Assessing Adaptability of Software Architectures for Cyber Physical Production Systems

Michael Mayrhofer[1], Christoph Mayr-Dorn[1,2], Alois Zoitl[2], Ouijdane Guiza[1], Georg Weichhart[3], and Alexander Egyed[2]

[1] Pro2Future GmbH, Austria
[2] Institute of Software Systems Engineering, Johannes Kepler University, Austria
[3] ProFactor GmbH, Austria
michael.mayrhofer@pro2future.at

Abstract. Cyber physical production systems (CPPS) focus on increasing the flexibility and adaptability of industrial production systems, systems that comprise hardware such as sensors and actuators in machines as well as software controlling and integrating these machines. The requirements of customised mass production imply that control software needs to be adaptable after deployment in a shop floor, possibly even without interrupting production. Software architecture plays a central role in achieving run-time adaptability. In this paper we describe five architectures, that define the structure and interaction of software components in CPPS. Three of them already are already well known and used in the field. The other two we contribute as possible solution to overcome limitations of the first three architectures. We analyse the architectures' ability to support adaptability based on Taylor et al.'s BASE framework. We compare the architectures and discuss how the implications of CPPS affect the analysis with BASE. We further highlight what lessons from "traditional" software architecture research can be applied to arrive at adaptable software architectures for cyber physical production systems.

Keywords: software architectures · manufacturing · reconfiguration · cyber-physical production systems · adaptability

1 Introduction

Cyber Physical Systems (CPS) tightly interweave software and physical components for integrating computation, networking, and physical processes in a feedback loop. In this feedback loop, software influences physical processes and vice versa. CPS in the manufacturing context are referred to as Cyber Physical Production Systems (CPPS). A production cell involving machines, robots, humans, and transport systems such as pick and place units are examples of a CPPS; Not considered are CPS in general: drones, smart buildings, or medical devices. CPPS increase the flexibility and adaptability of industrial production systems which enables reconfiguration of a physical plant layout with little effort and to produce a higher variety of products on the same layout.

Software, and specifically, software architecture plays a central role in achieving this goal. The general capabilities of a production plant depend on its physical layout. Yet, which capabilities are invoked, in which order and under which conditions is controlled mostly by software or human operators. Thus, fast and cheap reconfiguration can only happen through software designed to allow for adaptability and flexibility. Over the last decades, the software architecture community has focused intensely on these concerns in the scope of "traditional" software systems. In these systems physical aspects such as material flow, manipulation of physical objects, and physical layout of machines and humans, play no or only a marginal role. Little of the work in the software architecture community, however, addresses specifically CPPS. We believe that concepts, approaches, and ideas from software architecture are invaluable for guiding the design of CPPS. In return, we would expect that the constraints and characteristics of CPPS raises awareness in the software architecture community about the implications stemming from the physical world. Software systems inevitably will become increasingly fused with physical object as we can already observe with systems described as Smart Devices that are part of the Internet of Things: Software systems that inherently rely on appropriate software architectures for delivering long-term benefit to the user.

In the scope of this paper, we focus only on adaptability of CPPS (and refer for other, equally relevant, properties for example to [1] as well as future work). Adaptability in CPPS comes in two main categories: adaptation of the software (i.e., machine configuration, process configuration, etc.) and adaptation of the physical layout (i.e., relocating machine, mobile robots, autonomous guided vehicles). Both categories imply software adaptability (see Section 2 and 3). Whether the goal is assessing the current software architecture of an CPPS or deciding upon a future CPPS software architecture: in both cases we need a mechanism to analyse and compare an architecture's adaptability. Rather than determining criteria from scratch, we apply the BASE framework introduced by Oreizy, Medvidovic, and Taylor [2] (Section 4). This framework serves as our basis for evaluating and comparing CPPS architectures. This paper's core contribution is a comparison of five architectures for CPPS with an explicit focus on adaptability: the first three architectures describe the predominant approach to structuring production systems, the latter two architectures are proposed evolutions for increased adaptability (Section 5). We complete the paper with an overview of related work (Section 6) and an outlook on future work (Section 7).

2 Background

Similar to software centric companies, manufacturers aim to remain competitive through a higher innovation rate and an increase in product customization options. The former requires development processes with potential for both agile and parallel development. The ultimate goal is lot-size one: the ability to continuously produce ever-changing product configurations on the same product line at the same low costs as mass production.

