=1;s=Capability1".
- Remote Role: (optional) informational field to avoid having to browse the remote endpoint for retrieving additional wiring information, e.g., "Default".

Actors that only provide capabilities but don't require any themselves may choose to drop the *Wiring Endpoint* and thus won't expose any wiring information. Note that such actors still need to expose capability information in order to be discoverable. In this case, the shopfloor is recoverable from extracting the wiring information from actors that require and are wired to these capabilities.

Next, the *Internal Wiring Client* browses the identified remote capability to retrieve the references from the *ImplementedBy* property to obtain for each method, event, parameter of the capability its respective node id. If this is successful, it exposes the wiring information (4) in the capability's *Wirings* list as one *Wiring* entry (see Figure 3). The client may also decide to persist the wiring information on the local file system or another mechanism such as an external service to enable recovering the wiring information upon a system restart. The persistence mechanism is out of scope of this paper. The *Internal Wiring Client* then provides this list to the Subsystem Logic (5).[4] The client needs to be capability-aware as it has to know exactly which methods, parameters, events the subsystem logic requires and how the subsystem logic will make use of these. Ultimately, the subsystem logic accesses the required capabilities via an *OPC UA Client*.

In case of sending wiring information to the actor providing the capability (not shown in Figure 1), the *Internal Wiring Client* may merely expose the wiring information in the provided capability's *Wirings* list. It then informs the subsystem logic of the remote actor that will potentially make use of the locally provided capability. Whether the subsystem logic then prepares for such a usage (e.g., making system resource available, updating access rules, etc.) depends on the capability purpose and implementation.

C. Addressing the Challenges

Together, these elements enable to address the challenges outlined in Section IV:

- Determining the required capabilities to completely integrate a shopfloor participant is supported by inspecting the required capabilities.
- Missing and Incorrect Wiring are quickly found by identifying empty wiring information, respectively mismatching capability identifiers. Additionally the *Architecture Discovery Client* rejects wiring of incompatible capabilities, and then again the *Internal Wiring Client* ensures all required methods and field are indeed provided.
- Information hiding is achieved by limiting the need to inspect the OPC UA node set to those node areas identified by a capability.

VI. Use Case-based Evaluation

We demonstrate the feasibility of our approach by outlining how we realized the various wiring components in the scope of a lab-scale production cell model depicted in Figure 4.

A. Use case 1 - lab-scale production cell model

Our lab-scale production cell model aims at illustrating the concepts that enable flexible production and the need for software to achieve this. Our cell model has the capability to customize the drawings on a piece of paper at multiple plotting stations. The production plant consists of following machine types: input stations that provide pallets with paper, plotters that load the pallets and draw images, turntables that transport

[4]Depending on implementation and integration of the Internal Wiring Client, it may hand over an OPC UA client instance to the subsystem for immediate use (tighter integration) or merely write the now confirmed node ids to wellknown variables accessible by the subsystem logic (loose integration).