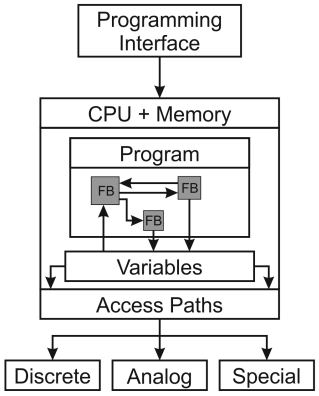

Fig. 1: Simplified PLC architecture layout

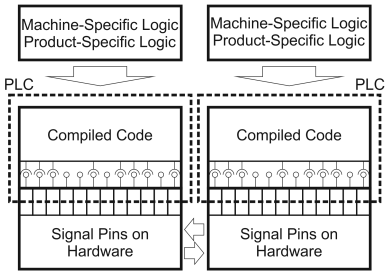

Fig. 2: Direct compilation of control code

From a software architecture point of view, manufacturers face a major challenge: the ability to reconfigure the production environment (the machines, processes, flows, etc) during production time without impairing production pace. At the most extreme end, no two products produced are the same and thus every machine needs involved in the production needs some (software) adaptation for each product produced.

The foundation of today's manufacturing systems are programmable logic controllers (PLCs). PLCs are microcomputers that read data from sensors in the physical world, process the measurement values, and set output voltages to control drives, valves, etc. (see Figure 1). They allow the automation of production steps by controlling a variety of machines across the shop floor: conveyor belts, robots, raw material processing machines, measuring devices, packaging, etc. PLC programs consists of function blocks that access and modify variables, and communicate with other function blocks. The variables map via access paths to the input/output ports. PLC-specific programming environments primarily aim to allow efficient and intuitive creation of such control software (mostly control algorithms). Engineers then compile these programs to the target platform and download/deploy them to the PLC for execution. Adaptation with this approach is cumbersome as we will show in subsection 5.1.

Software on PLSs fall into two major categories. First, low level control software handles sensor data and actuator signals, that is common for machines of the same type. This software can be compared to a hardware-centric API and its implementation: opening/closing a valve, setting the rotation speed of a drive. We refer to this software as *machine-specific code*. Second, there is software that defines parameters for and calls of low-level control, thereby specifying how a machine must behave on the shop floor: when to open a valve, at which force, how fast to run a drive. This software is tailored to a product, changing with a product revision. We, hence, refer to this software as *product-specific code*.

3 Motivating Scenario

We present a simple running scenario to provide more insights into the type of adaptation machines, respectively, software at the shop floor is subject to. We also use this scenario in subsequent sections to exemplify how the various architectures allow adaptation. In production automation, a machine rarely acts independently from other shop floor elements. Typical examples of machine-to-machine interaction include:
· Parts coming from a stamping machine fall onto a conveyor belt for further transportation.
· A robot feeds raw material to an automated machine tool (e.g., a milling machine) and retrieves the processed product.
· A robot holding a part, while a second robot processes it (e.g., spray painting or welding).

In our scenario, we assume a milling machine controlled with an attached robot arm for removing processed parts and sorting them on trays. Milling machine and robot are controlled by one PLC each for sake of simplicity. Even small scale industrial environments such as the VDMA OPC-UA demonstrator[4] are too complex to be described in adequate detail here, let along discussing its adaptability aspects. Our scenario picks out a part of such a setup that is sufficiently rich for discussing the impact of software architecture on adaptability. Traditionally, with little or no product change, engineers custom tailor the software for the PLCs specifically for a particular product. Here the software controls the movement, speed, and force of the milling machine's cutter as well as the robot arm's gripping position, force, and moving path.

With increasing demand for adaptability, two orthogonal adaptation dimensions emerge. On the one hand, we distinguish between the level of adaptation, and on the other hand we differentiate according to the locality of adaptation. The former describes adaptation of product-specific vs machine-specific code, while the latter separates adaptation within a machine invisible to the outside (local) from adaptations affecting multiple machines (distributed). Adaptation example for resulting four types include:

Machine-specific / Local Robot manufacturers continuously improve the control algorithms used in robots and offer frequent updates to existing robots on the shop floor. Robot manufacturers may introduce new algorithms that allow for simpler programming of gripping instructions or arm movements.

Product-specific / Local With lot-size one product customizations, the milling machine might have to cut away at different locations at the raw part, thus requiring different control parameter for each product.

Product-specific / Distributed With lot-size one production, when the raw part size changes between products, then (in addition to the milling machine control software) the robot arm control software needs new parameters for different gripping positions and movement paths to avoid dropping the processed product or bumping it against the milling machine.

[4] `https://www.youtube.com/watch?v=pUtSA8g9owY`

Machine-specific / Distributed the manufacturer decides to switch among the milling machines communications capabilities from WiFi to 5G for communicating with the robot. Assuming that the robot supports both wireless standards, now also the robot control software needs to switch connections.

4 Introduction to BASE

The BASE framework developed by Oreizy, Medvidovic, and Taylor defines four orthogonal criteria to evaluate software systems for their runtime adaptability. In this section we will summarise these criteria and outline how they match CPPS. For a detailed explanation of the framework itself refer to [2].

Behavior: How are changes to the behavior represented and applied? Is behavior limited to a combination of atomic capabilities or is it possible to introduce completely new behavior?
Changes to machine-specific behavior can come in different forms. New functionality (e.g., enabling the milling machine to create curves and arcs) can be introduced, or existing functionality can be improved (e.g., extending the current control algorithm) or replaced. Outdated functionality needs to be removed to create space for new functionality. Changes to the physical architecture (e.g., upgrading to the 5G communication standard) require updates of the drivers.
On the product level, the order of calling the different machine capabilities will change with every product. In addition, the machine configuration (e.g., cutting speeds and control parameters on milling machines, gripping forces and tool tip position on handling robots) needs to be altered, especially when the next product needs different hardware clamps, drills, etc. What looks like a matter of configuration is indeed (physical and software) adaptation (see also Sec.5.1).

Asynchrony: How does the update process affect the system's execution? Is it necessary to halt the system until the updated has completed, or can it resume after already after a partial update? How would correct execution be guaranteed in case of partial updates?
Given the combination of milling machine and handling robot, it might be desirable to update the robot's motion algorithm or positions for a new product while it is still handling the current product. In general, this aspect focuses on the architecture properties that ideally allow elements of a CPPS to be adapted without negatively affecting others, e.g., enabling the milling machine to start producing while the robot is still under reconfiguration.

State: How does the system react to changes in state? How does it deal with altered types? Does a state change require an interrupt of the system's execution? In CPPS, we primarily distinguish between managing product-specific state (i.e., which steps/phases are complete, which ones are currently active, what needs to be done next) and machine-specific state (e.g., current drill rotation speed or robot arm position, whether a product is inside the machine).

Execution context: Constraints on system parts that determine when adaptations are allowed. E.g. the system has to be in a safe state, heap has to be empty, system has to be disconnected from surrounding systems, ... While

Asynchony focuses on the timing of the ongoing adaptation actions, *Execution context* highlights adaptation pre-conditions. For example, does an architecture allow the algorithm controlling the tools position to be updated during execution or only when the gripper has released the part? Can we update the cutting force estimator on the fly? Do we need to shutdown the robot to alter the path planning? And, if milling machine and robot are working together in one cell, do we have to halt the milling machine while updating the robot? Might such dependencies cascade further across several machines, or even whole cells?

In the next section, we apply the BASE framework to evaluate the adaptability of five architectures: three reference architectures and two proposed evolutions thereof.

5 Architecture Analysis

Our goal is assessing how adaptable various CPPS architectures are. Ideally they are adaptable and response enough to change the behavior in nearly zero time. Recall, that we distinguish software according to product-specific code and machine-specific code. A major difference among the discussed architectures is how intertwined these two code types become at runtime (i.e., on the PLC). We assess each architecture with BASE in general and outline how the adaptation actions from our motivating scenarios may be implemented. Across all architecture Figures 2 to 6, arrows pointing down indicate transfer of artifacts (code and/or models) while left to right arrows indicate communication among machines.

5.1 Hardcoded and Physically Wired

In the most prevalent solutions, the engineer tightly weaves the product-specific code with the machine-specific code. Machine-specific code is available as include-files at compile time and is transferred upon each software update to the controller together with the product-specific code. Transfer occurs often at runtime when an Manufacturing Execution System (MES) deploys the software before each production process. This process of "direct compilation" is depicted in Figure 2. Communication among several PLCs occurs primarily via digital pins, thus hard-wired at the hardware level. This approach matches the strict resource limitations of cheap PLCs. Control code is translated directly to machine code, allowing for fast execution and minimizing memory footprint. On the downside, this architectural style comes with significant limitations:

Behavior: An adaptation implies changes to the software regardless whether is product-specific code or machine-specific code. To effect the changes, the complete application needs recompilation and retransfer to the PLC. Unsurprisingly, this approach allows adaptations of existing behavior as well as introduction of completely new behavior.

Asynchrony: the system is unavailable for the duration of shutting down, software replacement, and restarting.