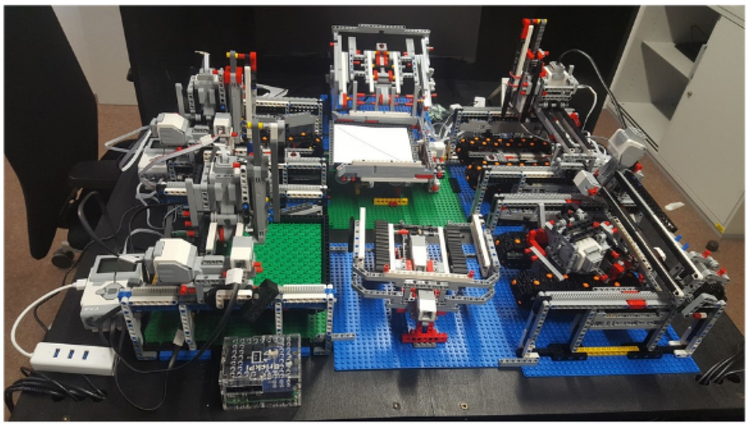


Fig. 4. The Factory in a Box, a lab-scale production cell

the pallets between plotters, and finally, output stations where the finished product (i.e., paper) is retrieved. We use Lego Mindstorm EV3-based PLCs as basis as this allows for rapid and cheap machine prototyping without the need to test individual subcomponents such as motors, actuators, and sensors. Also it ensures their seamless integration. Communication between the machines is purely based on OPC UA. Plotters and input/output stations are designed and programmed with the IEC 61499 industry standard using the open source IDE Eclipse 4diac and the respective FORTE runtime environment integrating the Open62541 OPC UA server. The turntables are implemented in Java, using the Eclipse Milo OPC UA framework. Hence, despite the toy character of the setup, the used software and communication infrastructure is industrial grade.

A turntable provides the central transport infrastructure. Each offers four sides for attaching machines, input/output stations, or other turntables. Hence, for our setup of two turntables placed side-by-side, we are able to dynamically connect six machines: input/output stations and plotters.

This model cell exhibits the same reasons for rewiring as outlined in the motivating scenario: At initial commissioning all these machine need a first wiring to know who to communicate with to synchronize the exchange of pallets. Product Variant-induced Changes: a plotter's print head may be replaced by a stamping head. Layout changes: whenever the layout changes, e.g., to replace a printer, it needs to be wired to a turntable. Dynamic Product Routing: adding another turntable requires the transport system to know exactly which turntable is wired to what other turntable via which synchronization capability instance.

To demonstrate dynamic rewiring, each input/output station and plotter exposes via OPC UA their single synchronization capability (see "SyncServer" capability in Figure 3 left). Turntables expose four such synchronization capabilities with roles "North", "South", "East", and "West". In our demonstrator only the turntables provide a Wiring capability (i.e., expose a Wiring endpoint) to demonstrate the ability of our approach to handle scenarios where only partial wiring information is available.

The same wiring challenges as outlined in the motivating scenario exist: A missing wiring between a turntable and plotter halts production as the two machines cannot coordinate the pallet handover. An incorrect wiring would result, e.g., in synchronizing with the wrong plotter that would then potentially unload the pallet towards an plotter that is oriented to another machine. An engineer wiring up plotters and turntable has all complexity of the turntable, conveyor, and plotting subsystems hidden from them and needs only to focus on the synchronization capability related OPC UA parameters.

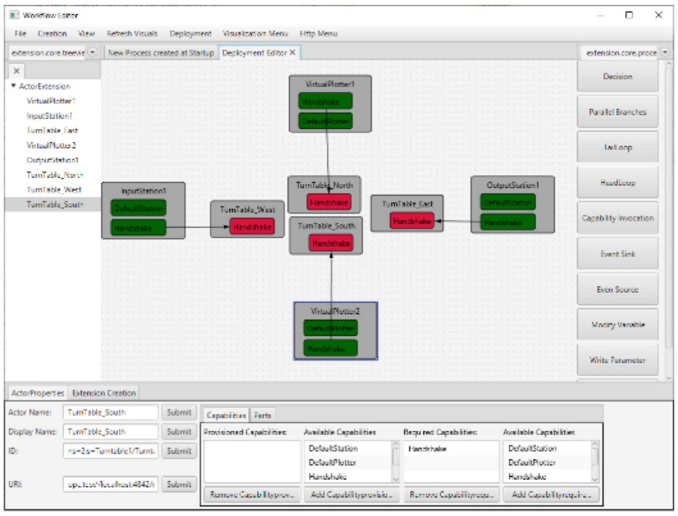
In out case, a dedicated visual editor provides even more support. With our editor (shown in Figure 5 we discover the shopfloor participants and their capabilities[5] (and any existing wirings) and subsequently allow the engineer to control how the capabilities need to be wired up without needing detailed expertise of the participants' OPC UA node set hierarchy. While the matching of required to provided capabilities occurs in the editor, the availability of all methods and variables part of the synchronization capability are done by the turntable, which thus are agnostic whether they synchronize with a plotter, an output station, or another turntable, each one having the synchronization capability at a different location in their OPC UA node set.