State: Due to wholesale software replacement and system shutdown, any state has to be persisted prior to shutdown or is lost. No separation of machine-specific state from product-specific state exists.

Execution context: The machine has to reach a safe state for shutdown. During software redeployment, therefore, the machine is unable to continue production or communicate with connected machines. Shutdown needs to be signalled to connected systems to allow them to gracefully react to the unavailable machine undergoing updating. Otherwise connected machines might malfunction due to missing signal values or alternatively have to be shutdown likewise.

Suppose the machines from our motivating scenario are implemented according to this reference architecture, the specific adaptation consequences are the following. The tight coupling of machine-specific and product-specific code implies that regardless whether the changes are new or improved gripping algorithms, or whether these are different milling parameter, the respective machine needs to reach a safe-state and subsequently be shutdown. In addition, the tight coupling among machines on the hardware level requires stopping (or even shutting down) and later restarting of the non-updated machine as well. An engineer, hence, needs to consider how the affected machine-under-adaptation is connected to other machines before effecting an update.

5.2 *Central Coordinator Architecture*

The *Central Coordinator Architecture* exhibits a clear separation of machine-specific logic and product-specific logic. Each PLC exposes its functionality (e.g., Function Blocks) as higher-level, composable endpoints (i.e., explicit interfaces). The endpoints' granularity depends how the underlying machine is typically used: i.e., how much fine-grained control is needed. See Figure 3 for an illustration. The defacto protocol for discovery, endpoint provisioning, and invocation in CPPS is OPC-Unified Architecture (OPC-UA) [3] (standardized in IEC 62451). The Centurio Engine [4] is an example for such an architecture.

The machine-specific details behind the exposed endpoints remain opaque to the production process engineer. Typically only engineers at the machine manufacturer—or dedicated integration experts that customize the machine for a particular shop floor—develop and adapt software at the PLC level (including middleware for exposing endpoints).

An engineer discovers the PLCs' endpoints and specifies the control-flow of endpoint invocations and invocation parameter values in a model. The engineer sends the model to the centralized coordinator and triggers its execution. Note that this coordinator is central only with respect to the involved PLCs and not with respect to the overall shop floor. Communication between PLCs occurs indirectly via the centralized coordinator. Production processes with time critical invocation sequences require locating the centralized coordinator close to the involved machines, respectively, PLCs, and/or communication over appropriate network infrastructure such as TSN (time-sensitive networking). Based on BASE, we make the following observations:

Behavior: An engineer specifies product-specific changes as changes to the production process model. The centralized coordinator's capabilities determine whether an updated process model replaces a currently active process wholesale, or whether it applies only the differences. Two options exist to obtain different behavior of machine-specific logic: On the one hand, choosing among different existing behavior occurs via the production process by invoking different endpoints (e.g., for a different algorithm) or using different invocation parameters. On the other hand, radically new functionality needs to be deployed to the PLC via side-channels.

Asynchrony: While switching among pre-existing functionality at the machine-specific level when triggered by the centralized controller is instantaneous, new functionality requires shutting down the machine for the duration of deploying new function blocks and making them available via the middleware. Such a shutdown implies pausing the current product model at the centralized coordinator, and thereby also potentially any other involved machine. However, scheduling a updated production process model for execution at the centralized coordinator upon completion of the currently running process are instantaneous. In-situ changes to running processes may require longer when the process needs to reach a certain stage before updating can safely occur. Changes to the production process become necessary when an interface of the exposed endpoints is affected. However, other machines, respectively PLCs, remain unaffected.

State: Product-specific state is managed in the centralized coordinator while machine-specific state remains within the PLC-level middleware. Updating the product-specific meta-model requires stopping the production, persisting the state, transforming the persisted state to the new meta-model. Such an adaptation typically also requires updating the centralized coordinator but not the machine-specific logic. Machine-specific state is represented by the underlying physical state of the machine and hence readily obtainable via reading from the PLC's hardaware signal pins.

Execution Context: Wholesale replacing product-specific logic requires the centralized coordinator to bring the current model to a safe state. A safe state typically describes a situation where the involved machines equally reach a safe state (e.g., idle) or require no input from the coordinator for the duration of the adaptation. In-situ adaptation of the product-specific logic requires product engineering know-how at which state fragments of the model can be updated quickly enough before the coordinator will access them and given the constraints among model fragments. Adaptation of the centralized coordinator itself requires putting all PLCs in a safe state. Switching among pre-existing machine-specific logic is only restricted by the machine-state, i.e., whether the desired invocation of an endpoint is valid at that particular time, but remains independent of the state of other machines. Adding new functionality at the machine-level typically requires PLC shutdown and hence requires the centralized coordinator to reach a save (product-specific) state first.