B. Discussion

The application of our approach the lab scale production cell using different OPC UA stacks demonstrates the general feasibility of our approach for dynamically wiring CPPS at the software architecture level.

We identified the OPC UA server as a suitable location on the shopfloor for exposing capability information as close to the implementing actor as possible. We note that the basic, out-of-the-box OPC UA mechanism for exposing an actor's information and means of communication limits dynamic wiring scenarios, as an OPC UA server by itself only describes the provided capabilities, but has no built-in dedicated mechanism

[5]Being work in progress, the editor cannot display the actor hierarchy yet.



Actor Name: TurnTable_South Submit
Display Name: TurnTable_South Submit
ID: ns=2;s=Turntable1/Turnt Submit
URI: opc.tcp://localhost:4842/ Submit

Provisioned Capabilities:	Available Capabilities:	Required Capabilities:	Available Capabilities:
	DefaultStation	Handshake	DefaultStation
	DefaultPlotter		DefaultPlotter
	Handshake		Handshake

Fig. 5. Architecture Discovery Client prototype for inspecting and reconfiguring wiring between systems: here an excerpt from the lab scale cell consisting of a turntable's synchonization subsystems in the middle (require capabilities displayed in red), and two plotters, input station, and output stations, all of them providing synchronization capabilities (depicted in green).

for signalling required capabilities. This paper provides an approach to overcome this limitation.

The use case also shows that by modeling explicit architectural information, a significant amount of details (and thus complexity) on the OPC UA servers can remain hidden. By accessing only the capability definitions, any client (e.g., the Architecture Discovery Client) does not need to understand the irrelevant parts of the OPC UA node set. For wiring up the input station, output station, turntables, and plotters, one does not need to know where to search a plotter for the transport synchronization methods and state variable.

The use cases also showed that discovery and wiring is feasible in a decentralized, dynamic manner. Instead of having to register all required and provided capabilities at a central entity (e.g., as the Arrowhead framework supports for service-oriented composition), participants just expose such information locally and apply a dedicated wiring mechanism with dedicated wiring logic (e.g., an expert) to connect the various shopfloor participants. Additionally, architecture recovery is possible without having to centrally aggregate and maintain a view of the whole shopfloor. Recovery may focus on a particular machine or shopfloor segment initially, then incrementally extend to the needs of the particular task at hand, especially if an engineer's tacit knowledge is required for deciding on wiring changes, e.g., understanding which plotting machine needs to be connected to which turntable when switching around plotters.

VII. Related Work

Software architecture research is an active topic in the cyber physical (production) systems community.

Alternatives to the automation pyramid are investigated. Thramboulidis et al. [6] investigate the usage of CPS as microservices. Pisching et al. [7] propose to use service-oriented architectures for CPPS and define a layout for CPS to behave as services. Others like Kastner and Ismail [8], Hussnain, Ferrer and Lastra [9], or Spinelli et al. [10] develop architectures, based on patterns studied well already in software architectures. Their intent is to improve the interoperability between components, but the prevalence of "vertical integration" mirrors the hierarchical concept of the "automation pyramid".
None of the works above considers frequent runtime reconfiguration of software interfaces.