Suppose the machines from our motivating scenario are implemented according to this reference architecture, the specific adaptation consequences are the

following. Product-specific updates are straight forward implemented via the model and loading this into the centralized coordinator. There is no differences whether the update affects only the milling machine or also the robot arm as neither machines maintain product-specific control software. Machine-specific adaptations are limited to the machine-under-adaptation: updating the gripping algorithm may not even require stopping the milling machine if sufficient time remains to deploy the new algorithm on the robot's PLC and bind it to the endpoint in use by the centralized controller. Alternatively, the centralized controller would bring the milling process to a save state and wait for continuation once the robot arm becomes operational again. Even machine-specific changes that affected multiple machines in *Baseline Architecture* become strongly decoupled. Switching to 5G on the milling machine, for example, would only affect the communication between the milling machine and the centralized controller (assuming that the controller supports this on the fly), but not the robot.

Overall, this architecture/approach is typically applied in the batch automation domain (e.g., pharma, food, beverages) where the product model is a so-called *recipe* (e.g., recipe for producing aspirin) defined in ISA 88.[5] The *Central Coordinator Architecture*, however, is not limited to this standard.

5.3 *61499 Architecture*

The IEC 61499 standard (and hence this architecture's name, Figure 4) defines a mechanism for specifying and loading product and machine-specific logic in the form of Function Blocks on the fly. To this end, each PLC hosts a run-time environment (RTE) that executes configurations of function blocks including the communication among function blocks across PLC boundaries. A central model consisting of Function Blocks (algorithmic units for computation, signaling, I/O control etc) and their wiring represent the product and machine-specific logic. Any separation between these to types is implicit and depends on a respective well designed model. While function blocks allow reuse and thus separation of machine-specific functionality, the RTE makes no such distinction and merely requires all logic (of all required function blocks) to be provided in an executable format. The mapping procedure of function blocks across PLCs (and respective RTEs) includes the automatic generation of communication proxies and hence allows function blocks to transparently communicate across PLC boundaries. Strasser et al. [5] describe an exemplary implementation of such an architecture. The VMDA demonstrator, referred to in the scenario description, shows the latest state-of-the-art realization of a shop floor by following *61499 Architecture*.

Behavior: The 61499 standard defines the ability how to change, replace, and rewire any function block on the fly.

Asynchrony: Given the finegrained adaptation capabilities, before adaptation, the impact of the adaptation must be evaluated to specify safe condition when to effect a change. Both changes in product and machine logic require

[5] `https://www.isa.org/templates/one-column.aspx?pageid=111294&productId=116649`

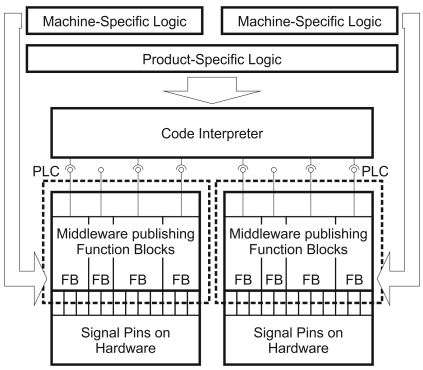

Fig. 3: *Central Coordinator Architecture*

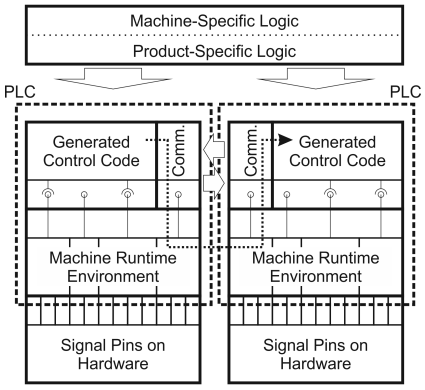

Fig. 4: *61499 Architecture*

compilation to intermediate code and to transfer it to the PLC. The RTE's mechanisms for code transfer support transferring deltas thus, reducing network load.

State: The RTE allows to employ algorithms for complete state transfer. This transfer has to be planned in detail beforehand, together with the code compilation. State required by dependent systems can be kept in memory until the adaptation is complete. This is safely possible as IEC 61499 assumes that physical states (positions, velocities, temperatures, ...) do not jump, thus do only deviate little from one time step to the next.