Oreizy, Medvidovic and Taylor [11] gathered an extensive survey on existing solutions and styles for flexible software. Patterns for delegate multi-agent systems [12] allow great reconfigurability at the level of replacing and rewiring components but haven't been evaluated in an manufacturing context yet.
Many agent-based works focus on reconfiguration. Many architectures are proposed and demonstrated based on platforms that do not consider ongoing standardisation efforts: Batista et al. [13] use OpenCOM, Rodrigues et al. [14] work directly at TCP/IP layer, Vogel-Heuser et al. [15] demonstrate their architecture via VPN channels and Web Sockets. Safi et al. [16], as well as Cornejo et al. [17] use Java environments.
A second group of multi-agent works focuses on code generation for machines, like Brennan et al. [18], or Lepuschitz et al. [19], whereas our approach only specifies the interface, and is unaffected by the choice of programming language.
Dias et al. base their framework on OPC UA and use AutomationML to model the node set. The approach only takes into account communication via data fields and omits the use of methods and events. Authors, like Malek, Mikic-Rakic and Medvidovic [20], or Hallsteinsen et al. [21], describe independent solutions, all based on platform-specific "connectors" or "bindings" and platform-independent coordination middleware. Connectors being platform-specific limits the interoperability of the involved machinery.
Prehofer and Zoitl [22] extend this concept of platform-specific access layer (a.k.a. "thin controller") with the capability to receive control code at runtime. Their approach is optimised to only transfer a few next instructions, making a central coordinator necessary. Atmojo et al. [23] solve the challenge of reconfiguration by replacing application logic on the server. The authors themselves state, that real-time requirements are subject to future work. Dassisti et al. [24] propose an architecture for application logic and communication that allows to dynamical regrouping of systems. They do not build on standards like OPC UA.
Above works tackle the reconfiguration aspect of distributed systems, but we find them either limited in their completeness or their applicability to a real production environment.
OPC UA and UML/SysML have been investigated for model-based approaches. Lee et al. [25] show the compatibility of OPC UA und UML by providing a QVT transformation, but do not mention any application to OPC UA server imple-

mentation. Pauker et al. [26] use UML to describe servers' capabilities. They generate code for OPC UA node sets at compile time. They do not consider dynamic reconfiguration. Brecher et al. [27] start from a SysML description of a production cell layout and behavior model to generate code for programmable logic controllers. They neither consider the aspect of dynamic reconfiguration.

We conclude, that the different groups of works presented in this section are not able to provide online reconfiguration of distributed communication between production machinery, as they either focus on the centralised, hierarchical *production pyramid*, are (in their current development state) *not applicable* to a manufacturing environment, or do *not consider reconfiguration* during runtime.

VIII. Conclusions

In this paper, we introduced a lightweight approach for dynamically wiring software architectures for cyber-physical production systems. The key concept of *Actors* exposing provided and required *Capabilities* via OPC-UA allows to decentrally recover/discover how shopfloor actors already communicate with each other, respectively how they can be correctly wired up. We demonstrated the feasibility of our approach on a lab-scale production cell model.

Future work will address how the distributed wiring information may be applied to distributed monitoring of the shopfloor in order to detect wiring inconsistencies or potential bottlenecks (e.g., when too many client require the capability of a single actor). We will also investigate the integration of UML/SysML based interface descriptions as an alternative to our capabilities meta model.

References

[1] P. Hoffmann, R. Schumann, T. M. Maksoud, and G. C. Premier, "Virtual commissioning of manufacturing systems a review and new approaches for simplification." in *ECMS*. Kuala Lumpur, Malaysia, 2010, pp. 175–181.

[2] J. Delsing, *Iot automation: Arrowhead framework*. CRC Press, 2017.

[3] H. Panetto, B. Iung, D. Ivanov, G. Weichhart, and X. Wang, "Challenges for the cyber-physical manufacturing enterprises of the future," *Annual Reviews in Control*, vol. 47, pp. 200 – 213, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1367578818302086

[4] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor, "A language and environment for architecture-based software development and evolution," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 44–53. [Online]. Available: http://doi.acm.org/10.1145/302405.302410

[5] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC unified architecture*. Springer Science & Business Media, 2009.

[6] K. Thramboulidis, D. C. Vachtsevanou, and A. Solanos, "Cyber-physical microservices: An iot-based framework for manufacturing systems," in *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, May 2018, pp. 232–239.

[7] M. A. Pisching, F. Junqueira, D. J. d. S. Filho, and P. E. Miyagi, "An architecture based on iot and cps to organize and locate services," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sept 2016, pp. 1–4.

[8] A. Ismail and W. Kastner, "A middleware architecture for vertical integration," in *2016 1st International Workshop on Cyber-Physical Production Systems (CPPS)*, April 2016, pp. 1–4.