Execution Context: With the RTE's capabilities of replacing code at runtime while keeping the state in memory (or, if necessary, updating state changes based on estimates) there are no restrictions to adaptation, from the software perspective. The planning of state transfer might become tedious, especially if states are removed or added, but not infeasible. The main limitation is, that the controlled system, the physical system, has to be in a safe state.

Without a clear, dedicated boundary between machine-specific and product-specific logic, any kind of local adaptation are possible on the fly if the timing permits, i.e., the change is completed before the change logic segment is accessed/used by the RTE again. Distributed changes such as switching to 5G or updating product dimension requires to synchronize the changes application on the milling machine and on the robot. Hence, adaptation planning requires in-depth domain knowhow of the milling machine and the robot at product and machine level to identify safe states.

5.4 *Coordination Middleware Architecture*

Having analysed the properties of these three architectures, we propose *Coordination Middleware Architecture* depicted in Figure 5 as the next logical evolution step towards more adaptability. Similar to the *Central Coordinator*

Architecture, an engineer describes the product-specific logic in a central model and subsequently assigns model fragments to various execution resources (i.e., the PLCs). In contrast to the *61499 Architecture*, there exists a strict separation of product-specific logic and machine-specific logic. A local middleware on each PLC interprets the product model fragments and calls the respective machine-specific code. The model fragments contain information for registering itself at the "shopfloor service bus" (SSB), in essence a coordination middleware. The SSB enables registering endpoints, subscribing to events, and dispatching messages. The SSB is responsible for routing messages and events among the participating PLCs. The local middleware obtains only local view of the overall production process without insights into which other entities are involved as the SSB is the only means for external communication. The SSB thereby constitutes a powerful location for adaptation support due to strong decoupling of machines: information mapping, message/event buffering, machine availability signalling, fail-over handling, etc.

Behavior: Adapting Product-specific logic implies transferring any changes from global model to local fragments. New functionality on machine level requires either recompilation of the middleware, if using a hardcoded interpretation middleware, or transfer of the deltas, if using a RTE as in *61499 Architecture*.

Asynchrony: Distributing updates to local product-specific logic fragments occurs independently from changes to other fragments while the machine continue to produce. Introduction of new machine-specific code without downtime is dependent on the capabilities of the middleware/RTE.

State: Product state needs to be persisted when adaption implies replacing a complete product fragment during production. Alternatively, applying deltas to the product model fragment preserves such state. For impact on machine-state, see *Central Coordinator Architecture*.

Execution Context: Updating (or replacing) a process fragment requires it to be in an safe state, i.e., where its not expected to react before the end of the adaptation procedure. The SSB enables PLCs to deregister during non-instantaneous adaptations or maintenance (both at product fragment level and machine logic level). The SSB may then signal other participants to suspend, involve a failover machine (e.g., use another robot), or it temporarily stores events and messages until the adapting PLC becomes available again. This limits the impact on other machines when a PLC needs to be shutdown for machine-level adaptations.

The specific adaptation consequences for our motivating scenario are very similar to *Central Coordinator Architecture* for machine-specific and product-specific adaptations. With respect to product-logic adaptation: adaptations can be effected on the fly. However, while distributed product-specific adaptations such as different product dimensions requiring different gripping locations may be distributed to machine and robot at different times, these adaptations have to be made effective simultaneously which incurs coordination overhead. Machine-specific distributed adaptation such as switching to 5G requires also the SSB

seamlessly use that communication means, making the change transparent to the robot.

5.5 *Distributed Middleware Architecture*

A further evolution of *Coordination Middleware Architecture* results in *Distributed Middleware Architecture*. It merges the strong separation of product-specific and machine specific logic of the *Central Coordinator Architecture* with the peer-to-peer communication and on-the-fly updating capabilities of the *61499 Architecture*, without having a central communication bottleneck as in the *Coordination Middleware Architecture* (see Figure 6. An SSB is often not feasible due to performance reasons (latency, throughput) or infrastructure availability. It effectively becomes distributed across the participating systems and integrated in the local middleware there. This implies that participating systems need to discover other participants, become aware of their role in the product-specific model, subscribe for events, and track their availability. Consequently adaptation support such as message caching, fail-over, etc becomes more complex. Given the similarities to the other architectures, the analysis with BASE yields few differences.