[9] A. Hussnain, B. R. Ferrer, and J. L. M. Lastra, "Towards the deployment of cloud robotics at factory shop floors: A prototype for smart material handling," in *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, May 2018, pp. 44–50.

[10] S. Spinelli, A. Cataldo, G. Pallucca, and A. Brusaferri, "A distributed control architecture for a reconfigurable manufacturing plant," in *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, May 2018, pp. 673–678.

[11] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Runtime software adaptation: framework, approaches, and styles," in *Companion of the 30th international conference on Software engineering*. ACM, 2008, pp. 899–910.

[12] T. Holvoet, D. Weyns, and P. Valckenaers, "Patterns of delegate mas," in *2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, Sept 2009, pp. 1–9.

[13] T. Batista, A. Joolia, and G. Coulson, "Managing dynamic reconfiguration in component-based systems," in *European workshop on software architecture*. Springer, 2005, pp. 1–17.

[14] N. Rodrigues, E. Oliveira, and P. Leitão, "Decentralized and on-the-fly agent-based service reconfiguration in manufacturing systems," *Computers in Industry*, vol. 101, pp. 81–90, 2018.

[15] B. Vogel-Heuser, C. Diedrich, D. Pantförder, and P. Göhner, "Coupling heterogeneous production systems by a multi-agent based cyber-physical production system," in *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 2014, pp. 713–719.

[16] Y. Al-Safi and V. Vyatkin, "An ontology-based reconfiguration agent for intelligent mechatronic systems," in *International Conference on Industrial Applications of Holonic and Multi-Agent Systems*. Springer, 2007, pp. 114–126.

[17] M. A. Cornejo, H. Garavel, R. Mateescu, and N. De Palma, "Specification and verification of a dynamic reconfiguration protocol for agent-based applications," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2001, pp. 229–242.

[18] R. W. Brennan, M. Fletcher, and D. H. Norrie, "An agent-based approach to reconfiguration of real-time distributed control systems," *IEEE transactions on Robotics and Automation*, vol. 18, no. 4, pp. 444–451, 2002.

[19] W. Lepuschitz, A. Zoitl, M. Vallée, and M. Merdan, "Toward self-reconfiguration of manufacturing systems using automation agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 41, no. 1, pp. 52–69, 2010.

[20] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A style-aware architectural middleware for resource-constrained, distributed systems," *IEEE Transactions onf Software Engineering*, 2005.

[21] S. Hallsteinsen, K. Geihs, N. Paspallis, F. Eliassen, G. Horn, J. Lorenzo, A. Mamelli, and G. A. Papadopoulos, "A development framework and methodology for self-adapting applications in ubiquitous computing environments," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2840–2859, 2012.

[22] C. Prehofer and A. Zoitl, "Towards flexible and adaptive productions systems based on virtual cloud-based control," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, Sept 2014, pp. 1–4.

[23] U. D. Atmojo, K. Gulzar, V. Vyatkin, R. Ma, A. Hopsu, H. Makkonen, A. Korhonen, and L. T. Phu, "Distributed control architecture for dynamic reconfiguration: Flexible assembly line case study," in *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*. IEEE, 2018, pp. 690–695.

[24] M. Dassisti, A. Giovannini, P. Merla, M. Chimienti, and H. Panetto, "Hybrid production-system control-architecture for smart manufacturing," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2017, pp. 5–15.

[25] B. Lee, D.-K. Kim, H. Yang, and S. Oh, "Model transformation between opc ua and uml," *Computer Standards Interfaces*, vol. 50, pp. 236 – 250, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0920548916300769

[26] F. Pauker, T. Frühwirth, B. Kittl, and W. Kastner, "A systematic approach to opc ua information model design," *Procedia CIRP*, vol. 57, pp. 321–326, 2016.

[27] C. Brecher, J. A. Nittinger, and A. Karlberger, "Model-based control of a handling system with sysml," *Procedia Computer Science*, vol. 16, pp. 197–205, 2013.