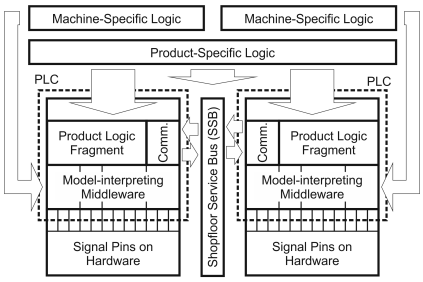

Fig. 5: *Coordination Middleware Architecture*

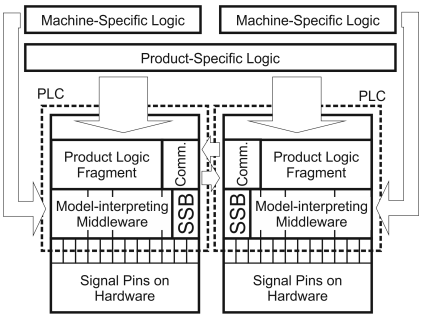

Fig. 6: *Distributed Middleware Architecture*

Behavior: similar to *Coordination Middleware Architecture*

Asynchrony: similar to *Coordination Middleware Architecture*, despite the fact, that there is no central, consistent view on the machine availability (formerly available at the SBB) but is maintained distributed and hence typically only eventually consistent.

State: similar to *Coordination Middleware Architecture*.

Execution Context: similar to *Coordination Middleware Architecture*, Adaptation that requires multiple model fragments to be simultaneously updated for correct production requires a dedicated coordinator mechanism for the

adaptations in sync. The middleware/coordinators now need to reach an agreement when to adapt, rather than merely exposing an simple adaptation endpoint for synchronization.

The adaptation implications for our motivating scenario are almost the same as for the *Coordination Middleware Architecture* architecture. Here, machine-specific distributed adaptation such as switching to 5G now requires all communicating parties to complete the switch at the same time.

5.6 Discussion

In the authoritative papers on the BASE framework [6], [7], highlight that Behavior, Asynchrony, State, and Execution identify the central techniques for achieving adaptability: In CPPS separating product and machine specific logic enables defining more precisely what should change, and how that can be changed while keeping the (side) effects local, and managing machine state separate from product(ion) state (see also [2]).

The two general strategies underlying these techniques are making bindings adaptable and using explicit events or messages for communication. These observations also hold true in CPPS. Malleable bindings imply that machines and robots are allocated to the individual production steps as late as possible, e.g., which robot instance maneuvers the product into and out of a particular milling machine instance. In CPPS the physical world limits the bindings to physically available machines, but having the flexibility at the software (architecture) level enables for increased flexibility at the physical level, e.g., replacing a robot, adding one to increase production pace, integrating autonomous transport vehicles. Architectures 2 to 5 make such late binding possible. Architecture 2 allows late binding of the machines to the production process steps, Architecture 3 explicitly focuses on the ability to change the bindings at runtime, Architecture 4 introduces an SSB with capabilities for dynamically routing messages to the right endpoints, with Architecture 5 doing the same but in a distributed manner.

Similarly, events/message achieve strong decoupling among components. There is no shared memory or tight binding. Events allow monitoring and thus provide feedback on the system state, informing adaptation mechanisms when and where to engage. Events further allow replaying, transforming, and enhancing to turn systems interoperable, see architectures 4 and 5.

Maintaining a model of the system (product-specific and/or machine-specific) is a key towards adaptability. Several approaches demonstrate the runtime adaptation based on linking a model, i.e., the system's architecture with its implementation, e.g., [6], [8], [9].

Ultimately, what architecture to select depends on the desired level of adaptability subject to the constraint of the physical properties of the production process and involved machines. An injection molding machine typically will produce many similar parts before the molding form is exchanged (a slow procedure) to produce a different product and thus has different requirements for run-time adaptation compared to a laser cutter that potentially cuts out a different form every time. A second selection criterion is whether the architecture meets the

real-time requirements of two communicating machines. When two robots need to interact to jointly lift a physical object, exchanging messages via an SSB in Architecture D might not be able to deliver messages quickly enough.

6 Related Work

Software architecture research is an active topic in the cyber physical (production) systems community. Ahmad and Babar [1] show that the last decades has seen adoption of software development paradigms in robotics. As robots are a specialisation of CPS, we expect a similar development for the CPS and CPPS community. Pisching et al. [10] propose to use service-oriented architectures for CPPS and define a layout for CPS to behave as services. Thramboulidis et al. [11] investigate the usage of CPS as microservices. Others develop architectures, usually based on patterns studied well already in software architectures [12], [13]. Their goal is to improve the compatibility between components, there is only little focus on runtime adaptation. None of the above works analysed considers frequent software reconfiguration or in-situ adaptation. This is a topic heavily investigated in the software architecture community. Several papers propose a plethora of approaches with many of them being relevant to CPS.

Oreizy, Medvidovic and Taylor [2] gathered an extensive survey on existing solutions and styles for flexible software. Michalik et al. [14] determine which code needs to be updated on a system, based on software product lines. The technology would be a key enabler for lot size one, yet it is left open how the actual software update is executed. Holvoet, Weyns and Valckenaers [15] identify patterns for delegate multi-agent systems that allow great reconfigurability at the level of replacing and rewiring components. They are great visions for future shopfloors, but might need several steps to be introduced in existing manufacturing environments. Fallah, Wolny and Wimmer [16] propose a framework that uses SysML to model and execute a production process. Their approach has a strong distinction between machines and machine operators, which we consider hampering when it comes to mixed scenarios, where machines should be replaced by humans or vice versa. Moreover, the tools of SysML are less suited to model dynamic processes compared to e.g. BPMN or SBPM. Other approaches introduce platform-specific "connectors" [17] or "bindings" [18] and platform-independent coordination middleware. Prehofer and Zoitl [19] extend this concept of platform-specific access layer (a.k.a. "thin controller") with the capability to receive control code at runtime. Though various architectures exist for robotic systems [1], [20], CPPS go in scope beyond a single machine or robot and hence have to satisfy stricter requirements [21], [22].

7 Conclusions

We motivated the need for architectural adaptability in cyber physical production systems. Using the BASE framework, we showed how Behavior, Asynchrony, State, and Execution aspects affect an architecture's adaptability. We presented

three existing and two novel architectures and discussed what makes them adaptable. While not the only architecture selection criterion, being aware of the limits of adaptability of a particular architecture is of uttermost importance when designing for future CPPS.

While this paper focused on the small scale interactions and adaptability of a few machines (and/or robots) for production, the adaptability on higher levels such as covering the complete shop floor are not very well understood yet. Our next steps focus on investigating how architectural styles and patterns apply for adapting at such higher-levels, especially in the presence of the various architectures presented in this paper.

Acknowledgement

Supported in part by ENGEL Austria GmbH and Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184

References

[1] A. Ahmad and M. A. Babar, "Software architectures for robotic systems: A systematic mapping study," *Journal of Systems and Software*, vol. 122, pp. 16–39, 2016.

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

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

[4] F. Pauker, J. Mangler, S. Rinderle-Ma, and C. Pollak, "Centurio.work - modular secure manufacturing orchestration," in *16th Int. Conf. on Business Process Management 2018*, 2018, pp. 164–171.

[5] T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sunder, A. Valentini, and A. Martel, "Framework for distributed industrial automation and control (4diac)," in *2008 6th IEEE Int. Conf. on Industrial Informatics*, 2008, pp. 283–288.

[6] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proc. of the 20th Int. Conf. on Software Engineering*, Washington, DC, USA: IEEE Computer Society, 1998, pp. 177–186.

[7] R. Taylor, N. Medvidovic, and P. Oreizy, "Architectural styles for runtime software adaptation," in *2009 Joint IEEE/IFIP Conf. on Software Architecture European Conf. on Software Architecture*, 2009, pp. 171–180.

[8] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[9] C. Dorn and R. N. Taylor, "Coupling software architecture and human architecture for collaboration-aware system adaptation," in *Proc. of the 2013 Int. Conf. on Software Engineering*, IEEE Press, 2013, pp. 53–62.

[10] 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 Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–4.

[11] 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)*, 2018, pp. 232–239.

[12] 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)*, 2018, pp. 44–50.

[13] 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)*, 2018, pp. 673–678.

[14] B. Michalik, D. Weyns, N. Boucke, and A. Helleboogh, "Supporting online updates of software product lines: A controlled experiment," in *2011 Int. Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 187–196.

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

[16] S. M. Fallah, S. Wolny, and M. Wimmer, "Towards model-integrated service-oriented manufacturing execution system," in *2016 1st Int. Workshop on Cyber-Physical Production Systems (CPPS)*, 2016, pp. 1–5.

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

[18] 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.

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

[20] J. C. Georgas and R. N. Taylor, "An architectural style perspective on dynamic robotic architectures," in *Proc. of the IEEE Second Int. Workshop on Software Development and Integration in Robotics (SDIR 2007), Rome, Italy*, 2007, p. 6.

[21] L. Hu, N. Xie, Z. Kuang, and K. Zhao, "Review of cyber-physical system architecture," in *2012 IEEE 15th Int. Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, 2012, pp. 25–30.

[22] M. Sadiku, Y. Wang, S. Cui, and S. Musa, "Cyber-physical systems: A literature review," *European Scientific Journal*, 2017.